

TLS WG
Internet-Draft
Intended status: Experimental
Expires: 13 October 2022

B. Schwartz
Google LLC
C. Patton
Cloudflare, Inc.
11 April 2022

The Pseudorandom Extension for cTLS
draft-cpbs-pseudorandom-ctls-01

Abstract

This draft describes a cTLS extension that allows each party to emit a purely pseudorandom bitstream.

Discussion Venues

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

Source for this draft and an issue tracker can be found at <https://github.com/bemasc/pseudorandom-ctls>.

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 13 October 2022.

Copyright Notice

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

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights

and restrictions with respect to this document. Code Components extracted from this document must include Revised BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Revised BSD License.

Table of Contents

1. Conventions and Definitions	2
2. Introduction	3
2.1. Background	3
2.2. Goal	3
2.2.1. Requirements	3
2.2.2. Non-requirements	4
2.2.3. Experimental status	4
3. The Pseudorandom Extension	4
3.1. Form	4
3.2. Use	5
3.2.1. Key Derivation	6
3.2.2. With Streaming Transports	6
3.2.3. With Datagram Transports	7
3.3. Protocol confusion defense	8
4. Plaintext Alerts	9
5. Operational Considerations	9
5.1. Multiple profiles and key rotation	9
5.2. Computational cost	10
6. Security Considerations	10
7. Privacy Considerations	11
8. IANA Considerations	11
8.1. TSPRP Registry	11
8.2. cTLS Template Key registry	11
8.3. TLS ContentType Registry	12
9. References	12
9.1. Normative References	12
9.2. Informative References	13
Acknowledgments	13
Authors' Addresses	13

1. Conventions and Definitions

The contents of a two-party protocol as perceived by a third party are called the "wire image".

A Tweakable Strong Pseudorandom Permutation (TSPRP) is a variable-input-length block cipher that is parameterized by a secret "key" and a public "tweak". Also known as a "super-pseudorandom permutation" or "wide block cipher".

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. Introduction

2.1. Background

Compact TLS [cTLS] is a compact representation of TLS 1.3 (or later), intended for uses where compatibility with previous versions of TLS is not required. It defines a pre-configuration object called a "template" that contains a profile of the capabilities and behaviors of a TLS server, which is known to both client and server before they initiate a connection. The template allows both parties to omit information that is irrelevant or redundant, allowing a secure connection to be established while exchanging fewer bits on the wire.

Every cTLS template potentially results in a distinct wire image, with important implications for user privacy and ossification risk.

One interesting consequence of conventional wire formats (i.e. not pseudorandom) is the risk of protocol confusion attacks. For example, in the NAT Slipstreaming attacks [SLIPSTREAM], a web server causes a browser to send HTTP data that can be confused for another protocol (e.g. SIP) that is processed by a firewall. Because firewalls are typically focused on attacks arriving from outside the network, malicious payloads sent from a trusted client can trick some firewalls into disabling their own protections.

2.2. Goal

The goal of this extension is to enable two endpoints to agree on a TLS-based protocol whose wire image is purely pseudorandom.

2.2.1. Requirements

- * **Privacy:** A third party without access to the template cannot tell whether two connections are using the same pseudorandom cTLS template, or two different pseudorandom cTLS templates.
- * **Ossification risk:** Every byte sent on the underlying transport is pseudorandom to an observer who does not know the cTLS template.
- * **Efficiency:** Zero size overhead and minimal CPU cost in the simplest case. Support for servers with many cTLS templates, when appropriately constructed.

- * Protocol confusion attack resistance: This attack assumes a malicious server or client that can coerce its peer into sending a ciphertext that could be misinterpreted as a different protocol by a third party. This extension must enable each peer to ensure that its own output is unlikely to resemble any other protocol.

2.2.2. Non-requirements

- * Efficient support for demultiplexing arbitrary cTLS templates.
- * Addressing information leakage in the length and timing of transmissions.

2.2.3. Experimental status

This specification has experimental status (INTENDED). The goals of this experiment include:

- * To assess the internet's tolerance of unrecognized protocols.
- * To gain experience with TSPRPs in a protocol context.
- * To exercise cTLS's extensibility features.
- * To support practical and theoretical research into protocol distinguishability.

3. The Pseudorandom Extension

3.1. Form

A cTLS template is structured as a JSON object. This extension is represented by an additional key, "pseudorandom", whose value is an object with at least two string-valued keys: "tsprp" (a name from the TSPRP registry (see Section 8.1)) and "key" (a base64-encoded shared secret whose length is specified by the TSPRP). For example, a cTLS template might contain an entry like:

```
"pseudorandom": {  
  "tsprp": "exp-hctr2",  
  "key": "nx2kEm50FCE...TyOhGOW477EHS"  
},
```

3.2. Use

The cTLS Record Layer protocol is comprised of AEAD-encrypted ciphertext fragments interleaved with plaintext fragments. Each record is prefixed by a plaintext header, and some records, like those containing the ClientHello and ServerHello, are not encrypted at all. The ciphertext fragments are pseudorandom already, so this extension specifies a transformation of the plaintext fragments that ensures that all bits written to the wire are pseudorandom.

Conceptually, the extension sits between the cTLS Record Layer and the underlying transport (e.g. TCP, UDP). The transformation is based on a TSPRP with the following syntax:

```
TSPRP-Encipher(key, tweak, message) -> ciphertext  
TSPRP-Decipher(key, tweak, ciphertext) -> message
```

The TSPRP specifies the length (in bytes) of the key. The tweak is a byte string of any length. The TSPRP uses the key and tweak to encipher the input message, which also may have any length. The output ciphertext has the same length as the input message.

Pseudorandom cTLS uses the TSPRP to encipher all plaintext handshake records, including the record headers. As long as there is sufficient entropy in the key_share extension or random field of the ClientHello (resp. ServerHello) the TSPRP output will be pseudorandom.

TODO: Check that the assumptions hold for HelloRetryRequest. As long as no handshake messages are repeated verbatim, it should be fine, but we need to check whether an active attacker can trigger a replay.

Pseudorandom cTLS also enciphers every record header. In addition to the header, 16 bytes of the AEAD ciphertext itself is enciphered to ensure the input has enough entropy. Any AEAD algorithm whose ciphertext overhead is less than 16 bytes is not compatible with this specification.

By default, Pseudorandom cTLS assumes that the TLS ciphertext is using an AEAD algorithm whose output is purely pseudorandom, such as AES-GCM and ChaCha20-Poly1305. If the ciphertext might not be pseudorandom (e.g. when handling hostile plaintext), the ciphertext can be "reciphered" to ensure pseudorandomness (see Section 3.3).

3.2.1. Key Derivation

To provide clear separation between data sent by the client and the server, the client and server encipher data using different keys, derived from the profile key as follows:

```
client_key = TSPRP-Encipher(key, "derive", zeros)
server_key = TSPRP-Encipher(key, "derive", ones)
```

where zeros and ones are messages the same size as key, with all bits set to zero and one respectively. This procedure guarantees that client_key and server_key are distinct and would appear unrelated to any party who does not know the profile key.

3.2.2. With Streaming Transports

When used over a streaming transport, Pseudorandom cTLS requires that headers have predictable lengths. Therefore, if a Connection ID is negotiated, it MUST always be included. Normally, when TLS runs on top of a streaming transport, Connection IDs are not enabled, so this is not expected to be a significant limitation.

The transformation performed by the sender uses TSPRP-Encipher() and client_key or server_key. The sender first constructs any CTLSPplaintext records as follows:

1. Set tweak = "hs".
2. Replace the message with TSPRP-Encipher(client/server_key, tweak, message).
3. Fragment the message if necessary, ensuring each fragment is at least 16 bytes long.
4. Change the content_type of the final fragment to ctls_handshake_end(TBD) (see Section 8.3).

Note: This procedure requires that handshake messages are at least 16 bytes long. This condition is automatically true in most configurations.

The sender then constructs cTLS records as usual, but applies the following transformation before sending each record:

1. Let prefix be the first 19 bytes of the record.
2. If the record is CTLSPplaintext, set tweak = "".

3. If the record is CTLSCiphertext, let tweak be the 64-bit Sequence Number in network byte order.
4. Replace prefix with TSPRP-Encipher(client/server_key, tweak, prefix).

OPEN ISSUE: How should we actually form the tweaks? Should we add some kind of chaining, within a stream or binding ServerHello to ClientHello?

3.2.3. With Datagram Transports

Pseudorandom cTLS applies to datagram applications of cTLS without restriction. If there are multiple records in the datagram, encipherment starts with the last record header and proceeds back-to-front.

Given the inputs:

- * payload, an entire datagram that may contain multiple cTLS records.
- * TSPRP-Decipher() and client_key or server_key
- * connection_id, the ID expected on incoming CTLSCiphertext records

The recipient deciphers the datagram as follows:

1. Let max_hdr_length = max(15, len(connection_id) + 5). This represents the most data that might be needed to read the CTLSP Plaintext and DTLS Handshake headers (Section 5.2 of [DTLS13]) or the CTLSCiphertext header.
2. Let index = 0.
3. While index != len(payload):
 1. Let prefix = payload[index : min(len(payload), index + max_hdr_length + 16)]
 2. Let tweak = "datagram" + len(payload) + index.
 3. Replace prefix with TSPRP-Decipher(client/server_key, tweak, prefix).
 4. Set index to the end of this record.

CTLSP Plaintext records are subject to an additional decipherment step:

1. Perform fragment reassembly to recover the complete `Handshake.body` (Section 5.5 of [DTLS13]).
2. Let `tweak` be `"datagram hs" + Handshake.msg_type`.
3. Replace `Handshake.body` with `TSPRP-Decipher(client/server_key, tweak, Handshake.body)`.

3.3. Protocol confusion defense

The procedure described in Section 3.2 is sufficient to render the bitstream pseudorandom to a third party when both peers are operating correctly. However, if a malicious client or server can coerce its peer into sending particular plaintext (as is common in web browsers), it can choose plaintext with knowledge of the encryption keys, in order to produce ciphertext that has visible structure to a third party. This technique can be used to mount protocol confusion attacks [SLIPSTREAM].

This attack is particularly straightforward when using the AES-GCM or ChaCha20-Poly1305 cipher suites, as much of the ciphertext is encrypted by XOR with a stream cipher. A malicious peer in this threat model can choose desired ciphertext, XOR it with the keystream to produce the malicious plaintext, and rely on the other peer's encryption stage to reverse the encryption and reveal the desired ciphertext.

As a defense against this attack, the Pseudorandom cTLS extension supports two optional keys named `"client-recipher"` and `"server-recipher"`. Each key's value is an integer `E` between 0 and 16 (inclusive) indicating how much entropy to add. When the `"client-recipher"` key is present, the client MUST prepend `E` fresh random bytes to `CTLSCiphertext.encrypted_record` before encipherment. The server MUST apply a similar transformation if the `"server-recipher"` key is present.

This transformation does not alter the Length field in the Unified Header, so it does not reduce the maximum plaintext record size. However, it does increase the output message size, which may impact MTU calculations in DTLS.

The sender MUST compute `R` using a cryptographically secure pseudorandom number generator (CSPRNG) whose seed contains at least 16 bytes of entropy that is not known to the peer.

In general, a malicious peer can still produce desired ciphertext with probability 2^{-8E} for each attempt by guessing a value of R . Accordingly, values of E less than 8 are NOT RECOMMENDED for defense against confusion attacks.

4. Plaintext Alerts

Representing plaintext alerts (i.e. `CTLSPplaintext` messages with `content_type = alert(21)`) requires additional steps, because Alert fragments have little entropy.

A standard TLS Alert fragment is always 2 bytes long. In Pseudorandom cTLS, senders MUST append at least 16 random bytes to any plaintext Alert fragment and increase `CTLSPplaintext.length` accordingly. Enciphering and deciphering then proceed identically to other `CTLSPplaintext` messages. The recipient MUST remove these bytes before passing the `CTLSPplaintext` to the cTLS implementation.

QUESTION: Are there client-issued Alerts in response to malformed `ServerHello`?

5. Operational Considerations

5.1. Multiple profiles and key rotation

Pseudorandom cTLS supports multiple profiles on the same server port. If all profiles share the same Pseudorandom cTLS configuration (and the same length of `connection_id` if applicable), the server simply deciphers the incoming data before reading the `profile_id` or `connection_id`.

If multiple Pseudorandom cTLS configurations are in use, the server can use trial deciphering to determine which profile applies to each new connection. A trial is confirmed as correct if the deciphered `ClientHello.profile_id` matches an expected value. To avoid false matches, server operators SHOULD choose a `profile_id` whose length is at least 8 bytes.

Pseudorandom cTLS key rotation can be represented as a transition from one profile to another. If the only difference between two profiles is the Pseudorandom cTLS configuration, the server MAY use the same `profile_id` for both profiles, relying on trial deciphering to identify which version is in use. Trial deciphering is also sufficient to determine whether the client is using Pseudorandom cTLS, so the "pseudorandom" key MAY appear in the template's "optional" section.

Pseudorandom cTLS does not support demultiplexing distinct configurations by `connection_id`. Such use would require both the client and server to perform trial deciphering on every datagram. Instead, clients that implement Pseudorandom cTLS MUST use a distinct transport session (e.g. UDP 5-tuple) for each cTLS profile.

5.2. Computational cost

Pseudorandom cTLS adds a constant, symmetric computational cost to sending and receiving every record, roughly similar to the cost of encrypting a very small record. The cryptographic cost of delivering small records will therefore be increased by a constant factor, and the computational cost of delivering large records will be almost unchanged.

The optional defense against ciphertext confusion attacks further increases the overall computational cost, generally at least doubling the cost of delivering large records. It also adds up to 16 bytes of overhead to each encrypted record.

6. Security Considerations

Pseudorandom cTLS operates as a layer between cTLS and its transport, so the security properties of cTLS are largely preserved. However, there are some small differences.

In datagram mode, the `profile_id` and `connection_id` fields allow a server to reject almost all packets from a sender who does not know the template (e.g. a DDoS attacker), with minimal CPU cost. Pseudorandom cTLS requires the server to apply a decryption operation to every incoming datagram before establishing whether it might be valid. This operation is $O(1)$ and uses only symmetric cryptography, so the impact is expected to be bearable in most deployments.

cTLS templates are presumed to be published by the server operator. In order to defend against ciphertext confusion attacks (Section 3.3), the client MUST refuse to connect unless the server provides a cTLS template with a sufficiently large "client-recipher" value.

TODO: More precise security properties and security proof. The goal we're after hasn't been widely considered in the literature so far, at least as far as we can tell. The basic idea is that the "real" protocol (Pseudorandom cTLS) should be indistinguishable from some "target" protocol that the network is known tolerate. The assumption is that middleboxes would not attempt to parse packets whose contents are pseudorandom. (The same idea underlies QUIC's wire encoding format [RFC9000].) A

starting point might be the formal notion of "Observational Equivalence" (<https://infsec.ethz.ch/content/dam/ethz/special-interest/infk/inst-infsec/information-security-group-dam/research/publications/pub2015/ASPObsEq.pdf>).

7. Privacy Considerations

Pseudorandom cTLS is intended to improve privacy in scenarios where the adversary can observe traffic to various servers but lacks access to their cTLS templates, by preventing the adversary from determining which profiles are in use by which clients and servers. If instead the adversary does have access to some cTLS templates, and these templates do not have distinctive `profile_id` values, Pseudorandom cTLS can reduce privacy, by enabling strong confirmation that a connection is using a specific profile.

When Pseudorandom cTLS is enabled, the adversary can still observe the length and timing of messages, so templates that differ in these can still be distinguished. Implementations MAY use TLS padding to reduce the observable patterns.

The adversary could also send random data to the server (a "probing attack") in order to learn the fraction of messages of each length that produce valid ClientHellos. This "probability fingerprint" could allow discrimination between profiles. Server operators that wish to defend against probing attacks SHOULD choose a sufficiently long `profile_id` that invalid ClientHellos are always rejected without eliciting a response. A 15-byte `profile_id` provides 128-bit security.

8. IANA Considerations

8.1. TSPRP Registry

This specification anticipates the existence of an IANA registry of Tweakable Strong Pseudorandom Permutations (TSPRPs). Until such a registry exists, the value "exp-hctr2" is reserved to indicate the HCTR2 TSPRP [HCTR2].

8.2. cTLS Template Key registry

This document requests that IANA add the following value to the "cTLS Template Keys" registry:

Key	JSON Type	Reference
pseudorandom	object	(This document)

Table 1

8.3. TLS ContentType Registry

IANA is requested to add the following codepoint to the TLS Content Types Registry

This document requests that a code point be allocated from the "TLS ContentType" registry. The row to be added in the registry has the following form:

Value	Description	DTLS-OK	Reference
TBD	ctls_handshake_end	Y	RFCXXXX

Table 2

9. References

9.1. Normative References

- [cTLS] Rescorla, E., Barnes, R., and H. Tschofenig, "Compact TLS 1.3", Work in Progress, Internet-Draft, draft-ietf-tls-ctls-05, 7 March 2022, <<https://datatracker.ietf.org/doc/html/draft-ietf-tls-ctls-05>>.
- [DTLS13] Rescorla, E., Tschofenig, H., and N. Modadugu, "The Datagram Transport Layer Security (DTLS) Protocol Version 1.3", Work in Progress, Internet-Draft, draft-ietf-tls-dtls13-43, 30 April 2021, <<https://datatracker.ietf.org/doc/html/draft-ietf-tls-dtls13-43>>.
- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <<https://www.rfc-editor.org/rfc/rfc2119>>.

- [RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <<https://www.rfc-editor.org/rfc/rfc8174>>.
- [RFC9000] Iyengar, J., Ed. and M. Thomson, Ed., "QUIC: A UDP-Based Multiplexed and Secure Transport", RFC 9000, DOI 10.17487/RFC9000, May 2021, <<https://www.rfc-editor.org/rfc/rfc9000>>.

9.2. Informative References

- [HCTR2] "Length-preserving encryption with HCTR2", n.d., <<https://eprint.iacr.org/2021/1441/20211027:085150>>.
- [SLIPSTREAM] "NAT Slipstreaming v2.0", n.d., <<https://samy.pl/slipstream/>>.

Acknowledgments

TODO

Authors' Addresses

Benjamin Schwartz
Google LLC
Email: bemasc@google.com

Christopher Patton
Cloudflare, Inc.
Email: cpatton@cloudflare.com

TLS
Internet-Draft
Intended status: Experimental
Expires: 23 May 2022

S. Farrell
Trinity College Dublin
19 November 2021

PEM file format for ECH
draft-farrell-tls-pemesni-02

Abstract

Encrypted ClientHello (ECH) key pairs need to be configured into TLS servers, some of which can be built with different TLS libraries, so there is a benefit and little cost in documenting a file format to use for these, similar to how RFC7468 defines other PEM file formats.

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 23 May 2022.

Copyright Notice

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

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

Table of Contents

1. Introduction	2
2. Terminology	2
3. ECHConfig file	3
4. Security Considerations	3
5. Acknowledgements	3
6. IANA Considerations	3
7. Normative References	3
Appendix A. Changes	4
Author's Address	4

1. Introduction

Encrypted ClientHello (ECH) [I-D.ietf-tls-esni] for TLS1.3 [RFC8446] defines a confidentiality mechanism for server names and other ClientHello content in TLS. That requires publication of an ECHConfigList data structure in an HTTPS or SVCB RR [I-D.ietf-dnsop-svcb-https] in the DNS. An ECHConfigList can contain one or more ECHConfig values. An ECHConfig structure contains the public component of a key pair that will typically be periodically (re-)generated by some key manager for a TLS server. TLS servers then need to be configured to use these key pairs, and given that various TLS servers can be built with different TLS libraries, there is a benefit in having a standard format for ECH key pairs, just as was done with [RFC7468].

[[This idea could: a) wither on the vine, b) be published as it's own RFC, or c) end up as a PR for [I-D.ietf-tls-esni]. There is no absolute need for this to be in the RFC that defines ECHO, so (b) seems feasible if there's enough interest, hence this draft. The source for this is in <https://github.com/sftcd/pemesni/> PRs are welcome there too.]]

2. 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.

3. ECHConfig file

The public and private keys MUST both be PEM encoded. The file contains the catenation of the PEM encoding of the private key followed by the PEM encoding of the public key as an ECHConfigList containing exactly one ECHConfig. The private key MUST be encoded as a PKCS#8 PrivateKey. The public key MUST be the base64 encoded form of an ECHConfigList value that can also be published in the DNS. The string "ECHCONFIG" MUST be used in the PEM file delimiter for the public key.

There MUST only be one key pair in each file even if a server publishes multiple public keys in the DNS in one ECHConfigList structure.

Figure 1 shows an example ECHConfig PEM File

```
-----BEGIN PRIVATE KEY-----
MC4CAQAwBQYDK2VuBCIEICjd4yGRdsoP9gU7YT7My8DHx1Tjme8GYDXrOMCi8v1V
-----END PRIVATE KEY-----
-----BEGIN ECHCONFIG-----
AD7+DQA65wAgACA8wVN2BtscOl3vQheUzHeIkVmKIiydUhDCliA4iyQRCwAEAAEA
AQALZXhhbXBsZS5jb20AAA==
-----END ECHCONFIG-----
```

Figure 1: Example ECHConfig PEM file

4. Security Considerations

Storing cryptographic keys in files leaves them vulnerable should anyone get shell access to the TLS server machine. So: Don't let that happen:-)

5. Acknowledgements

TBD, as needed

6. IANA Considerations

There are none so this section can be deleted later.

7. Normative References

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

- [RFC7468] Josefsson, S. and S. Leonard, "Textual Encodings of PKIX, PKCS, and CMS Structures", RFC 7468, DOI 10.17487/RFC7468, April 2015, <<https://www.rfc-editor.org/info/rfc7468>>.
- [RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <<https://www.rfc-editor.org/info/rfc8174>>.
- [RFC8446] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <<https://www.rfc-editor.org/info/rfc8446>>.
- [I-D.ietf-tls-esni]
Rescorla, E., Oku, K., Sullivan, N., and C. A. Wood, "TLS Encrypted Client Hello", Work in Progress, Internet-Draft, draft-ietf-tls-esni-13, 12 August 2021, <<https://www.ietf.org/archive/id/draft-ietf-tls-esni-13.txt>>.
- [I-D.ietf-dnsop-svcb-https]
Schwartz, B., Bishop, M., and E. Nygren, "Service binding and parameter specification via the DNS (DNS SVCB and HTTPS RRs)", Work in Progress, Internet-Draft, draft-ietf-dnsop-svcb-https-08, 12 October 2021, <<https://www.ietf.org/archive/id/draft-ietf-dnsop-svcb-https-08.txt>>.

Appendix A. Changes

From -01 to -02:

* ECHO -> ECH

From -00 to -01:

* ESNI -> ECHO

Author's Address

Stephen Farrell
Trinity College Dublin
Dublin
2
Ireland

Phone: +353-1-896-2354
Email: stephen.farrell@cs.tcd.ie

TLS
Internet-Draft
Intended status: Experimental
Expires: 3 June 2022

S. Farrell
Trinity College Dublin
30 November 2021

A well-known URI for publishing ECHConfigList values.
draft-farrell-tls-wkesni-02

Abstract

We propose use of a well-known URI at which web servers can publish ECHConfigList values as a way to help get those published in the DNS.

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 3 June 2022.

Copyright Notice

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

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

Table of Contents

1. Introduction	2
2. Terminology	3
3. Example use of the well-known URI for ECH	3
4. The ech well-known URI	4
5. The JSON structure for ECHConfigList values	4
6. Zone factory behaviour	5
7. Security Considerations	6
8. Acknowledgements	6
9. IANA Considerations	6
10. Normative References	6
Appendix A. Change Log	7
Author's Address	7

1. Introduction

Encrypted ClientHello (ECH) [I-D.ietf-tls-esni] for TLS1.3 [RFC8446] defines a confidentiality mechanism for server names and other ClientHello content in TLS. For many applications, that requires publication of ECHConfigList data structures in the DNS. An ECHConfigList structure contains a list of ECHConfig values. Each ECHConfig value contains the public component of a key pair that will typically be periodically (re-)generated by a web server. Many web infrastructures will have an API that can be used to dynamically update the DNS RR values containing ECHConfigList values. Some deployments however, will not, so web deployments could benefit from a mechanism to use in such cases.

We define such a mechanism here. Note that this is not intended for universal deployment, but rather for cases where the web server doesn't have write access to the relevant zone file (or equivalent). That zone file will eventually include an HTTPS or SVCB RR [I-D.ietf-dnsop-svcb-https] containing an ECHConfigList.

We use the term "zone factory" for the entity that does have write access to the zone file. We assume the zone factory (ZF) can also make HTTPS requests to the web server with the ECH keys.

We propose use of a well-known URI [RFC8615] on the web server that allows ZF to poll for changes to ECHConfigList values. For example, if a web server generates new ECHConfigList values hourly and publishes those at the well-known URI, ZF can poll that URI. When ZF sees new values, it can check if those work, and if they do, then update the zone file and re-publish the zone.

[[This idea could: a) wither on the vine, b) be published as it's own RFC, or c) end up as a PR for [I-D.ietf-tls-esni]. There is no absolute need for this to be in the RFC that defines ECH, so (b) seems feasible if there's enough interest, hence this draft. The source for this is in <https://github.com/sftcd/wkesni/> PRs are welcome there too.]]

2. 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.

3. Example use of the well-known URI for ECH

An example deployment could be as follows:

- * Web server generates new ECHConfigList values hourly at N past the hour via a cronjob
- * ECHConfigList values are "current" for an hour, published with a TTL of 1800, and remain usable for 3 hours from the time of generation
- * Web server has a set of "backend" sites - the DNS name for each such site is here represented as \$BACKEND, which will end up as an SNI value to be encrypted inside an ECH extension
- * Web server has a "front-end" site (\$FRONT), where \$FRONT will typically be the DNS name used in the ECHConfigList public_name field for ECHConfig version 0xff0d
- * A cronjob creates creates a JSON file for each backend site at [https://\\$FRONT/.well-known/ech/\\$BACKEND.json](https://$FRONT/.well-known/ech/$BACKEND.json)
- * Each JSON file contains an array with the ECHConfigList values values for that particular \$BACKEND as shown in Figure 1 - the values in Figure 1 with ellipses are the values we want to eventually see in the DNS
- * On the zone factory, a cronjob runs at N+3 past the hour, it knows all the names involved and checks to see if the content at those well-known URIs has changed or not
- * If the content has changed the cronjob attempts to use the ECHConfigList values, and for each \$BACKEND where that works, it updates the zone file and re-publishes the zone containing only the new ECHConfigList values

4. The ech well-known URI

When a web server (\$FRONT) wants to publish ECHConfigList information for a backend site (\$BACKEND) then it provides the JSON content defined in Section 5 at: `https://$FRONT/.well-known/ech/$BACKEND.json`

The well-known URI defined here MUST be an https URL and therefore the zone factory verifies the correct \$FRONT is being accessed. If there is any failure in accessing the well-known URI, then the zone factory MUST NOT modify the zone.

5. The JSON structure for ECHConfigList values

[[Since the specifics of the JSON structure in Figure 1 are very likely to change, this is mostly TBD. What is here for now, is what the author has currently implemented simply because it worked ok and was easy to do:-)]]

[[Might change this due to retry-configs and now that I've implemented split-mode]]

```
[
  {
    "desired-ttl": 1800,
    "ports": [ 443, 8413 ],
    "echconfiglist": "AD7+DQA65wAgAC..AA=="
  },
  {
    "desired-ttl": 1800,
    "ports": [ 443, 8413 ],
    "echconfiglist": "AD7+DQA65wAgAC..AA=="
  }
]
```

Figure 1: Sample JSON

The JSON file at the well-known URI MUST contain an array with one or more elements. Each element of the array MUST have these fields:

- * `desired-ttl`: contains a number indicating the TTL that the web server would like to see used for this RR. The zone factory MUST NOT use a longer TTL.
- * `ports`: this has a list of the TCP ports on which the web server with the relevant key pair will listen (needed to produce the correct zone file).
- * `ECHConfigList`: contains the value to be used as a base64 encoded string.

The JSON file contains an array for a couple of reasons:

- * As TLS authentication doesn't really distinguish ports, servers on the same host could in any case cheat on one another, so we may as well just read one JSON file per name.
- * Different ports could map to different sets of ECHConfigList values
- * As ECHConfigList is (regrettably:-) an extensible structure, it may be necessary to publish different ECHConfigList values to get best interoperability.

6. Zone factory behaviour

The zone factory SHOULD check that the presented ECHConfigList values work with the \$BACKEND server before publication. A "special" TLS client may be needed for this check, that does not require the ECHConfigList value to have already been published in the DNS. [[I guess that calls for the zone factory to know of a "safe" URL on \$BACKEND to try, or maybe it could use HTTP HEAD? Figuring that out is TBD. The ZF could also try a GREASEd ECH and see if the retry-configs it gets back is one of the ECHConfigList values in the ECHConfigList.]]

A careful zone factory could explode the ECHConfigList value presented into "singleton" values with one public key in each and test each for each port claimed.

The zone factory SHOULD publish all the ECHConfigList values that are presented in the JSON file, and that pass the check above.

The zone factory SHOULD only publish ECHConfigList values that are in the latest version of the JSON file. This leaves the control of "expiry" with the web server, so long as the ECHConfigList values presented actually work. [[An alternative could be to have the new values just be appended to the zone, but that'd require some form of "notAfter" value in the JSON file which seems unnecessary and more complex.]]

The SCVB and HTTPS RR specification [I-D.ietf-dnsop-svcb-https] defines how and where the ECHConfigList values for \$BACKEND needs to be published in the DNS. The zone factory is assumed to be in control of how ECHConfigList values are included in such RRs.

A possibly interesting (unintended) consequence of this design is that once a TLS client has first gotten an ECHConfigList from the DNS for \$BACKEND with the ECHConfigList structure containing the public_name field, the TLS client would know both \$FRONT and \$BACKEND and so could later probe for this .well-known as an alternative to doing so via DoT/DoH. Probably not something a web browser might do, but could be fun for other applications maybe.

[[The extent to which retry-configs could be used for a similar purpose might be worth considering. But the JSON stuff here may still be needed if implementations (such as mine:-) tend to only return one ECHConfig in retry-configs.]]

7. Security Considerations

This document defines another way to publish ECHConfigList values. If the wrong keys were read from here and published in the DNS, then clients using ECH would do the wrong thing, likely resulting in denial of service, or a privacy leak, or worse, when TLS clients attempt to use ECH with a backend web site. So: Don't do that:-)

8. Acknowledgements

Thanks to Niall O'Reilly for a quick review of -00.

9. IANA Considerations

[[TBD: IANA registration of a .well-known. Also TBD - how to handle I18N for \$FRONT and \$BACKEND within such a URL.]]

10. Normative References

- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <<https://www.rfc-editor.org/info/rfc2119>>.
- [RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <<https://www.rfc-editor.org/info/rfc8174>>.
- [RFC8446] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <<https://www.rfc-editor.org/info/rfc8446>>.
- [RFC8615] Nottingham, M., "Well-Known Uniform Resource Identifiers (URIs)", RFC 8615, DOI 10.17487/RFC8615, May 2019, <<https://www.rfc-editor.org/info/rfc8615>>.

[I-D.ietf-tls-esni]

Rescorla, E., Oku, K., Sullivan, N., and C. A. Wood, "TLS Encrypted Client Hello", Work in Progress, Internet-Draft, draft-ietf-tls-esni-13, 12 August 2021, <<https://www.ietf.org/archive/id/draft-ietf-tls-esni-13.txt>>.

[I-D.ietf-dnsop-svcb-https]

Schwartz, B., Bishop, M., and E. Nygren, "Service binding and parameter specification via the DNS (DNS SVCB and HTTPS RRs)", Work in Progress, Internet-Draft, draft-ietf-dnsop-svcb-https-08, 12 October 2021, <<https://www.ietf.org/archive/id/draft-ietf-dnsop-svcb-https-08.txt>>.

Appendix A. Change Log

[[RFC editor: please remove this before publication.]]

From -01 to -02:

- * General changes from ESNI to ECH.

From -00 to -01:

- * Re-structured a bit after re-reading rfc8615

Author's Address

Stephen Farrell
Trinity College Dublin
Dublin
2
Ireland

Phone: +353-1-896-2354
Email: stephen.farrell@cs.tcd.ie

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

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

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

Abstract

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

The DTLS 1.3 protocol is intentionally based on the Transport Layer Security (TLS) 1.3 protocol and provides equivalent security guarantees with the exception of order protection/non-replayability. Datagram semantics of the underlying transport are preserved by the DTLS protocol.

This document obsoletes RFC 6347.

Status of This Memo

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

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

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

This Internet-Draft will expire on 1 November 2021.

Copyright Notice

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

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

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

Table of Contents

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

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

1. Introduction

RFC EDITOR: PLEASE REMOVE THE FOLLOWING PARAGRAPH

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

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

There are applications that use UDP [RFC0768] as a transport and to offer communication security protection for those applications the Datagram Transport Layer Security (DTLS) protocol has been developed. DTLS is deliberately designed to be as similar to TLS as possible, both to minimize new security invention and to maximize the amount of code and infrastructure reuse.

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

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

2. Conventions and Terminology

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

The following terms are used:

- * **client**: The endpoint initiating the DTLS connection.
- * **association**: Shared state between two endpoints established with a DTLS handshake.
- * **connection**: Synonym for association.
- * **endpoint**: Either the client or server of the connection.
- * **epoch**: one set of cryptographic keys used for encryption and decryption.
- * **handshake**: An initial negotiation between client and server that establishes the parameters of the connection.
- * **peer**: An endpoint. When discussing a particular endpoint, "peer" refers to the endpoint that is remote to the primary subject of discussion.
- * **receiver**: An endpoint that is receiving records.
- * **sender**: An endpoint that is transmitting records.
- * **server**: The endpoint which did not initiate the DTLS connection.
- * **CID**: Connection ID
- * **MSL**: Maximum Segment Lifetime

The reader is assumed to be familiar with [TLS13]. As in TLS 1.3, the HelloRetryRequest has the same format as a ServerHello message, but for convenience we use the term HelloRetryRequest throughout this document as if it were a distinct message.

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

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

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

- * **'+'** indicates noteworthy extensions sent in the previously noted message.

- * `'**'` indicates optional or situation-dependent messages/extensions that are not always sent.
- * `'{}'` indicates messages protected using keys derived from a `[sender]_handshake_traffic_secret`.
- * `'[]'` indicates messages protected using keys derived from `traffic_secret_N`.

3. DTLS Design Rationale and Overview

The basic design philosophy of DTLS is to construct "TLS over datagram transport". Datagram transport does not require nor provide reliable or in-order delivery of data. The DTLS protocol preserves this property for application data. Applications, such as media streaming, Internet telephony, and online gaming use datagram transport for communication due to the delay-sensitive nature of transported data. The behavior of such applications is unchanged when the DTLS protocol is used to secure communication, since the DTLS protocol does not compensate for lost or reordered data traffic. Note that while low-latency streaming and gaming use DTLS to protect data (e.g. for protection of a WebRTC data channel), telephony utilizes DTLS for key establishment, and Secure Real-time Transport Protocol (SRTP) for protection of data [RFC5763].

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

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

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

3.1. Packet Loss

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

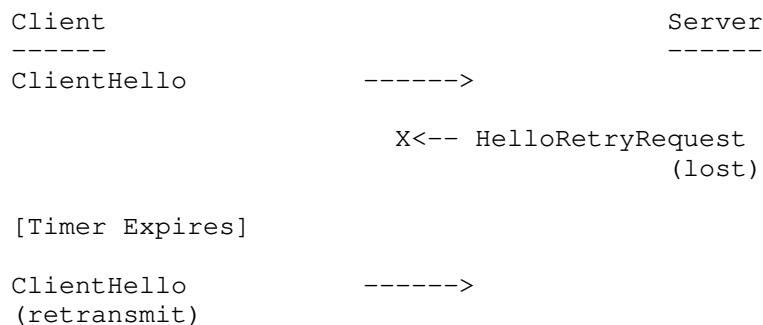


Figure 1: DTLS retransmission example

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

The server also maintains a retransmission timer for messages it sends (other than HelloRetryRequest) and retransmits when that timer expires. Not applying retransmissions to the HelloRetryRequest avoids the need to create state on the server. The HelloRetryRequest is designed to be small enough that it will not itself be fragmented, thus avoiding concerns about interleaving multiple HelloRetryRequests.

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

3.2. Reordering

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

3.3. Fragmentation

TLS and DTLS handshake messages can be quite large (in theory up to $2^{24}-1$ bytes, in practice many kilobytes). By contrast, UDP datagrams are often limited to less than 1500 bytes if IP fragmentation is not desired. In order to compensate for this limitation, each DTLS handshake message may be fragmented over several DTLS records, each of which is intended to fit in a single UDP datagram (see Section 4.4 for guidance). Each DTLS handshake message contains both a fragment offset and a fragment length. Thus, a recipient in possession of all bytes of a handshake message can reassemble the original unfragmented message.

3.4. Replay Detection

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

4. The DTLS Record Layer

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

1. The DTLSCiphertext structure omits the superfluous version number and type fields.
2. DTLS adds an epoch and sequence number to the TLS record header. This sequence number allows the recipient to correctly verify the DTLS MAC. However, the number of bits used for the epoch and sequence number fields in the DTLSCiphertext structure have been reduced from those in previous versions.
3. The DTLSCiphertext structure has a variable length header.

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

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

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

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

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

Figure 2: DTLS 1.3 Record Formats

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

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

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

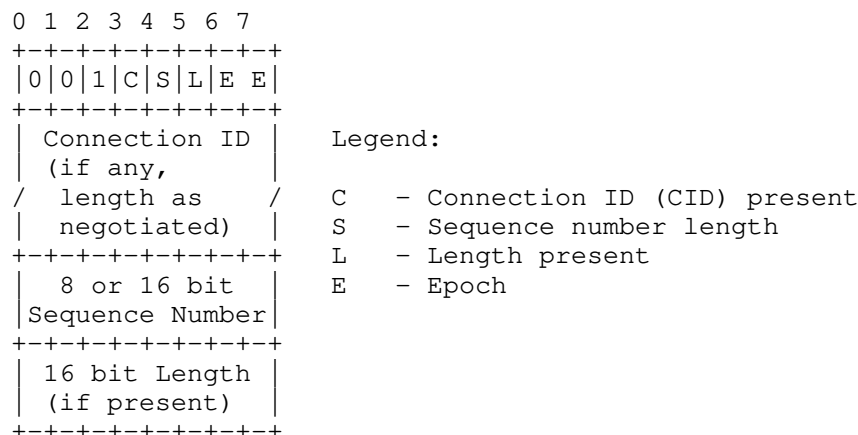


Figure 3: DTLS 1.3 Unified Header

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

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

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

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

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

Connection ID: Variable length CID. The CID functionality is described in [I-D.ietf-tls-dtls-connection-id]. An example can be found in Section 9.1.

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

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

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

Figure 4 illustrates different record headers.

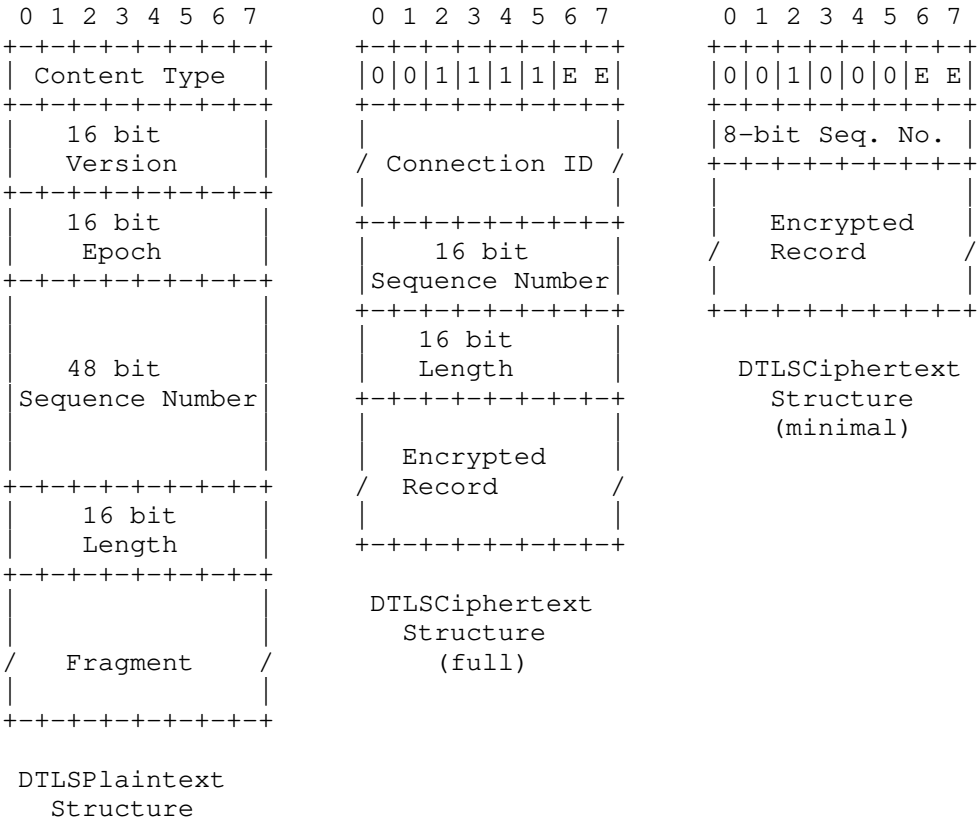


Figure 4: DTLS 1.3 Header Examples

The length field MAY be omitted by clearing the L bit, which means that the record consumes the entire rest of the datagram in the lower level transport. In this case it is not possible to have multiple DTLSCiphertext format records without length fields in the same datagram. Omitting the length field MUST only be used for the last record in a datagram. Implementations MAY mix records with and without length fields on the same connection.

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

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

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

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

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

4.1. Demultiplexing DTLS Records

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

- * If the first byte is alert(21), handshake(22), or ack(proposed, 26), the record MUST be interpreted as a DTLSPlaintext record.
- * If the first byte is any other value, then receivers MUST check to see if the leading bits of the first byte are 001. If so, the implementation MUST process the record as DTLSCiphertext; the true content type will be inside the protected portion.
- * Otherwise, the record MUST be rejected as if it had failed deprotection, as described in Section 4.5.2.

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

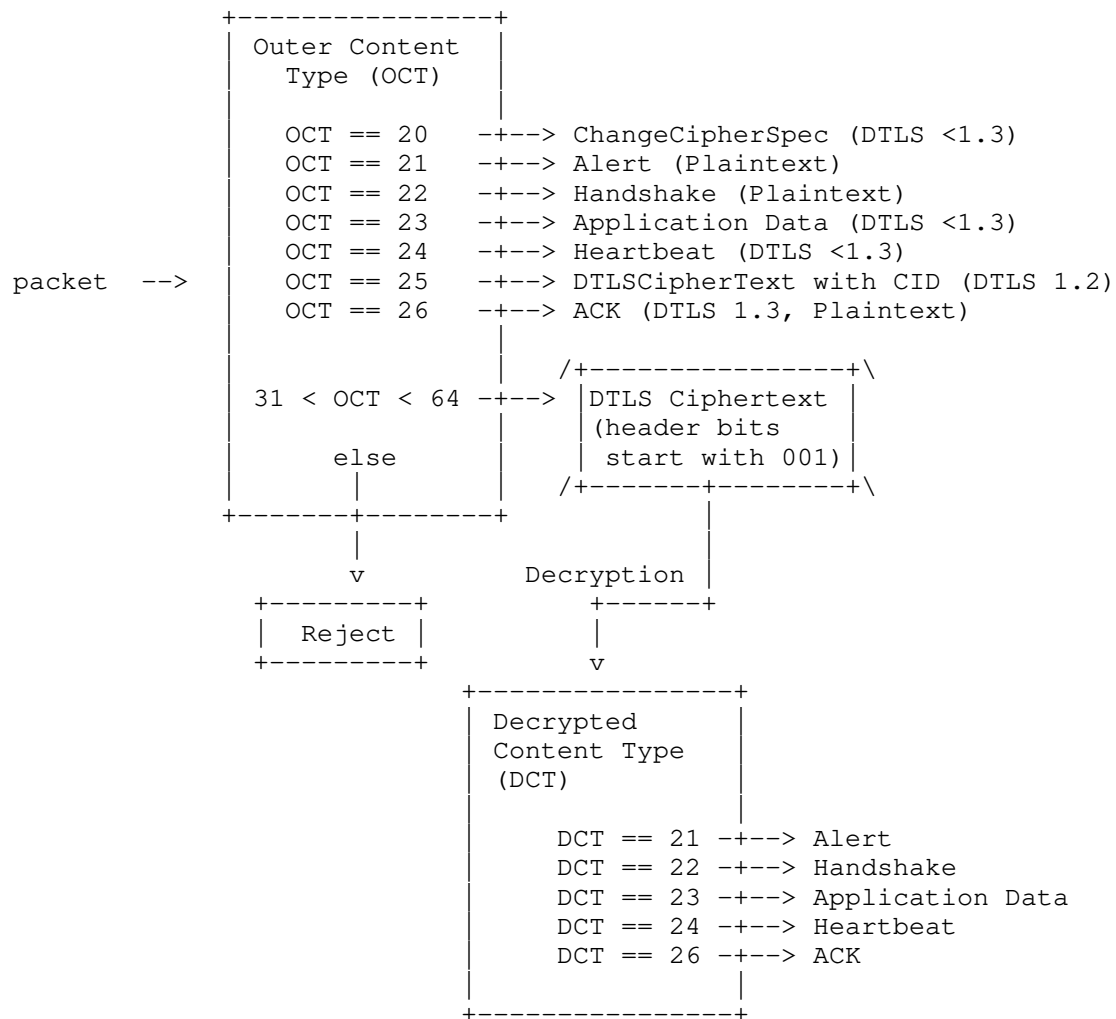


Figure 5: Demultiplexing DTLS 1.2 and DTLS 1.3 Records

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

4.2. Sequence Number and Epoch

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

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

4.2.1. Processing Guidelines

Because DTLS records could be reordered, a record from epoch M may be received after epoch N (where $N > M$) has begun. Implementations SHOULD discard records from earlier epochs, but MAY choose to retain keying material from previous epochs for up to the default MSL specified for TCP [RFC0793] to allow for packet reordering. (Note that the intention here is that implementers use the current guidance from the IETF for MSL, as specified in [RFC0793] or successors, not that they attempt to interrogate the MSL that the system TCP stack is using.)

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

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

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

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

4.2.2. Reconstructing the Sequence Number and Epoch

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

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

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

4.2.3. Record Number Encryption

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

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

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

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

```
Mask = ChaCha20(sn_key, Ciphertext[0..3], Ciphertext[4..15])
```

The `sn_key` is computed as follows:

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

[sender] denotes the sending side. The Secret value to be used is described in Section 7.3 of [TLS13]. Note that a new key is used for each epoch: because the epoch is sent in the clear, this does not result in ambiguity.

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

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

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

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

4.3. Transport Layer Mapping

DTLS messages MAY be fragmented into multiple DTLS records. Each DTLS record MUST fit within a single datagram. In order to avoid IP fragmentation, clients of the DTLS record layer SHOULD attempt to size records so that they fit within any Path MTU (PMTU) estimates obtained from the record layer. For more information about PMTU issues see Section 4.4.

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

DTLS records without CIDs do not contain any association identifiers and applications must arrange to multiplex between associations. With UDP, the host/port number is used to look up the appropriate security association for incoming records without CIDs.

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

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

4.4. PMTU Issues

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

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

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

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

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

- * For DTLS over DCCP, the upper layer protocol SHOULD be allowed to obtain the current estimate of the PMTU.
- * For DTLS over TCP or SCTP, which automatically fragment and reassemble datagrams, there is no PMTU limitation. However, the upper layer protocol MUST NOT write any record that exceeds the maximum record size of 2^{14} bytes.

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

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

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

The DTLS record layer SHOULD NOT interfere with upper layer protocols performing PMTU discovery, whether via [RFC1191] and [RFC4821] for IPv4 or via [RFC8201] for IPv6. In particular:

- * Where allowed by the underlying transport protocol, the upper layer protocol SHOULD be allowed to set the state of the DF bit (in IPv4) or prohibit local fragmentation (in IPv6).
- * If the underlying transport protocol allows the application to request PMTU probing (e.g., DCCP), the DTLS record layer SHOULD honor this request.

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

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

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

4.5. Record Payload Protection

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

4.5.1. Anti-Replay

Each DTLS record contains a sequence number to provide replay protection. Sequence number verification SHOULD be performed using the following sliding window procedure, borrowed from Section 3.4.3 of [RFC4303]. Because each epoch resets the sequence number space, a separate sliding window is needed for each epoch.

The received record counter for an epoch MUST be initialized to zero when that epoch is first used. For each received record, the receiver MUST verify that the record contains a sequence number that does not duplicate the sequence number of any other record received in that epoch during the lifetime of the association. This check SHOULD happen after deprotecting the record; otherwise the record discard might itself serve as a timing channel for the record number. Note that computing the full record number from the partial is still a potential timing channel for the record number, though a less powerful one than whether the record was deprotected.

Duplicates are rejected through the use of a sliding receive window. (How the window is implemented is a local matter, but the following text describes the functionality that the implementation must exhibit.) The receiver SHOULD pick a window large enough to handle any plausible reordering, which depends on the data rate. (The receiver does not notify the sender of the window size.)

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

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

4.5.2. Handling Invalid Records

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

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

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

4.5.3. AEAD Limits

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

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

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

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

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

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

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

5. The DTLS Handshake Protocol

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

1. To handle message loss, reordering, and fragmentation modifications to the handshake header are necessary.
2. Retransmission timers are introduced to handle message loss.
3. A new ACK content type has been added for reliable message delivery of handshake messages.

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

DTLS implementations do not use the TLS 1.3 "compatibility mode" described in Section D.4 of [TLS13]. DTLS servers MUST NOT echo the "legacy_session_id" value from the client and endpoints MUST NOT send ChangeCipherSpec messages.

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

5.1. Denial-of-Service Countermeasures

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

1. An attacker can consume excessive resources on the server by transmitting a series of handshake initiation requests, causing the server to allocate state and potentially to perform expensive cryptographic operations.
2. An attacker can use the server as an amplifier by sending connection initiation messages with a forged source address that belongs to a victim. The server then sends its response to the victim machine, thus flooding it. Depending on the selected parameters this response message can be quite large, as is the case for a Certificate message.

In order to counter both of these attacks, DTLS borrows the stateless cookie technique used by Photuris [RFC2522] and IKE [RFC7296]. When the client sends its ClientHello message to the server, the server

MAY respond with a HelloRetryRequest message. The HelloRetryRequest message, as well as the cookie extension, is defined in TLS 1.3. The HelloRetryRequest message contains a stateless cookie (see [TLS13]; Section 4.2.2). The client MUST send a new ClientHello with the cookie added as an extension. The server then verifies the cookie and proceeds with the handshake only if it is valid. This mechanism forces the attacker/client to be able to receive the cookie, which makes DoS attacks with spoofed IP addresses difficult. This mechanism does not provide any defense against DoS attacks mounted from valid IP addresses.

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

The exchange is shown in Figure 6. Note that the figure focuses on the cookie exchange; all other extensions are omitted.

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

                                   <----- HelloRetryRequest
                                   + cookie

ClientHello                         ----->
+ cookie

[Rest of handshake]
```

Figure 6: DTLS exchange with HelloRetryRequest containing the "cookie" extension

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

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

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

The handshake transcript is not reset with the second ClientHello and a stateless server-cookie implementation requires the content or hash of the initial ClientHello (and HelloRetryRequest) to be stored in the cookie. The initial ClientHello is included in the handshake transcript as a synthetic "message_hash" message, so only the hash value is needed for the handshake to complete, though the complete HelloRetryRequest contents are needed.

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

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

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

DTLS servers SHOULD perform a cookie exchange whenever a new handshake is being performed. If the server is being operated in an environment where amplification is not a problem, the server MAY be configured not to perform a cookie exchange. The default SHOULD be that the exchange is performed, however. In addition, the server MAY

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

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

As described in Section 4.1.4 of [TLS13], clients MUST abort the handshake with an "unexpected_message" alert in response to any second HelloRetryRequest which was sent in the same connection (i.e., where the ClientHello was itself in response to a HelloRetryRequest).

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

5.2. DTLS Handshake Message Format

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

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

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

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

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

DTLS implementations maintain (at least notionally) a next_receive_seq counter. This counter is initially set to zero. When a handshake message is received, if its message_seq value matches next_receive_seq, next_receive_seq is incremented and the message is processed. If the sequence number is less than next_receive_seq, the message MUST be discarded. If the sequence number is greater than next_receive_seq, the implementation SHOULD queue the message but MAY discard it. (This is a simple space/bandwidth tradeoff).

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

5.3. ClientHello Message

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

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

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

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

legacy_version: In previous versions of DTLS, this field was used for version negotiation and represented the highest version number supported by the client. Experience has shown that many servers do not properly implement version negotiation, leading to "version

intolerance" in which the server rejects an otherwise acceptable ClientHello with a version number higher than it supports. In DTLS 1.3, the client indicates its version preferences in the "supported_versions" extension (see Section 4.2.1 of [TLS13]) and the legacy_version field MUST be set to {254, 253}, which was the version number for DTLS 1.2. The supported_versions entries for DTLS 1.0 and DTLS 1.2 are 0xfeff and 0xfefd (to match the wire versions). The value 0xfefc is used to indicate DTLS 1.3.

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

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

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

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

legacy_compression_methods: Same as for TLS 1.3.

extensions: Same as for TLS 1.3.

5.4. ServerHello Message

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

5.5. Handshake Message Fragmentation and Reassembly

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

When transmitting the handshake message, the sender divides the message into a series of N contiguous data ranges. The ranges MUST NOT overlap. The sender then creates N handshake messages, all with the same message_seq value as the original handshake message. Each new message is labeled with the fragment_offset (the number of bytes contained in previous fragments) and the fragment_length (the length of this fragment). The length field in all messages is the same as the length field of the original message. An unfragmented message is a degenerate case with fragment_offset=0 and fragment_length=length. Each handshake message fragment that is placed into a record MUST be delivered in a single UDP datagram.

When a DTLS implementation receives a handshake message fragment corresponding to the next expected handshake message sequence number, it MUST buffer it until it has the entire handshake message. DTLS implementations MUST be able to handle overlapping fragment ranges. This allows senders to retransmit handshake messages with smaller fragment sizes if the PMTU estimate changes. Senders MUST NOT change handshake message bytes upon retransmission. Receivers MAY check that retransmitted bytes are identical and SHOULD abort the handshake with an "illegal_parameter" alert if the value of a byte changes.

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

5.6. End Of Early Data

The DTLS 1.3 handshake has one important difference from the TLS 1.3 handshake: the EndOfEarlyData message is omitted both from the wire and the handshake transcript: because DTLS records have epochs, EndOfEarlyData is not necessary to determine when the early data is complete, and because DTLS is lossy, attackers can trivially mount the deletion attacks that EndOfEarlyData prevents in TLS. Servers SHOULD NOT accept records from epoch 1 indefinitely once they are able to process records from epoch 3. Though reordering of IP packets can result in records from epoch 1 arriving after records from epoch 3, this is not likely to persist for very long relative to the round trip time. Servers could discard epoch 1 keys after the first epoch 3 data arrives, or retain keys for processing epoch 1 data for a short period. (See Section 6.1 for the definitions of each epoch.)

5.7. DTLS Handshake Flights

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

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

Table 1: Flight Handshake Message Combinations.

Remarks:

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

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

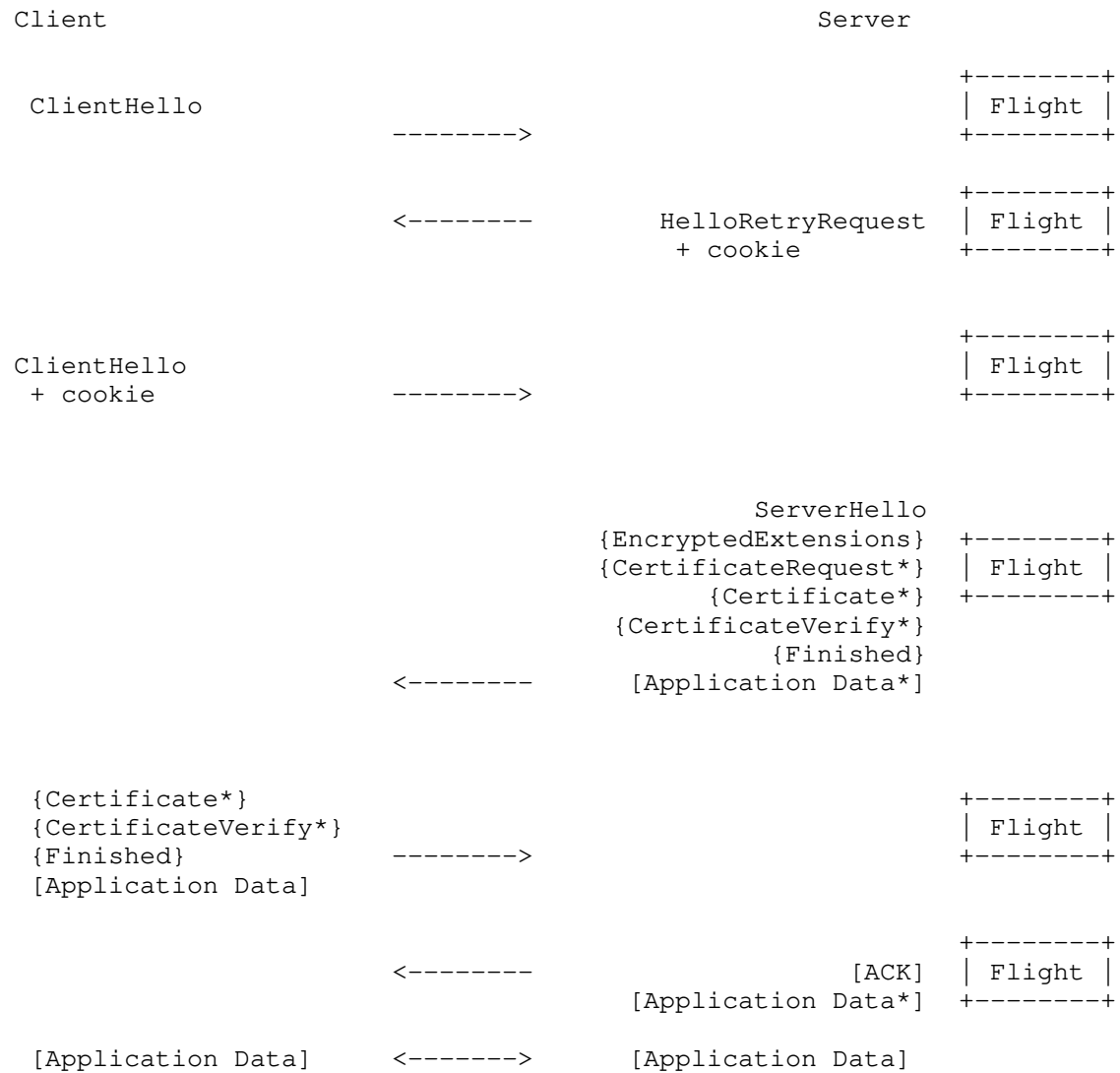


Figure 7: Message flights for a full DTLS Handshake (with cookie exchange)

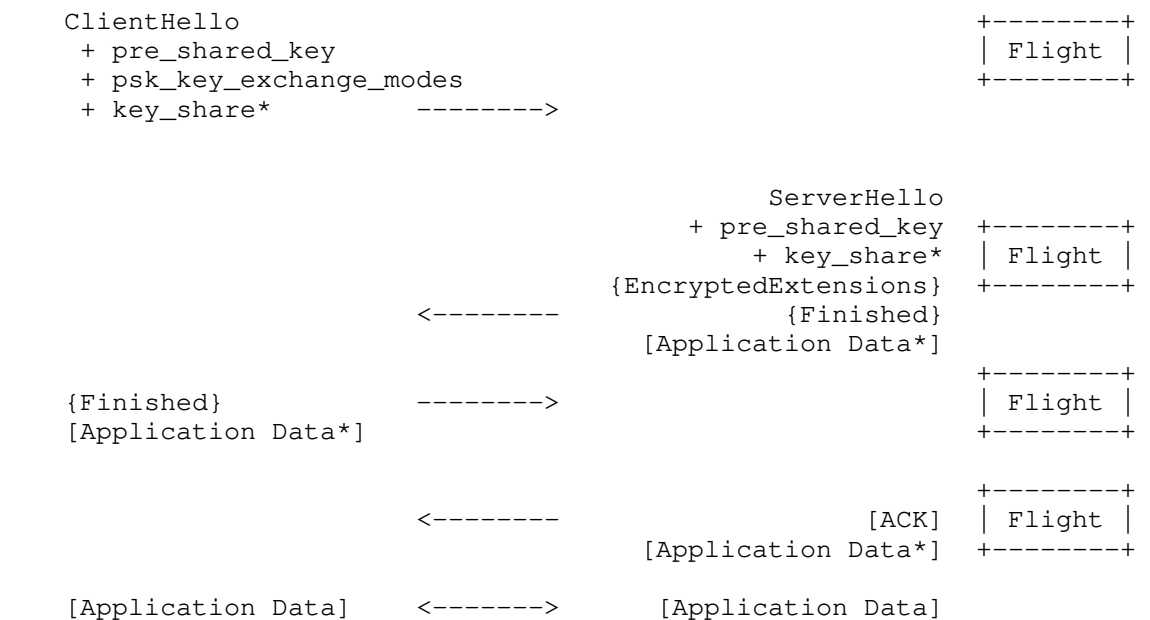


Figure 8: Message flights for resumption and PSK handshake (without cookie exchange)

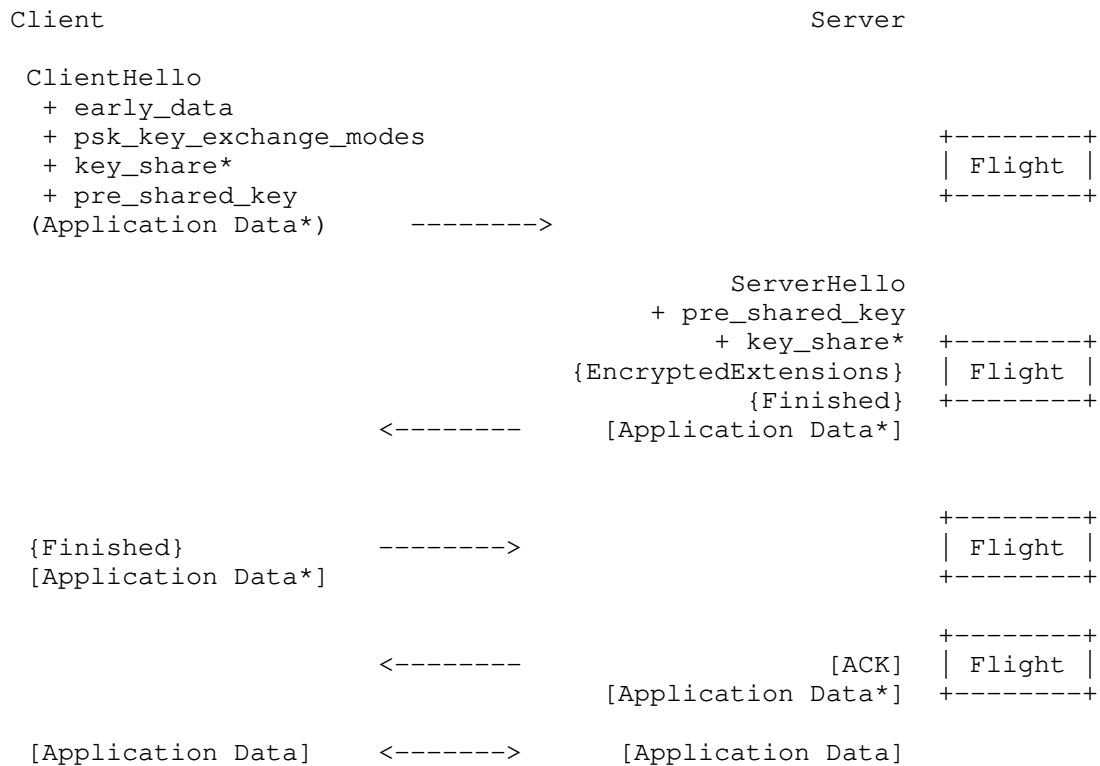
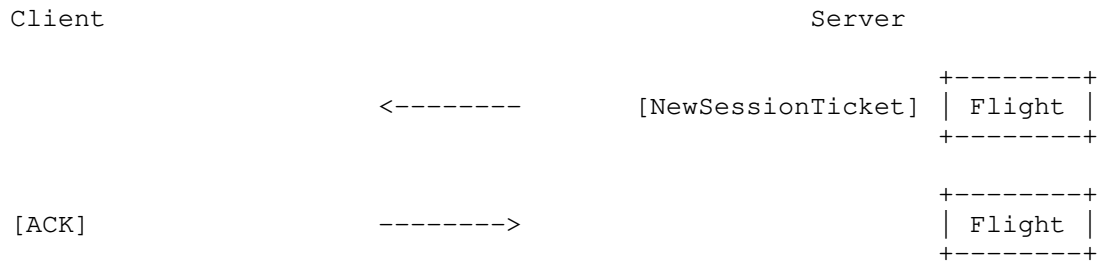


Figure 9: Message flights for the Zero-RTT handshake

Figure 10: Message flights for the `NewSessionTicket` message

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

5.8. Timeout and Retransmission

5.8.1. State Machine

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

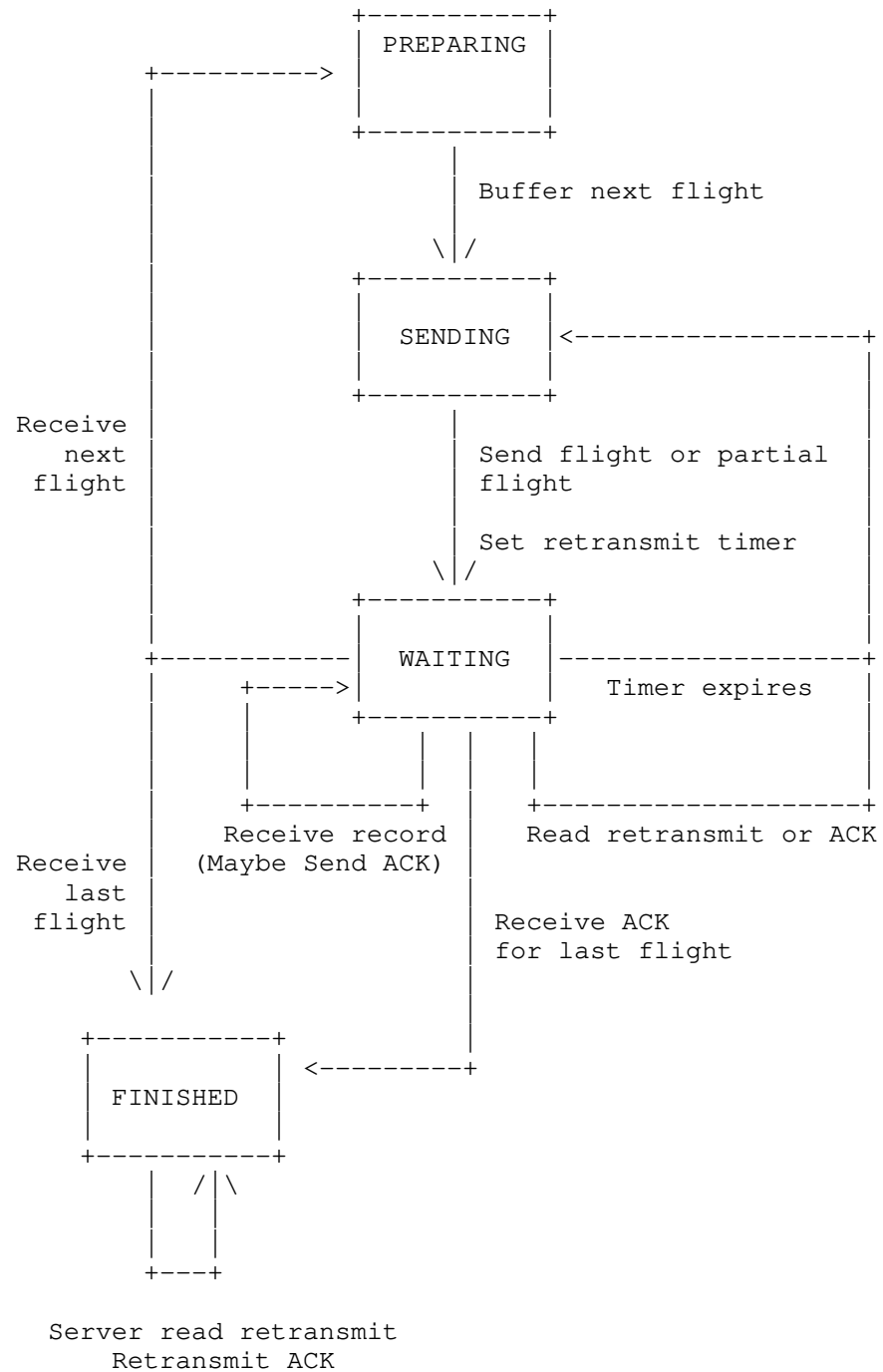


Figure 11: DTLS timeout and retransmission state machine

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

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

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

There are four ways to exit the WAITING state:

1. The retransmit timer expires: the implementation transitions to the SENDING state, where it retransmits the flight, adjusts and re-arms the retransmit timer (see Section 5.8.2), and returns to the WAITING state.
2. The implementation reads an ACK from the peer: upon receiving an ACK for a partial flight (as mentioned in Section 7.1), the implementation transitions to the SENDING state, where it retransmits the unacked portion of the flight, adjusts and re-arms the retransmit timer, and returns to the WAITING state. Upon receiving an ACK for a complete flight, the implementation cancels all retransmissions and either remains in WAITING, or, if the ACK was for the final flight, transitions to FINISHED.
3. The implementation reads a retransmitted flight from the peer: the implementation transitions to the SENDING state, where it retransmits the flight, adjusts and re-arms the retransmit timer, and returns to the WAITING state. The rationale here is that the receipt of a duplicate message is the likely result of timer expiry on the peer and therefore suggests that part of one's previous flight was lost.
4. The implementation receives some or all of the next flight of messages: if this is the final flight of messages, the implementation transitions to FINISHED. If the implementation needs to send a new flight, it transitions to the PREPARING state. Partial reads (whether partial messages or only some of the messages in the flight) may also trigger the implementation to send an ACK, as described in Section 7.1.

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

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

Note that because of packet loss, it is possible for one side to be sending application data even though the other side has not received the first side's Finished message. Implementations MUST either discard or buffer all application data records for epoch 3 and above until they have received the Finished message from the peer. Implementations MAY treat receipt of application data with a new epoch prior to receipt of the corresponding Finished message as evidence of reordering or packet loss and retransmit their final flight immediately, shortcutting the retransmission timer.

5.8.2. Timer Values

The configuration of timer settings varies with implementations, and certain deployment environments require timer value adjustments. Mishandling of the timer can lead to serious congestion problems, for example if many instances of a DTLS time out early and retransmit too quickly on a congested link.

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

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

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

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

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

5.8.3. Large Flight Sizes

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

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

5.8.4. State machine duplication for post-handshake messages

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

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

5. Post-handshake client authentication

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

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

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

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

5.9. CertificateVerify and Finished Messages

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

5.10. Cryptographic Label Prefix

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

5.11. Alert Messages

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

5.12. Establishing New Associations with Existing Parameters

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

blind attackers from destroying associations merely by sending forged ClientHellos.

Note: it is not always possible to distinguish which association a given record is from. For instance, if the client performs a handshake, abandons the connection, and then immediately starts a new handshake, it may not be possible to tell which connection a given protected record is for. In these cases, trial decryption may be necessary, though implementations could use CIDs to avoid the 5-tuple-based ambiguity.

6. Example of Handshake with Timeout and Retransmission

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

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

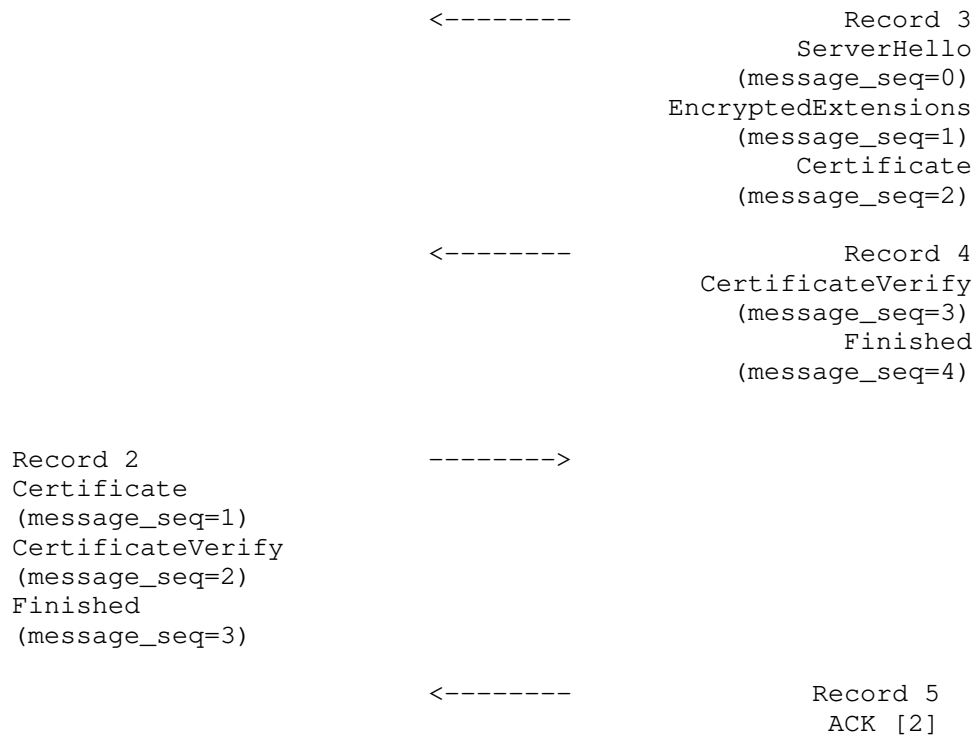


Figure 12: Example DTLS exchange illustrating message loss

6.1. Epoch Values and Rekeying

A recipient of a DTLS message needs to select the correct keying material in order to process an incoming message. With the possibility of message loss and re-ordering, an identifier is needed to determine which cipher state has been used to protect the record payload. The epoch value fulfills this role in DTLS. In addition to the TLS 1.3-defined key derivation steps, see Section 7 of [TLS13], a sender may want to rekey at any time during the lifetime of the connection. It therefore needs to indicate that it is updating its sending cryptographic keys.

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

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

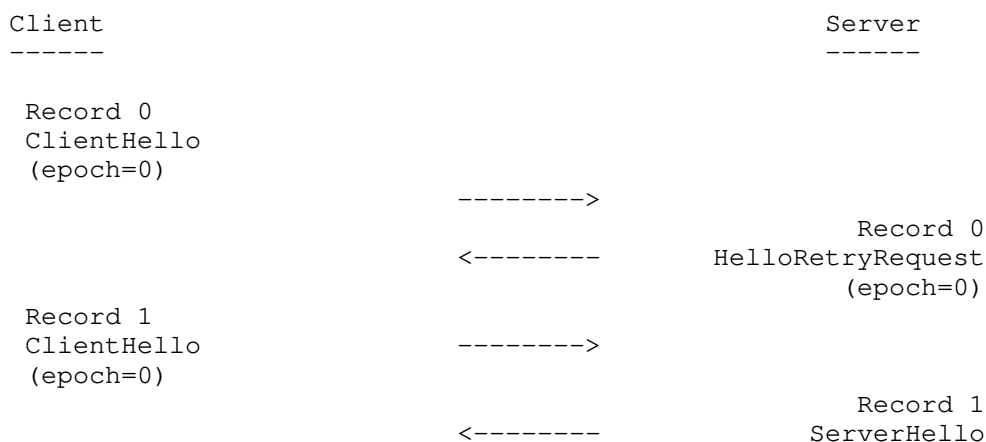
- * epoch value (1) is used for messages protected using keys derived from `client_early_traffic_secret`. Note this epoch is skipped if the client does not offer early data.
- * epoch value (2) is used for messages protected using keys derived from `[sender]_handshake_traffic_secret`. Messages transmitted during the initial handshake, such as `EncryptedExtensions`, `CertificateRequest`, `Certificate`, `CertificateVerify`, and `Finished` belong to this category. Note, however, post-handshake are protected under the appropriate application traffic key and are not included in this category.
- * epoch value (3) is used for payloads protected using keys derived from the initial `[sender]_application_traffic_secret_0`. This may include handshake messages, such as post-handshake messages (e.g., a `NewSessionTicket` message).
- * epoch value (4 to $2^{16}-1$) is used for payloads protected using keys from the `[sender]_application_traffic_secret_N` ($N>0$).

Using these reserved epoch values a receiver knows what cipher state has been used to encrypt and integrity protect a message. Implementations that receive a record with an epoch value for which no corresponding cipher state can be determined SHOULD handle it as a record which fails deprotection.

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

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

Figure 13 illustrates the epoch values in an example DTLS handshake.



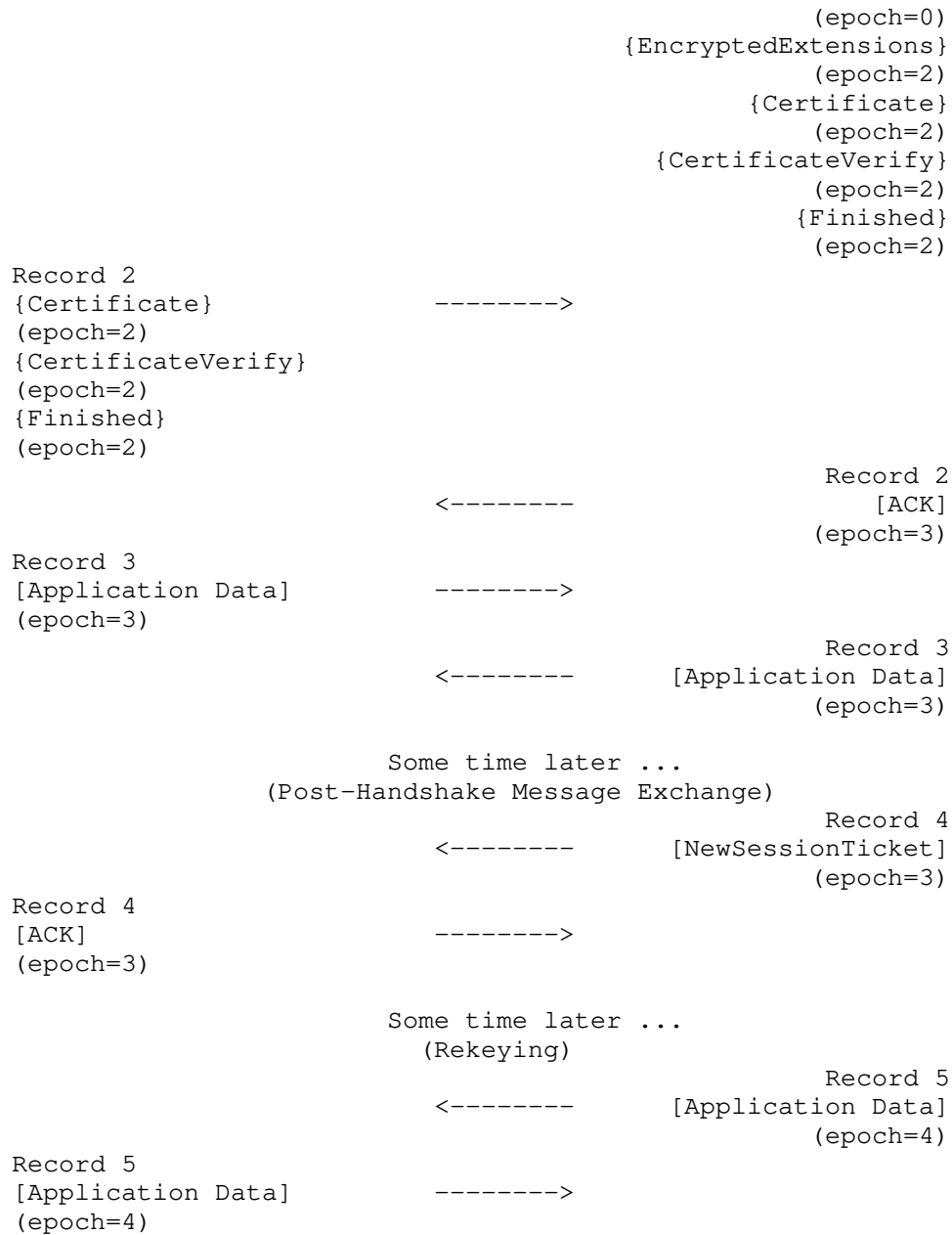


Figure 13: Example DTLS exchange with epoch information

7. ACK Message

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

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

record_numbers: a list of the records containing handshake messages in the current flight which the endpoint has received and either processed or buffered, in numerically increasing order.

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

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

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

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

7.1. Sending ACKs

When an implementation detects a disruption in the receipt of the current incoming flight, it SHOULD generate an ACK that covers the messages from that flight which it has received and processed so far. Implementations have some discretion about which events to treat as signs of disruption, but it is RECOMMENDED that they generate ACKs under two circumstances:

- * When they receive a message or fragment which is out of order, either because it is not the next expected message or because it is not the next piece of the current message.
- * When they have received part of a flight and do not immediately receive the rest of the flight (which may be in the same UDP datagram). "Immediately" is hard to define. One approach is to set a timer for 1/4 the current retransmit timer value when the first record in the flight is received and then send an ACK when that timer expires. Note: the 1/4 value here is somewhat arbitrary. Given that the round trip estimates in the DTLS handshake are generally very rough (or the default), any value will be an approximation, and there is an inherent compromise due to competition between retransmission due to over-aggressive ACKing and over-aggressive timeout-based retransmission. As a comparison point, QUIC's loss-based recovery algorithms ([I-D.ietf-quic-recovery]; Section 6.1.2) work out to a delay of about 1/3 of the retransmit timer.

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

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

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

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

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

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

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

7.2. Receiving ACKs

When an implementation receives an ACK, it SHOULD record that the messages or message fragments sent in the records being ACKed were received and omit them from any future retransmissions. Upon receipt of an ACK that leaves it with only some messages from a flight having been acknowledged an implementation SHOULD retransmit the unacknowledged messages or fragments. Note that this requires implementations to track which messages appear in which records. Once all the messages in a flight have been acknowledged, the implementation MUST cancel all retransmissions of that flight. Implementations MUST treat a record as having been acknowledged if it appears in any ACK; this prevents spurious retransmission in cases where a flight is very large and the receiver is forced to elide acknowledgements for records which have already been ACKed. As noted above, the receipt of any record responding to a given flight MUST be taken as an implicit acknowledgement for the entire flight to which it is responding.

7.3. Design Rationale

ACK messages are used in two circumstances, namely :

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

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

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

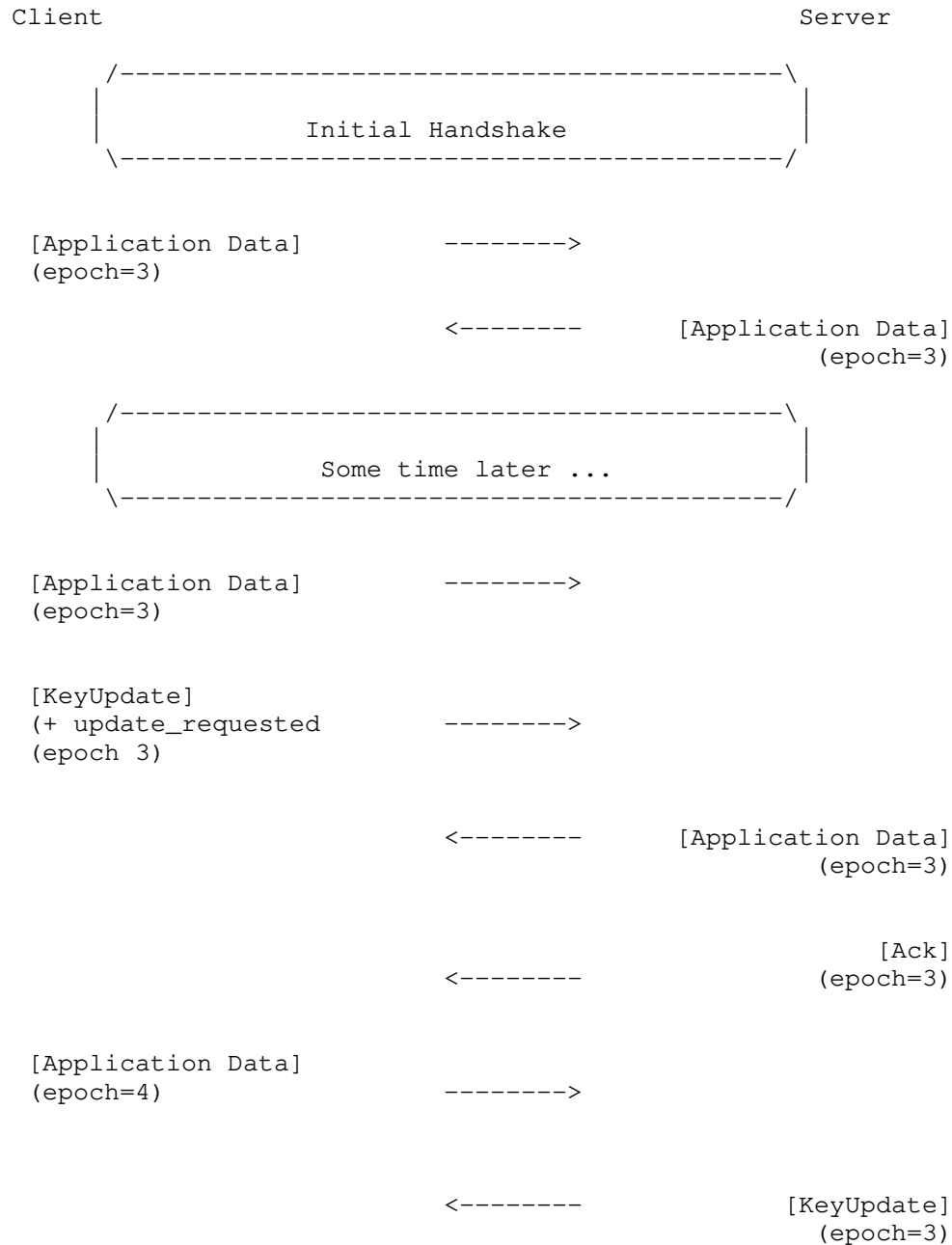
8. Key Updates

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

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

Due to the possibility of an ACK message for a KeyUpdate being lost and thereby preventing the sender of the KeyUpdate from updating its keying material, receivers MUST retain the pre-update keying material until receipt and successful decryption of a message using the new keys.

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



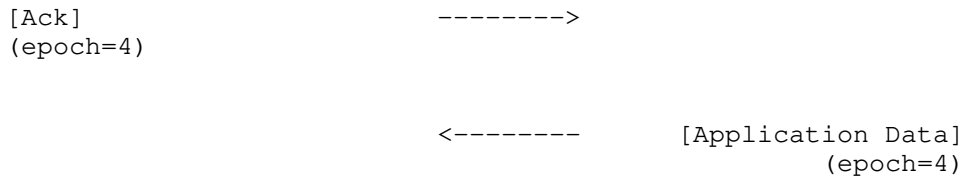


Figure 14: Example DTLS Key Update

9. Connection ID Updates

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

```

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

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

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

```

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

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

Endpoints SHOULD use receiver-provided CIDs in the order they were provided. Implementations which receive more spare CIDs than they wish to maintain MAY simply discard any extra CIDs. Endpoints MUST NOT have more than one NewConnectionId message outstanding.

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

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

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

num_cids The number of CIDs desired.

Endpoints SHOULD respond to RequestConnectionId by sending a NewConnectionId with usage "cid_spare" containing num_cid CIDs soon as possible. Endpoints MUST NOT send a RequestConnectionId message when an existing request is still unfulfilled; this implies that endpoints needs to request new CIDs well in advance. An endpoint MAY handle requests, which it considers excessive, by responding with a NewConnectionId message containing fewer than num_cid CIDs, including no CIDs at all. Endpoints MAY handle an excessive number of RequestConnectionId messages by terminating the connection using a "too_many_cids_requested" (alert number 52) alert.

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

9.1. Connection ID Example

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

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

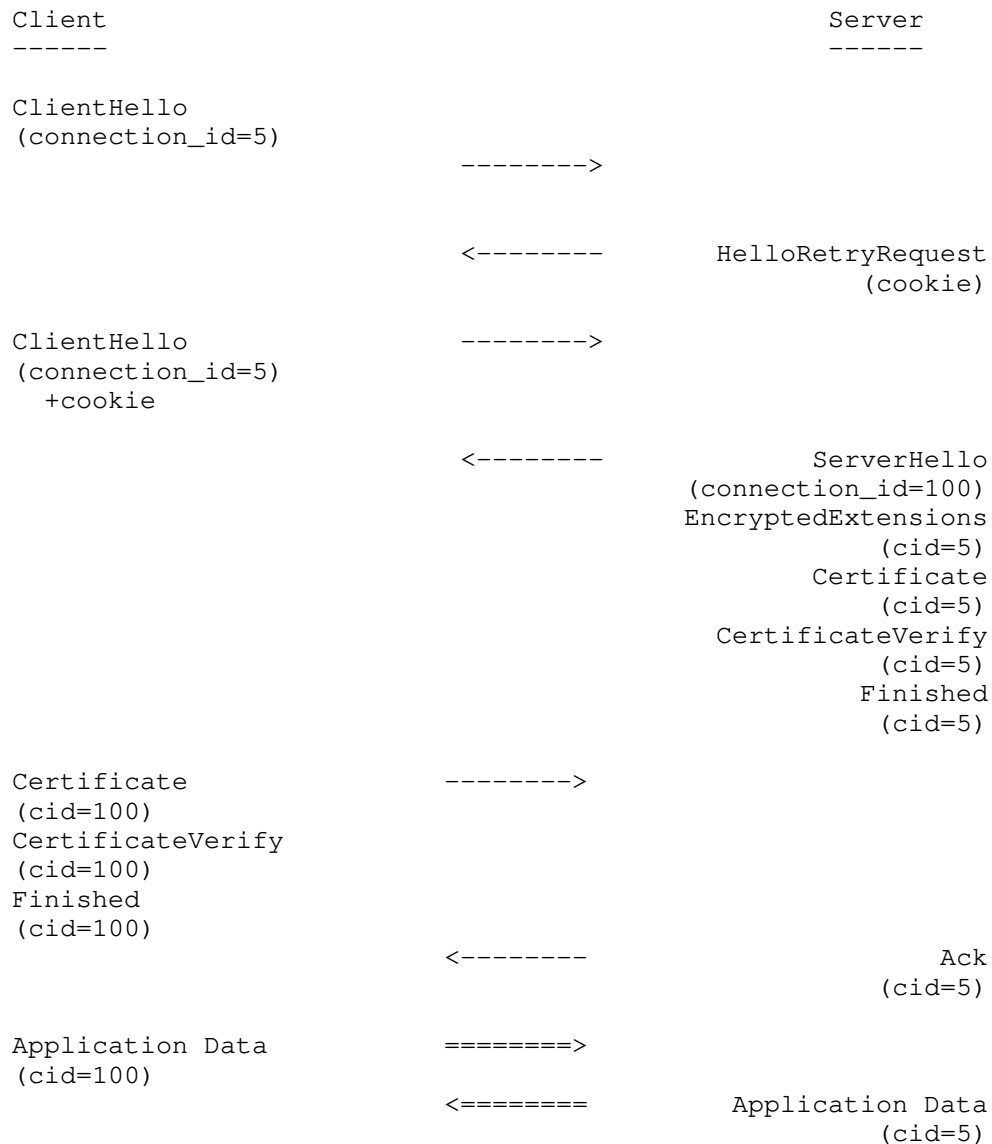


Figure 15: Example DTLS 1.3 Exchange with CIDs

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

10. Application Data Protocol

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

11. Security Considerations

Security issues are discussed primarily in [TLS13].

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

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

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

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

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

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

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

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

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

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

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

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

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

12. Changes since DTLS 1.2

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

- * New handshake pattern, which leads to a shorter message exchange
- * Only AEAD ciphers are supported. Additional data calculation has been simplified.
- * Removed support for weaker and older cryptographic algorithms
- * HelloRetryRequest of TLS 1.3 used instead of HelloVerifyRequest
- * More flexible ciphersuite negotiation
- * New session resumption mechanism

- * PSK authentication redefined
- * New key derivation hierarchy utilizing a new key derivation construct
- * Improved version negotiation
- * Optimized record layer encoding and thereby its size
- * Added CID functionality
- * Sequence numbers are encrypted.

13. Updates affecting DTLS 1.2

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

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

14. IANA Considerations

IANA is requested to allocate a new value in the "TLS ContentType" registry for the ACK message, defined in Section 7, with content type 26. The value for the "DTLS-OK" column is "Y". IANA is requested to reserve the content type range 32-63 so that content types in this range are not allocated.

IANA is requested to allocate "the too_many_cids_requested" alert in the "TLS Alerts" registry with value 52.

IANA is requested to allocate two values in the "TLS Handshake Type" registry, defined in [TLS13], for RequestConnectionId (TBD), and NewConnectionId (TBD), as defined in this document. The value for the "DTLS-OK" columns are "Y".

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

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

15. References

15.1. Normative References

- [CHACHA] Nir, Y. and A. Langley, "ChaCha20 and Poly1305 for IETF Protocols", RFC 8439, DOI 10.17487/RFC8439, June 2018, <<https://www.rfc-editor.org/info/rfc8439>>.
- [I-D.ietf-tls-dtls-connection-id] Rescorla, E., Tschofenig, H., Fossati, T., and A. Kraus, "Connection Identifiers for DTLS 1.2", Work in Progress, Internet-Draft, draft-ietf-tls-dtls-connection-id-11, 14 April 2021, <<https://www.ietf.org/internet-drafts/draft-ietf-tls-dtls-connection-id-11.txt>>.
- [RFC0768] Postel, J., "User Datagram Protocol", STD 6, RFC 768, DOI 10.17487/RFC0768, August 1980, <<https://www.rfc-editor.org/info/rfc768>>.
- [RFC0793] Postel, J., "Transmission Control Protocol", STD 7, RFC 793, DOI 10.17487/RFC0793, September 1981, <<https://www.rfc-editor.org/info/rfc793>>.
- [RFC1191] Mogul, J. and S. Deering, "Path MTU discovery", RFC 1191, DOI 10.17487/RFC1191, November 1990, <<https://www.rfc-editor.org/info/rfc1191>>.
- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <<https://www.rfc-editor.org/info/rfc2119>>.
- [RFC4443] Conta, A., Deering, S., and M. Gupta, Ed., "Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification", STD 89, RFC 4443, DOI 10.17487/RFC4443, March 2006, <<https://www.rfc-editor.org/info/rfc4443>>.
- [RFC4821] Mathis, M. and J. Heffner, "Packetization Layer Path MTU Discovery", RFC 4821, DOI 10.17487/RFC4821, March 2007, <<https://www.rfc-editor.org/info/rfc4821>>.

- [RFC6298] Paxson, V., Allman, M., Chu, J., and M. Sargent, "Computing TCP's Retransmission Timer", RFC 6298, DOI 10.17487/RFC6298, June 2011, <<https://www.rfc-editor.org/info/rfc6298>>.
- [RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <<https://www.rfc-editor.org/info/rfc8174>>.
- [TLS13] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <<https://www.rfc-editor.org/info/rfc8446>>.

15.2. Informative References

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

- [I-D.ietf-uta-tls13-iot-profile]
Tschofenig, H. and T. Fossati, "TLS/DTLS 1.3 Profiles for the Internet of Things", Work in Progress, Internet-Draft, draft-ietf-uta-tls13-iot-profile-01, 22 February 2021, <<https://www.ietf.org/internet-drafts/draft-ietf-uta-tls13-iot-profile-01.txt>>.
- [RFC2522] Karn, P. and W. Simpson, "Photuris: Session-Key Management Protocol", RFC 2522, DOI 10.17487/RFC2522, March 1999, <<https://www.rfc-editor.org/info/rfc2522>>.
- [RFC4303] Kent, S., "IP Encapsulating Security Payload (ESP)", RFC 4303, DOI 10.17487/RFC4303, December 2005, <<https://www.rfc-editor.org/info/rfc4303>>.
- [RFC4340] Kohler, E., Handley, M., and S. Floyd, "Datagram Congestion Control Protocol (DCCP)", RFC 4340, DOI 10.17487/RFC4340, March 2006, <<https://www.rfc-editor.org/info/rfc4340>>.
- [RFC4346] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.1", RFC 4346, DOI 10.17487/RFC4346, April 2006, <<https://www.rfc-editor.org/info/rfc4346>>.
- [RFC4347] Rescorla, E. and N. Modadugu, "Datagram Transport Layer Security", RFC 4347, DOI 10.17487/RFC4347, April 2006, <<https://www.rfc-editor.org/info/rfc4347>>.
- [RFC4960] Stewart, R., Ed., "Stream Control Transmission Protocol", RFC 4960, DOI 10.17487/RFC4960, September 2007, <<https://www.rfc-editor.org/info/rfc4960>>.
- [RFC5238] Phelan, T., "Datagram Transport Layer Security (DTLS) over the Datagram Congestion Control Protocol (DCCP)", RFC 5238, DOI 10.17487/RFC5238, May 2008, <<https://www.rfc-editor.org/info/rfc5238>>.
- [RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, DOI 10.17487/RFC5246, August 2008, <<https://www.rfc-editor.org/info/rfc5246>>.
- [RFC5763] Fischl, J., Tschofenig, H., and E. Rescorla, "Framework for Establishing a Secure Real-time Transport Protocol (SRTP) Security Context Using Datagram Transport Layer Security (DTLS)", RFC 5763, DOI 10.17487/RFC5763, May 2010, <<https://www.rfc-editor.org/info/rfc5763>>.

- [RFC5764] McGrew, D. and E. Rescorla, "Datagram Transport Layer Security (DTLS) Extension to Establish Keys for the Secure Real-time Transport Protocol (SRTP)", RFC 5764, DOI 10.17487/RFC5764, May 2010, <<https://www.rfc-editor.org/info/rfc5764>>.
- [RFC6066] Eastlake 3rd, D., "Transport Layer Security (TLS) Extensions: Extension Definitions", RFC 6066, DOI 10.17487/RFC6066, January 2011, <<https://www.rfc-editor.org/info/rfc6066>>.
- [RFC6347] Rescorla, E. and N. Modadugu, "Datagram Transport Layer Security Version 1.2", RFC 6347, DOI 10.17487/RFC6347, January 2012, <<https://www.rfc-editor.org/info/rfc6347>>.
- [RFC7296] Kaufman, C., Hoffman, P., Nir, Y., Eronen, P., and T. Kivinen, "Internet Key Exchange Protocol Version 2 (IKEv2)", STD 79, RFC 7296, DOI 10.17487/RFC7296, October 2014, <<https://www.rfc-editor.org/info/rfc7296>>.
- [RFC7525] Sheffer, Y., Holz, R., and P. Saint-Andre, "Recommendations for Secure Use of Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)", BCP 195, RFC 7525, DOI 10.17487/RFC7525, May 2015, <<https://www.rfc-editor.org/info/rfc7525>>.
- [RFC7924] Santesson, S. and H. Tschofenig, "Transport Layer Security (TLS) Cached Information Extension", RFC 7924, DOI 10.17487/RFC7924, July 2016, <<https://www.rfc-editor.org/info/rfc7924>>.
- [RFC7983] Petit-Huguenin, M. and G. Salgueiro, "Multiplexing Scheme Updates for Secure Real-time Transport Protocol (SRTP) Extension for Datagram Transport Layer Security (DTLS)", RFC 7983, DOI 10.17487/RFC7983, September 2016, <<https://www.rfc-editor.org/info/rfc7983>>.
- [RFC8201] McCann, J., Deering, S., Mogul, J., and R. Hinden, Ed., "Path MTU Discovery for IP version 6", STD 87, RFC 8201, DOI 10.17487/RFC8201, July 2017, <<https://www.rfc-editor.org/info/rfc8201>>.
- [RFC8445] Keranen, A., Holmberg, C., and J. Rosenberg, "Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal", RFC 8445, DOI 10.17487/RFC8445, July 2018, <<https://www.rfc-editor.org/info/rfc8445>>.

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

Appendix A. Protocol Data Structures and Constant Values

This section provides the normative protocol types and constants definitions.

A.1. Record Layer

```

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

```

```

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

```

```

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

```

```

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

```

Legend:

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

```

struct {
    uint16 epoch;
    uint48 sequence_number;
} RecordNumber;

```

A.2. Handshake Protocol

```
enum {
    hello_request_RESERVED(0),
    client_hello(1),
    server_hello(2),
    hello_verify_request_RESERVED(3),
    new_session_ticket(4),
    end_of_early_data(5),
    hello_retry_request_RESERVED(6),
    encrypted_extensions(8),
    certificate(11),
    server_key_exchange_RESERVED(12),
    certificate_request(13),
    server_hello_done_RESERVED(14),
    certificate_verify(15),
    client_key_exchange_RESERVED(16),
    finished(20),
    certificate_url_RESERVED(21),
    certificate_status_RESERVED(22),
    supplemental_data_RESERVED(23),
    key_update(24),
    message_hash(254),
    (255)
} HandshakeType;

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

uint16 ProtocolVersion;
opaque Random[32];

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

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

A.3. ACKs

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

A.4. Connection ID Management

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

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

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

struct {
    uint8 num_cids;
} RequestConnectionId;
```

Appendix B. Analysis of Limits on CCM Usage

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

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

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

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

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

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

B.1. Confidentiality Limits

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

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

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

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

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

B.2. Integrity Limits

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

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

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

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

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

B.3. Limits for AEAD_AES_128_CCM_8

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

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

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

$$v \leq 2^7$$

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

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

Appendix C. Implementation Pitfalls

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

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

Appendix D. History

RFC EDITOR: PLEASE REMOVE THE THIS SECTION

(*) indicates a change that may affect interoperability.

IETF Drafts draft-42

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

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

draft-40

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

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

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

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

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

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

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

draft-32: - Editorial improvements and clarifications.

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

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

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

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

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

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

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

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

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

draft-00 - First IETF Draft

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

draft-00

- * Initial version using TLS 1.3 as a baseline.
- * Use of epoch values instead of KeyUpdate message
- * Use of cookie extension instead of cookie field in ClientHello and HelloVerifyRequest messages
- * Added ACK message
- * Text about sequence number handling

Appendix E. Working Group Information

RFC EDITOR: PLEASE REMOVE THIS SECTION.

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

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

Appendix F. Contributors

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

In addition, we would like to thank:

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

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

Appendix G. Acknowledgements

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

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

Authors' Addresses

Eric Rescorla
RTFM, Inc.

Email: ekr@rtfm.com

Hannes Tschofenig
Arm Limited

Email: hannes.tschofenig@arm.com

Nagendra Modadugu
Google, Inc.

Email: nagendra@cs.stanford.edu

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

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

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

Abstract

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

Discussion Venues

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

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

Status of This Memo

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

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

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

This Internet-Draft will expire on 17 August 2022.

Copyright Notice

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

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

Table of Contents

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

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

1. Introduction

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

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

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

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

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

2. Conventions and Definitions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here. All TLS notation comes from [RFC8446], Section 3.

3. Overview

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

3.1. Topologies

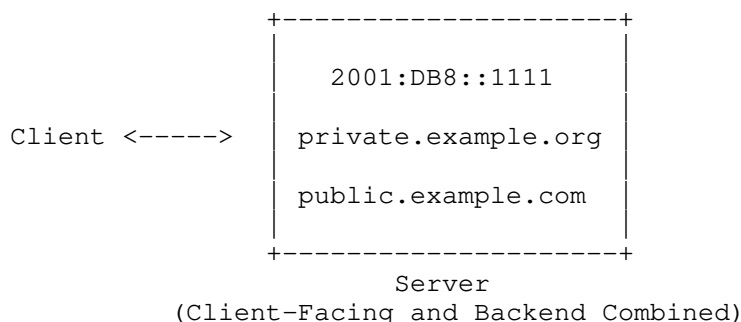


Figure 1: Shared Mode Topology

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

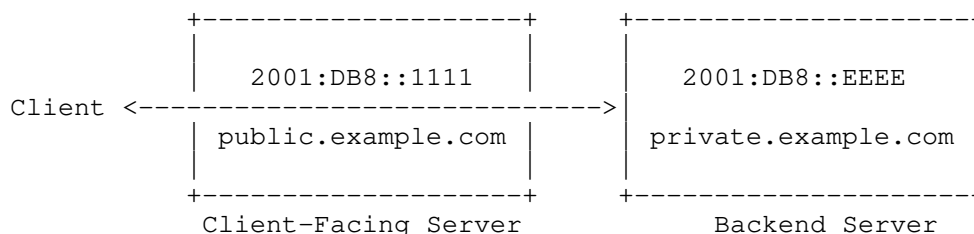


Figure 2: Split Mode Topology

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

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

3.2. Encrypted ClientHello (ECH)

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

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

The server takes one of the following actions:

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

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

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

4. Encrypted ClientHello Configuration

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

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

struct {
    HpkeKdfId kdf_id;
    HpkeAeadId aead_id;
} HpkeSymmetricCipherSuite;

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

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

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

The structure contains the following fields:

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

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

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

The ECHConfigContents structure contains the following fields:

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

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

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

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

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

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

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

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

The HpkeKeyConfig structure contains the following fields:

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

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

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

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

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

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

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

4.1. Configuration Identifiers

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

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

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

4.2. Configuration Extensions

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

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

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

5. The "encrypted_client_hello" Extension

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

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

The payload of the extension has the following structure:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

5.1. Encoding the ClientHelloInner

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

5.2. Authenticating the ClientHelloOuter

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

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

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

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

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

6. Client Behavior

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

6.1. Offering ECH

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

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

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

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

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

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

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

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

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

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

6.1.1. Encrypting the ClientHello

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

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

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

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

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

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

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

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

6.1.2. GREASE PSK

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

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

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

6.1.3. Recommended Padding Scheme

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

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

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

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

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

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

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

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

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

6.1.4. Determining ECH Acceptance

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

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

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

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

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

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

6.1.5. Handshaking with `ClientHelloInner`

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

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

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

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

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

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

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

6.1.6. Handshaking with ClientHelloOuter

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

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

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

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

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

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

6.1.7. Authenticating for the Public Name

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

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

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

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

6.2. GREASE ECH

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

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

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

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

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

7. Server Behavior

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

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

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

7.1. Client-Facing Server

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

7.1.1. Sending HelloRetryRequest

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

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

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

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

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

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

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

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

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

7.2. Backend Server

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

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

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

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

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

7.2.1. Sending HelloRetryRequest

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

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

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

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

8. Compatibility Issues

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

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

8.1. Misconfiguration and Deployment Concerns

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

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

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

8.2. Middleboxes

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

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

9. Compliance Requirements

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

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

10. Security Considerations

10.1. Security and Privacy Goals

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

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

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

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

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

10.2. Unauthenticated and Plaintext DNS

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

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

10.3. Client Tracking

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

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

10.4. Ignored Configuration Identifiers and Trial Decryption

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

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

10.5. Outer ClientHello

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

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

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

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

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

10.6. Related Privacy Leaks

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

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

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

10.7. Cookies

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

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

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

10.8. Attacks Exploiting Acceptance Confirmation

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

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

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

10.9. Comparison Against Criteria

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

10.9.1. Mitigate Cut-and-Paste Attacks

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

10.9.2. Avoid Widely Shared Secrets

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

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

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

10.9.4. Do Not Stick Out

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

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

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

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

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

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

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

10.9.5. Maintain Forward Secrecy

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

10.9.6. Enable Multi-party Security Contexts

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

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

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

10.9.7. Support Multiple Protocols

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

10.10. Padding Policy

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

10.11. Active Attack Mitigations

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

10.11.1. Client Reaction Attack Mitigation

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



Figure 3: Client reaction attack

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

10.11.2. HelloRetryRequest Hijack Mitigation

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

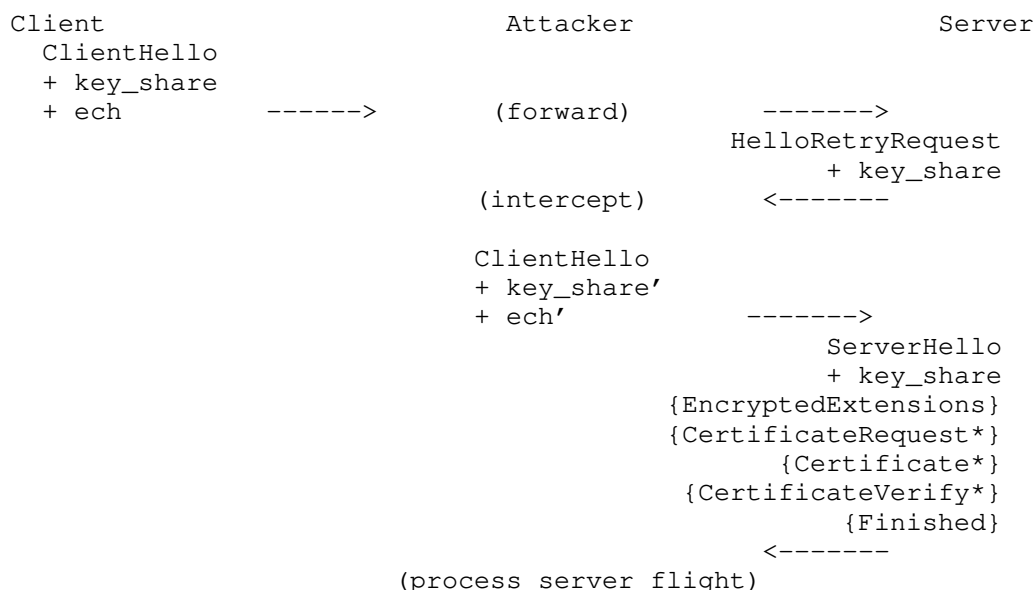


Figure 4: HelloRetryRequest hijack attack

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

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

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

10.11.3. ClientHello Malleability Mitigation

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

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

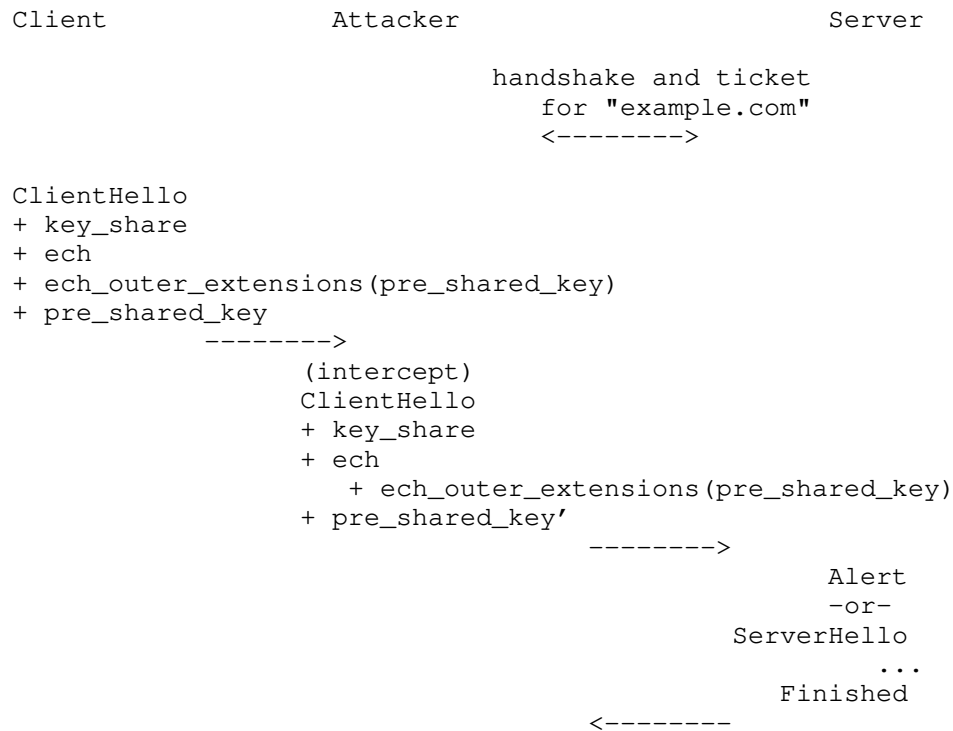


Figure 5: Message flow for malleable ClientHello

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

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

10.11.4. ClientHelloInner Packet Amplification Mitigation

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

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

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

11. IANA Considerations

11.1. Update of the TLS ExtensionType Registry

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

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

11.2. Update of the TLS Alert Registry

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

12. ECHConfig Extension Guidance

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

13. References

13.1. Normative References

- [HTTPS-RR] Schwartz, B., Bishop, M., and E. Nygren, "Service binding and parameter specification via the DNS (DNS SVCB and HTTPS RRs)", Work in Progress, Internet-Draft, draft-ietf-dnsop-svcb-https-08, 12 October 2021, <<https://www.ietf.org/archive/id/draft-ietf-dnsop-svcb-https-08.txt>>.
- [I-D.ietf-tls-exported-authenticator] Sullivan, N., "Exported Authenticators in TLS", Work in Progress, Internet-Draft, draft-ietf-tls-exported-authenticator-14, 25 January 2021, <<https://www.ietf.org/archive/id/draft-ietf-tls-exported-authenticator-14.txt>>.
- [I-D.irtf-cfrg-hpke] Barnes, R. L., Bhargavan, K., Lipp, B., and C. A. Wood, "Hybrid Public Key Encryption", Work in Progress, Internet-Draft, draft-irtf-cfrg-hpke-12, 2 September 2021, <<https://www.ietf.org/archive/id/draft-irtf-cfrg-hpke-12.txt>>.
- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <<https://www.rfc-editor.org/info/rfc2119>>.
- [RFC5890] Klensin, J., "Internationalized Domain Names for Applications (IDNA): Definitions and Document Framework", RFC 5890, DOI 10.17487/RFC5890, August 2010, <<https://www.rfc-editor.org/info/rfc5890>>.
- [RFC7918] Langley, A., Modadugu, N., and B. Moeller, "Transport Layer Security (TLS) False Start", RFC 7918, DOI 10.17487/RFC7918, August 2016, <<https://www.rfc-editor.org/info/rfc7918>>.
- [RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <<https://www.rfc-editor.org/info/rfc8174>>.
- [RFC8446] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <<https://www.rfc-editor.org/info/rfc8446>>.

13.2. Informative References

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

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

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

Appendix A. Alternative SNI Protection Designs

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

A.1. TLS-layer

A.1.1. TLS in Early Data

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

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

A.1.2. Combined Tickets

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

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

A.2. Application-layer

A.2.1. HTTP/2 CERTIFICATE Frames

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

Appendix B. Linear-time Outer Extension Processing

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

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

Appendix C. Acknowledgements

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

Appendix D. Change Log

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

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

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

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

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

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

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

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

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

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

Authors' Addresses

Eric Rescorla
RTFM, Inc.

Email: ekr@rtfm.com

Kazuho Oku
Fastly

Email: kazuhooku@gmail.com

Nick Sullivan
Cloudflare

Email: nick@cloudflare.com

Christopher A. Wood
Cloudflare

Email: caw@heapingbits.net

TLS
Internet-Draft
Intended status: Standards Track
Expires: 5 September 2022

N. Sullivan
Cloudflare Inc.
4 March 2022

Exported Authenticators in TLS
draft-ietf-tls-exported-authenticator-15

Abstract

This document describes a mechanism that builds on Transport Layer Security (TLS) or Datagram Transport Layer Security (DTLS) and enables peers to provide a proof of ownership of an identity, such as an X.509 certificate. This proof can be exported by one peer, transmitted out-of-band to the other peer, and verified by the receiving peer.

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 5 September 2022.

Copyright Notice

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

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

Table of Contents

1. Introduction	2
2. Conventions and Terminology	3
3. Message Sequences	4
4. Authenticator Request	4
5. Authenticator	6
5.1. Authenticator Keys	6
5.2. Authenticator Construction	7
5.2.1. Certificate	8
5.2.2. CertificateVerify	8
5.2.3. Finished	10
5.2.4. Authenticator Creation	10
6. Empty Authenticator	10
7. API considerations	11
7.1. The "request" API	11
7.2. The "get context" API	11
7.3. The "authenticate" API	11
7.4. The "validate" API	12
8. IANA Considerations	13
8.1. Update of the TLS ExtensionType Registry	13
8.2. Update of the TLS Exporter Labels Registry	13
8.3. Update of the TLS HandshakeType Registry	13
9. Security Considerations	13
10. Acknowledgements	14
11. References	14
11.1. Normative References	14
11.2. Informative References	15
Author's Address	16

1. Introduction

This document provides a way to authenticate one party of a Transport Layer Security (TLS) or Datagram Transport Layer Security (DTLS) connection to its peer using authentication messages created after the session has been established. This allows both the client and server to prove ownership of additional identities at any time after the handshake has completed. This proof of authentication can be exported and transmitted out-of-band from one party to be validated by its peer.

This mechanism provides two advantages over the authentication that TLS and DTLS natively provide:

multiple identities - Endpoints that are authoritative for multiple identities - but do not have a single certificate that includes all of the identities - can authenticate additional identities over a single connection.

spontaneous authentication - Endpoints can authenticate after a connection is established, in response to events in a higher-layer protocol, as well as integrating more context (such as context from the application).

Versions of TLS prior to TLS 1.3 used renegotiation as a way to enable post-handshake client authentication given an existing TLS connection. The mechanism described in this document may be used to replace the post-handshake authentication functionality provided by renegotiation. Unlike renegotiation, exported Authenticator-based post-handshake authentication does not require any changes at the TLS layer.

Post-handshake authentication is defined in section 4.6.3 of TLS 1.3 [RFC8446], but it has the disadvantage of requiring additional state to be stored as part of the TLS state machine. Furthermore, the authentication boundaries of TLS 1.3 post-handshake authentication align with TLS record boundaries, which are often not aligned with the authentication boundaries of the higher-layer protocol. For example, multiplexed connection protocols like HTTP/2 [RFC7540] do not have a notion of which TLS record a given message is a part of.

Exported Authenticators are meant to be used as a building block for application protocols. Mechanisms such as those required to advertise support and handle authentication errors are not handled by TLS (or DTLS).

The minimum version of TLS and DTLS required to implement the mechanisms described in this document are TLS 1.2 [RFC6347] and DTLS 1.2 [RFC5246].

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 uses terminology such as client, server, connection, handshake, endpoint, peer that are defined in section 1.1 of [RFC8446]. The term "initial connection" refers to the (D)TLS connection from which the exported authenticator messages are derived.

3. Message Sequences

There are two types of messages defined in this document: Authenticator Requests and Authenticators. These can be combined in the following three sequences:

Client Authentication

- * Server generates Authenticator Request
- * Client generates Authenticator from Server's Authenticator Request
- * Server validates Client's Authenticator

Server Authentication

- * Client generates Authenticator Request
- * Server generates Authenticator from Client's Authenticator Request
- * Client validates Server's Authenticator

Spontaneous Server Authentication

- * Server generates Authenticator
- * Client validates Server's Authenticator

4. Authenticator Request

The authenticator request is a structured message that can be created by either party of a (D)TLS connection using data exported from that connection. It can be transmitted to the other party of the (D)TLS connection at the application layer. The application layer protocol used to send the authenticator request SHOULD use a secure transport channel with equivalent security to TLS, such as QUIC [RFC9001], as its underlying transport to keep the request confidential. The application MAY use the existing (D)TLS connection to transport the authenticator.

An authenticator request message can be constructed by either the client or the server. Server-generated authenticator requests use the `CertificateRequest` message from Section 4.3.2 of [RFC8446]. Client-generated authenticator requests use a new message, called the `ClientCertificateRequest`, which uses the same structure as `CertificateRequest`. (Note that the latter is not a request for a client certificate, but rather a certificate request generated by the client.) These message structures are used even if the connection protocol is TLS 1.2 or DTLS 1.2.

The `CertificateRequest` and `ClientCertificateRequest` messages are used to define the parameters in a request for an authenticator. These are encoded as TLS handshake messages, including length and type fields. They do not include any TLS record layer framing and are not encrypted with a handshake or application-data key.

The structures are defined to be:

```
struct {
    opaque certificate_request_context<0..2^8-1>;
    Extension extensions<2..2^16-1>;
} ClientCertificateRequest;

struct {
    opaque certificate_request_context<0..2^8-1>;
    Extension extensions<2..2^16-1>;
} CertificateRequest;
```

`certificate_request_context`: An opaque string which identifies the authenticator request and which will be echoed in the authenticator message. A `certificate_request_context` value MUST be unique for each authenticator request within the scope of a connection (preventing replay and context confusion). The `certificate_request_context` SHOULD be chosen to be unpredictable to the peer (e.g., by randomly generating it) in order to prevent an attacker who has temporary access to the peer's private key from pre-computing valid authenticators. For example, the application may choose this value to correspond to a value used in an existing datastructure in the software to simplify implementation.

`extensions`: The set of extensions allowed in the `CertificateRequest` structure and the `ClientCertificateRequest` structure are those defined in the TLS ExtensionType Values IANA registry [RFC8447] containing CR in the TLS 1.3 column. In addition, the set of extensions in the `ClientCertificateRequest` structure MAY include the `server_name` [RFC6066] extension.

The uniqueness requirements of the `certificate_request_context` apply only to `CertificateRequest` and `ClientCertificateRequest` messages that are used as part of authenticator requests, but do apply across `CertificateRequest` and `ClientCertificateRequest` messages. A `certificate_request_context` value used in a `ClientCertificateRequest` cannot be used in an authenticator `CertificateRequest` on the same connection, and vice versa. There is no impact if the value of a `certificate_request_context` used in an authenticator request matches the value of a `certificate_request_context` in the handshake or in a post-handshake message.

5. Authenticator

The authenticator is a structured message that can be exported from either party of a (D)TLS connection. It can be transmitted to the other party of the (D)TLS connection at the application layer. The application layer protocol used to send the authenticator SHOULD use a secure transport channel with equivalent security to TLS, such as QUIC [RFC9001], as its underlying transport to keep the authenticator confidential. The application MAY use the existing (D)TLS connection to transport the authenticator.

An authenticator message can be constructed by either the client or the server given an established (D)TLS connection, an identity, such as an X.509 certificate, and a corresponding private key. Clients MUST NOT send an authenticator without a preceding authenticator request; for servers an authenticator request is optional. For authenticators that do not correspond to authenticator requests, the `certificate_request_context` is chosen by the server.

5.1. Authenticator Keys

Each authenticator is computed using a Handshake Context and Finished MAC Key derived from the (D)TLS connection. These values are derived using an exporter as described in Section 4 of [RFC5705] (for (D)TLS 1.2) or Section 7.5 of [RFC8446] (for (D)TLS 1.3). For (D)TLS 1.3, the `exporter_master_secret` MUST be used, not the `early_exporter_master_secret`. These values use different labels depending on the role of the sender:

- * The Handshake Context is an exporter value that is derived using the label "EXPORTER-client authenticator handshake context" or "EXPORTER-server authenticator handshake context" for authenticators sent by the client or server respectively.

- * The Finished MAC Key is an exporter value derived using the label "EXPORTER-client authenticator finished key" or "EXPORTER-server authenticator finished key" for authenticators sent by the client or server respectively.

The context_value used for the exporter is empty (zero length) for all four values. There is no need to include additional context information at this stage since the application-supplied context is included in the authenticator itself. The length of the exported value is equal to the length of the output of the hash function associated with the selected cipher suite (for TLS 1.3) or the hash function used for the pseudorandom function (PRF) (for (D)TLS 1.2). Exported authenticators cannot be used with (D)TLS 1.2 cipher suites that do not use the TLS PRF and with TLS 1.3 cipher suites that do not have an associated hash function. This hash is referred to as the authenticator hash.

To avoid key synchronization attacks, Exported Authenticators MUST NOT be generated or accepted on (D)TLS 1.2 connections that did not negotiate the extended master secret extension [RFC7627].

5.2. Authenticator Construction

An authenticator is formed from the concatenation of TLS 1.3 [RFC8446] Certificate, CertificateVerify, and Finished messages. These messages are encoded as TLS handshake messages, including length and type fields. They do not include any TLS record layer framing and are not encrypted with a handshake or application-data key.

If the peer populating the certificate_request_context field in an authenticator's Certificate message has already created or correctly validated an authenticator with the same value, then no authenticator should be constructed. If there is no authenticator request, the extensions are chosen from those presented in the (D)TLS handshake's ClientHello. Only servers can provide an authenticator without a corresponding request.

ClientHello extensions are used to determine permissible extensions in the server's unsolicited Certificate message in order to follow the general model for extensions in (D)TLS in which extensions can only be included as part of a Certificate message if they were previously sent as part of a CertificateRequest message or ClientHello message. This ensures that the recipient will be able to process such extensions.

5.2.1. Certificate

The Certificate message contains the identity to be used for authentication, such as the end-entity certificate and any supporting certificates in the chain. This structure is defined in [RFC8446], Section 4.4.2.

The Certificate message contains an opaque string called `certificate_request_context`, which is extracted from the authenticator request if present. If no authenticator request is provided, the `certificate_request_context` can be chosen arbitrarily but MUST be unique within the scope of the connection and be unpredictable to the peer.

Certificates chosen in the Certificate message MUST conform to the requirements of a Certificate message in the negotiated version of (D)TLS. In particular, the entries of `certificate_list` MUST be valid for the signature algorithms indicated by the peer in the `"signature_algorithms"` and `"signature_algorithms_cert"` extension, as described in Section 4.2.3 of [RFC8446] for (D)TLS 1.3 or from Sections 7.4.2 and 7.4.6 of [RFC5246] for (D)TLS 1.2.

In addition to `"signature_algorithms"` and `"signature_algorithms_cert"`, the `"server_name"` [RFC6066], `"certificate_authorities"` (Section 4.2.4. of [RFC8446]), and `"oid_filters"` (Section 4.2.5. of [RFC8446]) extensions are used to guide certificate selection.

Only the X.509 certificate type defined in [RFC8446] is supported. Alternative certificate formats such as [RFC7250] Raw Public Keys are not supported in this version of the specification and their use in this context has not yet been analysed.

If an authenticator request was provided, the Certificate message MUST contain only extensions present in the authenticator request. Otherwise, the Certificate message MUST contain only extensions present in the (D)TLS ClientHello. Unrecognized extensions in the authenticator request MUST be ignored.

5.2.2. CertificateVerify

This message is used to provide explicit proof that an endpoint possesses the private key corresponding to its identity. The format of this message is taken from TLS 1.3:

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

The algorithm field specifies the signature algorithm used (see Section 4.2.3 of [RFC8446] for the definition of this field). The signature is a digital signature using that algorithm.

The signature scheme MUST be a valid signature scheme for TLS 1.3. This excludes all RSASSA-PKCS1-v1_5 algorithms and combinations of ECDSA and hash algorithms that are not supported in TLS 1.3.

If an authenticator request is present, the signature algorithm MUST be chosen from one of the signature schemes present in the "signature_algorithms" extension of the authenticator request. Otherwise, with spontaneous server authentication, the signature algorithm used MUST be chosen from the "signature_algorithms" sent by the peer in the ClientHello of the (D)TLS handshake. If there are no available signature algorithms, then no authenticator should be constructed.

The signature is computed using the chosen signature scheme over the concatenation of:

- * A string that consists of octet 32 (0x20) repeated 64 times
- * The context string "Exported Authenticator" (which is not NUL-terminated)
- * A single 0 octet which serves as the separator
- * The hashed authenticator transcript

The authenticator transcript is the hash of the concatenated Handshake Context, authenticator request (if present), and Certificate message:

Hash(Handshake Context || authenticator request || Certificate)

Where Hash is the authenticator hash defined in section 4.1. If the authenticator request is not present, it is omitted from this construction, i.e., it is zero-length.

If the party that generates the exported authenticator does so with a different connection than the party that is validating it, then the Handshake Context will not match, resulting in a CertificateVerify message that does not validate. This includes situations in which

the application data is sent via TLS-terminating proxy. Given a failed CertificateVerify validation, it may be helpful for the application to confirm that both peers share the same connection using a value derived from the connection secrets (such as the Handshake Context) before taking a user-visible action.

5.2.3. Finished

An HMAC [HMAC] over the hashed authenticator transcript, which is the concatenation of the Handshake Context, authenticator request (if present), Certificate, and CertificateVerify. The HMAC is computed using the authenticator hash, using the Finished MAC Key as a key.

```
Finished = HMAC(Finished MAC Key, Hash(Handshake Context ||
    authenticator request || Certificate || CertificateVerify))
```

5.2.4. Authenticator Creation

An endpoint constructs an authenticator by serializing the Certificate, CertificateVerify, and Finished as TLS handshake messages and concatenating the octets:

```
Certificate || CertificateVerify || Finished
```

An authenticator is valid if the CertificateVerify message is correctly constructed given the authenticator request (if used) and the Finished message matches the expected value. When validating an authenticator, constant-time comparisons SHOULD be used for signature and MAC validation.

6. Empty Authenticator

If, given an authenticator request, the endpoint does not have an appropriate identity or does not want to return one, it constructs an authenticated refusal called an empty authenticator. This is a Finished message sent without a Certificate or CertificateVerify. This message is an HMAC over the hashed authenticator transcript with a Certificate message containing no CertificateEntries and the CertificateVerify message omitted. The HMAC is computed using the authenticator hash, using the Finished MAC Key as a key. This message is encoded as a TLS handshake message, including length and type field. It does not include TLS record layer framing and is not encrypted with a handshake or application-data key.

```
Finished = HMAC(Finished MAC Key, Hash(Handshake Context ||
    authenticator request || Certificate))
```

7. API considerations

The creation and validation of both authenticator requests and authenticators SHOULD be implemented inside the (D)TLS library even if it is possible to implement it at the application layer. (D)TLS implementations supporting the use of exported authenticators SHOULD provide application programming interfaces by which clients and servers may request and verify exported authenticator messages.

Notwithstanding the success conditions described below, all APIs MUST fail if:

- * the connection uses a (D)TLS version of 1.1 or earlier, or
- * the connection is (D)TLS 1.2 and the extended master secret extension [RFC7627] was not negotiated

The following sections describe APIs that are considered necessary to implement exported authenticators. These are informative only.

7.1. The "request" API

The "request" API takes as input:

- * `certificate_request_context` (from 0 to 255 octets)
- * set of extensions to include (this MUST include `signature_algorithms`) and the contents thereof

It returns an authenticator request, which is a sequence of octets that comprises a `CertificateRequest` or `ClientCertificateRequest` message.

7.2. The "get context" API

The "get context" API takes as input:

- * authenticator or authenticator request

It returns the `certificate_request_context`.

7.3. The "authenticate" API

The "authenticate" API takes as input:

- * a reference to the initial connection

- * an identity, such as a set of certificate chains and associated extensions (OCSP [RFC6960], SCT [RFC6962], etc.)
- * a signer (either the private key associated with the identity, or interface to perform private key operations) for each chain
- * an authenticator request or `certificate_request_context` (from 0 to 255 octets)

It returns either the exported authenticator or an empty authenticator as a sequence of octets. It is recommended that the logic for selecting the certificates and extensions to include in the exporter is implemented in the TLS library. Implementing this in the TLS library lets the implementer take advantage of existing extension and certificate selection logic and more easily remember which extensions were sent in the ClientHello.

It is also possible to implement this API outside of the TLS library using TLS exporters. This may be preferable in cases where the application does not have access to a TLS library with these APIs or when TLS is handled independently of the application layer protocol.

7.4. The "validate" API

The "validate" API takes as input:

- * a reference to the initial connection
- * an optional authenticator request
- * an authenticator
- * a function for validating a certificate chain

It returns a status to indicate whether the authenticator is valid or not after applying the function for validating the certificate chain to the chain contained in the authenticator. If validation is successful, it also returns the identity, such as the certificate chain and its extensions.

The API should return a failure if the `certificate_request_context` of the authenticator was used in a different authenticator that was previously validated. Well-formed empty authenticators are returned as invalid.

When validating an authenticator, constant-time comparison should be used.

8. IANA Considerations

8.1. Update of the TLS ExtensionType Registry

IANA is requested to update the entry for `server_name(0)` in the registry for ExtensionType (defined in [RFC8446]) by replacing the value in the "TLS 1.3" column with the value "CH, EE, CR" and adding this document in the "Reference" column.

IANA is also requested to add the following note to the registry:

The addition of the "CR" to the "TLS 1.3" column for the `server_name(0)` extension only marks the extension as valid in a ClientCertificateRequest created as part of client-generated authenticator requests.

8.2. Update of the TLS Exporter Labels Registry

IANA is requested to add the following entries to the registry for Exporter Labels (defined in [RFC5705]): "EXPORTER-client authenticator handshake context", "EXPORTER-server authenticator handshake context", "EXPORTER-client authenticator handshake context", "EXPORTER-client authenticator finished key" and "EXPORTER-server authenticator finished key" with "DTLS-OK" and "Recommended" set to "Y" and this document added to the "Reference" column.

8.3. Update of the TLS HandshakeType Registry

IANA is requested to add the following entry to the registry for HandshakeType (defined in [RFC8446]): "client_certificate_request" with "DTLS-OK" and "Recommended" set to "Y" and this document added to the "Reference" column with the following in the "Note" column: "Used in TLS versions prior to 1.3."

9. Security Considerations

The Certificate/Verify/Finished pattern intentionally looks like the TLS 1.3 pattern which now has been analyzed several times. For example, [SIGMAC] presents a relevant framework for analysis, and section 10. of [RFC8446] contains a comprehensive set of references.

Authenticators are independent and unidirectional. There is no explicit state change inside TLS when an authenticator is either created or validated. The application in possession of a validated authenticator can rely on any semantics associated with data in the `certificate_request_context`.

- * This property makes it difficult to formally prove that a server is jointly authoritative over multiple identities, rather than individually authoritative over each.
- * There is no indication in (D)TLS about which point in time an authenticator was computed. Any feedback about the time of creation or validation of the authenticator should be tracked as part of the application layer semantics if required.

The signatures generated with this API cover the context string "Exported Authenticator" and therefore cannot be transplanted into other protocols.

In TLS 1.3 the client can not explicitly learn from the TLS layer whether its Finished message was accepted. Because the application traffic keys are not dependent on the client's final flight, receiving messages from the server does not prove that the server received the client's Finished. To avoid disagreement between the client and server on the authentication status of EAs, servers MUST verify the client Finished before sending an EA or processing a received EA.

10. Acknowledgements

Comments on this proposal were provided by Martin Thomson.
Suggestions for Section 9 were provided by Karthikeyan Bhargavan.

11. References

11.1. Normative References

- [HMAC] Krawczyk, H., Bellare, M., and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication", RFC 2104, DOI 10.17487/RFC2104, February 1997, <<https://www.rfc-editor.org/info/rfc2104>>.
- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <<https://www.rfc-editor.org/info/rfc2119>>.
- [RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, DOI 10.17487/RFC5246, August 2008, <<https://www.rfc-editor.org/info/rfc5246>>.

- [RFC5705] Rescorla, E., "Keying Material Exporters for Transport Layer Security (TLS)", RFC 5705, DOI 10.17487/RFC5705, March 2010, <<https://www.rfc-editor.org/info/rfc5705>>.
- [RFC6066] Eastlake 3rd, D., "Transport Layer Security (TLS) Extensions: Extension Definitions", RFC 6066, DOI 10.17487/RFC6066, January 2011, <<https://www.rfc-editor.org/info/rfc6066>>.
- [RFC6347] Rescorla, E. and N. Modadugu, "Datagram Transport Layer Security Version 1.2", RFC 6347, DOI 10.17487/RFC6347, January 2012, <<https://www.rfc-editor.org/info/rfc6347>>.
- [RFC7627] Bhargavan, K., Ed., Delignat-Lavaud, A., Pironti, A., Langley, A., and M. Ray, "Transport Layer Security (TLS) Session Hash and Extended Master Secret Extension", RFC 7627, DOI 10.17487/RFC7627, September 2015, <<https://www.rfc-editor.org/info/rfc7627>>.
- [RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <<https://www.rfc-editor.org/info/rfc8174>>.
- [RFC8446] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <<https://www.rfc-editor.org/info/rfc8446>>.
- [RFC8447] Salowey, J. and S. Turner, "IANA Registry Updates for TLS and DTLS", RFC 8447, DOI 10.17487/RFC8447, August 2018, <<https://www.rfc-editor.org/info/rfc8447>>.

11.2. Informative References

- [RFC6960] Santesson, S., Myers, M., Ankney, R., Malpani, A., Galperin, S., and C. Adams, "X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP", RFC 6960, DOI 10.17487/RFC6960, June 2013, <<https://www.rfc-editor.org/info/rfc6960>>.
- [RFC6962] Laurie, B., Langley, A., and E. Kasper, "Certificate Transparency", RFC 6962, DOI 10.17487/RFC6962, June 2013, <<https://www.rfc-editor.org/info/rfc6962>>.
- [RFC7250] Wouters, P., Ed., Tschofenig, H., Ed., Gilmore, J., Weiler, S., and T. Kivinen, "Using Raw Public Keys in Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)", RFC 7250, DOI 10.17487/RFC7250, June 2014, <<https://www.rfc-editor.org/info/rfc7250>>.

- [RFC7540] Belshe, M., Peon, R., and M. Thomson, Ed., "Hypertext Transfer Protocol Version 2 (HTTP/2)", RFC 7540, DOI 10.17487/RFC7540, May 2015, <<https://www.rfc-editor.org/info/rfc7540>>.
- [RFC9001] Thomson, M., Ed. and S. Turner, Ed., "Using TLS to Secure QUIC", RFC 9001, DOI 10.17487/RFC9001, May 2021, <<https://www.rfc-editor.org/info/rfc9001>>.
- [SIGMAC] Krawczyk, H., "A Unilateral-to-Mutual Authentication Compiler for Key Exchange (with Applications to Client Authentication in TLS 1.3)", 2016, <<https://eprint.iacr.org/2016/711.pdf>>.

Author's Address

Nick Sullivan
Cloudflare Inc.
Email: nick@cloudflare.com

Network Working Group
Internet-Draft

Obsoletes: 5077, 5246, 6961, 8446 (if approved)
Updates: 5705, 6066, 7627, 8422 (if approved)
Intended status: Standards Track
Expires: 8 September 2022

E. Rescorla
Mozilla
7 March 2022

The Transport Layer Security (TLS) Protocol Version 1.3
draft-ietf-tls-rfc8446bis-04

Abstract

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

This document updates RFCs 5705, 6066, 7627, and 8422 and obsoletes RFCs 5077, 5246, 6961, and 8446. This document also specifies new requirements for TLS 1.2 implementations.

Status of This Memo

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

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

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

This Internet-Draft will expire on 8 September 2022.

Copyright Notice

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

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights

and restrictions with respect to this document. Code Components extracted from this document must include Revised BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Revised BSD License.

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	5
1.1. Conventions and Terminology	6
1.2. Relationship to RFC 8446	7
1.3. Major Differences from TLS 1.2	7
1.4. Updates Affecting TLS 1.2	8
2. Protocol Overview	9
2.1. Incorrect DHE Share	12
2.2. Resumption and Pre-Shared Key (PSK)	13
2.3. 0-RTT Data	15
3. Presentation Language	17
3.1. Basic Block Size	17
3.2. Miscellaneous	17
3.3. Numbers	18
3.4. Vectors	18
3.5. Enumerateds	19
3.6. Constructed Types	20
3.7. Constants	20
3.8. Variants	21
4. Handshake Protocol	22
4.1. Key Exchange Messages	23
4.1.1. Cryptographic Negotiation	23
4.1.2. Client Hello	24
4.1.3. Server Hello	27
4.1.4. Hello Retry Request	30
4.2. Extensions	31
4.2.1. Supported Versions	35
4.2.2. Cookie	36
4.2.3. Signature Algorithms	37
4.2.4. Certificate Authorities	41

4.2.5.	OID Filters	41
4.2.6.	Post-Handshake Certificate-Based Client Authentication	42
4.2.7.	Supported Groups	43
4.2.8.	Key Share	44
4.2.9.	Pre-Shared Key Exchange Modes	47
4.2.10.	Early Data Indication	48
4.2.11.	Pre-Shared Key Extension	51
4.3.	Server Parameters	55
4.3.1.	Encrypted Extensions	55
4.3.2.	Certificate Request	55
4.4.	Authentication Messages	56
4.4.1.	The Transcript Hash	58
4.4.2.	Certificate	59
4.4.3.	Certificate Verify	64
4.4.4.	Finished	66
4.5.	End of Early Data	67
4.6.	Post-Handshake Messages	68
4.6.1.	New Session Ticket Message	68
4.6.2.	Post-Handshake Authentication	70
4.6.3.	Key and Initialization Vector Update	71
5.	Record Protocol	72
5.1.	Record Layer	73
5.2.	Record Payload Protection	75
5.3.	Per-Record Nonce	77
5.4.	Record Padding	78
5.5.	Limits on Key Usage	79
6.	Alert Protocol	79
6.1.	Closure Alerts	81
6.2.	Error Alerts	82
7.	Cryptographic Computations	85
7.1.	Key Schedule	85
7.2.	Updating Traffic Secrets	89
7.3.	Traffic Key Calculation	89
7.4.	(EC)DHE Shared Secret Calculation	90
7.4.1.	Finite Field Diffie-Hellman	90
7.4.2.	Elliptic Curve Diffie-Hellman	90
7.5.	Exporters	91
8.	0-RTT and Anti-Replay	92
8.1.	Single-Use Tickets	93
8.2.	Client Hello Recording	94
8.3.	Freshness Checks	95
9.	Compliance Requirements	96
9.1.	Mandatory-to-Implement Cipher Suites	96
9.2.	Mandatory-to-Implement Extensions	97
9.3.	Protocol Invariants	98
10.	Security Considerations	99
11.	IANA Considerations	99

12. References	102
12.1. Normative References	102
12.2. Informative References	105
Appendix A. State Machine	113
A.1. Client	113
A.2. Server	114
Appendix B. Protocol Data Structures and Constant Values	115
B.1. Record Layer	116
B.2. Alert Messages	116
B.3. Handshake Protocol	117
B.3.1. Key Exchange Messages	118
B.3.2. Server Parameters Messages	123
B.3.3. Authentication Messages	124
B.3.4. Ticket Establishment	125
B.3.5. Updating Keys	125
B.4. Cipher Suites	126
Appendix C. Implementation Notes	127
C.1. Random Number Generation and Seeding	127
C.2. Certificates and Authentication	128
C.3. Implementation Pitfalls	128
C.4. Client Tracking Prevention	130
C.5. Unauthenticated Operation	130
Appendix D. Updates to TLS 1.2	131
Appendix E. Backward Compatibility	131
E.1. Negotiating with an Older Server	132
E.2. Negotiating with an Older Client	133
E.3. 0-RTT Backward Compatibility	133
E.4. Middlebox Compatibility Mode	134
E.5. Security Restrictions Related to Backward Compatibility	134
Appendix F. Overview of Security Properties	135
F.1. Handshake	135
F.1.1. Key Derivation and HKDF	139
F.1.2. Certificate-Based Client Authentication	140
F.1.3. 0-RTT	140
F.1.4. Exporter Independence	140
F.1.5. Post-Compromise Security	141
F.1.6. External References	141
F.2. Record Layer	141
F.2.1. External References	142
F.3. Traffic Analysis	142
F.4. Side Channel Attacks	143
F.5. Replay Attacks on 0-RTT	144
F.5.1. Replay and Exporters	146
F.6. PSK Identity Exposure	146
F.7. Sharing PSKs	146
F.8. Attacks on Static RSA	147
Appendix G. Changes Since -00	147

Contributors	147
Author's Address	155

1. Introduction

RFC EDITOR: PLEASE REMOVE THE FOLLOWING PARAGRAPH The source for this draft is maintained in GitHub. Suggested changes should be submitted as pull requests at <https://github.com/ekr/tls13-spec>. Instructions are on that page as well.

The primary goal of TLS is to provide a secure channel between two communicating peers; the only requirement from the underlying transport is a reliable, in-order, data stream. Specifically, the secure channel should provide the following properties:

- * Authentication: The server side of the channel is always authenticated; the client side is optionally authenticated. Authentication can happen via asymmetric cryptography (e.g., RSA [RSA], the Elliptic Curve Digital Signature Algorithm (ECDSA) [ECDSA], or the Edwards-Curve Digital Signature Algorithm (EdDSA) [RFC8032]) or a symmetric pre-shared key (PSK).
- * Confidentiality: Data sent over the channel after establishment is only visible to the endpoints. TLS does not hide the length of the data it transmits, though endpoints are able to pad TLS records in order to obscure lengths and improve protection against traffic analysis techniques.
- * Integrity: Data sent over the channel after establishment cannot be modified by attackers without detection.

These properties should be true even in the face of an attacker who has complete control of the network, as described in [RFC3552]. See Appendix F for a more complete statement of the relevant security properties.

TLS consists of two primary components:

- * A handshake protocol (Section 4) that authenticates the communicating parties, negotiates cryptographic modes and parameters, and establishes shared keying material. The handshake protocol is designed to resist tampering; an active attacker should not be able to force the peers to negotiate different parameters than they would if the connection were not under attack.

- * A record protocol (Section 5) that uses the parameters established by the handshake protocol to protect traffic between the communicating peers. The record protocol divides traffic up into a series of records, each of which is independently protected using the traffic keys.

TLS is application protocol independent; higher-level protocols can layer on top of TLS transparently. The TLS standard, however, does not specify how protocols add security with TLS; how to initiate TLS handshaking and how to interpret the authentication certificates exchanged are left to the judgment of the designers and implementors of protocols that run on top of TLS.

This document defines TLS version 1.3. While TLS 1.3 is not directly compatible with previous versions, all versions of TLS incorporate a versioning mechanism which allows clients and servers to interoperably negotiate a common version if one is supported by both peers.

This document supersedes and obsoletes previous versions of TLS, including version 1.2 [RFC5246]. It also obsoletes the TLS ticket mechanism defined in [RFC5077] and replaces it with the mechanism defined in Section 2.2. Because TLS 1.3 changes the way keys are derived, it updates [RFC5705] as described in Section 7.5. It also changes how Online Certificate Status Protocol (OCSP) messages are carried and therefore updates [RFC6066] and obsoletes [RFC6961] as described in Section 4.4.2.1.

1.1. Conventions and Terminology

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

The following terms are used:

client: The endpoint initiating the TLS 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 subsequent interactions within TLS.

peer: An endpoint. When discussing a particular endpoint, "peer" refers to the endpoint that is not the primary subject of discussion.

receiver: An endpoint that is receiving records.

sender: An endpoint that is transmitting records.

server: The endpoint that did not initiate the TLS connection.

1.2. Relationship to RFC 8446

TLS 1.3 was originally specified in [RFC8446]. This document is solely an editorial update. It contains updated text in areas which were found to be unclear as well as other editorial improvements. In addition, it removes the use of the term "master" as applied to secrets in favor of the term "main" or shorter names where no term was necessary.

1.3. Major Differences from TLS 1.2

The following is a list of the major functional differences between TLS 1.2 and TLS 1.3. It is not intended to be exhaustive, and there are many minor differences.

- * The list of supported symmetric encryption algorithms has been pruned of all algorithms that are considered legacy. Those that remain are all Authenticated Encryption with Associated Data (AEAD) algorithms. The cipher suite concept has been changed to separate the authentication and key exchange mechanisms from the record protection algorithm (including secret key length) and a hash to be used with both the key derivation function and handshake message authentication code (MAC).
- * A zero round-trip time (0-RTT) mode was added, saving a round trip at connection setup for some application data, at the cost of certain security properties.
- * Static RSA and Diffie-Hellman cipher suites have been removed; all public-key based key exchange mechanisms now provide forward secrecy.
- * All handshake messages after the ServerHello are now encrypted. The newly introduced EncryptedExtensions message allows various extensions previously sent in the clear in the ServerHello to also enjoy confidentiality protection.

- * The key derivation function has been redesigned. The new design allows easier analysis by cryptographers due to their improved key separation properties. The HMAC-based Extract-and-Expand Key Derivation Function (HKDF) is used as an underlying primitive.
- * The handshake state machine has been significantly restructured to be more consistent and to remove superfluous messages such as ChangeCipherSpec (except when needed for middlebox compatibility).
- * Elliptic curve algorithms are now in the base spec, and new signature algorithms, such as EdDSA, are included. TLS 1.3 removed point format negotiation in favor of a single point format for each curve.
- * Other cryptographic improvements were made, including changing the RSA padding to use the RSA Probabilistic Signature Scheme (RSASSA-PSS), and the removal of compression, the Digital Signature Algorithm (DSA), and custom Ephemeral Diffie-Hellman (DHE) groups.
- * The TLS 1.2 version negotiation mechanism has been deprecated in favor of a version list in an extension. This increases compatibility with existing servers that incorrectly implemented version negotiation.
- * Session resumption with and without server-side state as well as the PSK-based cipher suites of earlier TLS versions have been replaced by a single new PSK exchange.
- * References have been updated to point to the updated versions of RFCs, as appropriate (e.g., RFC 5280 rather than RFC 3280).

1.4. Updates Affecting TLS 1.2

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

- * A version downgrade protection mechanism is described in Section 4.1.3.
- * RSASSA-PSS signature schemes are defined in Section 4.2.3.
- * The "supported_versions" ClientHello extension can be used to negotiate the version of TLS to use, in preference to the legacy_version field of the ClientHello.

- * The "signature_algorithms_cert" extension allows a client to indicate which signature algorithms it can validate in X.509 certificates.
- * The term "master" as applied to secrets has been removed, and the "extended_master_secret" extension [RFC7627] has been renamed to "extended_main_secret".

Additionally, this document clarifies some compliance requirements for earlier versions of TLS; see Section 9.3.

2. Protocol Overview

The cryptographic parameters used by the secure channel are produced by the TLS handshake protocol. This sub-protocol of TLS is used by the client and server when first communicating with each other. The handshake protocol allows peers to negotiate a protocol version, select cryptographic algorithms, authenticate each other (with client authentication being optional), and establish shared secret keying material. Once the handshake is complete, the peers use the established keys to protect the application-layer traffic.

A failure of the handshake or other protocol error triggers the termination of the connection, optionally preceded by an alert message (Section 6).

TLS supports three basic key exchange modes:

- * (EC)DHE (Diffie-Hellman over either finite fields or elliptic curves)
- * PSK-only
- * PSK with (EC)DHE

Figure 1 below shows the basic full TLS handshake:

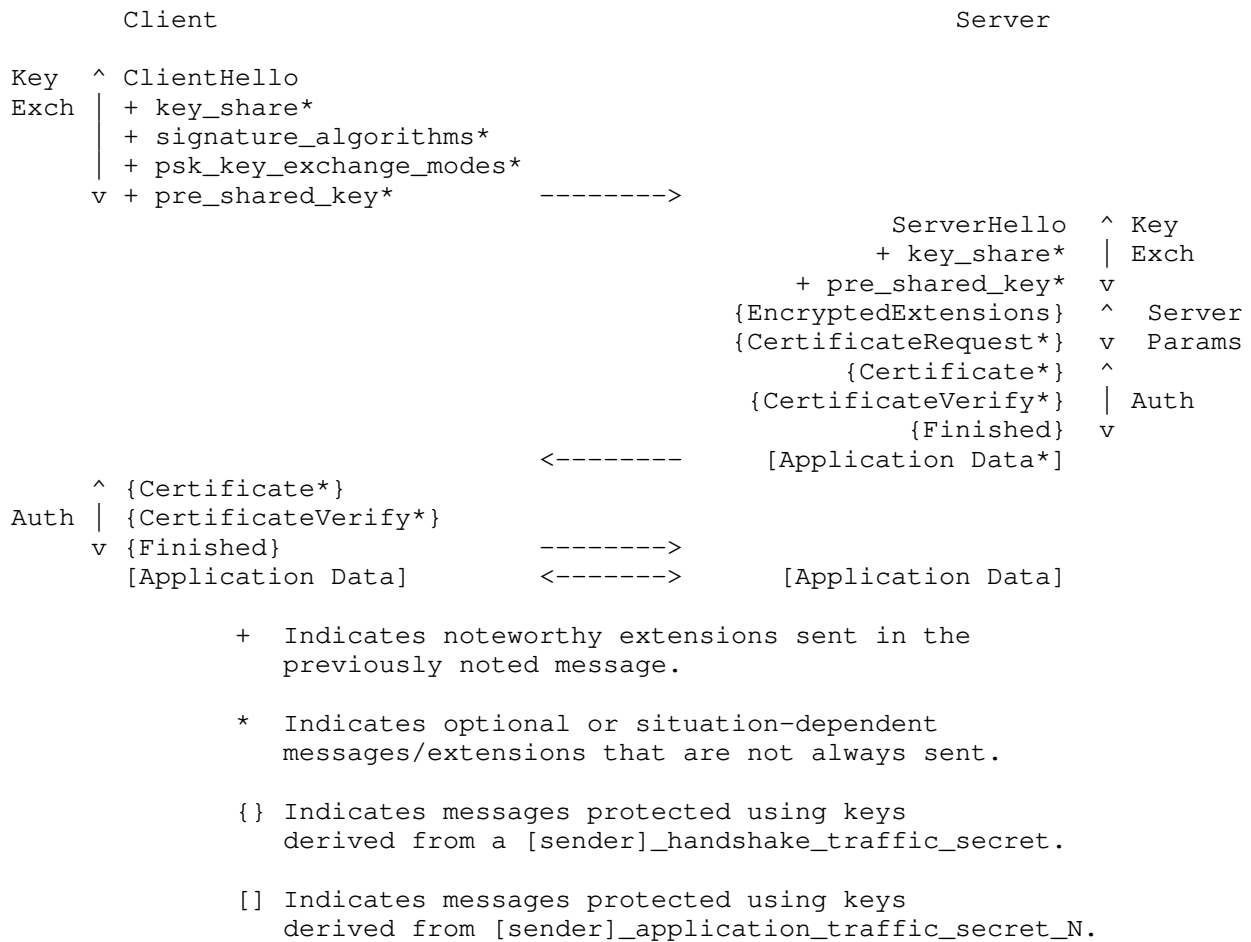


Figure 1: Message Flow for Full TLS Handshake

The handshake can be thought of as having three phases (indicated in the diagram above):

- * **Key Exchange:** Establish shared keying material and select the cryptographic parameters. Everything after this phase is encrypted.
- * **Server Parameters:** Establish other handshake parameters (whether the client is authenticated, application-layer protocol support, etc.).
- * **Authentication:** Authenticate the server (and, optionally, the client) and provide key confirmation and handshake integrity.

In the Key Exchange phase, the client sends the ClientHello (Section 4.1.2) message, which contains a random nonce (ClientHello.random); its offered protocol versions; a list of symmetric cipher/HKDF hash pairs; either a list of Diffie-Hellman key shares (in the "key_share" (Section 4.2.8) extension), a list of pre-shared key labels (in the "pre_shared_key" (Section 4.2.11) extension), or both; and potentially additional extensions. Additional fields and/or messages may also be present for middlebox compatibility.

The server processes the ClientHello and determines the appropriate cryptographic parameters for the connection. It then responds with its own ServerHello (Section 4.1.3), which indicates the negotiated connection parameters. The combination of the ClientHello and the ServerHello determines the shared keys. If (EC)DHE key establishment is in use, then the ServerHello contains a "key_share" extension with the server's ephemeral Diffie-Hellman share; the server's share MUST be in the same group as one of the client's shares. If PSK key establishment is in use, then the ServerHello contains a "pre_shared_key" extension indicating which of the client's offered PSKs was selected. Note that implementations can use (EC)DHE and PSK together, in which case both extensions will be supplied.

The server then sends two messages to establish the Server Parameters:

EncryptedExtensions: responses to ClientHello extensions that are not required to determine the cryptographic parameters, other than those that are specific to individual certificates.
[Section 4.3.1]

CertificateRequest: if certificate-based client authentication is desired, the desired parameters for that certificate. This message is omitted if client authentication is not desired.
[Section 4.3.2]

Finally, the client and server exchange Authentication messages. TLS uses the same set of messages every time that certificate-based authentication is needed. (PSK-based authentication happens as a side effect of key exchange.) Specifically:

Certificate: The certificate of the endpoint and any per-certificate

extensions. This message is omitted by the server if not authenticating with a certificate and by the client if the server did not send CertificateRequest (thus indicating that the client should not authenticate with a certificate). Note that if raw public keys [RFC7250] or the cached information extension [RFC7924] are in use, then this message will not contain a certificate but rather some other value corresponding to the server's long-term key. [Section 4.4.2]

CertificateVerify: A signature over the entire handshake using the private key corresponding to the public key in the Certificate message. This message is omitted if the endpoint is not authenticating via a certificate. [Section 4.4.3]

Finished: A MAC (Message Authentication Code) over the entire handshake. This message provides key confirmation, binds the endpoint's identity to the exchanged keys, and in PSK mode also authenticates the handshake. [Section 4.4.4]

Upon receiving the server's messages, the client responds with its Authentication messages, namely Certificate and CertificateVerify (if requested), and Finished.

At this point, the handshake is complete, and the client and server derive the keying material required by the record layer to exchange application-layer data protected through authenticated encryption. Application Data MUST NOT be sent prior to sending the Finished message, except as specified in Section 2.3. Note that while the server may send Application Data prior to receiving the client's Authentication messages, any data sent at that point is, of course, being sent to an unauthenticated peer.

2.1. Incorrect DHE Share

If the client has not provided a sufficient "key_share" extension (e.g., it includes only DHE or ECDHE groups unacceptable to or unsupported by the server), the server corrects the mismatch with a HelloRetryRequest and the client needs to restart the handshake with an appropriate "key_share" extension, as shown in Figure 2. If no common cryptographic parameters can be negotiated, the server MUST abort the handshake with an appropriate alert.



Figure 2: Message Flow for a Full Handshake with Mismatched Parameters

Note: The handshake transcript incorporates the initial ClientHello/HelloRetryRequest exchange; it is not reset with the new ClientHello.

TLS also allows several optimized variants of the basic handshake, as described in the following sections.

2.2. Resumption and Pre-Shared Key (PSK)

Although TLS PSKs can be established externally, PSKs can also be established in a previous connection and then used to establish a new connection ("session resumption" or "resuming" with a PSK). Once a handshake has completed, the server can send the client a PSK identity that corresponds to a unique key derived from the initial handshake (see Section 4.6.1). The client can then use that PSK identity in future handshakes to negotiate the use of the associated PSK. If the server accepts the PSK, then the security context of the new connection is cryptographically tied to the original connection and the key derived from the initial handshake is used to bootstrap the cryptographic state instead of a full handshake. In TLS 1.2 and below, this functionality was provided by "session IDs" and "session tickets" [RFC5077]. Both mechanisms are obsoleted in TLS 1.3.

PSKs can be used with (EC)DHE key exchange in order to provide forward secrecy in combination with shared keys, or can be used alone, at the cost of losing forward secrecy for the application data.

Figure 3 shows a pair of handshakes in which the first handshake establishes a PSK and the second handshake uses it:



Figure 3: Message Flow for Resumption and PSK

As the server is authenticating via a PSK, it does not send a Certificate or a CertificateVerify message. When a client offers resumption via a PSK, it SHOULD also supply a "key_share" extension to the server to allow the server to decline resumption and fall back to a full handshake, if needed. The server responds with a

"pre_shared_key" extension to negotiate the use of PSK key establishment and can (as shown here) respond with a "key_share" extension to do (EC)DHE key establishment, thus providing forward secrecy.

When PSKs are provisioned externally, the PSK identity and the KDF hash algorithm to be used with the PSK MUST also be provisioned.

Note: When using an externally provisioned pre-shared secret, a critical consideration is using sufficient entropy during the key generation, as discussed in [RFC4086]. Deriving a shared secret from a password or other low-entropy sources is not secure. A low-entropy secret, or password, is subject to dictionary attacks based on the PSK binder. The specified PSK authentication is not a strong password-based authenticated key exchange even when used with Diffie-Hellman key establishment. Specifically, it does not prevent an attacker that can observe the handshake from performing a brute-force attack on the password/pre-shared key.

2.3. 0-RTT Data

When clients and servers share a PSK (either obtained externally or via a previous handshake), TLS 1.3 allows clients to send data on the first flight ("early data"). The client uses the PSK to authenticate the server and to encrypt the early data.

As shown in Figure 4, the 0-RTT data is just added to the 1-RTT handshake in the first flight. The rest of the handshake uses the same messages as for a 1-RTT handshake with PSK resumption.

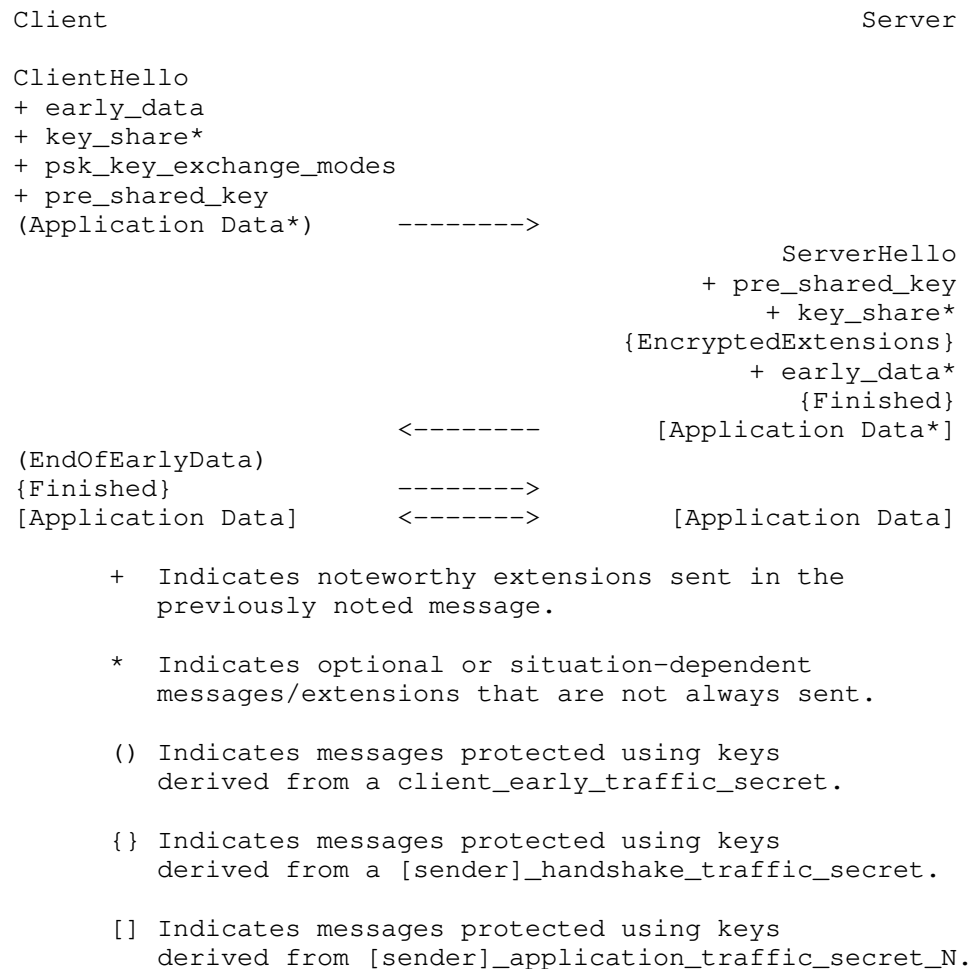


Figure 4: Message Flow for a 0-RTT Handshake

IMPORTANT NOTE: The security properties for 0-RTT data are weaker than those for other kinds of TLS data. Specifically:

1. The protocol does not provide any forward secrecy guarantees for this data. The server's behavior determines what forward secrecy guarantees, if any, apply (see Section 8.1). This behavior is not communicated to the client as part of the protocol. Therefore, absent out-of-band knowledge of the server's behavior, the client should assume that this data is not forward secret.

2. There are no guarantees of non-replay between connections. Protection against replay for ordinary TLS 1.3 1-RTT data is provided via the server's Random value, but 0-RTT data does not depend on the ServerHello and therefore has weaker guarantees. This is especially relevant if the data is authenticated either with TLS client authentication or inside the application protocol. The same warnings apply to any use of the `early_exporter_secret`.

0-RTT data cannot be duplicated within a connection (i.e., the server will not process the same data twice for the same connection), and an attacker will not be able to make 0-RTT data appear to be 1-RTT data (because it is protected with different keys). Appendix F.5 contains a description of potential attacks, and Section 8 describes mechanisms which the server can use to limit the impact of replay.

3. Presentation Language

This document deals with the formatting of data in an external representation. The following very basic and somewhat casually defined presentation syntax will be used.

3.1. Basic Block Size

The representation of all data items is explicitly specified. The basic data block size is one byte (i.e., 8 bits). Multiple-byte data items are concatenations of bytes, from left to right, from top to bottom. From the byte stream, a multi-byte item (a numeric in the following example) is formed (using C notation) by:

```
value = (byte[0] << 8*(n-1)) | (byte[1] << 8*(n-2)) |  
        ... | byte[n-1];
```

This byte ordering for multi-byte values is the commonplace network byte order or big-endian format.

3.2. Miscellaneous

Comments begin with `/*` and end with `*/`.

Optional components are denoted by enclosing them in `"[]"` (double brackets).

Single-byte entities containing uninterpreted data are of type `opaque`.

A type alias `T'` for an existing type `T` is defined by:


```
T T';
```

3.3. Numbers

The basic numeric data type is an unsigned byte (uint8). All larger numeric data types are constructed from a fixed-length series of bytes concatenated as described in Section 3.1 and are also unsigned. The following numeric types are predefined.

```
uint8 uint16[2];
uint8 uint24[3];
uint8 uint32[4];
uint8 uint64[8];
```

All values, here and elsewhere in the specification, are transmitted in network byte (big-endian) order; the uint32 represented by the hex bytes 01 02 03 04 is equivalent to the decimal value 16909060.

3.4. Vectors

A vector (single-dimensioned array) is a stream of homogeneous data elements. For presentation purposes, this specification refers to vectors as lists. The size of the vector may be specified at documentation time or left unspecified until runtime. In either case, the length declares the number of bytes, not the number of elements, in the vector. The syntax for specifying a new type, T', that is a fixed-length vector of type T is

```
T T'[n];
```

Here, T' occupies n bytes in the data stream, where n is a multiple of the size of T. The length of the vector is not included in the encoded stream.

In the following example, Datum is defined to be three consecutive bytes that the protocol does not interpret, while Data is three consecutive Datum, consuming a total of nine bytes.

```
opaque Datum[3];      /* three uninterpreted bytes */
Datum Data[9];         /* three consecutive 3-byte vectors */
```

Variable-length vectors are defined by specifying a subrange of legal lengths, inclusively, using the notation <floor..ceiling>. When these are encoded, the actual length precedes the vector's contents in the byte stream. The length will be in the form of a number consuming as many bytes as required to hold the vector's specified maximum (ceiling) length. A variable-length vector with an actual length field of zero is referred to as an empty vector.

```
T T'<floor..ceiling>;
```

In the following example, "mandatory" is a vector that must contain between 300 and 400 bytes of type opaque. It can never be empty. The actual length field consumes two bytes, a uint16, which is sufficient to represent the value 400 (see Section 3.3). Similarly, "longer" can represent up to 800 bytes of data, or 400 uint16 elements, and it may be empty. Its encoding will include a two-byte actual length field prepended to the vector. The length of an encoded vector must be an exact multiple of the length of a single element (e.g., a 17-byte vector of uint16 would be illegal).

```
opaque mandatory<300..400>;
/* length field is two bytes, cannot be empty */
uint16 longer<0..800>;
/* zero to 400 16-bit unsigned integers */
```

3.5. Enumerateds

An additional sparse data type, called "enum" or "enumerated", is available. Each definition is a different type. Only enumerateds of the same type may be assigned or compared. Every element of an enumerated must be assigned a value, as demonstrated in the following example. Since the elements of the enumerated are not ordered, they can be assigned any unique value, in any order.

```
enum { e1(v1), e2(v2), ... , en(vn) [[, (n)]] } Te;
```

Future extensions or additions to the protocol may define new values. Implementations need to be able to parse and ignore unknown values unless the definition of the field states otherwise.

An enumerated occupies as much space in the byte stream as would its maximal defined ordinal value. The following definition would cause one byte to be used to carry fields of type Color.

```
enum { red(3), blue(5), white(7) } Color;
```

One may optionally specify a value without its associated tag to force the width definition without defining a superfluous element.

In the following example, Taste will consume two bytes in the data stream but can only assume the values 1, 2, or 4 in the current version of the protocol.

```
enum { sweet(1), sour(2), bitter(4), (32000) } Taste;
```

The names of the elements of an enumeration are scoped within the defined type. In the first example, a fully qualified reference to the second element of the enumeration would be `Color.blue`. Such qualification is not required if the target of the assignment is well specified.

```
Color color = Color.blue;      /* overspecified, legal */
Color color = blue;           /* correct, type implicit */
```

The names assigned to enumerations do not need to be unique. The numerical value can describe a range over which the same name applies. The value includes the minimum and maximum inclusive values in that range, separated by two period characters. This is principally useful for reserving regions of the space.

```
enum { sad(0), meh(1..254), happy(255) } Mood;
```

3.6. Constructed Types

Structure types may be constructed from primitive types for convenience. Each specification declares a new, unique type. The syntax used for definitions is much like that of C.

```
struct {
    T1 f1;
    T2 f2;
    ...
    Tn fn;
} T;
```

Fixed- and variable-length list (vector) fields are allowed using the standard list syntax. Structures V1 and V2 in the variants example (Section 3.8) demonstrate this.

The fields within a structure may be qualified using the type's name, with a syntax much like that available for enumerations. For example, `T.f2` refers to the second field of the previous declaration.

3.7. Constants

Fields and variables may be assigned a fixed value using "=", as in:

```
struct {
    T1 f1 = 8; /* T.f1 must always be 8 */
    T2 f2;
} T;
```

3.8. Variants

Defined structures may have variants based on some knowledge that is available within the environment. The selector must be an enumerated type that defines the possible variants the structure defines. Each arm of the select (below) specifies the type of that variant's field and an optional field label. The mechanism by which the variant is selected at runtime is not prescribed by the presentation language.

```
struct {  
    T1 f1;  
    T2 f2;  
    ....  
    Tn fn;  
    select (E) {  
        case e1: T1 [[fe1]];  
        case e2: T2 [[fe2]];  
        ....  
        case en: Tn [[fen]];  
    };  
} Tv;
```

For example:

```
enum { apple(0), orange(1) } VariantTag;  
  
struct {  
    uint16 number;  
    opaque string<0..10>; /* variable length */  
} V1;  
  
struct {  
    uint32 number;  
    opaque string[10];    /* fixed length */  
} V2;  
  
struct {  
    VariantTag type;  
    select (VariantRecord.type) {  
        case apple: V1;  
        case orange: V2;  
    };  
} VariantRecord;
```

4. Handshake Protocol

The handshake protocol is used to negotiate the security parameters of a connection. Handshake messages are supplied to the TLS record layer, where they are encapsulated within one or more TLSPlaintext or TLSCiphertext structures which are processed and transmitted as specified by the current active connection state.

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

Protocol messages MUST be sent in the order defined in Section 4.4.1 and shown in the diagrams in Section 2. A peer which receives a handshake message in an unexpected order MUST abort the handshake with an "unexpected_message" alert.

New handshake message types are assigned by IANA as described in Section 11.

4.1. Key Exchange Messages

The key exchange messages are used to determine the security capabilities of the client and the server and to establish shared secrets, including the traffic keys used to protect the rest of the handshake and the data.

4.1.1. Cryptographic Negotiation

In TLS, the cryptographic negotiation proceeds by the client offering the following four sets of options in its ClientHello:

- * A list of cipher suites which indicates the AEAD algorithm/HKDF hash pairs which the client supports.
- * A "supported_groups" (Section 4.2.7) extension which indicates the (EC)DHE groups which the client supports and a "key_share" (Section 4.2.8) extension which contains (EC)DHE shares for some or all of these groups.
- * A "signature_algorithms" (Section 4.2.3) extension which indicates the signature algorithms which the client can accept. A "signature_algorithms_cert" extension (Section 4.2.3) may also be added to indicate certificate-specific signature algorithms.
- * A "pre_shared_key" (Section 4.2.11) extension which contains a list of symmetric key identities known to the client and a "psk_key_exchange_modes" (Section 4.2.9) extension which indicates the key exchange modes that may be used with PSKs.

If the server does not select a PSK, then the first three of these options are entirely orthogonal: the server independently selects a cipher suite, an (EC)DHE group and key share for key establishment, and a signature algorithm/certificate pair to authenticate itself to the client. If there is no overlap between the received "supported_groups" and the groups supported by the server, then the server MUST abort the handshake with a "handshake_failure" or an "insufficient_security" alert.

If the server selects a PSK, then it MUST also select a key establishment mode from the list indicated by the client's "psk_key_exchange_modes" extension (at present, PSK alone or with (EC)DHE). Note that if the PSK can be used without (EC)DHE, then non-overlap in the "supported_groups" parameters need not be fatal, as it is in the non-PSK case discussed in the previous paragraph.

If the server selects an (EC)DHE group and the client did not offer a compatible "key_share" extension in the initial ClientHello, the server MUST respond with a HelloRetryRequest (Section 4.1.4) message.

If the server successfully selects parameters and does not require a HelloRetryRequest, it indicates the selected parameters in the ServerHello as follows:

- * If PSK is being used, then the server will send a "pre_shared_key" extension indicating the selected key.
- * When (EC)DHE is in use, the server will also provide a "key_share" extension. If PSK is not being used, then (EC)DHE and certificate-based authentication are always used.
- * When authenticating via a certificate, the server will send the Certificate (Section 4.4.2) and CertificateVerify (Section 4.4.3) messages. In TLS 1.3 as defined by this document, either a PSK or a certificate is always used, but not both. Future documents may define how to use them together.

If the server is unable to negotiate a supported set of parameters (i.e., there is no overlap between the client and server parameters), it MUST abort the handshake with either a "handshake_failure" or "insufficient_security" fatal alert (see Section 6).

4.1.2. Client Hello

When a client first connects to a server, it is REQUIRED to send the ClientHello as its first TLS message. The client will also send a ClientHello when the server has responded to its ClientHello with a HelloRetryRequest. In that case, the client MUST send the same ClientHello without modification, except as follows:

- * If a "key_share" extension was supplied in the HelloRetryRequest, replacing the list of shares with a list containing a single KeyShareEntry from the indicated group.
- * Removing the "early_data" extension (Section 4.2.10) if one was present. Early data is not permitted after a HelloRetryRequest.
- * Including a "cookie" extension if one was provided in the HelloRetryRequest.
- * Updating the "pre_shared_key" extension if present by recomputing the "obfuscated_ticket_age" and binder values and (optionally) removing any PSKs which are incompatible with the server's indicated cipher suite.

- * Optionally adding, removing, or changing the length of the "padding" extension [RFC7685].
- * Other modifications that may be allowed by an extension defined in the future and present in the HelloRetryRequest.

Because TLS 1.3 forbids renegotiation, if a server has negotiated TLS 1.3 and receives a ClientHello at any other time, it MUST terminate the connection with an "unexpected_message" alert.

If a server established a TLS connection with a previous version of TLS and receives a TLS 1.3 ClientHello in a renegotiation, it MUST retain the previous protocol version. In particular, it MUST NOT negotiate TLS 1.3.

Structure of this message:

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

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

struct {
    ProtocolVersion legacy_version = 0x0303;    /* TLS v1.2 */
    Random random;
    opaque legacy_session_id<0..32>;
    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 TLS, 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 TLS 1.3, the client indicates its version preferences in the "supported_versions" extension (Section 4.2.1) and the legacy_version field MUST be set to 0x0303, which is the version number for TLS 1.2. TLS 1.3 ClientHellos are identified as having a legacy_version of 0x0303 and a supported_versions extension present with 0x0304 as the highest version indicated therein. (See Appendix E for details about backward compatibility.) A server which receives a legacy_version value not equal to 0x0303 MUST abort the handshake with an "illegal_parameter" alert.

random: 32 bytes generated by a secure random number generator. See

Appendix C for additional information.

legacy_session_id: Versions of TLS before TLS 1.3 supported a "session resumption" feature which has been merged with pre-shared keys in this version (see Section 2.2). A client which has a cached session ID set by a pre-TLS 1.3 server SHOULD set this field to that value. In compatibility mode (see Appendix E.4), this field MUST be non-empty, so a client not offering a pre-TLS 1.3 session MUST generate a new 32-byte value. This value need not be random but SHOULD be unpredictable to avoid implementations fixating on a specific value (also known as ossification). Otherwise, it MUST be set as a zero-length list (i.e., a zero-valued single byte length field).

cipher_suites: A list of the symmetric cipher options supported by the client, specifically the record protection algorithm (including secret key length) and a hash to be used with HKDF, in descending order of client preference. Values are defined in Appendix B.4. If the list contains cipher suites that the server does not recognize, support, or wish to use, the server MUST ignore those cipher suites and process the remaining ones as usual. If the client is attempting a PSK key establishment, it SHOULD advertise at least one cipher suite indicating a Hash associated with the PSK.

legacy_compression_methods: Versions of TLS before 1.3 supported compression with the list of supported compression methods being sent in this field. For every TLS 1.3 ClientHello, this list MUST contain exactly one byte, set to zero, which corresponds to the "null" compression method in prior versions of TLS. If a TLS 1.3 ClientHello is received with any other value in this field, the server MUST abort the handshake with an "illegal_parameter" alert. Note that TLS 1.3 servers might receive TLS 1.2 or prior ClientHellos which contain other compression methods and (if negotiating such a prior version) MUST follow the procedures for the appropriate prior version of TLS.

extensions: Clients request extended functionality from servers by sending data in the extensions field. The actual "Extension" format is defined in Section 4.2. In TLS 1.3, the use of certain extensions is mandatory, as functionality has moved into extensions to preserve ClientHello compatibility with previous versions of TLS. Servers MUST ignore unrecognized extensions.

All versions of TLS allow an extensions field to optionally follow the compression_methods field. TLS 1.3 ClientHello messages always contain extensions (minimally "supported_versions", otherwise, they will be interpreted as TLS 1.2 ClientHello messages). However, TLS

1.3 servers might receive ClientHello messages without an extensions field from prior versions of TLS. The presence of extensions can be detected by determining whether there are bytes following the compression_methods field at the end of the ClientHello. Note that this method of detecting optional data differs from the normal TLS method of having a variable-length field, but it is used for compatibility with TLS before extensions were defined. TLS 1.3 servers will need to perform this check first and only attempt to negotiate TLS 1.3 if the "supported_versions" extension is present. If negotiating a version of TLS prior to 1.3, a server MUST check that the message either contains no data after legacy_compression_methods or that it contains a valid extensions block with no data following. If not, then it MUST abort the handshake with a "decode_error" alert.

In the event that a client requests additional functionality using extensions and this functionality is not supplied by the server, the client MAY abort the handshake.

After sending the ClientHello message, the client waits for a ServerHello or HelloRetryRequest message. If early data is in use, the client may transmit early Application Data (Section 2.3) while waiting for the next handshake message.

4.1.3. Server Hello

The server will send this message in response to a ClientHello message to proceed with the handshake if it is able to negotiate an acceptable set of handshake parameters based on the ClientHello.

Structure of this message:

```
struct {
    ProtocolVersion legacy_version = 0x0303;    /* TLS v1.2 */
    Random random;
    opaque legacy_session_id_echo<0..32>;
    CipherSuite cipher_suite;
    uint8 legacy_compression_method = 0;
    Extension extensions<6..2^16-1>;
} ServerHello;
```

legacy_version: In previous versions of TLS, this field was used for

version negotiation and represented the selected version number for the connection. Unfortunately, some middleboxes fail when presented with new values. In TLS 1.3, the TLS server indicates its version using the "supported_versions" extension (Section 4.2.1), and the legacy_version field MUST be set to 0x0303, which is the version number for TLS 1.2. (See Appendix E for details about backward compatibility.)

random: 32 bytes generated by a secure random number generator. See Appendix C for additional information. The last 8 bytes MUST be overwritten as described below if negotiating TLS 1.2 or TLS 1.1, but the remaining bytes MUST be random. This structure is generated by the server and MUST be generated independently of the ClientHello.random.

legacy_session_id_echo: The contents of the client's legacy_session_id field. Note that this field is echoed even if the client's value corresponded to a cached pre-TLS 1.3 session which the server has chosen not to resume. A client which receives a legacy_session_id_echo field that does not match what it sent in the ClientHello MUST abort the handshake with an "illegal_parameter" alert.

cipher_suite: The single cipher suite selected by the server from the ClientHello.cipher_suites list. A client which receives a cipher suite that was not offered MUST abort the handshake with an "illegal_parameter" alert.

legacy_compression_method: A single byte which MUST have the value 0.

extensions: A list of extensions. The ServerHello MUST only include extensions which are required to establish the cryptographic context and negotiate the protocol version. All TLS 1.3 ServerHello messages MUST contain the "supported_versions" extension. Current ServerHello messages additionally contain either the "pre_shared_key" extension or the "key_share" extension, or both (when using a PSK with (EC)DHE key establishment). Other extensions (see Section 4.2) are sent separately in the EncryptedExtensions message.

For reasons of backward compatibility with middleboxes (see Appendix E.4), the HelloRetryRequest message uses the same structure as the ServerHello, but with Random set to the special value of the SHA-256 of "HelloRetryRequest":

```
CF 21 AD 74 E5 9A 61 11 BE 1D 8C 02 1E 65 B8 91
C2 A2 11 16 7A BB 8C 5E 07 9E 09 E2 C8 A8 33 9C
```

Upon receiving a message with type `server_hello`, implementations MUST first examine the Random value and, if it matches this value, process it as described in Section 4.1.4).

TLS 1.3 has a downgrade protection mechanism embedded in the server's random value. TLS 1.3 servers which negotiate TLS 1.2 or below in response to a `ClientHello` MUST set the last 8 bytes of their Random value specially in their `ServerHello`.

If negotiating TLS 1.2, TLS 1.3 servers MUST set the last 8 bytes of their Random value to the bytes:

```
44 4F 57 4E 47 52 44 01
```

If negotiating TLS 1.1 or below, TLS 1.3 servers MUST, and TLS 1.2 servers SHOULD, set the last 8 bytes of their `ServerHello.Random` value to the bytes:

```
44 4F 57 4E 47 52 44 00
```

Note that [RFC8996] and Appendix E.5 forbid the negotiation of TLS versions below 1.2; implementations which do not follow that guidance MUST behave as described above.

TLS 1.3 clients receiving a `ServerHello` indicating TLS 1.2 or below MUST check that the last 8 bytes are not equal to either of these values. TLS 1.2 clients SHOULD also check that the last 8 bytes are not equal to the second value if the `ServerHello` indicates TLS 1.1 or below. If a match is found, the client MUST abort the handshake with an "illegal_parameter" alert. This mechanism provides limited protection against downgrade attacks over and above what is provided by the `Finished` exchange: because the `ServerKeyExchange`, a message present in TLS 1.2 and below, includes a signature over both random values, it is not possible for an active attacker to modify the random values without detection as long as ephemeral ciphers are used. It does not provide downgrade protection when static RSA is used.

Note: This is a change from [RFC5246], so in practice many TLS 1.2 clients and servers will not behave as specified above.

A legacy TLS client performing renegotiation with TLS 1.2 or prior and which receives a TLS 1.3 `ServerHello` during renegotiation MUST abort the handshake with a "protocol_version" alert. Note that renegotiation is not possible when TLS 1.3 has been negotiated.

4.1.4. Hello Retry Request

The server will send this message in response to a ClientHello message if it is able to find an acceptable set of parameters but the ClientHello does not contain sufficient information to proceed with the handshake. As discussed in Section 4.1.3, the HelloRetryRequest has the same format as a ServerHello message, and the legacy_version, legacy_session_id_echo, cipher_suite, and legacy_compression_method fields have the same meaning. However, for convenience we discuss "HelloRetryRequest" throughout this document as if it were a distinct message.

The server's extensions MUST contain "supported_versions". Additionally, it SHOULD contain the minimal set of extensions necessary for the client to generate a correct ClientHello pair. As with the ServerHello, a HelloRetryRequest MUST NOT contain any extensions that were not first offered by the client in its ClientHello, with the exception of optionally the "cookie" (see Section 4.2.2) extension.

Upon receipt of a HelloRetryRequest, the client MUST check the legacy_version, legacy_session_id_echo, cipher_suite, and legacy_compression_method as specified in Section 4.1.3 and then process the extensions, starting with determining the version using "supported_versions". Clients MUST abort the handshake with an "illegal_parameter" alert if the HelloRetryRequest would not result in any change in the ClientHello. If a client receives a second HelloRetryRequest in the same connection (i.e., where the ClientHello was itself in response to a HelloRetryRequest), it MUST abort the handshake with an "unexpected_message" alert.

Otherwise, the client MUST process all extensions in the HelloRetryRequest and send a second updated ClientHello. The HelloRetryRequest extensions defined in this specification are:

- * supported_versions (see Section 4.2.1)
- * cookie (see Section 4.2.2)
- * key_share (see Section 4.2.8)

A client which receives a cipher suite that was not offered MUST abort the handshake. Servers MUST ensure that they negotiate the same cipher suite when receiving a conformant updated ClientHello (if the server selects the cipher suite as the first step in the negotiation, then this will happen automatically). Upon receiving the ServerHello, clients MUST check that the cipher suite supplied in the ServerHello is the same as that in the HelloRetryRequest and otherwise abort the handshake with an "illegal_parameter" alert.

In addition, in its updated ClientHello, the client SHOULD NOT offer any pre-shared keys associated with a hash other than that of the selected cipher suite. This allows the client to avoid having to compute partial hash transcripts for multiple hashes in the second ClientHello.

The value of `selected_version` in the HelloRetryRequest "supported_versions" extension MUST be retained in the ServerHello, and a client MUST abort the handshake with an "illegal_parameter" alert if the value changes.

4.2. Extensions

A number of TLS messages contain tag-length-value encoded extensions structures.

```
struct {
    ExtensionType extension_type;
    opaque extension_data<0..2^16-1>;
} Extension;

enum {
    server_name(0), /* RFC 6066 */
    max_fragment_length(1), /* RFC 6066 */
    status_request(5), /* RFC 6066 */
    supported_groups(10), /* RFC 8422, 7919 */
    signature_algorithms(13), /* RFC 8446 */
    use_srtp(14), /* RFC 5764 */
    heartbeat(15), /* RFC 6520 */
    application_layer_protocol_negotiation(16), /* RFC 7301 */
    signed_certificate_timestamp(18), /* RFC 6962 */
    client_certificate_type(19), /* RFC 7250 */
    server_certificate_type(20), /* RFC 7250 */
    padding(21), /* RFC 7685 */
    pre_shared_key(41), /* RFC 8446 */
    early_data(42), /* RFC 8446 */
    supported_versions(43), /* RFC 8446 */
    cookie(44), /* RFC 8446 */
    psk_key_exchange_modes(45), /* RFC 8446 */
    certificate_authorities(47), /* RFC 8446 */
    oid_filters(48), /* RFC 8446 */
    post_handshake_auth(49), /* RFC 8446 */
    signature_algorithms_cert(50), /* RFC 8446 */
    key_share(51), /* RFC 8446 */
    (65535)
} ExtensionType;
```

Here:

- * "extension_type" identifies the particular extension type.
- * "extension_data" contains information specific to the particular extension type.

The contents of the "extension_data" field are typically defined by an extension-specific structure defined in the TLS presentation language. Unless otherwise specified, trailing data is forbidden. That is, senders MUST NOT include data after the structure in the "extension_data" field. When processing an extension, receivers MUST abort the handshake with a "decode_error" alert if there is data left over after parsing the structure. This does not apply if the receiver does not implement or is configured to ignore an extension.

The list of extension types is maintained by IANA as described in Section 11.

Extensions are generally structured in a request/response fashion, though some extensions are just requests with no corresponding response (i.e., indications). The client sends its extension requests in the ClientHello message, and the server sends its extension responses in the ServerHello, EncryptedExtensions, HelloRetryRequest, and Certificate messages. The server sends extension requests in the CertificateRequest message which a client MAY respond to with a Certificate message. The server MAY also send unsolicited extensions in the NewSessionTicket, though the client does not respond directly to these.

Implementations MUST NOT send extension responses if the remote endpoint did not send the corresponding extension requests, with the exception of the "cookie" extension in the HelloRetryRequest. Upon receiving such an extension, an endpoint MUST abort the handshake with an "unsupported_extension" alert.

The table below indicates the messages where a given extension may appear, using the following notation: CH (ClientHello), SH (ServerHello), EE (EncryptedExtensions), CT (Certificate), CR (CertificateRequest), NST (NewSessionTicket), and HRR (HelloRetryRequest). If an implementation receives an extension which it recognizes and which is not specified for the message in which it appears, it MUST abort the handshake with an "illegal_parameter" alert.

Extension	TLS 1.3
server_name [RFC6066]	CH, EE
max_fragment_length [RFC6066]	CH, EE
status_request [RFC6066]	CH, CR, CT
supported_groups [RFC7919]	CH, EE
signature_algorithms (RFC8446)	CH, CR
use_srtp [RFC5764]	CH, EE
heartbeat [RFC6520]	CH, EE
application_layer_protocol_negotiation [RFC7301]	CH, EE

signed_certificate_timestamp [RFC6962]	CH, CR, CT
+-----+-----+	+-----+-----+
client_certificate_type [RFC7250]	CH, EE
+-----+-----+	+-----+-----+
server_certificate_type [RFC7250]	CH, EE
+-----+-----+	+-----+-----+
padding [RFC7685]	CH
+-----+-----+	+-----+-----+
cached_info [RFC7924]	CH, EE
+-----+-----+	+-----+-----+
key_share (RFC 8446)	CH, SH, HRR
+-----+-----+	+-----+-----+
pre_shared_key (RFC 8446)	CH, SH
+-----+-----+	+-----+-----+
psk_key_exchange_modes (RFC 8446)	CH
+-----+-----+	+-----+-----+
early_data (RFC 8446)	CH, EE, NST
+-----+-----+	+-----+-----+
cookie (RFC 8446)	CH, HRR
+-----+-----+	+-----+-----+
supported_versions (RFC 8446)	CH, SH, HRR
+-----+-----+	+-----+-----+
certificate_authorities (RFC 8446)	CH, CR
+-----+-----+	+-----+-----+
oid_filters (RFC 8446)	CR
+-----+-----+	+-----+-----+
post_handshake_auth (RFC 8446)	CH
+-----+-----+	+-----+-----+
signature_algorithms_cert (RFC 8446)	CH, CR
+-----+-----+	+-----+-----+

Table 1: TLS Extensions

When multiple extensions of different types are present, the extensions MAY appear in any order, with the exception of "pre_shared_key" (Section 4.2.11) which MUST be the last extension in the ClientHello (but can appear anywhere in the ServerHello extensions block). There MUST NOT be more than one extension of the same type in a given extension block.

In TLS 1.3, unlike TLS 1.2, extensions are negotiated for each handshake even when in resumption-PSK mode. However, 0-RTT parameters are those negotiated in the previous handshake; mismatches may require rejecting 0-RTT (see Section 4.2.10).

There are subtle (and not so subtle) interactions that may occur in this protocol between new features and existing features which may result in a significant reduction in overall security. The following considerations should be taken into account when designing new extensions:

- * Some cases where a server does not agree to an extension are error conditions (e.g., the handshake cannot continue), and some are simply refusals to support particular features. In general, error alerts should be used for the former and a field in the server extension response for the latter.
- * Extensions should, as far as possible, be designed to prevent any attack that forces use (or non-use) of a particular feature by manipulation of handshake messages. This principle should be followed regardless of whether the feature is believed to cause a security problem. Often the fact that the extension fields are included in the inputs to the Finished message hashes will be sufficient, but extreme care is needed when the extension changes the meaning of messages sent in the handshake phase. Designers and implementors should be aware of the fact that until the handshake has been authenticated, active attackers can modify messages and insert, remove, or replace extensions.

4.2.1. Supported Versions

```
struct {
    select (Handshake.msg_type) {
        case client_hello:
            ProtocolVersion versions<2..254>;

        case server_hello: /* and HelloRetryRequest */
            ProtocolVersion selected_version;
    };
} SupportedVersions;
```

The "supported_versions" extension is used by the client to indicate which versions of TLS it supports and by the server to indicate which version it is using. The extension contains a list of supported versions in preference order, with the most preferred version first. Implementations of this specification MUST send this extension in the ClientHello containing all versions of TLS which they are prepared to negotiate (for this specification, that means minimally 0x0304, but if previous versions of TLS are allowed to be negotiated, they MUST be present as well).

If this extension is not present, servers which are compliant with this specification and which also support TLS 1.2 MUST negotiate TLS 1.2 or prior as specified in [RFC5246], even if ClientHello.legacy_version is 0x0304 or later. Servers MAY abort the handshake upon receiving a ClientHello with legacy_version 0x0304 or later.

If this extension is present in the ClientHello, servers MUST NOT use the ClientHello.legacy_version value for version negotiation and MUST use only the "supported_versions" extension to determine client preferences. Servers MUST only select a version of TLS present in that extension and MUST ignore any unknown versions that are present in that extension. Note that this mechanism makes it possible to negotiate a version prior to TLS 1.2 if one side supports a sparse range. Implementations of TLS 1.3 which choose to support prior versions of TLS SHOULD support TLS 1.2. Servers MUST be prepared to receive ClientHellos that include this extension but do not include 0x0304 in the list of versions.

A server which negotiates a version of TLS prior to TLS 1.3 MUST set ServerHello.version and MUST NOT send the "supported_versions" extension. A server which negotiates TLS 1.3 MUST respond by sending a "supported_versions" extension containing the selected version value (0x0304). It MUST set the ServerHello.legacy_version field to 0x0303 (TLS 1.2).

After checking ServerHello.random to determine if the server handshake message is a ServerHello or HelloRetryRequest, clients MUST check for this extension prior to processing the rest of the ServerHello. This will require clients to parse the ServerHello in order to read the extension. If this extension is present, clients MUST ignore the ServerHello.legacy_version value and MUST use only the "supported_versions" extension to determine the selected version. If the "supported_versions" extension in the ServerHello contains a version not offered by the client or contains a version prior to TLS 1.3, the client MUST abort the handshake with an "illegal_parameter" alert.

4.2.2. Cookie

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

Cookies serve two primary purposes:

- * Allowing the server to force the client to demonstrate reachability at their apparent network address (thus providing a measure of DoS protection). This is primarily useful for non-connection-oriented transports (see [RFC6347] for an example of this).
- * Allowing the server to offload state to the client, thus allowing it to send a HelloRetryRequest without storing any state. The server can do this by storing the hash of the ClientHello in the HelloRetryRequest cookie (protected with some suitable integrity protection algorithm).

When sending a HelloRetryRequest, the server MAY provide a "cookie" extension to the client (this is an exception to the usual rule that the only extensions that may be sent are those that appear in the ClientHello). When sending the new ClientHello, the client MUST copy the contents of the extension received in the HelloRetryRequest into a "cookie" extension in the new ClientHello. Clients MUST NOT use cookies in their initial ClientHello in subsequent connections.

When a server is operating statelessly, it may receive an unprotected record of type change_cipher_spec between the first and second ClientHello (see Section 5). Since the server is not storing any state, this will appear as if it were the first message to be received. Servers operating statelessly MUST ignore these records.

4.2.3. Signature Algorithms

TLS 1.3 provides two extensions for indicating which signature algorithms may be used in digital signatures. The "signature_algorithms_cert" extension applies to signatures in certificates, and the "signature_algorithms" extension, which originally appeared in TLS 1.2, applies to signatures in CertificateVerify messages. The keys found in certificates MUST also be of appropriate type for the signature algorithms they are used with. This is a particular issue for RSA keys and PSS signatures, as described below. If no "signature_algorithms_cert" extension is present, then the "signature_algorithms" extension also applies to signatures appearing in certificates. Clients which desire the server to authenticate itself via a certificate MUST send the "signature_algorithms" extension. If a server is authenticating via a certificate and the client has not sent a "signature_algorithms" extension, then the server MUST abort the handshake with a "missing_extension" alert (see Section 9.2).

The "signature_algorithms_cert" extension was added to allow implementations which supported different sets of algorithms for certificates and in TLS itself to clearly signal their capabilities.

TLS 1.2 implementations SHOULD also process this extension. Implementations which have the same policy in both cases MAY omit the "signature_algorithms_cert" extension.

The "extension_data" field of these extensions contains a SignatureSchemeList value:

```
enum {
    /* RSASSA-PKCS1-v1_5 algorithms */
    rsa_pkcs1_sha256(0x0401),
    rsa_pkcs1_sha384(0x0501),
    rsa_pkcs1_sha512(0x0601),

    /* ECDSA algorithms */
    ecdsa_secp256r1_sha256(0x0403),
    ecdsa_secp384r1_sha384(0x0503),
    ecdsa_secp521r1_sha512(0x0603),

    /* RSASSA-PSS algorithms with public key OID rsaEncryption */
    rsa_pss_rsae_sha256(0x0804),
    rsa_pss_rsae_sha384(0x0805),
    rsa_pss_rsae_sha512(0x0806),

    /* EdDSA algorithms */
    ed25519(0x0807),
    ed448(0x0808),

    /* RSASSA-PSS algorithms with public key OID RSASSA-PSS */
    rsa_pss_pss_sha256(0x0809),
    rsa_pss_pss_sha384(0x080a),
    rsa_pss_pss_sha512(0x080b),

    /* Legacy algorithms */
    rsa_pkcs1_shal(0x0201),
    ecdsa_shal(0x0203),

    /* Reserved Code Points */
    private_use(0xFE00..0xFFFF),
    (0xFFFF)
} SignatureScheme;

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

Note: This enum is named "SignatureScheme" because there is already a "SignatureAlgorithm" type in TLS 1.2, which this replaces. We use the term "signature algorithm" throughout the text.

Each SignatureScheme value lists a single signature algorithm that the client is willing to verify. The values are indicated in descending order of preference. Note that a signature algorithm takes as input an arbitrary-length message, rather than a digest. Algorithms which traditionally act on a digest should be defined in TLS to first hash the input with a specified hash algorithm and then proceed as usual. The code point groups listed above have the following meanings:

RSASSA-PKCS1-v1_5 algorithms: Indicates a signature algorithm using RSASSA-PKCS1-v1_5 [RFC8017] with the corresponding hash algorithm as defined in [SHS]. These values refer solely to signatures which appear in certificates (see Section 4.4.2.2) and are not defined for use in signed TLS handshake messages, although they MAY appear in "signature_algorithms" and "signature_algorithms_cert" for backward compatibility with TLS 1.2.

ECDSA algorithms: Indicates a signature algorithm using ECDSA [ECDSA], the corresponding curve as defined in ANSI X9.62 [ECDSA] and FIPS 186-4 [DSS], and the corresponding hash algorithm as defined in [SHS]. The signature is represented as a DER-encoded [X690] ECDSA-Sig-Value structure as defined in [RFC4492].

RSASSA-PSS RSAE algorithms: Indicates a signature algorithm using RSASSA-PSS [RFC8017] with mask generation function 1. The digest used in the mask generation function and the digest being signed are both the corresponding hash algorithm as defined in [SHS]. The length of the Salt MUST be equal to the length of the output of the digest algorithm. If the public key is carried in an X.509 certificate, it MUST use the rsaEncryption OID [RFC5280].

EdDSA algorithms: Indicates a signature algorithm using EdDSA as defined in [RFC8032] or its successors. Note that these correspond to the "PureEdDSA" algorithms and not the "prehash" variants.

RSASSA-PSS PSS algorithms: Indicates a signature algorithm using RSASSA-PSS [RFC8017] with mask generation function 1. The digest used in the mask generation function and the digest being signed are both the corresponding hash algorithm as defined in [SHS]. The length of the Salt MUST be equal to the length of the digest algorithm. If the public key is carried in an X.509 certificate, it MUST use the RSASSA-PSS OID [RFC5756]. When used in certificate signatures, the algorithm parameters MUST be DER encoded. If the corresponding public key's parameters are present, then the parameters in the signature MUST be identical to those in the public key.

Legacy algorithms: Indicates algorithms which are being deprecated because they use algorithms with known weaknesses, specifically SHA-1 which is used in this context with either (1) RSA using RSASSA-PKCS1-v1_5 or (2) ECDSA. These values refer solely to signatures which appear in certificates (see Section 4.4.2.2) and are not defined for use in signed TLS handshake messages, although they MAY appear in "signature_algorithms" and "signature_algorithms_cert" for backward compatibility with TLS 1.2. Endpoints SHOULD NOT negotiate these algorithms but are permitted to do so solely for backward compatibility. Clients offering these values MUST list them as the lowest priority (listed after all other algorithms in SignatureSchemeList). TLS 1.3 servers MUST NOT offer a SHA-1 signed certificate unless no valid certificate chain can be produced without it (see Section 4.4.2.2).

The signatures on certificates that are self-signed or certificates that are trust anchors are not validated, since they begin a certification path (see [RFC5280], Section 3.2). A certificate that begins a certification path MAY use a signature algorithm that is not advertised as being supported in the "signature_algorithms" and "signature_algorithms_cert" extensions.

Note that TLS 1.2 defines this extension differently. TLS 1.3 implementations willing to negotiate TLS 1.2 MUST behave in accordance with the requirements of [RFC5246] when negotiating that version. In particular:

- * TLS 1.2 ClientHellos MAY omit this extension.
- * In TLS 1.2, the extension contained hash/signature pairs. The pairs are encoded in two octets, so SignatureScheme values have been allocated to align with TLS 1.2's encoding. Some legacy pairs are left unallocated. These algorithms are deprecated as of TLS 1.3. They MUST NOT be offered or negotiated by any implementation. In particular, MD5 [SLOTH], SHA-224, and DSA MUST NOT be used.
- * ECDSA signature schemes align with TLS 1.2's ECDSA hash/signature pairs. However, the old semantics did not constrain the signing curve. If TLS 1.2 is negotiated, implementations MUST be prepared to accept a signature that uses any curve that they advertised in the "supported_groups" extension.
- * Implementations that advertise support for RSASSA-PSS (which is mandatory in TLS 1.3) MUST be prepared to accept a signature using that scheme even when TLS 1.2 is negotiated. In TLS 1.2, RSASSA-PSS is used with RSA cipher suites.

4.2.4. Certificate Authorities

The "certificate_authorities" extension is used to indicate the certificate authorities (CAs) which an endpoint supports and which SHOULD be used by the receiving endpoint to guide certificate selection.

The body of the "certificate_authorities" extension consists of a CertificateAuthoritiesExtension structure.

```
opaque DistinguishedName<1..2^16-1>;

struct {
    DistinguishedName authorities<3..2^16-1>;
} CertificateAuthoritiesExtension;
```

authorities: A list of the distinguished names [X501] of acceptable certificate authorities, represented in DER-encoded [X690] format. These distinguished names specify a desired distinguished name for a trust anchor or subordinate CA; thus, this message can be used to describe known trust anchors as well as a desired authorization space.

The client MAY send the "certificate_authorities" extension in the ClientHello message. The server MAY send it in the CertificateRequest message.

The "trusted_ca_keys" extension [RFC6066], which serves a similar purpose, but is more complicated, is not used in TLS 1.3 (although it may appear in ClientHello messages from clients which are offering prior versions of TLS).

4.2.5. OID Filters

The "oid_filters" extension allows servers to provide a list of OID/value pairs which it would like the client's certificate to match. This extension, if provided by the server, MUST only be sent in the CertificateRequest message.

```
struct {
    opaque certificate_extension_oid<1..2^8-1>;
    opaque certificate_extension_values<0..2^16-1>;
} OIDFilter;

struct {
    OIDFilter filters<0..2^16-1>;
} OIDFilterExtension;
```


filters: A list of certificate extension OIDs [RFC5280] with their allowed value(s) and represented in DER-encoded [X690] format. Some certificate extension OIDs allow multiple values (e.g., Extended Key Usage). If the server has included a non-empty filters list, the client certificate included in the response MUST contain all of the specified extension OIDs that the client recognizes. For each extension OID recognized by the client, all of the specified values MUST be present in the client certificate (but the certificate MAY have other values as well). However, the client MUST ignore and skip any unrecognized certificate extension OIDs. If the client ignored some of the required certificate extension OIDs and supplied a certificate that does not satisfy the request, the server MAY at its discretion either continue the connection without client authentication or abort the handshake with an "unsupported_certificate" alert. Any given OID MUST NOT appear more than once in the filters list.

PKIX RFCs define a variety of certificate extension OIDs and their corresponding value types. Depending on the type, matching certificate extension values are not necessarily bitwise-equal. It is expected that TLS implementations will rely on their PKI libraries to perform certificate selection using certificate extension OIDs.

This document defines matching rules for two standard certificate extensions defined in [RFC5280]:

- * The Key Usage extension in a certificate matches the request when all key usage bits asserted in the request are also asserted in the Key Usage certificate extension.
- * The Extended Key Usage extension in a certificate matches the request when all key purpose OIDs present in the request are also found in the Extended Key Usage certificate extension. The special anyExtendedKeyUsage OID MUST NOT be used in the request.

Separate specifications may define matching rules for other certificate extensions.

4.2.6. Post-Handshake Certificate-Based Client Authentication

The "post_handshake_auth" extension is used to indicate that a client is willing to perform post-handshake authentication (Section 4.6.2). Servers MUST NOT send a post-handshake CertificateRequest to clients which do not offer this extension. Servers MUST NOT send this extension.

```
struct {} PostHandshakeAuth;
```

The "extension_data" field of the "post_handshake_auth" extension is zero length.

4.2.7. Supported Groups

When sent by the client, the "supported_groups" extension indicates the named groups which the client supports for key exchange, ordered from most preferred to least preferred.

Note: In versions of TLS prior to TLS 1.3, this extension was named "elliptic_curves" and only contained elliptic curve groups. See [RFC8422] and [RFC7919]. This extension was also used to negotiate ECDSA curves. Signature algorithms are now negotiated independently (see Section 4.2.3).

The "extension_data" field of this extension contains a "NamedGroupList" value:

```
enum {  
  
    /* Elliptic Curve Groups (ECDHE) */  
    secp256r1(0x0017), secp384r1(0x0018), secp521r1(0x0019),  
    x25519(0x001D), x448(0x001E),  
  
    /* Finite Field Groups (DHE) */  
    ffdhe2048(0x0100), ffdhe3072(0x0101), ffdhe4096(0x0102),  
    ffdhe6144(0x0103), ffdhe8192(0x0104),  
  
    /* Reserved Code Points */  
    ffdhe_private_use(0x01FC..0x01FF),  
    ecdhe_private_use(0xFE00..0xFEFF),  
    (0xFFFF)  
} NamedGroup;  
  
struct {  
    NamedGroup named_group_list<2..2^16-1>;  
} NamedGroupList;
```

Elliptic Curve Groups (ECDHE): Indicates support for the corresponding named curve, defined in either FIPS 186-4 [DSS] or in [RFC7748]. Values 0xFE00 through 0xFEFF are reserved for Private Use [RFC8126].

Finite Field Groups (DHE): Indicates support for the corresponding finite field group, defined in [RFC7919]. Values 0x01FC through 0x01FF are reserved for Private Use.

Items in "named_group_list" are ordered according to the sender's preferences (most preferred choice first).

As of TLS 1.3, servers are permitted to send the "supported_groups" extension to the client. Clients MUST NOT act upon any information found in "supported_groups" prior to successful completion of the handshake but MAY use the information learned from a successfully completed handshake to change what groups they use in their "key_share" extension in subsequent connections. If the server has a group it prefers to the ones in the "key_share" extension but is still willing to accept the ClientHello, it SHOULD send "supported_groups" to update the client's view of its preferences; this extension SHOULD contain all groups the server supports, regardless of whether they are currently supported by the client.

4.2.8. Key Share

The "key_share" extension contains the endpoint's cryptographic parameters.

Clients MAY send an empty client_shares list in order to request group selection from the server, at the cost of an additional round trip (see Section 4.1.4).

```
struct {
    NamedGroup group;
    opaque key_exchange<1..216-1>;
} KeyShareEntry;
```

group: The named group for the key being exchanged.

key_exchange: Key exchange information. The contents of this field are determined by the specified group and its corresponding definition. Finite Field Diffie-Hellman [DH76] parameters are described in Section 4.2.8.1; Elliptic Curve Diffie-Hellman parameters are described in Section 4.2.8.2.

In the ClientHello message, the "extension_data" field of this extension contains a "KeyShareClientHello" value:

```
struct {
    KeyShareEntry client_shares<0..216-1>;
} KeyShareClientHello;
```

client_shares: A list of offered KeyShareEntry values in descending order of client preference.

This list MAY be empty if the client is requesting a HelloRetryRequest. Each KeyShareEntry value MUST correspond to a group offered in the "supported_groups" extension and MUST appear in the same order. However, the values MAY be a non-contiguous subset of the "supported_groups" extension and MAY omit the most preferred groups. Such a situation could arise if the most preferred groups are new and unlikely to be supported in enough places to make pregenerating key shares for them efficient.

Clients can offer as many KeyShareEntry values as the number of supported groups it is offering, each representing a single set of key exchange parameters. For instance, a client might offer shares for several elliptic curves or multiple FFDHE groups. The key_exchange values for each KeyShareEntry MUST be generated independently. Clients MUST NOT offer multiple KeyShareEntry values for the same group. Clients MUST NOT offer any KeyShareEntry values for groups not listed in the client's "supported_groups" extension. Servers MAY check for violations of these rules and abort the handshake with an "illegal_parameter" alert if one is violated.

In a HelloRetryRequest message, the "extension_data" field of this extension contains a KeyShareHelloRetryRequest value:

```
struct {  
    NamedGroup selected_group;  
} KeyShareHelloRetryRequest;
```

selected_group: The mutually supported group the server intends to negotiate and is requesting a retried ClientHello/KeyShare for.

Upon receipt of this extension in a HelloRetryRequest, the client MUST verify that (1) the selected_group field corresponds to a group which was provided in the "supported_groups" extension in the original ClientHello and (2) the selected_group field does not correspond to a group which was provided in the "key_share" extension in the original ClientHello. If either of these checks fails, then the client MUST abort the handshake with an "illegal_parameter" alert. Otherwise, when sending the new ClientHello, the client MUST replace the original "key_share" extension with one containing only a new KeyShareEntry for the group indicated in the selected_group field of the triggering HelloRetryRequest.

In a ServerHello message, the "extension_data" field of this extension contains a KeyShareServerHello value:

```
struct {  
    KeyShareEntry server_share;  
} KeyShareServerHello;
```

server_share: A single KeyShareEntry value that is in the same group as one of the client's shares.

If using (EC)DHE key establishment, servers offer exactly one KeyShareEntry in the ServerHello. This value MUST be in the same group as the KeyShareEntry value offered by the client that the server has selected for the negotiated key exchange. Servers MUST NOT send a KeyShareEntry for any group not indicated in the client's "supported_groups" extension and MUST NOT send a KeyShareEntry when using the "psk_ke" PskKeyExchangeMode. If using (EC)DHE key establishment and a HelloRetryRequest containing a "key_share" extension was received by the client, the client MUST verify that the selected NamedGroup in the ServerHello is the same as that in the HelloRetryRequest. If this check fails, the client MUST abort the handshake with an "illegal_parameter" alert.

4.2.8.1. Diffie-Hellman Parameters

Diffie-Hellman [DH76] parameters for both clients and servers are encoded in the opaque key_exchange field of a KeyShareEntry in a KeyShare structure. The opaque value contains the Diffie-Hellman public value ($Y = g^X \bmod p$) for the specified group (see [RFC7919] for group definitions) encoded as a big-endian integer and padded to the left with zeros to the size of p in bytes.

Note: For a given Diffie-Hellman group, the padding results in all public keys having the same length.

Peers MUST validate each other's public key Y by ensuring that $1 < Y < p-1$. This check ensures that the remote peer is properly behaved and isn't forcing the local system into a small subgroup.

4.2.8.2. ECDHE Parameters

ECDHE parameters for both clients and servers are encoded in the opaque key_exchange field of a KeyShareEntry in a KeyShare structure.

For secp256r1, secp384r1, and secp521r1, the contents are the serialized value of the following struct:

```
struct {
    uint8 legacy_form = 4;
    opaque X[coordinate_length];
    opaque Y[coordinate_length];
} UncompressedPointRepresentation;
```

X and Y, respectively, are the binary representations of the x and y values in network byte order. There are no internal length markers, so each number representation occupies as many octets as implied by the curve parameters. For P-256, this means that each of X and Y use 32 octets, padded on the left by zeros if necessary. For P-384, they take 48 octets each. For P-521, they take 66 octets each.

For the curves secp256r1, secp384r1, and secp521r1, peers MUST validate each other's public value Q by ensuring that the point is a valid point on the elliptic curve. The appropriate validation procedures are defined in Section 4.3.7 of [ECDSA] and alternatively in Section 5.6.2.3 of [KEYAGREEMENT]. This process consists of three steps: (1) verify that Q is not the point at infinity (O), (2) verify that for Q = (x, y) both integers x and y are in the correct interval, and (3) ensure that (x, y) is a correct solution to the elliptic curve equation. For these curves, implementors do not need to verify membership in the correct subgroup.

For X25519 and X448, the contents of the public value is the K_A or K_B value described in Section 6 of [RFC7748]. This is 32 bytes for X25519 and 56 bytes for X448.

Note: Versions of TLS prior to 1.3 permitted point format negotiation; TLS 1.3 removes this feature in favor of a single point format for each curve.

4.2.9. Pre-Shared Key Exchange Modes

In order to use PSKs, clients MUST also send a "psk_key_exchange_modes" extension. The semantics of this extension are that the client only supports the use of PSKs with these modes, which restricts both the use of PSKs offered in this ClientHello and those which the server might supply via NewSessionTicket.

A client MUST provide a "psk_key_exchange_modes" extension if it offers a "pre_shared_key" extension. If clients offer "pre_shared_key" without a "psk_key_exchange_modes" extension, servers MUST abort the handshake. Servers MUST NOT select a key exchange mode that is not listed by the client. This extension also restricts the modes for use with PSK resumption. Servers SHOULD NOT send NewSessionTicket with tickets that are not compatible with the advertised modes; however, if a server does so, the impact will just be that the client's attempts at resumption fail.

The server MUST NOT send a "psk_key_exchange_modes" extension.

```
enum { psk_ke(0), psk_dhe_ke(1), (255) } PskKeyExchangeMode;
```

```
struct {  
    PskKeyExchangeMode ke_modes<1..255>;  
} PskKeyExchangeModes;
```

psk_ke: PSK-only key establishment. In this mode, the server MUST NOT supply a "key_share" value.

psk_dhe_ke: PSK with (EC)DHE key establishment. In this mode, the client and server MUST supply "key_share" values as described in Section 4.2.8.

Any future values that are allocated must ensure that the transmitted protocol messages unambiguously identify which mode was selected by the server; at present, this is indicated by the presence of the "key_share" in the ServerHello.

4.2.10. Early Data Indication

When a PSK is used and early data is allowed for that PSK (see for instance Appendix B.3.4), the client can send Application Data in its first flight of messages. If the client opts to do so, it MUST supply both the "pre_shared_key" and "early_data" extensions.

The "extension_data" field of this extension contains an "EarlyDataIndication" value.

```
struct {} Empty;
```

```
struct {  
    select (Handshake.msg_type) {  
        case new_session_ticket:    uint32 max_early_data_size;  
        case client_hello:          Empty;  
        case encrypted_extensions: Empty;  
    };  
} EarlyDataIndication;
```

See Section 4.6.1 for details regarding the use of the max_early_data_size field.

The parameters for the 0-RTT data (version, symmetric cipher suite, Application-Layer Protocol Negotiation (ALPN) [RFC7301] protocol, etc.) are those associated with the PSK in use. For externally provisioned PSKs, the associated values are those provisioned along with the key. For PSKs established via a NewSessionTicket message, the associated values are those which were negotiated in the connection which established the PSK. The PSK used to encrypt the early data MUST be the first PSK listed in the client's "pre_shared_key" extension.

For PSKs provisioned via NewSessionTicket, a server MUST validate that the ticket age for the selected PSK identity (computed by subtracting `ticket_age_add` from `PskIdentity.obfuscated_ticket_age` modulo 2^{32}) is within a small tolerance of the time since the ticket was issued (see Section 8). If it is not, the server SHOULD proceed with the handshake but reject 0-RTT, and SHOULD NOT take any other action that assumes that this ClientHello is fresh.

0-RTT messages sent in the first flight have the same (encrypted) content types as messages of the same type sent in other flights (handshake and `application_data`) but are protected under different keys. After receiving the server's Finished message, if the server has accepted early data, an EndOfEarlyData message will be sent to indicate the key change. This message will be encrypted with the 0-RTT traffic keys.

A server which receives an "early_data" extension MUST behave in one of three ways:

- * Ignore the extension and return a regular 1-RTT response. The server then skips past early data by attempting to deprotect received records using the handshake traffic key, discarding records which fail deprotection (up to the configured `max_early_data_size`). Once a record is deprotected successfully, it is treated as the start of the client's second flight and the server proceeds as with an ordinary 1-RTT handshake.
- * Request that the client send another ClientHello by responding with a HelloRetryRequest. A client MUST NOT include the "early_data" extension in its followup ClientHello. The server then ignores early data by skipping all records with an external content type of "application_data" (indicating that they are encrypted), up to the configured `max_early_data_size`.

- * Return its own "early_data" extension in EncryptedExtensions, indicating that it intends to process the early data. It is not possible for the server to accept only a subset of the early data messages. Even though the server sends a message accepting early data, the actual early data itself may already be in flight by the time the server generates this message.

In order to accept early data, the server MUST have selected the first key offered in the client's "pre_shared_key" extension. In addition, it MUST verify that the following values are the same as those associated with the selected PSK:

- * The selected TLS version number
- * The selected cipher suite
- * The selected ALPN [RFC7301] protocol, if any

These requirements are a superset of those needed to perform a 1-RTT handshake using the PSK in question.

Future extensions MUST define their interaction with 0-RTT.

If any of these checks fail, the server MUST NOT respond with the extension and must discard all the first-flight data using one of the first two mechanisms listed above (thus falling back to 1-RTT or 2-RTT). If the client attempts a 0-RTT handshake but the server rejects it, the server will generally not have the 0-RTT record protection keys and must instead use trial decryption (either with the 1-RTT handshake keys or by looking for a cleartext ClientHello in the case of a HelloRetryRequest) to find the first non-0-RTT message.

If the server chooses to accept the "early_data" extension, then it MUST comply with the same error-handling requirements specified for all records when processing early data records. Specifically, if the server fails to decrypt a 0-RTT record following an accepted "early_data" extension, it MUST terminate the connection with a "bad_record_mac" alert as per Section 5.2.

If the server rejects the "early_data" extension, the client application MAY opt to retransmit the Application Data previously sent in early data once the handshake has been completed. Note that automatic retransmission of early data could result in incorrect assumptions regarding the status of the connection. For instance, when the negotiated connection selects a different ALPN protocol from what was used for the early data, an application might need to construct different messages. Similarly, if early data assumes anything about the connection state, it might be sent in error after the handshake completes.

A TLS implementation SHOULD NOT automatically resend early data; applications are in a better position to decide when retransmission is appropriate. A TLS implementation MUST NOT automatically resend early data unless the negotiated connection selects the same ALPN protocol.

4.2.11. Pre-Shared Key Extension

The "pre_shared_key" extension is used to negotiate the identity of the pre-shared key to be used with a given handshake in association with PSK key establishment.

The "extension_data" field of this extension contains a "PreSharedKeyExtension" value:

```
struct {
    opaque identity<1..2^16-1>;
    uint32 obfuscated_ticket_age;
} PskIdentity;

opaque PskBinderEntry<32..255>;

struct {
    PskIdentity identities<7..2^16-1>;
    PskBinderEntry binders<33..2^16-1>;
} OfferedPsks;

struct {
    select (Handshake.msg_type) {
        case client_hello: OfferedPsks;
        case server_hello: uint16 selected_identity;
    };
} PreSharedKeyExtension;
```

identity: A label for a key. For instance, a ticket (as defined in Appendix B.3.4) or a label for a pre-shared key established externally.

obfuscated_ticket_age: An obfuscated version of the age of the key. Section 4.2.11.1 describes how to form this value for identities established via the NewSessionTicket message. For identities established externally, an **obfuscated_ticket_age** of 0 SHOULD be used, and servers MUST ignore the value.

identities: A list of the identities that the client is willing to negotiate with the server. If sent alongside the "early_data" extension (see Section 4.2.10), the first identity is the one used for 0-RTT data.

binders: A series of HMAC values, one for each value in the identities list and in the same order, computed as described below.

selected_identity: The server's chosen identity expressed as a (0-based) index into the identities in the client's "OfferedPsk.identities" list.

Each PSK is associated with a single Hash algorithm. For PSKs established via the ticket mechanism (Section 4.6.1), this is the KDF Hash algorithm on the connection where the ticket was established. For externally established PSKs, the Hash algorithm MUST be set when the PSK is established or default to SHA-256 if no such algorithm is defined. The server MUST ensure that it selects a compatible PSK (if any) and cipher suite.

In TLS versions prior to TLS 1.3, the Server Name Indication (SNI) value was intended to be associated with the session (Section 3 of [RFC6066]), with the server being required to enforce that the SNI value associated with the session matches the one specified in the resumption handshake. However, in reality the implementations were not consistent on which of two supplied SNI values they would use, leading to the consistency requirement being de facto enforced by the clients. In TLS 1.3, the SNI value is always explicitly specified in the resumption handshake, and there is no need for the server to associate an SNI value with the ticket. Clients, however, SHOULD store the SNI with the PSK to fulfill the requirements of Section 4.6.1.

Implementor's note: When session resumption is the primary use case of PSKs, the most straightforward way to implement the PSK/cipher suite matching requirements is to negotiate the cipher suite first and then exclude any incompatible PSKs. Any unknown PSKs (e.g., ones not in the PSK database or encrypted with an unknown key) SHOULD simply be ignored. If no acceptable PSKs are found, the server SHOULD perform a non-PSK handshake if possible. If backward compatibility is important, client-provided, externally established PSKs SHOULD influence cipher suite selection.

Prior to accepting PSK key establishment, the server MUST validate the corresponding binder value (see Section 4.2.11.2 below). If this value is not present or does not validate, the server MUST abort the handshake. Servers SHOULD NOT attempt to validate multiple binders; rather, they SHOULD select a single PSK and validate solely the binder that corresponds to that PSK. See Section 8.2 and Appendix F.6 for the security rationale for this requirement. In order to accept PSK key establishment, the server sends a "pre_shared_key" extension indicating the selected identity.

Clients MUST verify that the server's selected_identity is within the range supplied by the client, that the server selected a cipher suite indicating a Hash associated with the PSK, and that a server "key_share" extension is present if required by the ClientHello "psk_key_exchange_modes" extension. If these values are not consistent, the client MUST abort the handshake with an "illegal_parameter" alert.

If the server supplies an "early_data" extension, the client MUST verify that the server's selected_identity is 0. If any other value is returned, the client MUST abort the handshake with an "illegal_parameter" alert.

The "pre_shared_key" extension MUST be the last extension in the ClientHello (this facilitates implementation as described below). Servers MUST check that it is the last extension and otherwise fail the handshake with an "illegal_parameter" alert.

4.2.11.1. Ticket Age

The client's view of the age of a ticket is the time since the receipt of the NewSessionTicket message. Clients MUST NOT attempt to use tickets which have ages greater than the "ticket_lifetime" value which was provided with the ticket. The "obfuscated_ticket_age" field of each PskIdentity contains an obfuscated version of the ticket age formed by taking the age in milliseconds and adding the "ticket_age_add" value that was included with the ticket (see Section 4.6.1), modulo 2^{32} . This addition prevents passive

observers from correlating connections unless tickets are reused. Note that the "ticket_lifetime" field in the NewSessionTicket message is in seconds but the "obfuscated_ticket_age" is in milliseconds. Because ticket lifetimes are restricted to a week, 32 bits is enough to represent any plausible age, even in milliseconds.

4.2.11.2. PSK Binder

The PSK binder value forms a binding between a PSK and the current handshake, as well as a binding between the handshake in which the PSK was generated (if via a NewSessionTicket message) and the current handshake. Each entry in the binders list is computed as an HMAC over a transcript hash (see Section 4.4.1) containing a partial ClientHello up to and including the PreSharedKeyExtension.identities field. That is, it includes all of the ClientHello but not the binders list itself. The length fields for the message (including the overall length, the length of the extensions block, and the length of the "pre_shared_key" extension) are all set as if binders of the correct lengths were present.

The PskBinderEntry is computed in the same way as the Finished message (Section 4.4.4) but with the BaseKey being the binder_key derived via the key schedule from the corresponding PSK which is being offered (see Section 7.1).

If the handshake includes a HelloRetryRequest, the initial ClientHello and HelloRetryRequest are included in the transcript along with the new ClientHello. For instance, if the client sends ClientHello1, its binder will be computed over:

```
Transcript-Hash(Truncate(ClientHello1))
```

Where Truncate() removes the binders list from the ClientHello.

If the server responds with a HelloRetryRequest and the client then sends ClientHello2, its binder will be computed over:

```
Transcript-Hash(ClientHello1,  
                  HelloRetryRequest,  
                  Truncate(ClientHello2))
```

The full ClientHello1/ClientHello2 is included in all other handshake hash computations. Note that in the first flight, Truncate(ClientHello1) is hashed directly, but in the second flight, ClientHello1 is hashed and then reinjected as a "message_hash" message, as described in Section 4.4.1.

4.2.11.3. Processing Order

Clients are permitted to "stream" 0-RTT data until they receive the server's Finished, only then sending the EndOfEarlyData message, followed by the rest of the handshake. In order to avoid deadlocks, when accepting "early_data", servers MUST process the client's ClientHello and then immediately send their flight of messages, rather than waiting for the client's EndOfEarlyData message before sending its ServerHello.

4.3. Server Parameters

The next two messages from the server, EncryptedExtensions and CertificateRequest, contain information from the server that determines the rest of the handshake. These messages are encrypted with keys derived from the server_handshake_traffic_secret.

4.3.1. Encrypted Extensions

In all handshakes, the server MUST send the EncryptedExtensions message immediately after the ServerHello message. This is the first message that is encrypted under keys derived from the server_handshake_traffic_secret.

The EncryptedExtensions message contains extensions that can be protected, i.e., any which are not needed to establish the cryptographic context but which are not associated with individual certificates. The client MUST check EncryptedExtensions for the presence of any forbidden extensions and if any are found MUST abort the handshake with an "illegal_parameter" alert.

Structure of this message:

```
struct {  
    Extension extensions<0..2^16-1>;  
} EncryptedExtensions;
```

extensions: A list of extensions. For more information, see the table in Section 4.2.

4.3.2. Certificate Request

A server which is authenticating with a certificate MAY optionally request a certificate from the client. This message, if sent, MUST follow EncryptedExtensions.

Structure of this message:

```
struct {  
    opaque certificate_request_context<0..2^8-1>;  
    Extension extensions<0..2^16-1>;  
} CertificateRequest;
```

certificate_request_context: An opaque string which identifies the certificate request and which will be echoed in the client's Certificate message. The `certificate_request_context` MUST be unique within the scope of this connection (thus preventing replay of client CertificateVerify messages). This field SHALL be zero length unless used for the post-handshake authentication exchanges described in Section 4.6.2. When requesting post-handshake authentication, the server SHOULD make the context unpredictable to the client (e.g., by randomly generating it) in order to prevent an attacker who has temporary access to the client's private key from pre-computing valid CertificateVerify messages.

extensions: A list of extensions describing the parameters of the certificate being requested. The "signature_algorithms" extension MUST be specified, and other extensions may optionally be included if defined for this message. Clients MUST ignore unrecognized extensions.

In prior versions of TLS, the CertificateRequest message carried a list of signature algorithms and certificate authorities which the server would accept. In TLS 1.3, the former is expressed by sending the "signature_algorithms" and optionally "signature_algorithms_cert" extensions. The latter is expressed by sending the "certificate_authorities" extension (see Section 4.2.4).

Servers which are authenticating with a resumption PSK MUST NOT send the CertificateRequest message in the main handshake, though they MAY send it in post-handshake authentication (see Section 4.6.2) provided that the client has sent the "post_handshake_auth" extension (see Section 4.2.6). Servers which are authenticating with an external PSK MUST NOT send the CertificateRequest message either in the main handshake or request post-handshake authentication. Future specifications MAY provide an extension to permit this.

4.4. Authentication Messages

As discussed in Section 2, TLS generally uses a common set of messages for authentication, key confirmation, and handshake integrity: Certificate, CertificateVerify, and Finished. (The PSK binders also perform key confirmation, in a similar fashion.) These three messages are always sent as the last messages in their handshake flight. The Certificate and CertificateVerify messages are only sent under certain circumstances, as defined below. The

Finished message is always sent as part of the Authentication Block. These messages are encrypted under keys derived from the [sender]_handshake_traffic_secret.

The computations for the Authentication messages all uniformly take the following inputs:

- * The certificate and signing key to be used.
- * A Handshake Context consisting of the list of messages to be included in the transcript hash.
- * A Base Key to be used to compute a MAC key.

Based on these inputs, the messages then contain:

Certificate The certificate to be used for authentication, and any supporting certificates in the chain. Note that certificate-based client authentication is not available in PSK handshake flows (including 0-RTT).

CertificateVerify: A signature over the value Transcript-Hash(Handshake Context, Certificate)

Finished: A MAC over the value Transcript-Hash(Handshake Context, Certificate, CertificateVerify) using a MAC key derived from the Base Key.

The following table defines the Handshake Context and MAC Base Key for each scenario:

Mode	Handshake Context	Base Key
Server	ClientHello ... later of EncryptedExtensions/ CertificateRequest	server_handshake_traffic_secret
Client	ClientHello ... later of server Finished/ EndOfEarlyData	client_handshake_traffic_secret
Post- Handshake	ClientHello ... client Finished + CertificateRequest	client_application_traffic_secret_N

Table 2: Authentication Inputs

4.4.1. The Transcript Hash

Many of the cryptographic computations in TLS make use of a transcript hash. This value is computed by hashing the concatenation of each included handshake message, including the handshake message header carrying the handshake message type and length fields, but not including record layer headers. I.e.,

$$\text{Transcript-Hash}(M_1, M_2, \dots, M_n) = \text{Hash}(M_1 \parallel M_2 \parallel \dots \parallel M_n)$$

As an exception to this general rule, when the server responds to a ClientHello with a HelloRetryRequest, the value of ClientHello1 is replaced with a special synthetic handshake message of handshake type "message_hash" containing Hash(ClientHello1). I.e.,

```
Transcript-Hash(ClientHello1, HelloRetryRequest, ... Mn) =
  Hash(message_hash ||           /* Handshake type */
        00 00 Hash.length ||     /* Handshake message length (bytes) */
        Hash(ClientHello1) ||    /* Hash of ClientHello1 */
        HelloRetryRequest || ... || Mn)
```

The reason for this construction is to allow the server to do a stateless HelloRetryRequest by storing just the hash of ClientHello1 in the cookie, rather than requiring it to export the entire intermediate hash state (see Section 4.2.2).

For concreteness, the transcript hash is always taken from the following sequence of handshake messages, starting at the first ClientHello and including only those messages that were sent: ClientHello, HelloRetryRequest, ClientHello, ServerHello, EncryptedExtensions, server CertificateRequest, server Certificate, server CertificateVerify, server Finished, EndOfEarlyData, client Certificate, client CertificateVerify, client Finished.

In general, implementations can implement the transcript by keeping a running transcript hash value based on the negotiated hash. Note, however, that subsequent post-handshake authentications do not include each other, just the messages through the end of the main handshake.

4.4.2. Certificate

This message conveys the endpoint's certificate chain to the peer.

The server **MUST** send a Certificate message whenever the agreed-upon key exchange method uses certificates for authentication (this includes all key exchange methods defined in this document except PSK).

The client **MUST** send a Certificate message if and only if the server has requested certificate-based client authentication via a CertificateRequest message (Section 4.3.2). If the server requests certificate-based client authentication but no suitable certificate is available, the client **MUST** send a Certificate message containing no certificates (i.e., with the "certificate_list" field having length 0). A Finished message **MUST** be sent regardless of whether the Certificate message is empty.

Structure of this message:

```
enum {
    X509(0),
    RawPublicKey(2),
    (255)
} CertificateType;

struct {
    select (certificate_type) {
        case RawPublicKey:
            /* From RFC 7250 ASN.1_subjectPublicKeyInfo */
            opaque ASN1_subjectPublicKeyInfo<1..2^24-1>;

        case X509:
            opaque cert_data<1..2^24-1>;
    };
    Extension extensions<0..2^16-1>;
} CertificateEntry;

struct {
    opaque certificate_request_context<0..2^8-1>;
    CertificateEntry certificate_list<0..2^24-1>;
} Certificate;
```

certificate_request_context: If this message is in response to a CertificateRequest, the value of certificate_request_context in that message. Otherwise (in the case of server authentication), this field SHALL be zero length.

certificate_list: A list (chain) of CertificateEntry structures, each containing a single certificate and list of extensions.

extensions: A list of extension values for the CertificateEntry. The "Extension" format is defined in Section 4.2. Valid extensions for server certificates at present include the OCSP Status extension [RFC6066] and the SignedCertificateTimestamp extension [RFC6962]; future extensions may be defined for this message as well. Extensions in the Certificate message from the server MUST correspond to ones from the ClientHello message. Extensions in the Certificate message from the client MUST correspond to extensions in the CertificateRequest message from the server. If an extension applies to the entire chain, it SHOULD be included in the first CertificateEntry.

If the corresponding certificate type extension ("server_certificate_type" or "client_certificate_type") was not negotiated in EncryptedExtensions, or the X.509 certificate type was negotiated, then each CertificateEntry contains a DER-encoded X.509 certificate. The sender's certificate MUST come in the first

CertificateEntry in the list. Each following certificate SHOULD directly certify the one immediately preceding it. Because certificate validation requires that trust anchors be distributed independently, a certificate that specifies a trust anchor MAY be omitted from the chain, provided that supported peers are known to possess any omitted certificates.

Note: Prior to TLS 1.3, "certificate_list" ordering required each certificate to certify the one immediately preceding it; however, some implementations allowed some flexibility. Servers sometimes send both a current and deprecated intermediate for transitional purposes, and others are simply configured incorrectly, but these cases can nonetheless be validated properly. For maximum compatibility, all implementations SHOULD be prepared to handle potentially extraneous certificates and arbitrary orderings from any TLS version, with the exception of the end-entity certificate which MUST be first.

If the RawPublicKey certificate type was negotiated, then the certificate_list MUST contain no more than one CertificateEntry, which contains an ASN1_subjectPublicKeyInfo value as defined in [RFC7250], Section 3.

The OpenPGP certificate type [RFC6091] MUST NOT be used with TLS 1.3.

The server's certificate_list MUST always be non-empty. A client will send an empty certificate_list if it does not have an appropriate certificate to send in response to the server's authentication request.

4.4.2.1. OCSP Status and SCT Extensions

[RFC6066] and [RFC6961] provide extensions to negotiate the server sending OCSP responses to the client. In TLS 1.2 and below, the server replies with an empty extension to indicate negotiation of this extension and the OCSP information is carried in a CertificateStatus message. In TLS 1.3, the server's OCSP information is carried in an extension in the CertificateEntry containing the associated certificate. Specifically, the body of the "status_request" extension from the server MUST be a CertificateStatus structure as defined in [RFC6066], which is interpreted as defined in [RFC6960].

Note: The `status_request_v2` extension [RFC6961] is deprecated. TLS 1.3 servers MUST NOT act upon its presence or information in it when processing `ClientHello` messages; in particular, they MUST NOT send the `status_request_v2` extension in the `EncryptedExtensions`, `CertificateRequest`, or `Certificate` messages. TLS 1.3 servers MUST be able to process `ClientHello` messages that include it, as it MAY be sent by clients that wish to use it in earlier protocol versions.

A server MAY request that a client present an OCSP response with its certificate by sending an empty `"status_request"` extension in its `CertificateRequest` message. If the client opts to send an OCSP response, the body of its `"status_request"` extension MUST be a `CertificateStatus` structure as defined in [RFC6066].

Similarly, [RFC6962] provides a mechanism for a server to send a Signed Certificate Timestamp (SCT) as an extension in the `ServerHello` in TLS 1.2 and below. In TLS 1.3, the server's SCT information is carried in an extension in the `CertificateEntry`.

4.4.2.2. Server Certificate Selection

The following rules apply to the certificates sent by the server:

- * The certificate type MUST be X.509v3 [RFC5280], unless explicitly negotiated otherwise (e.g., [RFC7250]).
- * The end-entity certificate MUST allow the key to be used for signing with a signature scheme indicated in the client's `"signature_algorithms"` extension (see Section 4.2.3). That is, the `digitalSignature` bit MUST be set if the Key Usage extension is present, and the public key (with associated restrictions) MUST be compatible with some supported signature scheme.
- * The `"server_name"` [RFC6066] and `"certificate_authorities"` extensions are used to guide certificate selection. As servers MAY require the presence of the `"server_name"` extension, clients SHOULD send this extension when the server is identified by name.

All certificates provided by the server MUST be signed by a signature algorithm advertised by the client, if it is able to provide such a chain (see Section 4.2.3). Certificates that are self-signed or certificates that are expected to be trust anchors are not validated as part of the chain and therefore MAY be signed with any algorithm.

If the server cannot produce a certificate chain that is signed only via the indicated supported algorithms, then it SHOULD continue the handshake by sending the client a certificate chain of its choice that may include algorithms that are not known to be supported by the

client. This fallback chain SHOULD NOT use the deprecated SHA-1 hash algorithm in general, but MAY do so if the client's advertisement permits it, and MUST NOT do so otherwise.

If the client cannot construct an acceptable chain using the provided certificates and decides to abort the handshake, then it MUST abort the handshake with an appropriate certificate-related alert (by default, "unsupported_certificate"; see Section 6.2 for more information).

If the server has multiple certificates, it chooses one of them based on the above-mentioned criteria (in addition to other criteria, such as transport-layer endpoint, local configuration, and preferences).

4.4.2.3. Client Certificate Selection

The following rules apply to certificates sent by the client:

- * The certificate type MUST be X.509v3 [RFC5280], unless explicitly negotiated otherwise (e.g., [RFC7250]).
- * If the "certificate_authorities" extension in the CertificateRequest message was present, at least one of the certificates in the certificate chain SHOULD be issued by one of the listed CAs.
- * The certificates MUST be signed using an acceptable signature algorithm, as described in Section 4.3.2. Note that this relaxes the constraints on certificate-signing algorithms found in prior versions of TLS.
- * If the CertificateRequest message contained a non-empty "oid_filters" extension, the end-entity certificate MUST match the extension OIDs that are recognized by the client, as described in Section 4.2.5.

4.4.2.4. Receiving a Certificate Message

In general, detailed certificate validation procedures are out of scope for TLS (see [RFC5280]). This section provides TLS-specific requirements.

If the server supplies an empty Certificate message, the client MUST abort the handshake with a "decode_error" alert.

If the client does not send any certificates (i.e., it sends an empty Certificate message), the server MAY at its discretion either continue the handshake without client authentication, or abort the

handshake with a "certificate_required" alert. Also, if some aspect of the certificate chain was unacceptable (e.g., it was not signed by a known, trusted CA), the server MAY at its discretion either continue the handshake (considering the client unauthenticated) or abort the handshake.

Any endpoint receiving any certificate which it would need to validate using any signature algorithm using an MD5 hash MUST abort the handshake with a "bad_certificate" alert. SHA-1 is deprecated and it is RECOMMENDED that any endpoint receiving any certificate which it would need to validate using any signature algorithm using a SHA-1 hash abort the handshake with a "bad_certificate" alert. For clarity, this means that endpoints can accept these algorithms for certificates that are self-signed or are trust anchors.

All endpoints are RECOMMENDED to transition to SHA-256 or better as soon as possible to maintain interoperability with implementations currently in the process of phasing out SHA-1 support.

Note that a certificate containing a key for one signature algorithm MAY be signed using a different signature algorithm (for instance, an RSA key signed with an ECDSA key).

4.4.3. Certificate Verify

This message is used to provide explicit proof that an endpoint possesses the private key corresponding to its certificate. The CertificateVerify message also provides integrity for the handshake up to this point. Servers MUST send this message when authenticating via a certificate. Clients MUST send this message whenever authenticating via a certificate (i.e., when the Certificate message is non-empty). When sent, this message MUST appear immediately after the Certificate message and immediately prior to the Finished message.

Structure of this message:

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

The algorithm field specifies the signature algorithm used (see Section 4.2.3 for the definition of this type). The signature is a digital signature using that algorithm. The content that is covered under the signature is the hash output as described in Section 4.4.1, namely:

Transcript-Hash(Handshake Context, Certificate)

The digital signature is then computed over the concatenation of:

- * A string that consists of octet 32 (0x20) repeated 64 times
- * The context string (defined below)
- * A single 0 byte which serves as the separator
- * The content to be signed

This structure is intended to prevent an attack on previous versions of TLS in which the ServerKeyExchange format meant that attackers could obtain a signature of a message with a chosen 32-byte prefix (ClientHello.random). The initial 64-byte pad clears that prefix along with the server-controlled ServerHello.random.

The context string for a server signature is "TLS 1.3, server CertificateVerify" The context string for a client signature is "TLS 1.3, client CertificateVerify" It is used to provide separation between signatures made in different contexts, helping against potential cross-protocol attacks.

For example, if the transcript hash was 32 bytes of 01 (this length would make sense for SHA-256), the content covered by the digital signature for a server CertificateVerify would be:

```
2020202020202020202020202020202020202020202020202020202020202020
2020202020202020202020202020202020202020202020202020202020202020
544c5320312e332c207365727665722043657274696669636174655665726966
79
00
0101010101010101010101010101010101010101010101010101010101010101
```

On the sender side, the process for computing the signature field of the CertificateVerify message takes as input:

- * The content covered by the digital signature
- * The private signing key corresponding to the certificate sent in the previous message

If the CertificateVerify message is sent by a server, the signature algorithm MUST be one offered in the client's "signature_algorithms" extension unless no valid certificate chain can be produced without unsupported algorithms (see Section 4.2.3).

If sent by a client, the signature algorithm used in the signature MUST be one of those present in the `supported_signature_algorithms` field of the `"signature_algorithms"` extension in the `CertificateRequest` message.

In addition, the signature algorithm MUST be compatible with the key in the sender's end-entity certificate. RSA signatures MUST use an RSASSA-PSS algorithm, regardless of whether RSASSA-PKCS1-v1_5 algorithms appear in `"signature_algorithms"`. The SHA-1 algorithm MUST NOT be used in any signatures of `CertificateVerify` messages. All SHA-1 signature algorithms in this specification are defined solely for use in legacy certificates and are not valid for `CertificateVerify` signatures.

The receiver of a `CertificateVerify` message MUST verify the signature field. The verification process takes as input:

- * The content covered by the digital signature
- * The public key contained in the end-entity certificate found in the associated `Certificate` message
- * The digital signature received in the signature field of the `CertificateVerify` message

If the verification fails, the receiver MUST terminate the handshake with a `"decrypt_error"` alert.

4.4.4. Finished

The `Finished` message is the final message in the Authentication Block. It is essential for providing authentication of the handshake and of the computed keys.

Recipients of `Finished` messages MUST verify that the contents are correct and if incorrect MUST terminate the connection with a `"decrypt_error"` alert.

Once a side has sent its `Finished` message and has received and validated the `Finished` message from its peer, it may begin to send and receive Application Data over the connection. There are two settings in which it is permitted to send data prior to receiving the peer's `Finished`:

1. Clients sending 0-RTT data as described in Section 4.2.10.

2. Servers MAY send data after sending their first flight, but because the handshake is not yet complete, they have no assurance of either the peer's identity or its liveness (i.e., the ClientHello might have been replayed).

The key used to compute the Finished message is computed from the Base Key defined in Section 4.4 using HKDF (see Section 7.1). Specifically:

```
finished_key =  
    HKDF-Expand-Label(BaseKey, "finished", "", Hash.length)
```

Structure of this message:

```
struct {  
    opaque verify_data[Hash.length];  
} Finished;
```

The verify_data value is computed as follows:

```
verify_data =  
    HMAC(finished_key,  
        Transcript-Hash(Handshake Context,  
                        Certificate*, CertificateVerify*))
```

* Only included if present.

HMAC [RFC2104] uses the Hash algorithm for the handshake. As noted above, the HMAC input can generally be implemented by a running hash, i.e., just the handshake hash at this point.

In previous versions of TLS, the verify_data was always 12 octets long. In TLS 1.3, it is the size of the HMAC output for the Hash used for the handshake.

Note: Alerts and any other non-handshake record types are not handshake messages and are not included in the hash computations.

Any records following a Finished message MUST be encrypted under the appropriate application traffic key as described in Section 7.2. In particular, this includes any alerts sent by the server in response to client Certificate and CertificateVerify messages.

4.5. End of Early Data

```
struct {} EndOfEarlyData;
```

If the server sent an "early_data" extension in EncryptedExtensions, the client MUST send an EndOfEarlyData message after receiving the server Finished. If the server does not send an "early_data" extension in EncryptedExtensions, then the client MUST NOT send an EndOfEarlyData message. This message indicates that all 0-RTT application_data messages, if any, have been transmitted and that the following records are protected under handshake traffic keys. Servers MUST NOT send this message, and clients receiving it MUST terminate the connection with an "unexpected_message" alert. This message is encrypted under keys derived from the client_early_traffic_secret.

4.6. Post-Handshake Messages

TLS also allows other messages to be sent after the main handshake. These messages use a handshake content type and are encrypted under the appropriate application traffic key.

4.6.1. New Session Ticket Message

At any time after the server has received the client Finished message, it MAY send a NewSessionTicket message. This message creates a unique association between the ticket value and a secret PSK derived from the resumption secret (see Section 7).

The client MAY use this PSK for future handshakes by including the ticket value in the "pre_shared_key" extension in its ClientHello (Section 4.2.11). Resumption MAY be done while the original connection is still open. Servers MAY send multiple tickets on a single connection, either immediately after each other or after specific events (see Appendix C.4). For instance, the server might send a new ticket after post-handshake authentication in order to encapsulate the additional client authentication state. Multiple tickets are useful for clients for a variety of purposes, including:

- * Opening multiple parallel HTTP connections.
- * Performing connection racing across interfaces and address families via (for example) Happy Eyeballs [RFC8305] or related techniques.

Any ticket MUST only be resumed with a cipher suite that has the same KDF hash algorithm as that used to establish the original connection.

Clients MUST only resume if the new SNI value is valid for the server certificate presented in the original session, and SHOULD only resume if the SNI value matches the one used in the original session. The latter is a performance optimization: normally, there is no reason to

expect that different servers covered by a single certificate would be able to accept each other's tickets; hence, attempting resumption in that case would waste a single-use ticket. If such an indication is provided (externally or by any other means), clients MAY resume with a different SNI value.

On resumption, if reporting an SNI value to the calling application, implementations MUST use the value sent in the resumption ClientHello rather than the value sent in the previous session. Note that if a server implementation declines all PSK identities with different SNI values, these two values are always the same.

Note: Although the resumption secret depends on the client's second flight, a server which does not request certificate-based client authentication MAY compute the remainder of the transcript independently and then send a NewSessionTicket immediately upon sending its Finished rather than waiting for the client Finished. This might be appropriate in cases where the client is expected to open multiple TLS connections in parallel and would benefit from the reduced overhead of a resumption handshake, for example.

```
struct {
    uint32 ticket_lifetime;
    uint32 ticket_age_add;
    opaque ticket_nonce<0..255>;
    opaque ticket<1..2^16-1>;
    Extension extensions<0..2^16-1>;
} NewSessionTicket;
```

ticket_lifetime: Indicates the lifetime in seconds as a 32-bit unsigned integer in network byte order from the time of ticket issuance. Servers MUST NOT use any value greater than 604800 seconds (7 days). The value of zero indicates that the ticket should be discarded immediately. Clients MUST NOT use tickets for longer than 7 days after issuance, regardless of the ticket_lifetime, and MAY delete tickets earlier based on local policy. A server MAY treat a ticket as valid for a shorter period of time than what is stated in the ticket_lifetime.

ticket_age_add: A securely generated, random 32-bit value that is used to obscure the age of the ticket that the client includes in the "pre_shared_key" extension. The client-side ticket age is added to this value modulo 2^{32} to obtain the value that is transmitted by the client. The server MUST generate a fresh value for each ticket it sends.

ticket_nonce: A per-ticket value that is unique across all tickets issued on this connection.

ticket: The value of the ticket to be used as the PSK identity. The ticket itself is an opaque label. It MAY be either a database lookup key or a self-encrypted and self-authenticated value.

extensions: A list of extension values for the ticket. The "Extension" format is defined in Section 4.2. Clients MUST ignore unrecognized extensions.

The sole extension currently defined for NewSessionTicket is "early_data", indicating that the ticket may be used to send 0-RTT data (Section 4.2.10). It contains the following value:

max_early_data_size: The maximum amount of 0-RTT data that the client is allowed to send when using this ticket, in bytes. Only Application Data payload (i.e., plaintext but not padding or the inner content type byte) is counted. A server receiving more than max_early_data_size bytes of 0-RTT data SHOULD terminate the connection with an "unexpected_message" alert. Note that servers that reject early data due to lack of cryptographic material will be unable to differentiate padding from content, so clients SHOULD NOT depend on being able to send large quantities of padding in early data records.

The PSK associated with the ticket is computed as:

```
HKDF-Expand-Label(resumption_secret,  
                  "resumption", ticket_nonce, Hash.length)
```

Because the ticket_nonce value is distinct for each NewSessionTicket message, a different PSK will be derived for each ticket.

Note that in principle it is possible to continue issuing new tickets which indefinitely extend the lifetime of the keying material originally derived from an initial non-PSK handshake (which was most likely tied to the peer's certificate). It is RECOMMENDED that implementations place limits on the total lifetime of such keying material; these limits should take into account the lifetime of the peer's certificate, the likelihood of intervening revocation, and the time since the peer's online CertificateVerify signature.

4.6.2. Post-Handshake Authentication

When the client has sent the "post_handshake_auth" extension (see Section 4.2.6), a server MAY request certificate-based client authentication at any time after the handshake has completed by sending a CertificateRequest message. The client MUST respond with the appropriate Authentication messages (see Section 4.4). If the client chooses to authenticate, it MUST send Certificate,

CertificateVerify, and Finished. If it declines, it MUST send a Certificate message containing no certificates followed by Finished. All of the client's messages for a given response MUST appear consecutively on the wire with no intervening messages of other types.

A client that receives a CertificateRequest message without having sent the "post_handshake_auth" extension MUST send an "unexpected_message" fatal alert.

Note: Because certificate-based client authentication could involve prompting the user, servers MUST be prepared for some delay, including receiving an arbitrary number of other messages between sending the CertificateRequest and receiving a response. In addition, clients which receive multiple CertificateRequests in close succession MAY respond to them in a different order than they were received (the certificate_request_context value allows the server to disambiguate the responses).

4.6.3. Key and Initialization Vector Update

The KeyUpdate handshake message is used to indicate that the sender is updating its sending cryptographic keys. This message can be sent by either peer after it has sent a Finished message. Implementations that receive a KeyUpdate message prior to receiving a Finished message MUST terminate the connection with an "unexpected_message" alert. After sending a KeyUpdate message, the sender SHALL send all its traffic using the next generation of keys, computed as described in Section 7.2. Upon receiving a KeyUpdate, the receiver MUST update its receiving keys.

```
enum {
    update_not_requested(0), update_requested(1), (255)
} KeyUpdateRequest;

struct {
    KeyUpdateRequest request_update;
} KeyUpdate;
```

request_update: Indicates whether the recipient of the KeyUpdate should respond with its own KeyUpdate. If an implementation receives any other value, it MUST terminate the connection with an "illegal_parameter" alert.

If the request_update field is set to "update_requested", then the receiver MUST send a KeyUpdate of its own with request_update set to "update_not_requested" prior to sending its next Application Data record. This mechanism allows either side to force an update to the

entire connection, but causes an implementation which receives multiple KeyUpdates while it is silent to respond with a single update. Note that implementations may receive an arbitrary number of messages between sending a KeyUpdate with `request_update` set to "update_requested" and receiving the peer's KeyUpdate, because those messages may already be in flight. However, because send and receive keys are derived from independent traffic secrets, retaining the receive traffic secret does not threaten the forward secrecy of data sent before the sender changed keys.

If implementations independently send their own KeyUpdates with `request_update` set to "update_requested", and they cross in flight, then each side will also send a response, with the result that each side increments by two generations.

Both sender and receiver MUST encrypt their KeyUpdate messages with the old keys. Additionally, both sides MUST enforce that a KeyUpdate with the old key is received before accepting any messages encrypted with the new key. Failure to do so may allow message truncation attacks.

5. Record Protocol

The TLS record protocol takes messages to be transmitted, fragments the data into manageable blocks, protects the records, and transmits the result. Received data is verified, decrypted, reassembled, and then delivered to higher-level clients.

TLS records are typed, which allows multiple higher-level protocols to be multiplexed over the same record layer. This document specifies four content types: `handshake`, `application_data`, `alert`, and `change_cipher_spec`. The `change_cipher_spec` record is used only for compatibility purposes (see Appendix E.4).

An implementation may receive an unencrypted record of type `change_cipher_spec` consisting of the single byte value `0x01` at any time after the first `ClientHello` message has been sent or received and before the peer's `Finished` message has been received and MUST simply drop it without further processing. Note that this record may appear at a point at the handshake where the implementation is expecting protected records, and so it is necessary to detect this condition prior to attempting to deprotect the record. An implementation which receives any other `change_cipher_spec` value or which receives a protected `change_cipher_spec` record MUST abort the handshake with an `"unexpected_message"` alert. If an implementation detects a `change_cipher_spec` record received before the first `ClientHello` message or after the peer's `Finished` message, it MUST be treated as an unexpected record type (though stateless servers may not be able to distinguish these cases from allowed cases).

Implementations MUST NOT send record types not defined in this document unless negotiated by some extension. If a TLS implementation receives an unexpected record type, it MUST terminate the connection with an `"unexpected_message"` alert. New record content type values are assigned by IANA in the TLS `ContentType` registry as described in Section 11.

5.1. Record Layer

The record layer fragments information blocks into `TLSPlaintext` records carrying data in chunks of 2^{14} bytes or less. Message boundaries are handled differently depending on the underlying `ContentType`. Any future content types MUST specify appropriate rules. Note that these rules are stricter than what was enforced in TLS 1.2.

Handshake messages MAY be coalesced into a single `TLSPlaintext` record or fragmented across several records, provided that:

- * Handshake messages MUST NOT be interleaved with other record types. That is, if a handshake message is split over two or more records, there MUST NOT be any other records between them.
- * Handshake messages MUST NOT span key changes. Implementations MUST verify that all messages immediately preceding a key change align with a record boundary; if not, then they MUST terminate the connection with an `"unexpected_message"` alert. Because the `ClientHello`, `EndOfEarlyData`, `ServerHello`, `Finished`, and `KeyUpdate` messages can immediately precede a key change, implementations MUST send these messages in alignment with a record boundary.

Implementations MUST NOT send zero-length fragments of Handshake types, even if those fragments contain padding.

Alert messages (Section 6) MUST NOT be fragmented across records, and multiple alert messages MUST NOT be coalesced into a single TLSPlaintext record. In other words, a record with an Alert type MUST contain exactly one message.

Application Data messages contain data that is opaque to TLS. Application Data messages are always protected. Zero-length fragments of Application Data MAY be sent, as they are potentially useful as a traffic analysis countermeasure. Application Data fragments MAY be split across multiple records or coalesced into a single record.

```
enum {
    invalid(0),
    change_cipher_spec(20),
    alert(21),
    handshake(22),
    application_data(23),
    (255)
} ContentType;

struct {
    ContentType type;
    ProtocolVersion legacy_record_version;
    uint16 length;
    opaque fragment[TLSPlaintext.length];
} TLSPlaintext;
```

type: The higher-level protocol used to process the enclosed fragment.

legacy_record_version: MUST be set to 0x0303 for all records generated by a TLS 1.3 implementation other than an initial ClientHello (i.e., one not generated after a HelloRetryRequest), where it MAY also be 0x0301 for compatibility purposes. This field is deprecated and MUST be ignored for all purposes. Previous versions of TLS would use other values in this field under some circumstances.

length: The length (in bytes) of the following TLSPlaintext.fragment. The length MUST NOT exceed 2^{14} bytes. An endpoint that receives a record that exceeds this length MUST terminate the connection with a "record_overflow" alert.

fragment The data being transmitted. This value is transparent and

is treated as an independent block to be dealt with by the higher-level protocol specified by the type field.

This document describes TLS 1.3, which uses the version 0x0304. This version value is historical, deriving from the use of 0x0301 for TLS 1.0 and 0x0300 for SSL 3.0. In order to maximize backward compatibility, a record containing an initial ClientHello SHOULD have version 0x0301 (reflecting TLS 1.0) and a record containing a second ClientHello or a ServerHello MUST have version 0x0303 (reflecting TLS 1.2). When negotiating prior versions of TLS, endpoints follow the procedure and requirements provided in Appendix E.

When record protection has not yet been engaged, TLSPlaintext structures are written directly onto the wire. Once record protection has started, TLSPlaintext records are protected and sent as described in the following section. Note that Application Data records MUST NOT be written to the wire unprotected (see Section 2 for details).

5.2. Record Payload Protection

The record protection functions translate a TLSPlaintext structure into a TLSCiphertext structure. The deprotection functions reverse the process. In TLS 1.3, as opposed to previous versions of TLS, all ciphers are modeled as "Authenticated Encryption with Associated Data" (AEAD) [RFC5116]. AEAD functions provide a unified encryption and authentication operation which turns plaintext into authenticated ciphertext and back again. Each encrypted record consists of a plaintext header followed by an encrypted body, which itself contains a type and optional padding.

```
struct {
    opaque content[TLSPlaintext.length];
    ContentType type;
    uint8 zeros[length_of_padding];
} TLSInnerPlaintext;

struct {
    ContentType opaque_type = application_data; /* 23 */
    ProtocolVersion legacy_record_version = 0x0303; /* TLS v1.2 */
    uint16 length;
    opaque encrypted_record[TLSCiphertext.length];
} TLSCiphertext;
```

content: The TLSPlaintext.fragment value, containing the byte encoding of a handshake or an alert message, or the raw bytes of the application's data to send.

`type`: The `TLSP Plaintext.type` value containing the content type of the record.

`zeros`: An arbitrary-length run of zero-valued bytes may appear in the cleartext after the type field. This provides an opportunity for senders to pad any TLS record by a chosen amount as long as the total stays within record size limits. See Section 5.4 for more details.

`opaque_type`: The outer `opaque_type` field of a `TLSCiphertext` record is always set to the value 23 (`application_data`) for outward compatibility with middleboxes accustomed to parsing previous versions of TLS. The actual content type of the record is found in `TLSP InnerPlaintext.type` after decryption.

`legacy_record_version`: The `legacy_record_version` field is always 0x0303. TLS 1.3 `TLSCiphertexts` are not generated until after TLS 1.3 has been negotiated, so there are no historical compatibility concerns where other values might be received. Note that the handshake protocol, including the `ClientHello` and `ServerHello` messages, authenticates the protocol version, so this value is redundant.

`length`: The length (in bytes) of the following `TLSCiphertext.encrypted_record`, which is the sum of the lengths of the content and the padding, plus one for the inner content type, plus any expansion added by the AEAD algorithm. The length MUST NOT exceed $2^{14} + 256$ bytes. An endpoint that receives a record that exceeds this length MUST terminate the connection with a "record_overflow" alert.

`encrypted_record`: The AEAD-encrypted form of the serialized `TLSP InnerPlaintext` structure.

AEAD algorithms take as input a single key, a nonce, a plaintext, and "additional data" to be included in the authentication check, as described in Section 2.1 of [RFC5116]. The key is either the `client_write_key` or the `server_write_key`, the nonce is derived from the sequence number and the `client_write_iv` or `server_write_iv` (see Section 5.3), and the additional data input is the record header. I.e.,

```
additional_data = TLSCiphertext.opaque_type ||
                  TLSCiphertext.legacy_record_version ||
                  TLSCiphertext.length
```

The plaintext input to the AEAD algorithm is the encoded `TLSInnerPlaintext` structure. Derivation of traffic keys is defined in Section 7.3.

The AEAD output consists of the ciphertext output from the AEAD encryption operation. The length of the plaintext is greater than the corresponding `TLSPlaintext.length` due to the inclusion of `TLSInnerPlaintext.type` and any padding supplied by the sender. The length of the AEAD output will generally be larger than the plaintext, but by an amount that varies with the AEAD algorithm. Since the ciphers might incorporate padding, the amount of overhead could vary with different lengths of plaintext. Symbolically,

```
AEADEncrypted =  
    AEAD-Encrypt(write_key, nonce, additional_data, plaintext)
```

The `encrypted_record` field of `TLSCiphertext` is set to `AEADEncrypted`.

In order to decrypt and verify, the cipher takes as input the key, nonce, additional data, and the `AEADEncrypted` value. The output is either the plaintext or an error indicating that the decryption failed. There is no separate integrity check. Symbolically,

```
plaintext of encrypted_record =  
    AEAD-Decrypt(peer_write_key, nonce, additional_data, AEADEncrypted)
```

If the decryption fails, the receiver **MUST** terminate the connection with a "bad_record_mac" alert.

An AEAD algorithm used in TLS 1.3 **MUST NOT** produce an expansion greater than 255 octets. An endpoint that receives a record from its peer with `TLSCiphertext.length` larger than $2^{14} + 256$ octets **MUST** terminate the connection with a "record_overflow" alert. This limit is derived from the maximum `TLSInnerPlaintext` length of 2^{14} octets + 1 octet for `ContentType` + the maximum AEAD expansion of 255 octets.

5.3. Per-Record Nonce

A 64-bit sequence number is maintained separately for reading and writing records. The appropriate sequence number is incremented by one after reading or writing each record. Each sequence number is set to zero at the beginning of a connection and whenever the key is changed; the first record transmitted under a particular traffic key **MUST** use sequence number 0.

Because the size of sequence numbers is 64-bit, they should not wrap. If a TLS implementation would need to wrap a sequence number, it **MUST** either rekey (Section 4.6.3) or terminate the connection.

Each AEAD algorithm will specify a range of possible lengths for the per-record nonce, from N_MIN bytes to N_MAX bytes of input [RFC5116]. The length of the TLS per-record nonce (iv_length) is set to the larger of 8 bytes and N_MIN for the AEAD algorithm (see [RFC5116], Section 4). An AEAD algorithm where N_MAX is less than 8 bytes MUST NOT be used with TLS. The per-record nonce for the AEAD construction is formed as follows:

1. The 64-bit record sequence number is encoded in network byte order and padded to the left with zeros to iv_length.
2. The padded sequence number is XORed with either the static client_write_iv or server_write_iv (depending on the role).

The resulting quantity (of length iv_length) is used as the per-record nonce.

Note: This is a different construction from that in TLS 1.2, which specified a partially explicit nonce.

5.4. Record Padding

All encrypted TLS records can be padded to inflate the size of the TLSCiphertext. This allows the sender to hide the size of the traffic from an observer.

When generating a TLSCiphertext record, implementations MAY choose to pad. An unpadded record is just a record with a padding length of zero. Padding is a string of zero-valued bytes appended to the ContentType field before encryption. Implementations MUST set the padding octets to all zeros before encrypting.

Application Data records may contain a zero-length TLSInnerPlaintext.content if the sender desires. This permits generation of plausibly sized cover traffic in contexts where the presence or absence of activity may be sensitive. Implementations MUST NOT send Handshake and Alert records that have a zero-length TLSInnerPlaintext.content; if such a message is received, the receiving implementation MUST terminate the connection with an "unexpected_message" alert.

The padding sent is automatically verified by the record protection mechanism; upon successful decryption of a TLSCiphertext.encrypted_record, the receiving implementation scans the field from the end toward the beginning until it finds a non-zero octet. This non-zero octet is the content type of the message. This padding scheme was selected because it allows padding of any encrypted TLS record by an arbitrary size (from zero up to TLS record

size limits) without introducing new content types. The design also enforces all-zero padding octets, which allows for quick detection of padding errors.

Implementations MUST limit their scanning to the cleartext returned from the AEAD decryption. If a receiving implementation does not find a non-zero octet in the cleartext, it MUST terminate the connection with an "unexpected_message" alert.

The presence of padding does not change the overall record size limitations: the full encoded TLSInnerPlaintext MUST NOT exceed $2^{14} + 1$ octets. If the maximum fragment length is reduced -- as for example by the record_size_limit extension from [RFC8449] -- then the reduced limit applies to the full plaintext, including the content type and padding.

Selecting a padding policy that suggests when and how much to pad is a complex topic and is beyond the scope of this specification. If the application-layer protocol on top of TLS has its own padding, it may be preferable to pad Application Data TLS records within the application layer. Padding for encrypted Handshake or Alert records must still be handled at the TLS layer, though. Later documents may define padding selection algorithms or define a padding policy request mechanism through TLS extensions or some other means.

5.5. Limits on Key Usage

There are cryptographic limits on the amount of plaintext which can be safely encrypted under a given set of keys. [AEAD-LIMITS] provides an analysis of these limits under the assumption that the underlying primitive (AES or ChaCha20) has no weaknesses. Implementations SHOULD do a key update as described in Section 4.6.3 prior to reaching these limits. Note that it is not possible to perform a KeyUpdate for early data and therefore implementations SHOULD not exceed the limits when sending early data.

For AES-GCM, up to $2^{24.5}$ full-size records (about 24 million) may be encrypted on a given connection while keeping a safety margin of approximately 2^{-57} for Authenticated Encryption (AE) security. For ChaCha20/Poly1305, the record sequence number would wrap before the safety limit is reached.

6. Alert Protocol

TLS provides an Alert content type to indicate closure information and errors. Like other messages, alert messages are encrypted as specified by the current connection state.

Alert messages convey a description of the alert and a legacy field that conveyed the severity level of the message in previous versions of TLS. Alerts are divided into two classes: closure alerts and error alerts. In TLS 1.3, the severity is implicit in the type of alert being sent, and the "level" field can safely be ignored. The "close_notify" alert is used to indicate orderly closure of one direction of the connection. Upon receiving such an alert, the TLS implementation SHOULD indicate end-of-data to the application.

Error alerts indicate abortive closure of the connection (see Section 6.2). Upon receiving an error alert, the TLS implementation SHOULD indicate an error to the application and MUST NOT allow any further data to be sent or received on the connection. Servers and clients MUST forget the secret values and keys established in failed connections, with the exception of the PSKs associated with session tickets, which SHOULD be discarded if possible.

All the alerts listed in Section 6.2 MUST be sent with AlertLevel=fatal and MUST be treated as error alerts when received regardless of the AlertLevel in the message. Unknown Alert types MUST be treated as error alerts.

Note: TLS defines two generic alerts (see Section 6) to use upon failure to parse a message. Peers which receive a message which cannot be parsed according to the syntax (e.g., have a length extending beyond the message boundary or contain an out-of-range length) MUST terminate the connection with a "decode_error" alert. Peers which receive a message which is syntactically correct but semantically invalid (e.g., a DHE share of $p - 1$, or an invalid enum) MUST terminate the connection with an "illegal_parameter" alert.

```
enum { warning(1), fatal(2), (255) } AlertLevel;

enum {
    close_notify(0),
    unexpected_message(10),
    bad_record_mac(20),
    record_overflow(22),
    handshake_failure(40),
    bad_certificate(42),
    unsupported_certificate(43),
    certificate_revoked(44),
    certificate_expired(45),
    certificate_unknown(46),
    illegal_parameter(47),
    unknown_ca(48),
    access_denied(49),
    decode_error(50),
    decrypt_error(51),
    protocol_version(70),
    insufficient_security(71),
    internal_error(80),
    inappropriate_fallback(86),
    user_canceled(90),
    missing_extension(109),
    unsupported_extension(110),
    unrecognized_name(112),
    bad_certificate_status_response(113),
    unknown_psk_identity(115),
    certificate_required(116),
    no_application_protocol(120),
    (255)
} AlertDescription;

struct {
    AlertLevel level;
    AlertDescription description;
} Alert;
```

6.1. Closure Alerts

The client and the server must share knowledge that the connection is ending in order to avoid a truncation attack.

close_notify: This alert notifies the recipient that the sender will not send any more messages on this connection. Any data received after a closure alert has been received **MUST** be ignored.

user_canceled: This alert notifies the recipient that the sender is

canceling the handshake for some reason unrelated to a protocol failure. If a user cancels an operation after the handshake is complete, just closing the connection by sending a "close_notify" is more appropriate. This alert SHOULD be followed by a "close_notify". This alert generally has AlertLevel=warning.

Either party MAY initiate a close of its write side of the connection by sending a "close_notify" alert. Any data received after a closure alert has been received MUST be ignored. If a transport-level close is received prior to a "close_notify", the receiver cannot know that all the data that was sent has been received.

Each party MUST send a "close_notify" alert before closing its write side of the connection, unless it has already sent some error alert. This does not have any effect on its read side of the connection. Note that this is a change from versions of TLS prior to TLS 1.3 in which implementations were required to react to a "close_notify" by discarding pending writes and sending an immediate "close_notify" alert of their own. That previous requirement could cause truncation in the read side. Both parties need not wait to receive a "close_notify" alert before closing their read side of the connection, though doing so would introduce the possibility of truncation.

If the application protocol using TLS provides that any data may be carried over the underlying transport after the TLS connection is closed, the TLS implementation MUST receive a "close_notify" alert before indicating end-of-data to the application layer. No part of this standard should be taken to dictate the manner in which a usage profile for TLS manages its data transport, including when connections are opened or closed.

Note: It is assumed that closing the write side of a connection reliably delivers pending data before destroying the transport.

6.2. Error Alerts

Error handling in TLS is very simple. When an error is detected, the detecting party sends a message to its peer. Upon transmission or receipt of a fatal alert message, both parties MUST immediately close the connection.

Whenever an implementation encounters a fatal error condition, it SHOULD send an appropriate fatal alert and MUST close the connection without sending or receiving any additional data. Throughout this specification, when the phrases "terminate the connection" and "abort the handshake" are used without a specific alert it means that the implementation SHOULD send the alert indicated by the descriptions

below. The phrases "terminate the connection with an X alert" and "abort the handshake with an X alert" mean that the implementation MUST send alert X if it sends any alert. All alerts defined below in this section, as well as all unknown alerts, are universally considered fatal as of TLS 1.3 (see Section 6). The implementation SHOULD provide a way to facilitate logging the sending and receiving of alerts.

The following error alerts are defined:

`unexpected_message`: An inappropriate message (e.g., the wrong handshake message, premature Application Data, etc.) was received. This alert should never be observed in communication between proper implementations.

`bad_record_mac`: This alert is returned if a record is received which cannot be deprotected. Because AEAD algorithms combine decryption and verification, and also to avoid side-channel attacks, this alert is used for all deprotection failures. This alert should never be observed in communication between proper implementations, except when messages were corrupted in the network.

`record_overflow`: A `TLSCiphertext` record was received that had a length more than $2^{14} + 256$ bytes, or a record decrypted to a `TLSPlaintext` record with more than 2^{14} bytes (or some other negotiated limit). This alert should never be observed in communication between proper implementations, except when messages were corrupted in the network.

`handshake_failure`: Receipt of a "handshake_failure" alert message indicates that the sender was unable to negotiate an acceptable set of security parameters given the options available.

`bad_certificate`: A certificate was corrupt, contained signatures that did not verify correctly, etc.

`unsupported_certificate`: A certificate was of an unsupported type.

`certificate_revoked`: A certificate was revoked by its signer.

`certificate_expired`: A certificate has expired or is not currently valid.

`certificate_unknown`: Some other (unspecified) issue arose in processing the certificate, rendering it unacceptable.

`illegal_parameter`: A field in the handshake was incorrect or

inconsistent with other fields. This alert is used for errors which conform to the formal protocol syntax but are otherwise incorrect.

unknown_ca: A valid certificate chain or partial chain was received, but the certificate was not accepted because the CA certificate could not be located or could not be matched with a known trust anchor.

access_denied: A valid certificate or PSK was received, but when access control was applied, the sender decided not to proceed with negotiation.

decode_error: A message could not be decoded because some field was out of the specified range or the length of the message was incorrect. This alert is used for errors where the message does not conform to the formal protocol syntax. This alert should never be observed in communication between proper implementations, except when messages were corrupted in the network.

decrypt_error: A handshake (not record layer) cryptographic operation failed, including being unable to correctly verify a signature or validate a Finished message or a PSK binder.

protocol_version: The protocol version the peer has attempted to negotiate is recognized but not supported (see Appendix E).

insufficient_security: Returned instead of "handshake_failure" when a negotiation has failed specifically because the server requires parameters more secure than those supported by the client.

internal_error: An internal error unrelated to the peer or the correctness of the protocol (such as a memory allocation failure) makes it impossible to continue.

inappropriate_fallback: Sent by a server in response to an invalid connection retry attempt from a client (see [RFC7507]).

missing_extension: Sent by endpoints that receive a handshake message not containing an extension that is mandatory to send for the offered TLS version or other negotiated parameters.

unsupported_extension: Sent by endpoints receiving any handshake message containing an extension known to be prohibited for inclusion in the given handshake message, or including any extensions in a ServerHello or Certificate not first offered in the corresponding ClientHello or CertificateRequest.

`unrecognized_name`: Sent by servers when no server exists identified by the name provided by the client via the `"server_name"` extension (see [RFC6066]).

`bad_certificate_status_response`: Sent by clients when an invalid or unacceptable OCSP response is provided by the server via the `"status_request"` extension (see [RFC6066]).

`unknown_psk_identity`: Sent by servers when PSK key establishment is desired but no acceptable PSK identity is provided by the client. Sending this alert is OPTIONAL; servers MAY instead choose to send a `"decrypt_error"` alert to merely indicate an invalid PSK identity.

`certificate_required`: Sent by servers when a client certificate is desired but none was provided by the client.

`no_application_protocol`: Sent by servers when a client `"application_layer_protocol_negotiation"` extension advertises only protocols that the server does not support (see [RFC7301]).

New Alert values are assigned by IANA as described in Section 11.

7. Cryptographic Computations

The TLS handshake establishes one or more input secrets which are combined to create the actual working keying material, as detailed below. The key derivation process incorporates both the input secrets and the handshake transcript. Note that because the handshake transcript includes the random values from the Hello messages, any given handshake will have different traffic secrets, even if the same input secrets are used, as is the case when the same PSK is used for multiple connections.

7.1. Key Schedule

The key derivation process makes use of the HKDF-Extract and HKDF-Expand functions as defined for HKDF [RFC5869], as well as the functions defined below:

```
HKDF-Expand-Label(Secret, Label, Context, Length) =  
    HKDF-Expand(Secret, HkdfLabel, Length)
```

Where HkdfLabel is specified as:

```
struct {  
    uint16 length = Length;  
    opaque label<7..255> = "tls13 " + Label;  
    opaque context<0..255> = Context;  
} HkdfLabel;
```

```
Derive-Secret(Secret, Label, Messages) =  
    HKDF-Expand-Label(Secret, Label,  
        Transcript-Hash(Messages), Hash.length)
```

The Hash function used by Transcript-Hash and HKDF is the cipher suite hash algorithm. Hash.length is its output length in bytes. Messages is the concatenation of the indicated handshake messages, including the handshake message type and length fields, but not including record layer headers. Note that in some cases a zero-length Context (indicated by "") is passed to HKDF-Expand-Label. The labels specified in this document are all ASCII strings and do not include a trailing NUL byte.

Note: With common hash functions, any label longer than 12 characters requires an additional iteration of the hash function to compute. The labels in this specification have all been chosen to fit within this limit.

Keys are derived from two input secrets using the HKDF-Extract and Derive-Secret functions. The general pattern for adding a new secret is to use HKDF-Extract with the Salt being the current secret state and the Input Keying Material (IKM) being the new secret to be added. In this version of TLS 1.3, the two input secrets are:

- * PSK (a pre-shared key established externally or derived from the resumption_secret value from a previous connection)
- * (EC)DHE shared secret (Section 7.4)

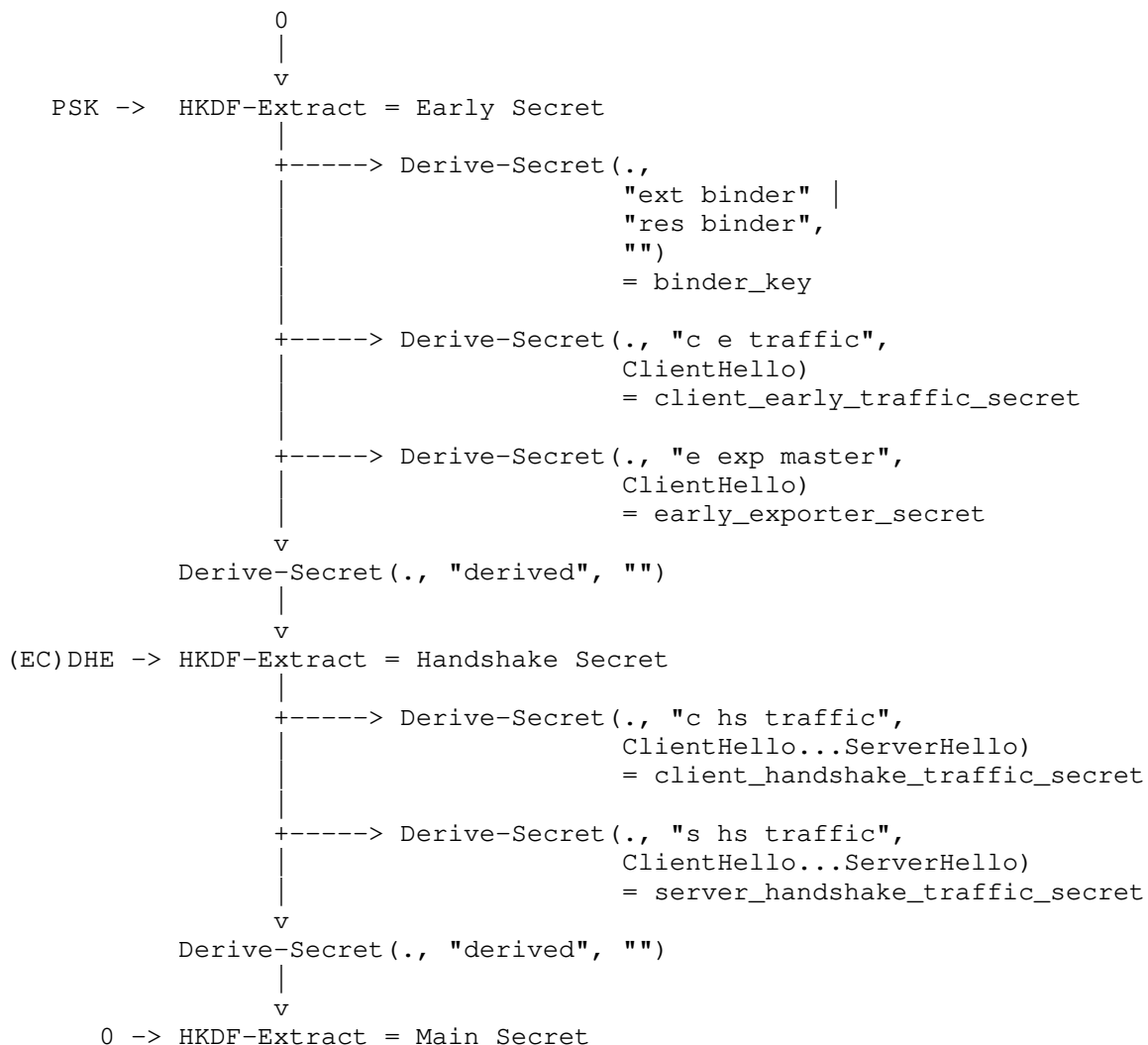
This produces a full key derivation schedule shown in the diagram below. In this diagram, the following formatting conventions apply:

- * HKDF-Extract is drawn as taking the Salt argument from the top and the IKM argument from the left, with its output to the bottom and the name of the output on the right.

- * Derive-Secret's Secret argument is indicated by the incoming arrow. For instance, the Early Secret is the Secret for generating the `client_early_traffic_secret`.
- * "0" indicates a string of `Hash.length` bytes set to zero.

Note: the key derivation labels use the string "master" even though the values are referred to as "main" secrets. This mismatch is a result of renaming the values while retaining compatibility.

[[OPEN ISSUE: Replace the strings with hex value?]]



```

|
+-----> Derive-Secret(., "c ap traffic",
|                                     ClientHello...server Finished)
|                                     = client_application_traffic_secret_0
+-----> Derive-Secret(., "s ap traffic",
|                                     ClientHello...server Finished)
|                                     = server_application_traffic_secret_0
+-----> Derive-Secret(., "exp master",
|                                     ClientHello...server Finished)
|                                     = exporter_secret
+-----> Derive-Secret(., "res master",
|                                     ClientHello...client Finished)
|                                     = resumption_secret

```

The general pattern here is that the secrets shown down the left side of the diagram are just raw entropy without context, whereas the secrets down the right side include Handshake Context and therefore can be used to derive working keys without additional context. Note that the different calls to Derive-Secret may take different Messages arguments, even with the same secret. In a 0-RTT exchange, Derive-Secret is called with four distinct transcripts; in a 1-RTT-only exchange, it is called with three distinct transcripts.

If a given secret is not available, then the 0-value consisting of a string of Hash.length bytes set to zeros is used. Note that this does not mean skipping rounds, so if PSK is not in use, Early Secret will still be HKDF-Extract(0, 0). For the computation of the binder_key, the label is "ext binder" for external PSKs (those provisioned outside of TLS) and "res binder" for resumption PSKs (those provisioned as the resumption secret of a previous handshake). The different labels prevent the substitution of one type of PSK for the other.

There are multiple potential Early Secret values, depending on which PSK the server ultimately selects. The client will need to compute one for each potential PSK; if no PSK is selected, it will then need to compute the Early Secret corresponding to the zero PSK.

Once all the values which are to be derived from a given secret have been computed, that secret SHOULD be erased.

7.2. Updating Traffic Secrets

Once the handshake is complete, it is possible for either side to update its sending traffic keys using the KeyUpdate handshake message defined in Section 4.6.3. The next generation of traffic keys is computed by generating `client_/server_application_traffic_secret_N+1` from `client_/server_application_traffic_secret_N` as described in this section and then re-deriving the traffic keys as described in Section 7.3.

The next-generation `application_traffic_secret` is computed as:

```
application_traffic_secret_N+1 =  
    HKDF-Expand-Label(application_traffic_secret_N,  
                        "traffic upd", "", Hash.length)
```

Once `client_/server_application_traffic_secret_N+1` and its associated traffic keys have been computed, implementations SHOULD delete `client_/server_application_traffic_secret_N` and its associated traffic keys.

7.3. Traffic Key Calculation

The traffic keying material is generated from the following input values:

- * A secret value
- * A purpose value indicating the specific value being generated
- * The length of the key being generated

The traffic keying material is generated from an input traffic secret value using:

```
[sender]_write_key = HKDF-Expand-Label(Secret, "key", "", key_length)  
[sender]_write_iv  = HKDF-Expand-Label(Secret, "iv", "", iv_length)
```

[sender] denotes the sending side. The value of Secret for each category of data is shown in the table below.

Data Type	Secret
0-RTT Application and EndOfEarlyData	client_early_traffic_secret
Initial Handshake	[sender]_handshake_traffic_secret
Post-Handshake and Application Data	[sender]_application_traffic_secret_N

Table 3: Secrets for Traffic Keys

Alerts are sent with the then current sending key (or as plaintext if no such key has been established.) All the traffic keying material is recomputed whenever the underlying Secret changes (e.g., when changing from the handshake to Application Data keys or upon a key update).

7.4. (EC)DHE Shared Secret Calculation

7.4.1. Finite Field Diffie-Hellman

For finite field groups, a conventional Diffie-Hellman [DH76] computation is performed. The negotiated key (Z) is converted to a byte string by encoding in big-endian form and left-padded with zeros up to the size of the prime. This byte string is used as the shared secret in the key schedule as specified above.

Note that this construction differs from previous versions of TLS which remove leading zeros.

7.4.2. Elliptic Curve Diffie-Hellman

For secp256r1, secp384r1 and secp521r1, ECDH calculations (including parameter and key generation as well as the shared secret calculation) are performed according to [IEEE1363] using the ECKAS-DH1 scheme with the identity map as the key derivation function (KDF), so that the shared secret is the x-coordinate of the ECDH shared secret elliptic curve point represented as an octet string. Note that this octet string ("Z" in IEEE 1363 terminology) as output by FE2OSP (the Field Element to Octet String Conversion Primitive) has constant length for any given field; leading zeros found in this octet string MUST NOT be truncated.

(Note that this use of the identity KDF is a technicality. The complete picture is that ECDH is employed with a non-trivial KDF because TLS does not directly use this secret for anything other than for computing other secrets.)

For X25519 and X448, the ECDH calculations are as follows:

- * The public key to put into the `KeyShareEntry.key_exchange` structure is the result of applying the ECDH scalar multiplication function to the secret key of appropriate length (into scalar input) and the standard public basepoint (into u-coordinate point input).
- * The ECDH shared secret is the result of applying the ECDH scalar multiplication function to the secret key (into scalar input) and the peer's public key (into u-coordinate point input). The output is used raw, with no processing.

For these curves, implementations SHOULD use the approach specified in [RFC7748] to calculate the Diffie-Hellman shared secret. Implementations MUST check whether the computed Diffie-Hellman shared secret is the all-zero value and abort if so, as described in Section 6 of [RFC7748]. If implementors use an alternative implementation of these elliptic curves, they SHOULD perform the additional checks specified in Section 7 of [RFC7748].

7.5. Exporters

[RFC5705] defines keying material exporters for TLS in terms of the TLS pseudorandom function (PRF). This document replaces the PRF with HKDF, thus requiring a new construction. The exporter interface remains the same.

The exporter value is computed as:

```
TLS-Exporter(label, context_value, key_length) =  
    HKDF-Expand-Label(Derive-Secret(Secret, label, ""),  
                      "exporter", Hash(context_value), key_length)
```

Where `Secret` is either the `early_exporter_secret` or the `exporter_secret`. Implementations MUST use the `exporter_secret` unless explicitly specified by the application. The `early_exporter_secret` is defined for use in settings where an exporter is needed for 0-RTT data. A separate interface for the early exporter is RECOMMENDED; this avoids the exporter user accidentally using an early exporter when a regular one is desired or vice versa.

If no context is provided, the context_value is zero length. Consequently, providing no context computes the same value as providing an empty context. This is a change from previous versions of TLS where an empty context produced a different output than an absent context. As of this document's publication, no allocated exporter label is used both with and without a context. Future specifications MUST NOT define a use of exporters that permit both an empty context and no context with the same label. New uses of exporters SHOULD provide a context in all exporter computations, though the value could be empty.

Requirements for the format of exporter labels are defined in Section 4 of [RFC5705].

8. 0-RTT and Anti-Replay

As noted in Section 2.3 and Appendix F.5, TLS does not provide inherent replay protections for 0-RTT data. There are two potential threats to be concerned with:

- * Network attackers who mount a replay attack by simply duplicating a flight of 0-RTT data.
- * Network attackers who take advantage of client retry behavior to arrange for the server to receive multiple copies of an application message. This threat already exists to some extent because clients that value robustness respond to network errors by attempting to retry requests. However, 0-RTT adds an additional dimension for any server system which does not maintain globally consistent server state. Specifically, if a server system has multiple zones where tickets from zone A will not be accepted in zone B, then an attacker can duplicate a ClientHello and early data intended for A to both A and B. At A, the data will be accepted in 0-RTT, but at B the server will reject 0-RTT data and instead force a full handshake. If the attacker blocks the ServerHello from A, then the client will complete the handshake with B and probably retry the request, leading to duplication on the server system as a whole.

The first class of attack can be prevented by sharing state to guarantee that the 0-RTT data is accepted at most once. Servers SHOULD provide that level of replay safety by implementing one of the methods described in this section or by equivalent means. It is understood, however, that due to operational concerns not all deployments will maintain state at that level. Therefore, in normal operation, clients will not know which, if any, of these mechanisms servers actually implement and hence MUST only send early data which they deem safe to be replayed.

In addition to the direct effects of replays, there is a class of attacks where even operations normally considered idempotent could be exploited by a large number of replays (timing attacks, resource limit exhaustion and others, as described in Appendix F.5). Those can be mitigated by ensuring that every 0-RTT payload can be replayed only a limited number of times. The server **MUST** ensure that any instance of it (be it a machine, a thread, or any other entity within the relevant serving infrastructure) would accept 0-RTT for the same 0-RTT handshake at most once; this limits the number of replays to the number of server instances in the deployment. Such a guarantee can be accomplished by locally recording data from recently received ClientHellos and rejecting repeats, or by any other method that provides the same or a stronger guarantee. The "at most once per server instance" guarantee is a minimum requirement; servers **SHOULD** limit 0-RTT replays further when feasible.

The second class of attack cannot be prevented at the TLS layer and **MUST** be dealt with by any application. Note that any application whose clients implement any kind of retry behavior already needs to implement some sort of anti-replay defense.

8.1. Single-Use Tickets

The simplest form of anti-replay defense is for the server to only allow each session ticket to be used once. For instance, the server can maintain a database of all outstanding valid tickets, deleting each ticket from the database as it is used. If an unknown ticket is provided, the server would then fall back to a full handshake.

If the tickets are not self-contained but rather are database keys, and the corresponding PSKs are deleted upon use, then connections established using PSKs enjoy not only anti-replay protection, but also forward secrecy once all copies of the PSK from the database entry have been deleted. This mechanism also improves security for PSK usage when PSK is used without (EC)DHE.

Because this mechanism requires sharing the session database between server nodes in environments with multiple distributed servers, it may be hard to achieve high rates of successful PSK 0-RTT connections when compared to self-encrypted tickets. Unlike session databases, session tickets can successfully do PSK-based session establishment even without consistent storage, though when 0-RTT is allowed they still require consistent storage for anti-replay of 0-RTT data, as detailed in the following section.

8.2. Client Hello Recording

An alternative form of anti-replay is to record a unique value derived from the ClientHello (generally either the random value or the PSK binder) and reject duplicates. Recording all ClientHellos causes state to grow without bound, but a server can instead record ClientHellos within a given time window and use the "obfuscated_ticket_age" to ensure that tickets aren't reused outside that window.

In order to implement this, when a ClientHello is received, the server first verifies the PSK binder as described in Section 4.2.11. It then computes the expected_arrival_time as described in the next section and rejects 0-RTT if it is outside the recording window, falling back to the 1-RTT handshake.

If the expected_arrival_time is in the window, then the server checks to see if it has recorded a matching ClientHello. If one is found, it either aborts the handshake with an "illegal_parameter" alert or accepts the PSK but rejects 0-RTT. If no matching ClientHello is found, then it accepts 0-RTT and then stores the ClientHello for as long as the expected_arrival_time is inside the window. Servers MAY also implement data stores with false positives, such as Bloom filters, in which case they MUST respond to apparent replay by rejecting 0-RTT but MUST NOT abort the handshake.

The server MUST derive the storage key only from validated sections of the ClientHello. If the ClientHello contains multiple PSK identities, then an attacker can create multiple ClientHellos with different binder values for the less-preferred identity on the assumption that the server will not verify it (as recommended by Section 4.2.11). I.e., if the client sends PSKs A and B but the server prefers A, then the attacker can change the binder for B without affecting the binder for A. If the binder for B is part of the storage key, then this ClientHello will not appear as a duplicate, which will cause the ClientHello to be accepted, and may cause side effects such as replay cache pollution, although any 0-RTT data will not be decryptable because it will use different keys. If the validated binder or the ClientHello.random is used as the storage key, then this attack is not possible.

Because this mechanism does not require storing all outstanding tickets, it may be easier to implement in distributed systems with high rates of resumption and 0-RTT, at the cost of potentially weaker anti-replay defense because of the difficulty of reliably storing and retrieving the received ClientHello messages. In many such systems, it is impractical to have globally consistent storage of all the received ClientHellos. In this case, the best anti-replay protection

is provided by having a single storage zone be authoritative for a given ticket and refusing 0-RTT for that ticket in any other zone. This approach prevents simple replay by the attacker because only one zone will accept 0-RTT data. A weaker design is to implement separate storage for each zone but allow 0-RTT in any zone. This approach limits the number of replays to once per zone. Application message duplication of course remains possible with either design.

When implementations are freshly started, they SHOULD reject 0-RTT as long as any portion of their recording window overlaps the startup time. Otherwise, they run the risk of accepting replays which were originally sent during that period.

Note: If the client's clock is running much faster than the server's, then a ClientHello may be received that is outside the window in the future, in which case it might be accepted for 1-RTT, causing a client retry, and then acceptable later for 0-RTT. This is another variant of the second form of attack described in Section 8.

8.3. Freshness Checks

Because the ClientHello indicates the time at which the client sent it, it is possible to efficiently determine whether a ClientHello was likely sent reasonably recently and only accept 0-RTT for such a ClientHello, otherwise falling back to a 1-RTT handshake. This is necessary for the ClientHello storage mechanism described in Section 8.2 because otherwise the server needs to store an unlimited number of ClientHellos, and is a useful optimization for self-contained single-use tickets because it allows efficient rejection of ClientHellos which cannot be used for 0-RTT.

In order to implement this mechanism, a server needs to store the time that the server generated the session ticket, offset by an estimate of the round-trip time between client and server. I.e.,

$$\text{adjusted_creation_time} = \text{creation_time} + \text{estimated_RTT}$$

This value can be encoded in the ticket, thus avoiding the need to keep state for each outstanding ticket. The server can determine the client's view of the age of the ticket by subtracting the ticket's "ticket_age_add" value from the "obfuscated_ticket_age" parameter in the client's "pre_shared_key" extension. The server can determine the expected_arrival_time of the ClientHello as:

$$\text{expected_arrival_time} = \text{adjusted_creation_time} + \text{clients_ticket_age}$$

When a new ClientHello is received, the `expected_arrival_time` is then compared against the current server wall clock time and if they differ by more than a certain amount, 0-RTT is rejected, though the 1-RTT handshake can be allowed to complete.

There are several potential sources of error that might cause mismatches between the `expected_arrival_time` and the measured time. Variations in client and server clock rates are likely to be minimal, though potentially the absolute times may be off by large values. Network propagation delays are the most likely causes of a mismatch in legitimate values for elapsed time. Both the NewSessionTicket and ClientHello messages might be retransmitted and therefore delayed, which might be hidden by TCP. For clients on the Internet, this implies windows on the order of ten seconds to account for errors in clocks and variations in measurements; other deployment scenarios may have different needs. Clock skew distributions are not symmetric, so the optimal tradeoff may involve an asymmetric range of permissible mismatch values.

Note that freshness checking alone is not sufficient to prevent replays because it does not detect them during the error window, which -- depending on bandwidth and system capacity -- could include billions of replays in real-world settings. In addition, this freshness checking is only done at the time the ClientHello is received, and not when subsequent early Application Data records are received. After early data is accepted, records may continue to be streamed to the server over a longer time period.

9. Compliance Requirements

9.1. Mandatory-to-Implement Cipher Suites

In the absence of an application profile standard specifying otherwise:

A TLS-compliant application MUST implement the TLS_AES_128_GCM_SHA256 [GCM] cipher suite and SHOULD implement the TLS_AES_256_GCM_SHA384 [GCM] and TLS_CHACHA20_POLY1305_SHA256 [RFC8439] cipher suites (see Appendix B.4).

A TLS-compliant application MUST support digital signatures with `rsa_pkcs1_sha256` (for certificates), `rsa_pss_rsae_sha256` (for CertificateVerify and certificates), and `ecdsa_secp256r1_sha256`. A TLS-compliant application MUST support key exchange with `secp256r1` (NIST P-256) and SHOULD support key exchange with X25519 [RFC7748].

9.2. Mandatory-to-Implement Extensions

In the absence of an application profile standard specifying otherwise, a TLS-compliant application MUST implement the following TLS extensions:

- * Supported Versions ("supported_versions"; Section 4.2.1)
- * Cookie ("cookie"; Section 4.2.2)
- * Signature Algorithms ("signature_algorithms"; Section 4.2.3)
- * Signature Algorithms Certificate ("signature_algorithms_cert"; Section 4.2.3)
- * Negotiated Groups ("supported_groups"; Section 4.2.7)
- * Key Share ("key_share"; Section 4.2.8)
- * Server Name Indication ("server_name"; Section 3 of [RFC6066])

All implementations MUST send and use these extensions when offering applicable features:

- * "supported_versions" is REQUIRED for all ClientHello, ServerHello, and HelloRetryRequest messages.
- * "signature_algorithms" is REQUIRED for certificate authentication.
- * "supported_groups" is REQUIRED for ClientHello messages using DHE or ECDHE key exchange.
- * "key_share" is REQUIRED for DHE or ECDHE key exchange.
- * "pre_shared_key" is REQUIRED for PSK key agreement.
- * "psk_key_exchange_modes" is REQUIRED for PSK key agreement.

A client is considered to be attempting to negotiate using this specification if the ClientHello contains a "supported_versions" extension with 0x0304 contained in its body. Such a ClientHello message MUST meet the following requirements:

- * If not containing a "pre_shared_key" extension, it MUST contain both a "signature_algorithms" extension and a "supported_groups" extension.

- * If containing a "supported_groups" extension, it MUST also contain a "key_share" extension, and vice versa. An empty KeyShare.client_shares list is permitted.

Servers receiving a ClientHello which does not conform to these requirements MUST abort the handshake with a "missing_extension" alert.

Additionally, all implementations MUST support the use of the "server_name" extension with applications capable of using it. Servers MAY require clients to send a valid "server_name" extension. Servers requiring this extension SHOULD respond to a ClientHello lacking a "server_name" extension by terminating the connection with a "missing_extension" alert.

9.3. Protocol Invariants

This section describes invariants that TLS endpoints and middleboxes MUST follow. It also applies to earlier versions of TLS.

TLS is designed to be securely and compatibly extensible. Newer clients or servers, when communicating with newer peers, should negotiate the most preferred common parameters. The TLS handshake provides downgrade protection: Middleboxes passing traffic between a newer client and newer server without terminating TLS should be unable to influence the handshake (see Appendix F.1). At the same time, deployments update at different rates, so a newer client or server MAY continue to support older parameters, which would allow it to interoperate with older endpoints.

For this to work, implementations MUST correctly handle extensible fields:

- * A client sending a ClientHello MUST support all parameters advertised in it. Otherwise, the server may fail to interoperate by selecting one of those parameters.
- * A server receiving a ClientHello MUST correctly ignore all unrecognized cipher suites, extensions, and other parameters. Otherwise, it may fail to interoperate with newer clients. In TLS 1.3, a client receiving a CertificateRequest or NewSessionTicket MUST also ignore all unrecognized extensions.
- * A middlebox which terminates a TLS connection MUST behave as a compliant TLS server (to the original client), including having a certificate which the client is willing to accept, and also as a compliant TLS client (to the original server), including verifying the original server's certificate. In particular, it MUST

generate its own ClientHello containing only parameters it understands, and it MUST generate a fresh ServerHello random value, rather than forwarding the endpoint's value.

Note that TLS's protocol requirements and security analysis only apply to the two connections separately. Safely deploying a TLS terminator requires additional security considerations which are beyond the scope of this document.

- * A middlebox which forwards ClientHello parameters it does not understand MUST NOT process any messages beyond that ClientHello. It MUST forward all subsequent traffic unmodified. Otherwise, it may fail to interoperate with newer clients and servers.

Forwarded ClientHellos may contain advertisements for features not supported by the middlebox, so the response may include future TLS additions the middlebox does not recognize. These additions MAY change any message beyond the ClientHello arbitrarily. In particular, the values sent in the ServerHello might change, the ServerHello format might change, and the TLSCiphertext format might change.

The design of TLS 1.3 was constrained by widely deployed non-compliant TLS middleboxes (see Appendix E.4); however, it does not relax the invariants. Those middleboxes continue to be non-compliant.

10. Security Considerations

Security issues are discussed throughout this memo, especially in Appendix C, Appendix E, and Appendix F.

11. IANA Considerations

[[OPEN ISSUE: Should we remove this? I am reluctant to create a situation where one needs to read 8446 to process this document.]]

[[OPEN ISSUE: Add some text to rename the `extended_master_secret` entry in the extensions registry to `extended_main_secret`, after the above is resolved.]]

This document uses several registries that were originally created in [RFC4346] and updated in [RFC8447]. IANA has updated these to reference this document. The registries and their allocation policies are below:

- * TLS Cipher Suites registry: values with the first byte in the range 0-254 (decimal) are assigned via Specification Required [RFC8126]. Values with the first byte 255 (decimal) are reserved for Private Use [RFC8126].

IANA has added the cipher suites listed in Appendix B.4 to the registry. The "Value" and "Description" columns are taken from the table. The "DTLS-OK" and "Recommended" columns are both marked as "Y" for each new cipher suite.

- * TLS ContentType registry: Future values are allocated via Standards Action [RFC8126].
- * TLS Alerts registry: Future values are allocated via Standards Action [RFC8126]. IANA has populated this registry with the values from Appendix B.2. The "DTLS-OK" column is marked as "Y" for all such values. Values marked as "_RESERVED" have comments describing their previous usage.
- * TLS HandshakeType registry: Future values are allocated via Standards Action [RFC8126]. IANA has updated this registry to rename item 4 from "NewSessionTicket" to "new_session_ticket" and populated this registry with the values from Appendix B.3. The "DTLS-OK" column is marked as "Y" for all such values. Values marked "_RESERVED" have comments describing their previous or temporary usage.

This document also uses the TLS ExtensionType Values registry originally created in [RFC4366]. IANA has updated it to reference this document. Changes to the registry follow:

- * IANA has updated the registration policy as follows:

Values with the first byte in the range 0-254 (decimal) are assigned via Specification Required [RFC8126]. Values with the first byte 255 (decimal) are reserved for Private Use [RFC8126].

- * IANA has updated this registry to include the "key_share", "pre_shared_key", "psk_key_exchange_modes", "early_data", "cookie", "supported_versions", "certificate_authorities", "oid_filters", "post_handshake_auth", and "signature_algorithms_cert" extensions with the values defined in this document and the "Recommended" value of "Y".

- * IANA has updated this registry to include a "TLS 1.3" column which lists the messages in which the extension may appear. This column has been initially populated from the table in Section 4.2, with any extension not listed there marked as "-" to indicate that it is not used by TLS 1.3.

This document updates an entry in the TLS Certificate Types registry originally created in [RFC6091] and updated in [RFC8447]. IANA has updated the entry for value 1 to have the name "OpenPGP_RESERVED", "Recommended" value "N", and comment "Used in TLS versions prior to 1.3." IANA has updated the entry for value 0 to have the name "X509", "Recommended" value "Y", and comment "Was X.509 before TLS 1.3".

This document updates an entry in the TLS Certificate Status Types registry originally created in [RFC6961]. IANA has updated the entry for value 2 to have the name "ocsp_multi_RESERVED" and comment "Used in TLS versions prior to 1.3".

This document updates two entries in the TLS Supported Groups registry (created under a different name by [RFC4492]; now maintained by [RFC8422]) and updated by [RFC7919] and [RFC8447]. The entries for values 29 and 30 (x25519 and x448) have been updated to also refer to this document.

In addition, this document defines two new registries that are maintained by IANA:

- * TLS SignatureScheme registry: Values with the first byte in the range 0-253 (decimal) are assigned via Specification Required [RFC8126]. Values with the first byte 254 or 255 (decimal) are reserved for Private Use [RFC8126]. Values with the first byte in the range 0-6 or with the second byte in the range 0-3 that are not currently allocated are reserved for backward compatibility. This registry has a "Recommended" column. The registry has been initially populated with the values described in Section 4.2.3. The following values are marked as "Recommended":
ecdsa_secp256r1_sha256, ecdsa_secp384r1_sha384,
rsa_pss_rsae_sha256, rsa_pss_rsae_sha384, rsa_pss_rsae_sha512,
rsa_pss_pss_sha256, rsa_pss_pss_sha384, rsa_pss_pss_sha512, and
ed25519. The "Recommended" column is assigned a value of "N" unless explicitly requested, and adding a value with a "Recommended" value of "Y" requires Standards Action [RFC8126]. IESG Approval is REQUIRED for a Y->N transition.
- * TLS PskKeyExchangeMode registry: Values in the range 0-253 (decimal) are assigned via Specification Required [RFC8126]. The values 254 and 255 (decimal) are reserved for Private Use

[RFC8126]. This registry has a "Recommended" column. The registry has been initially populated with `psk_ke` (0) and `psk_dhe_ke` (1). Both are marked as "Recommended". The "Recommended" column is assigned a value of "N" unless explicitly requested, and adding a value with a "Recommended" value of "Y" requires Standards Action [RFC8126]. IESG Approval is REQUIRED for a Y->N transition.

12. References

12.1. Normative References

- [DH76] Diffie, W. and M. Hellman, "New directions in cryptography", IEEE Transactions on Information Theory Vol. 22, pp. 644-654, DOI 10.1109/tit.1976.1055638, November 1976, <<https://doi.org/10.1109/tit.1976.1055638>>.
- [ECDSA] American National Standards Institute, "Public Key Cryptography for the Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA)", ANSI ANS X9.62-2005, November 2005.
- [GCM] Dworkin, M., "Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC", NIST Special Publication 800-38D, November 2007.
- [IEEE1363] "IEEE Standard Specifications for Public-Key Cryptography", IEEE standard, DOI 10.1109/ieeestd.2000.92292, n.d., <<https://doi.org/10.1109/ieeestd.2000.92292>>.
- [RFC2104] Krawczyk, H., Bellare, M., and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication", RFC 2104, DOI 10.17487/RFC2104, February 1997, <<https://www.rfc-editor.org/info/rfc2104>>.
- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <<https://www.rfc-editor.org/info/rfc2119>>.
- [RFC5116] McGrew, D., "An Interface and Algorithms for Authenticated Encryption", RFC 5116, DOI 10.17487/RFC5116, January 2008, <<https://www.rfc-editor.org/info/rfc5116>>.

- [RFC5280] Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R., and W. Polk, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", RFC 5280, DOI 10.17487/RFC5280, May 2008, <<https://www.rfc-editor.org/info/rfc5280>>.
- [RFC5705] Rescorla, E., "Keying Material Exporters for Transport Layer Security (TLS)", RFC 5705, DOI 10.17487/RFC5705, March 2010, <<https://www.rfc-editor.org/info/rfc5705>>.
- [RFC5756] Turner, S., Brown, D., Yiu, K., Housley, R., and T. Polk, "Updates for RSAES-OAEP and RSASSA-PSS Algorithm Parameters", RFC 5756, DOI 10.17487/RFC5756, January 2010, <<https://www.rfc-editor.org/info/rfc5756>>.
- [RFC5869] Krawczyk, H. and P. Eronen, "HMAC-based Extract-and-Expand Key Derivation Function (HKDF)", RFC 5869, DOI 10.17487/RFC5869, May 2010, <<https://www.rfc-editor.org/info/rfc5869>>.
- [RFC6066] Eastlake 3rd, D., "Transport Layer Security (TLS) Extensions: Extension Definitions", RFC 6066, DOI 10.17487/RFC6066, January 2011, <<https://www.rfc-editor.org/info/rfc6066>>.
- [RFC6655] McGrew, D. and D. Bailey, "AES-CCM Cipher Suites for Transport Layer Security (TLS)", RFC 6655, DOI 10.17487/RFC6655, July 2012, <<https://www.rfc-editor.org/info/rfc6655>>.
- [RFC6960] Santesson, S., Myers, M., Ankney, R., Malpani, A., Galperin, S., and C. Adams, "X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP", RFC 6960, DOI 10.17487/RFC6960, June 2013, <<https://www.rfc-editor.org/info/rfc6960>>.
- [RFC6961] Pettersen, Y., "The Transport Layer Security (TLS) Multiple Certificate Status Request Extension", RFC 6961, DOI 10.17487/RFC6961, June 2013, <<https://www.rfc-editor.org/info/rfc6961>>.
- [RFC6962] Laurie, B., Langley, A., and E. Kasper, "Certificate Transparency", RFC 6962, DOI 10.17487/RFC6962, June 2013, <<https://www.rfc-editor.org/info/rfc6962>>.

- [RFC6979] Pornin, T., "Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA)", RFC 6979, DOI 10.17487/RFC6979, August 2013, <<https://www.rfc-editor.org/info/rfc6979>>.
- [RFC7301] Friedl, S., Popov, A., Langley, A., and E. Stephan, "Transport Layer Security (TLS) Application-Layer Protocol Negotiation Extension", RFC 7301, DOI 10.17487/RFC7301, July 2014, <<https://www.rfc-editor.org/info/rfc7301>>.
- [RFC7507] Moeller, B. and A. Langley, "TLS Fallback Signaling Cipher Suite Value (SCSV) for Preventing Protocol Downgrade Attacks", RFC 7507, DOI 10.17487/RFC7507, April 2015, <<https://www.rfc-editor.org/info/rfc7507>>.
- [RFC7627] Bhargavan, K., Ed., Delignat-Lavaud, A., Pironti, A., Langley, A., and M. Ray, "Transport Layer Security (TLS) Session Hash and Extended Master Secret Extension", RFC 7627, DOI 10.17487/RFC7627, September 2015, <<https://www.rfc-editor.org/info/rfc7627>>.
- [RFC7748] Langley, A., Hamburg, M., and S. Turner, "Elliptic Curves for Security", RFC 7748, DOI 10.17487/RFC7748, January 2016, <<https://www.rfc-editor.org/info/rfc7748>>.
- [RFC7919] Gillmor, D., "Negotiated Finite Field Diffie-Hellman Ephemeral Parameters for Transport Layer Security (TLS)", RFC 7919, DOI 10.17487/RFC7919, August 2016, <<https://www.rfc-editor.org/info/rfc7919>>.
- [RFC8017] Moriarty, K., Ed., Kaliski, B., Jonsson, J., and A. Rusch, "PKCS #1: RSA Cryptography Specifications Version 2.2", RFC 8017, DOI 10.17487/RFC8017, November 2016, <<https://www.rfc-editor.org/info/rfc8017>>.
- [RFC8032] Josefsson, S. and I. Liusvaara, "Edwards-Curve Digital Signature Algorithm (EdDSA)", RFC 8032, DOI 10.17487/RFC8032, January 2017, <<https://www.rfc-editor.org/info/rfc8032>>.
- [RFC8126] Cotton, M., Leiba, B., and T. Narten, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 8126, DOI 10.17487/RFC8126, June 2017, <<https://www.rfc-editor.org/info/rfc8126>>.
- [RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <<https://www.rfc-editor.org/info/rfc8174>>.

- [RFC8439] Nir, Y. and A. Langley, "ChaCha20 and Poly1305 for IETF Protocols", RFC 8439, DOI 10.17487/RFC8439, June 2018, <<https://www.rfc-editor.org/info/rfc8439>>.
- [RFC8996] Moriarty, K. and S. Farrell, "Deprecating TLS 1.0 and TLS 1.1", BCP 195, RFC 8996, DOI 10.17487/RFC8996, March 2021, <<https://www.rfc-editor.org/info/rfc8996>>.
- [SHS] Dang, Q., "Secure Hash Standard", National Institute of Standards and Technology report, DOI 10.6028/nist.fips.180-4, July 2015, <<https://doi.org/10.6028/nist.fips.180-4>>.
- [X690] ITU-T, "Information technology - ASN.1 encoding Rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)", ISO/IEC 8824-1:2021 , February 2021.

12.2. Informative References

- [AEAD-LIMITS] Luykx, A. and K. Paterson, "Limits on Authenticated Encryption Use in TLS", August 2017, <<http://www.isg.rhul.ac.uk/~kp/TLS-AEbounds.pdf>>.
- [BBFGKZ16] Bhargavan, K., Brzuska, C., Fournet, C., Green, M., Kohlweiss, M., and S. Zanella-Beguelin, "Downgrade Resilience in Key-Exchange Protocols", 2016 IEEE Symposium on Security and Privacy (SP), DOI 10.1109/sp.2016.37, May 2016, <<https://doi.org/10.1109/sp.2016.37>>.
- [BBK17] Bhargavan, K., Blanchet, B., and N. Kobeissi, "Verified Models and Reference Implementations for the TLS 1.3 Standard Candidate", 2017 IEEE Symposium on Security and Privacy (SP), DOI 10.1109/sp.2017.26, May 2017, <<https://doi.org/10.1109/sp.2017.26>>.
- [BDFKPPRSZZ16] Bhargavan, K., Delignat-Lavaud, A., Fournet, C., Kohlweiss, M., Pan, J., Protzenko, J., Rastogi, A., Swamy, N., Zanella-Beguelin, S., and J. Zinzindohoue, "Implementing and Proving the TLS 1.3 Record Layer", Proceedings of IEEE Symposium on Security and Privacy (San Jose) 2017 , December 2016, <<https://eprint.iacr.org/2016/1178>>.

- [Ben17a] Benjamin, D., "Presentation before the TLS WG at IETF 100", 2017, <<https://datatracker.ietf.org/meeting/100/materials/slides-100-tls-sessa-tls13/>>.
- [Ben17b] Benjamin, D., "Additional TLS 1.3 results from Chrome", 2017, <<https://www.ietf.org/mail-archive/web/tls/current/msg25168.html>>.
- [Blei98] Bleichenbacher, D., "Chosen Ciphertext Attacks against Protocols Based on RSA Encryption Standard PKCS #1", Proceedings of CRYPTO '98 , 1998.
- [BMMRT15] Badertscher, C., Matt, C., Maurer, U., Rogaway, P., and B. Tackmann, "Augmented Secure Channels and the Goal of the TLS 1.3 Record Layer", ProvSec 2015 , September 2015, <<https://eprint.iacr.org/2015/394>>.
- [BT16] Bellare, M. and B. Tackmann, "The Multi-User Security of Authenticated Encryption: AES-GCM in TLS 1.3", Proceedings of CRYPTO 2016 , July 2016, <<https://eprint.iacr.org/2016/564>>.
- [CCG16] Cohn-Gordon, K., Cremers, C., and L. Garratt, "On Post-compromise Security", 2016 IEEE 29th Computer Security Foundations Symposium (CSF), DOI 10.1109/csf.2016.19, June 2016, <<https://doi.org/10.1109/csf.2016.19>>.
- [CHECKOWAY] Checkoway, S., Maskiewicz, J., Garman, C., Fried, J., Cohny, S., Green, M., Heninger, N., Weinmann, R., Rescorla, E., and H. Shacham, "A Systematic Analysis of the Juniper Dual EC Incident", Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, DOI 10.1145/2976749.2978395, October 2016, <<https://doi.org/10.1145/2976749.2978395>>.
- [CHHSV17] Cremers, C., Horvat, M., Hoyland, J., van der Merwe, T., and S. Scott, "Awkward Handshake: Possible mismatch of client/server view on client authentication in post-handshake mode in Revision 18", message to the TLS mailing list , February 2017, <<https://www.ietf.org/mail-archive/web/tls/current/msg22382.html>>.

- [CHSV16] Cremers, C., Horvat, M., Scott, S., and T. van der Merwe, "Automated Analysis and Verification of TLS 1.3: 0-RTT, Resumption and Delayed Authentication", 2016 IEEE Symposium on Security and Privacy (SP), DOI 10.1109/sp.2016.35, May 2016, <<https://doi.org/10.1109/sp.2016.35>>.
- [CK01] Canetti, R. and H. Krawczyk, "Analysis of Key-Exchange Protocols and Their Use for Building Secure Channels", Lecture Notes in Computer Science pp. 453-474, DOI 10.1007/3-540-44987-6_28, 2001, <https://doi.org/10.1007/3-540-44987-6_28>.
- [CLINIC] Miller, B., Huang, L., Joseph, A., and J. Tygar, "I Know Why You Went to the Clinic: Risks and Realization of HTTPS Traffic Analysis", Privacy Enhancing Technologies pp. 143-163, DOI 10.1007/978-3-319-08506-7_8, 2014, <https://doi.org/10.1007/978-3-319-08506-7_8>.
- [DFGS15] Dowling, B., Fischlin, M., Guenther, F., and D. Stebila, "A Cryptographic Analysis of the TLS 1.3 draft-10 Full and Pre-shared Key Handshake Protocol", Proceedings of ACM CCS 2015 , October 2016, <<https://eprint.iacr.org/2015/914>>.
- [DFGS16] Dowling, B., Fischlin, M., Guenther, F., and D. Stebila, "A Cryptographic Analysis of the TLS 1.3 draft-10 Full and Pre-shared Key Handshake Protocol", TRON 2016 , February 2016, <<https://eprint.iacr.org/2016/081>>.
- [DOW92] Diffie, W., Van Oorschot, P., and M. Wiener, "Authentication and authenticated key exchanges", Designs, Codes and Cryptography Vol. 2, pp. 107-125, DOI 10.1007/bf00124891, June 1992, <<https://doi.org/10.1007/bf00124891>>.
- [DSA-1571-1] The Debian Project, "openssl -- predictable random number generator", May 2008, <<https://www.debian.org/security/2008/dsa-1571>>.
- [DSS] "Digital Signature Standard (DSS)", National Institute of Standards and Technology report, DOI 10.6028/nist.fips.186-4, July 2013, <<https://doi.org/10.6028/nist.fips.186-4>>.
- [FETCH] WHATWG, "Fetch Standard", March 2022, <<https://fetch.spec.whatwg.org/>>.

- [FG17] Fischlin, M. and F. Guenther, "Replay Attacks on Zero Round-Trip Time: The Case of the TLS 1.3 Handshake Candidates", Proceedings of Euro S&P 2017 , 2017, <<https://eprint.iacr.org/2017/082>>.
- [FGSW16] Fischlin, M., Guenther, F., Schmidt, B., and B. Warinschi, "Key Confirmation in Key Exchange: A Formal Treatment and Implications for TLS 1.3", Proceedings of IEEE Symposium on Security and Privacy (Oakland) 2016 , 2016, <<http://ieeexplore.ieee.org/document/7546517/>>.
- [FW15] Weimer, F., "Factoring RSA Keys With TLS Perfect Forward Secrecy", September 2015.
- [HCJC16] Husák, M., ermák, M., Jirsík, T., and P. eleda, "HTTPS traffic analysis and client identification using passive SSL/TLS fingerprinting", EURASIP Journal on Information Security Vol. 2016, DOI 10.1186/s13635-016-0030-7, February 2016, <<https://doi.org/10.1186/s13635-016-0030-7>>.
- [HGFS15] Hlauschek, C., Gruber, M., Fankhauser, F., and C. Schanes, "Prying Open Pandora's Box: KCI Attacks against TLS", Proceedings of USENIX Workshop on Offensive Technologies , 2015.
- [JSS15] Jager, T., Schwenk, J., and J. Somorovsky, "On the Security of TLS 1.3 and QUIC Against Weaknesses in PKCS#1 v1.5 Encryption", Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, DOI 10.1145/2810103.2813657, October 2015, <<https://doi.org/10.1145/2810103.2813657>>.
- [KEYAGREEMENT] Barker, E., Chen, L., Roginsky, A., and M. Smid, "Recommendation for Pair-Wise Key Establishment Schemes Using Discrete Logarithm Cryptography", National Institute of Standards and Technology report, DOI 10.6028/nist.sp.800-56ar2, May 2013, <<https://doi.org/10.6028/nist.sp.800-56ar2>>.
- [Kraw10] Krawczyk, H., "Cryptographic Extraction and Key Derivation: The HKDF Scheme", Proceedings of CRYPTO 2010 , 2010, <<https://eprint.iacr.org/2010/264>>.

- [Kraw16] Krawczyk, H., "A Unilateral-to-Mutual Authentication Compiler for Key Exchange (with Applications to Client Authentication in TLS 1.3", Proceedings of ACM CCS 2016 , October 2016, <<https://eprint.iacr.org/2016/711>>.
- [KW16] Krawczyk, H. and H. Wee, "The OPTLS Protocol and TLS 1.3", Proceedings of Euro S&P 2016 , 2016, <<https://eprint.iacr.org/2015/978>>.
- [LXZFH16] Li, X., Xu, J., Zhang, Z., Feng, D., and H. Hu, "Multiple Handshakes Security of TLS 1.3 Candidates", 2016 IEEE Symposium on Security and Privacy (SP), DOI 10.1109/sp.2016.36, May 2016, <<https://doi.org/10.1109/sp.2016.36>>.
- [Mac17] MacCarthaigh, C., "Security Review of TLS1.3 0-RTT", March 2017, <<https://github.com/tlswg/tls13-spec/issues/1001>>.
- [PS18] Patton, C. and T. Shrimpton, "Partially specified channels: The TLS 1.3 record layer without elision", 2018, <<https://eprint.iacr.org/2018/634>>.
- [PSK-FINISHED] Cremers, C., Horvat, M., van der Merwe, T., and S. Scott, "Revision 10: possible attack if client authentication is allowed during PSK", message to the TLS mailing list, , 2015, <<https://www.ietf.org/mail-archive/web/tls/current/msg18215.html>>.
- [REKEY] Abdalla, M. and M. Bellare, "Increasing the Lifetime of a Key: A Comparative Analysis of the Security of Re-keying Techniques", Advances in Cryptology - ASIACRYPT 2000 pp. 546-559, DOI 10.1007/3-540-44448-3_42, 2000, <https://doi.org/10.1007/3-540-44448-3_42>.
- [Res17a] Rescorla, E., "Preliminary data on Firefox TLS 1.3 Middlebox experiment", message to the TLS mailing list , 2017, <<https://www.ietf.org/mail-archive/web/tls/current/msg25091.html>>.
- [Res17b] Rescorla, E., "More compatibility measurement results", message to the TLS mailing list , December 2017, <<https://www.ietf.org/mail-archive/web/tls/current/msg25179.html>>.
- [RFC2246] Dierks, T. and C. Allen, "The TLS Protocol Version 1.0", RFC 2246, DOI 10.17487/RFC2246, January 1999, <<https://www.rfc-editor.org/info/rfc2246>>.

- [RFC3552] Rescorla, E. and B. Korver, "Guidelines for Writing RFC Text on Security Considerations", BCP 72, RFC 3552, DOI 10.17487/RFC3552, July 2003, <<https://www.rfc-editor.org/info/rfc3552>>.
- [RFC4086] Eastlake 3rd, D., Schiller, J., and S. Crocker, "Randomness Requirements for Security", BCP 106, RFC 4086, DOI 10.17487/RFC4086, June 2005, <<https://www.rfc-editor.org/info/rfc4086>>.
- [RFC4346] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.1", RFC 4346, DOI 10.17487/RFC4346, April 2006, <<https://www.rfc-editor.org/info/rfc4346>>.
- [RFC4366] Blake-Wilson, S., Nystrom, M., Hopwood, D., Mikkelsen, J., and T. Wright, "Transport Layer Security (TLS) Extensions", RFC 4366, DOI 10.17487/RFC4366, April 2006, <<https://www.rfc-editor.org/info/rfc4366>>.
- [RFC4492] Blake-Wilson, S., Bolyard, N., Gupta, V., Hawk, C., and B. Moeller, "Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS)", RFC 4492, DOI 10.17487/RFC4492, May 2006, <<https://www.rfc-editor.org/info/rfc4492>>.
- [RFC5077] Salowey, J., Zhou, H., Eronen, P., and H. Tschofenig, "Transport Layer Security (TLS) Session Resumption without Server-Side State", RFC 5077, DOI 10.17487/RFC5077, January 2008, <<https://www.rfc-editor.org/info/rfc5077>>.
- [RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, DOI 10.17487/RFC5246, August 2008, <<https://www.rfc-editor.org/info/rfc5246>>.
- [RFC5764] McGrew, D. and E. Rescorla, "Datagram Transport Layer Security (DTLS) Extension to Establish Keys for the Secure Real-time Transport Protocol (SRTP)", RFC 5764, DOI 10.17487/RFC5764, May 2010, <<https://www.rfc-editor.org/info/rfc5764>>.
- [RFC5929] Altman, J., Williams, N., and L. Zhu, "Channel Bindings for TLS", RFC 5929, DOI 10.17487/RFC5929, July 2010, <<https://www.rfc-editor.org/info/rfc5929>>.

- [RFC6091] Mavrogiannopoulos, N. and D. Gillmor, "Using OpenPGP Keys for Transport Layer Security (TLS) Authentication", RFC 6091, DOI 10.17487/RFC6091, February 2011, <<https://www.rfc-editor.org/info/rfc6091>>.
- [RFC6101] Freier, A., Karlton, P., and P. Kocher, "The Secure Sockets Layer (SSL) Protocol Version 3.0", RFC 6101, DOI 10.17487/RFC6101, August 2011, <<https://www.rfc-editor.org/info/rfc6101>>.
- [RFC6176] Turner, S. and T. Polk, "Prohibiting Secure Sockets Layer (SSL) Version 2.0", RFC 6176, DOI 10.17487/RFC6176, March 2011, <<https://www.rfc-editor.org/info/rfc6176>>.
- [RFC6347] Rescorla, E. and N. Modadugu, "Datagram Transport Layer Security Version 1.2", RFC 6347, DOI 10.17487/RFC6347, January 2012, <<https://www.rfc-editor.org/info/rfc6347>>.
- [RFC6520] Seggelmann, R., Tuexen, M., and M. Williams, "Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS) Heartbeat Extension", RFC 6520, DOI 10.17487/RFC6520, February 2012, <<https://www.rfc-editor.org/info/rfc6520>>.
- [RFC7230] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing", RFC 7230, DOI 10.17487/RFC7230, June 2014, <<https://www.rfc-editor.org/info/rfc7230>>.
- [RFC7250] Wouters, P., Ed., Tschofenig, H., Ed., Gilmore, J., Weiler, S., and T. Kivinen, "Using Raw Public Keys in Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)", RFC 7250, DOI 10.17487/RFC7250, June 2014, <<https://www.rfc-editor.org/info/rfc7250>>.
- [RFC7465] Popov, A., "Prohibiting RC4 Cipher Suites", RFC 7465, DOI 10.17487/RFC7465, February 2015, <<https://www.rfc-editor.org/info/rfc7465>>.
- [RFC7568] Barnes, R., Thomson, M., Pironti, A., and A. Langley, "Deprecating Secure Sockets Layer Version 3.0", RFC 7568, DOI 10.17487/RFC7568, June 2015, <<https://www.rfc-editor.org/info/rfc7568>>.

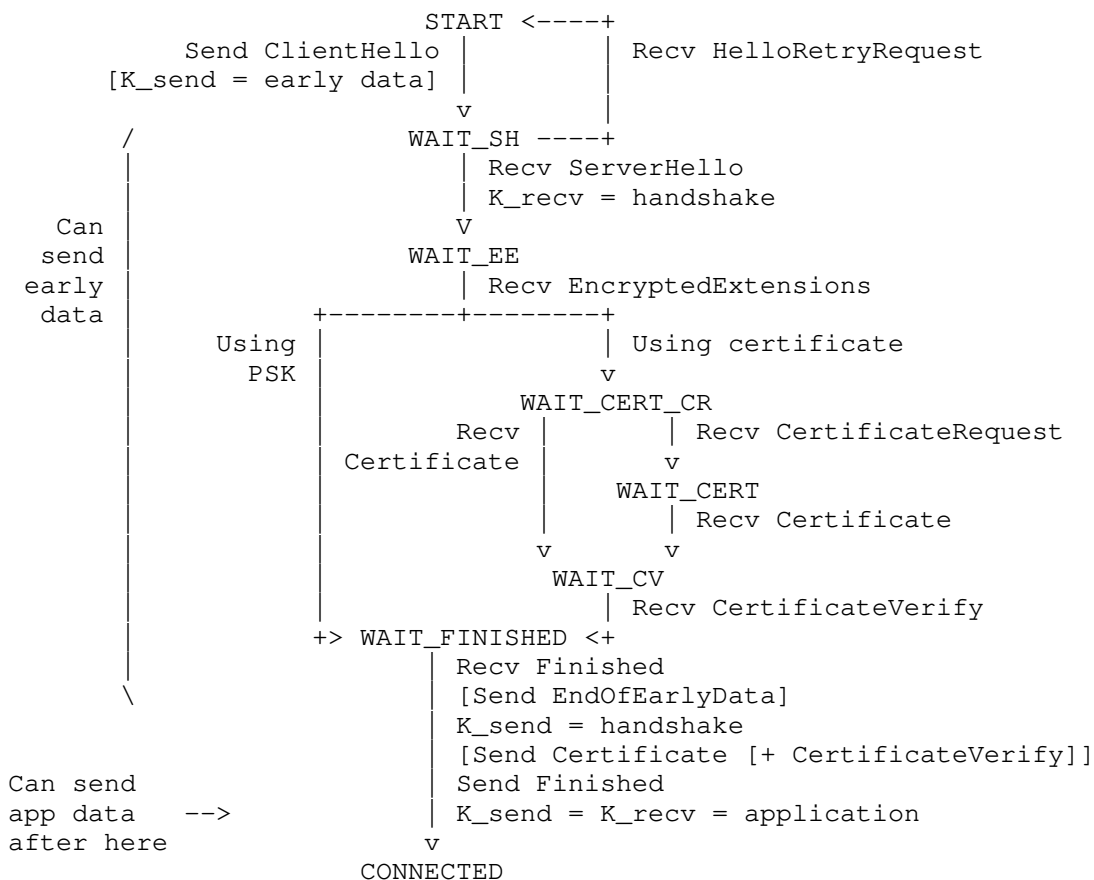
- [RFC7624] Barnes, R., Schneier, B., Jennings, C., Hardie, T., Trammell, B., Huitema, C., and D. Borkmann, "Confidentiality in the Face of Pervasive Surveillance: A Threat Model and Problem Statement", RFC 7624, DOI 10.17487/RFC7624, August 2015, <<https://www.rfc-editor.org/info/rfc7624>>.
- [RFC7685] Langley, A., "A Transport Layer Security (TLS) ClientHello Padding Extension", RFC 7685, DOI 10.17487/RFC7685, October 2015, <<https://www.rfc-editor.org/info/rfc7685>>.
- [RFC7924] Santesson, S. and H. Tschofenig, "Transport Layer Security (TLS) Cached Information Extension", RFC 7924, DOI 10.17487/RFC7924, July 2016, <<https://www.rfc-editor.org/info/rfc7924>>.
- [RFC8305] Schinazi, D. and T. Pauly, "Happy Eyeballs Version 2: Better Connectivity Using Concurrency", RFC 8305, DOI 10.17487/RFC8305, December 2017, <<https://www.rfc-editor.org/info/rfc8305>>.
- [RFC8422] Nir, Y., Josefsson, S., and M. Pegourie-Gonnard, "Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS) Versions 1.2 and Earlier", RFC 8422, DOI 10.17487/RFC8422, August 2018, <<https://www.rfc-editor.org/info/rfc8422>>.
- [RFC8446] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <<https://www.rfc-editor.org/info/rfc8446>>.
- [RFC8447] Salowey, J. and S. Turner, "IANA Registry Updates for TLS and DTLS", RFC 8447, DOI 10.17487/RFC8447, August 2018, <<https://www.rfc-editor.org/info/rfc8447>>.
- [RFC8448] Thomson, M., "Example Handshake Traces for TLS 1.3", RFC 8448, DOI 10.17487/RFC8448, January 2019, <<https://www.rfc-editor.org/info/rfc8448>>.
- [RFC8449] Thomson, M., "Record Size Limit Extension for TLS", RFC 8449, DOI 10.17487/RFC8449, August 2018, <<https://www.rfc-editor.org/info/rfc8449>>.
- [RFC8879] Ghedini, A. and V. Vasiliev, "TLS Certificate Compression", RFC 8879, DOI 10.17487/RFC8879, December 2020, <<https://www.rfc-editor.org/info/rfc8879>>.

- [RFC8937] Cremers, C., Garratt, L., Smyshlyaev, S., Sullivan, N., and C. Wood, "Randomness Improvements for Security Protocols", RFC 8937, DOI 10.17487/RFC8937, October 2020, <<https://www.rfc-editor.org/info/rfc8937>>.
- [RSA] Rivest, R., Shamir, A., and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems", Communications of the ACM Vol. 21, pp. 120-126, DOI 10.1145/359340.359342, February 1978, <<https://doi.org/10.1145/359340.359342>>.
- [SIGMA] Krawczyk, H., "SIGMA: The SIGn-and-MAC Approach to Authenticated Diffie-Hellman and Its Use in the IKE Protocols", Advances in Cryptology - CRYPTO 2003 pp. 400-425, DOI 10.1007/978-3-540-45146-4_24, 2003, <https://doi.org/10.1007/978-3-540-45146-4_24>.
- [SLOTH] Bhargavan, K. and G. Leurent, "Transcript Collision Attacks: Breaking Authentication in TLS, IKE, and SSH", Proceedings 2016 Network and Distributed System Security Symposium, DOI 10.14722/ndss.2016.23418, 2016, <<https://doi.org/10.14722/ndss.2016.23418>>.
- [SSL2] Hickman, K., "The SSL Protocol", 9 February 1995.
- [TIMING] Boneh, D. and D. Brumley, "Remote Timing Attacks Are Practical", USENIX Security Symposium, 2003.
- [X501] ITU-T, "Information Technology - Open Systems Interconnection - The Directory: Models", ISO/IEC 9594-2:2020 , October 2019.

Appendix A. State Machine

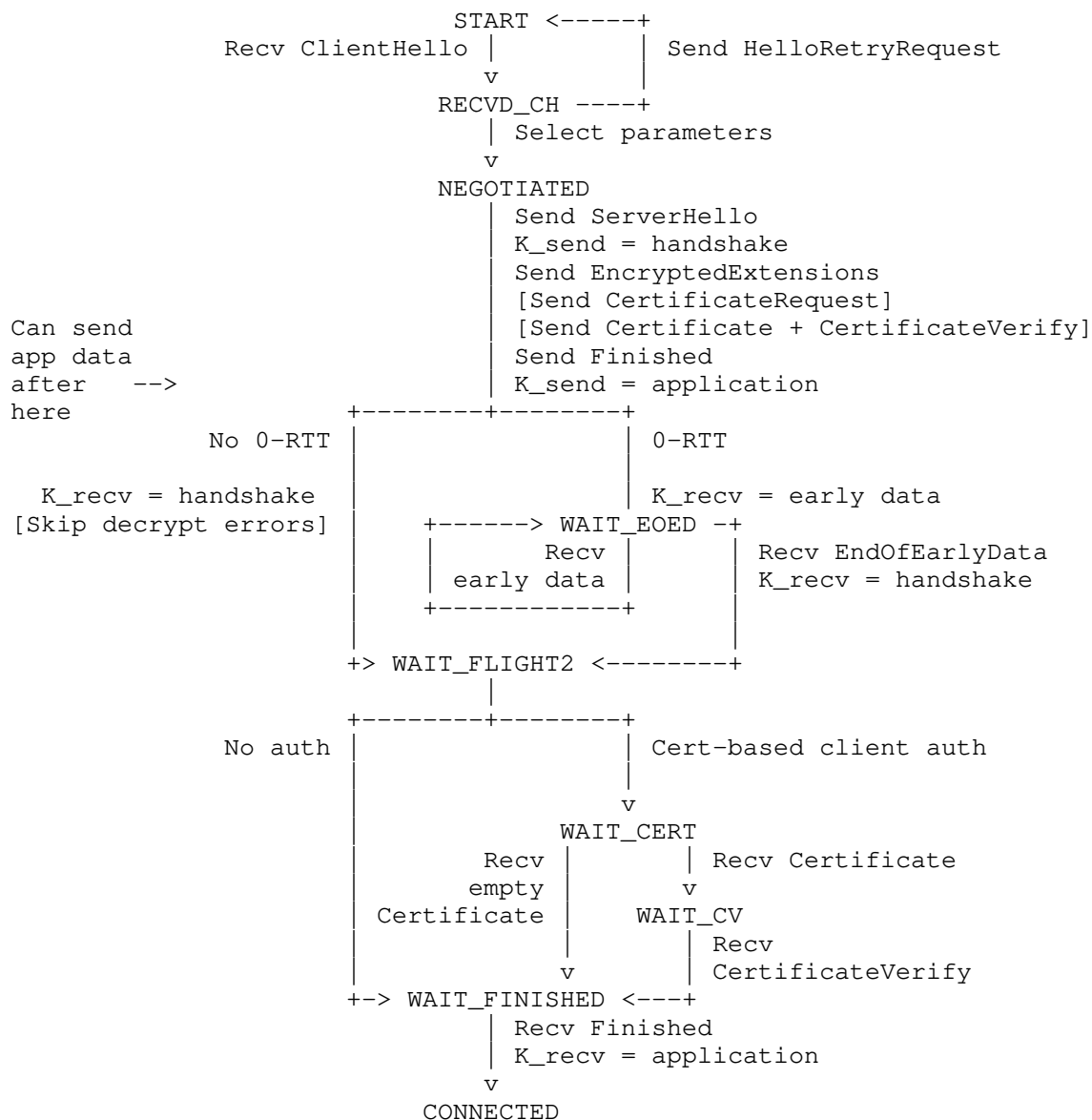
This appendix provides a summary of the legal state transitions for the client and server handshakes. State names (in all capitals, e.g., START) have no formal meaning but are provided for ease of comprehension. Actions which are taken only in certain circumstances are indicated in []. The notation "K_{send,recv} = foo" means "set the send/recv key to the given key".

A.1. Client



Note that with the transitions as shown above, clients may send alerts that derive from post-ServerHello messages in the clear or with the early data keys. If clients need to send such alerts, they SHOULD first rekey to the handshake keys if possible.

A.2. Server



Appendix B. Protocol Data Structures and Constant Values

This appendix provides the normative protocol types and the definitions for constants. Values listed as "**_RESERVED**" were used in previous versions of TLS and are listed here for completeness. TLS 1.3 implementations **MUST NOT** send them but might receive them from older TLS implementations.

B.1. Record Layer

```
enum {
    invalid(0),
    change_cipher_spec(20),
    alert(21),
    handshake(22),
    application_data(23),
    (255)
} ContentType;

struct {
    ContentType type;
    ProtocolVersion legacy_record_version;
    uint16 length;
    opaque fragment[TLSPlaintext.length];
} TLSPlaintext;

struct {
    opaque content[TLSPlaintext.length];
    ContentType type;
    uint8 zeros[length_of_padding];
} TLSInnerPlaintext;

struct {
    ContentType opaque_type = application_data; /* 23 */
    ProtocolVersion legacy_record_version = 0x0303; /* TLS v1.2 */
    uint16 length;
    opaque encrypted_record[TLSCiphertext.length];
} TLSCiphertext;
```

B.2. Alert Messages

```
enum { warning(1), fatal(2), (255) } AlertLevel;

enum {
    close_notify(0),
    unexpected_message(10),
    bad_record_mac(20),
    decryption_failed_RESERVED(21),
    record_overflow(22),
    decompression_failure_RESERVED(30),
    handshake_failure(40),
    no_certificate_RESERVED(41),
    bad_certificate(42),
    unsupported_certificate(43),
    certificate_revoked(44),
    certificate_expired(45),
    certificate_unknown(46),
    illegal_parameter(47),
    unknown_ca(48),
    access_denied(49),
    decode_error(50),
    decrypt_error(51),
    export_restriction_RESERVED(60),
    protocol_version(70),
    insufficient_security(71),
    internal_error(80),
    inappropriate_fallback(86),
    user_canceled(90),
    no_renegotiation_RESERVED(100),
    missing_extension(109),
    unsupported_extension(110),
    certificate_unobtainable_RESERVED(111),
    unrecognized_name(112),
    bad_certificate_status_response(113),
    bad_certificate_hash_value_RESERVED(114),
    unknown_psk_identity(115),
    certificate_required(116),
    no_application_protocol(120),
    (255)
} AlertDescription;

struct {
    AlertLevel level;
    AlertDescription description;
} Alert;
```

B.3. Handshake Protocol

```
enum {
    hello_request_RESERVED(0),
    client_hello(1),
    server_hello(2),
    hello_verify_request_RESERVED(3),
    new_session_ticket(4),
    end_of_early_data(5),
    hello_retry_request_RESERVED(6),
    encrypted_extensions(8),
    certificate(11),
    server_key_exchange_RESERVED(12),
    certificate_request(13),
    server_hello_done_RESERVED(14),
    certificate_verify(15),
    client_key_exchange_RESERVED(16),
    finished(20),
    certificate_url_RESERVED(21),
    certificate_status_RESERVED(22),
    supplemental_data_RESERVED(23),
    key_update(24),
    message_hash(254),
    (255)
} HandshakeType;

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

B.3.1. Key Exchange Messages

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

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

struct {
    ProtocolVersion legacy_version = 0x0303;    /* TLS v1.2 */
    Random random;
    opaque legacy_session_id<0..32>;
    CipherSuite cipher_suites<2..2^16-2>;
    opaque legacy_compression_methods<1..2^8-1>;
    Extension extensions<8..2^16-1>;
} ClientHello;

struct {
    ProtocolVersion legacy_version = 0x0303;    /* TLS v1.2 */
    Random random;
    opaque legacy_session_id_echo<0..32>;
    CipherSuite cipher_suite;
    uint8 legacy_compression_method = 0;
    Extension extensions<6..2^16-1>;
} ServerHello;

struct {
    ExtensionType extension_type;
    opaque extension_data<0..2^16-1>;
} Extension;

enum {
    server_name(0),                /* RFC 6066 */
    max_fragment_length(1),        /* RFC 6066 */
    status_request(5),             /* RFC 6066 */
    supported_groups(10),          /* RFC 8422, 7919 */
    signature_algorithms(13),      /* RFC 8446 */
    use_srtp(14),                  /* RFC 5764 */
    heartbeat(15),                 /* RFC 6520 */
    application_layer_protocol_negotiation(16), /* RFC 7301 */
    signed_certificate_timestamp(18), /* RFC 6962 */
    client_certificate_type(19),   /* RFC 7250 */
    server_certificate_type(20),   /* RFC 7250 */
    padding(21),                   /* RFC 7685 */
    pre_shared_key(41),            /* RFC 8446 */
    early_data(42),                /* RFC 8446 */
    supported_versions(43),        /* RFC 8446 */
    cookie(44),                    /* RFC 8446 */
    psk_key_exchange_modes(45),    /* RFC 8446 */
    certificate_authorities(47),   /* RFC 8446 */
    oid_filters(48),               /* RFC 8446 */
}
```

```
        post_handshake_auth(49),                /* RFC 8446 */
        signature_algorithms_cert(50),          /* RFC 8446 */
        key_share(51),                          /* RFC 8446 */
        (65535)
    } ExtensionType;

    struct {
        NamedGroup group;
        opaque key_exchange<1..2^16-1>;
    } KeyShareEntry;

    struct {
        KeyShareEntry client_shares<0..2^16-1>;
    } KeyShareClientHello;

    struct {
        NamedGroup selected_group;
    } KeyShareHelloRetryRequest;

    struct {
        KeyShareEntry server_share;
    } KeyShareServerHello;

    struct {
        uint8 legacy_form = 4;
        opaque X[coordinate_length];
        opaque Y[coordinate_length];
    } UncompressedPointRepresentation;

    enum { psk_ke(0), psk_dhe_ke(1), (255) } PskKeyExchangeMode;

    struct {
        PskKeyExchangeMode ke_modes<1..255>;
    } PskKeyExchangeModes;

    struct {} Empty;

    struct {
        select (Handshake.msg_type) {
            case new_session_ticket:    uint32 max_early_data_size;
            case client_hello:          Empty;
            case encrypted_extensions: Empty;
        };
    } EarlyDataIndication;

    struct {
        opaque identity<1..2^16-1>;
        uint32 obfuscated_ticket_age;
```

```
    } PskIdentity;

    opaque PskBinderEntry<32..255>;

    struct {
        PskIdentity identities<7..2^16-1>;
        PskBinderEntry binders<33..2^16-1>;
    } OfferedPsks;

    struct {
        select (Handshake.msg_type) {
            case client_hello: OfferedPsks;
            case server_hello: uint16 selected_identity;
        };
    } PreSharedKeyExtension;
```

B.3.1.1. Version Extension

```
    struct {
        select (Handshake.msg_type) {
            case client_hello:
                ProtocolVersion versions<2..254>;

            case server_hello: /* and HelloRetryRequest */
                ProtocolVersion selected_version;
        };
    } SupportedVersions;
```

B.3.1.2. Cookie Extension

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

B.3.1.3. Signature Algorithm Extension


```
enum {
    /* RSASSA-PKCS1-v1_5 algorithms */
    rsa_pkcs1_sha256(0x0401),
    rsa_pkcs1_sha384(0x0501),
    rsa_pkcs1_sha512(0x0601),

    /* ECDSA algorithms */
    ecdsa_secp256r1_sha256(0x0403),
    ecdsa_secp384r1_sha384(0x0503),
    ecdsa_secp521r1_sha512(0x0603),

    /* RSASSA-PSS algorithms with public key OID rsaEncryption */
    rsa_pss_rsae_sha256(0x0804),
    rsa_pss_rsae_sha384(0x0805),
    rsa_pss_rsae_sha512(0x0806),

    /* EdDSA algorithms */
    ed25519(0x0807),
    ed448(0x0808),

    /* RSASSA-PSS algorithms with public key OID RSASSA-PSS */
    rsa_pss_pss_sha256(0x0809),
    rsa_pss_pss_sha384(0x080a),
    rsa_pss_pss_sha512(0x080b),

    /* Legacy algorithms */
    rsa_pkcs1_sha1(0x0201),
    ecdsa_sha1(0x0203),

    /* Reserved Code Points */
    obsolete_RESERVED(0x0000..0x0200),
    dsa_sha1_RESERVED(0x0202),
    obsolete_RESERVED(0x0204..0x0400),
    dsa_sha256_RESERVED(0x0402),
    obsolete_RESERVED(0x0404..0x0500),
    dsa_sha384_RESERVED(0x0502),
    obsolete_RESERVED(0x0504..0x0600),
    dsa_sha512_RESERVED(0x0602),
    obsolete_RESERVED(0x0604..0x06FF),
    private_use(0xFE00..0xFFFF),
    (0xFFFF)
} SignatureScheme;

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

B.3.1.4. Supported Groups Extension

```
enum {
    unallocated_RESERVED(0x0000),

    /* Elliptic Curve Groups (ECDHE) */
    obsolete_RESERVED(0x0001..0x0016),
    secp256r1(0x0017), secp384r1(0x0018), secp521r1(0x0019),
    obsolete_RESERVED(0x001A..0x001C),
    x25519(0x001D), x448(0x001E),

    /* Finite Field Groups (DHE) */
    ffdhe2048(0x0100), ffdhe3072(0x0101), ffdhe4096(0x0102),
    ffdhe6144(0x0103), ffdhe8192(0x0104),

    /* Reserved Code Points */
    ffdhe_private_use(0x01FC..0x01FF),
    ecdhe_private_use(0xFE00..0xFEFF),
    obsolete_RESERVED(0xFF01..0xFF02),
    (0xFFFF)
} NamedGroup;

struct {
    NamedGroup named_group_list<2..2^16-1>;
} NamedGroupList;
```

Values within "obsolete_RESERVED" ranges are used in previous versions of TLS and MUST NOT be offered or negotiated by TLS 1.3 implementations. The obsolete curves have various known/theoretical weaknesses or have had very little usage, in some cases only due to unintentional server configuration issues. They are no longer considered appropriate for general use and should be assumed to be potentially unsafe. The set of curves specified here is sufficient for interoperability with all currently deployed and properly configured TLS implementations.

B.3.2. Server Parameters Messages

```
opaque DistinguishedName<1..2^16-1>;

struct {
    DistinguishedName authorities<3..2^16-1>;
} CertificateAuthoritiesExtension;

struct {
    opaque certificate_extension_oid<1..2^8-1>;
    opaque certificate_extension_values<0..2^16-1>;
} OIDFilter;

struct {
    OIDFilter filters<0..2^16-1>;
} OIDFilterExtension;

struct {} PostHandshakeAuth;

struct {
    Extension extensions<0..2^16-1>;
} EncryptedExtensions;

struct {
    opaque certificate_request_context<0..2^8-1>;
    Extension extensions<0..2^16-1>;
} CertificateRequest;
```

B.3.3. Authentication Messages

```
enum {
    X509(0),
    OpenPGP_RESERVED(1),
    RawPublicKey(2),
    (255)
} CertificateType;

struct {
    select (certificate_type) {
        case RawPublicKey:
            /* From RFC 7250 ASN.1_subjectPublicKeyInfo */
            opaque ASN1_subjectPublicKeyInfo<1..2^24-1>;

        case X509:
            opaque cert_data<1..2^24-1>;
    };
    Extension extensions<0..2^16-1>;
} CertificateEntry;

struct {
    opaque certificate_request_context<0..2^8-1>;
    CertificateEntry certificate_list<0..2^24-1>;
} Certificate;

struct {
    SignatureScheme algorithm;
    opaque signature<0..2^16-1>;
} CertificateVerify;

struct {
    opaque verify_data[Hash.length];
} Finished;
```

B.3.4. Ticket Establishment

```
struct {
    uint32 ticket_lifetime;
    uint32 ticket_age_add;
    opaque ticket_nonce<0..255>;
    opaque ticket<1..2^16-1>;
    Extension extensions<0..2^16-1>;
} NewSessionTicket;
```

B.3.5. Updating Keys

```

struct {} EndOfEarlyData;

enum {
    update_not_requested(0), update_requested(1), (255)
} KeyUpdateRequest;

struct {
    KeyUpdateRequest request_update;
} KeyUpdate;

```

B.4. Cipher Suites

A cipher suite defines the pair of the AEAD algorithm and hash algorithm to be used with HKDF. Cipher suite names follow the naming convention:

```
CipherSuite TLS_AEAD_HASH = VALUE;
```

Component	Contents
TLS	The string "TLS"
AEAD	The AEAD algorithm used for record protection
HASH	The hash algorithm used with HKDF
VALUE	The two byte ID assigned for this cipher suite

Table 4: Cipher Suite Name Structure

This specification defines the following cipher suites for use with TLS 1.3.

Description	Value
TLS_AES_128_GCM_SHA256	{0x13,0x01}
TLS_AES_256_GCM_SHA384	{0x13,0x02}
TLS_CHACHA20_POLY1305_SHA256	{0x13,0x03}
TLS_AES_128_CCM_SHA256	{0x13,0x04}
TLS_AES_128_CCM_8_SHA256	{0x13,0x05}

Table 5: Cipher Suite List

The corresponding AEAD algorithms AEAD_AES_128_GCM, AEAD_AES_256_GCM, and AEAD_AES_128_CCM are defined in [RFC5116]. AEAD_CHACHA20_POLY1305 is defined in [RFC8439]. AEAD_AES_128_CCM_8 is defined in [RFC6655]. The corresponding hash algorithms are defined in [SHS].

Although TLS 1.3 uses the same cipher suite space as previous versions of TLS, TLS 1.3 cipher suites are defined differently, only specifying the symmetric ciphers, and cannot be used for TLS 1.2. Similarly, cipher suites for TLS 1.2 and lower cannot be used with TLS 1.3.

New cipher suite values are assigned by IANA as described in Section 11.

Appendix C. Implementation Notes

The TLS protocol cannot prevent many common security mistakes. This appendix provides several recommendations to assist implementors. [RFC8448] provides test vectors for TLS 1.3 handshakes.

C.1. Random Number Generation and Seeding

TLS requires a cryptographically secure pseudorandom number generator (CSPRNG). In most cases, the operating system provides an appropriate facility such as /dev/urandom, which should be used absent other (e.g., performance) concerns. It is RECOMMENDED to use an existing CSPRNG implementation in preference to crafting a new one. Many adequate cryptographic libraries are already available under favorable license terms. Should those prove unsatisfactory, [RFC4086] provides guidance on the generation of random values.

TLS uses random values (1) in public protocol fields such as the public Random values in the ClientHello and ServerHello and (2) to generate keying material. With a properly functioning CSPRNG, this does not present a security problem, as it is not feasible to determine the CSPRNG state from its output. However, with a broken CSPRNG, it may be possible for an attacker to use the public output to determine the CSPRNG internal state and thereby predict the keying material, as documented in [CHECKOWAY] and [DSA-1571-1].

Implementations can provide extra security against this form of attack by using separate CSPRNGs to generate public and private values.

[RFC8937] describes a way way for security protocol implementations to augment their (pseudo)random number generators using a long-term private key and a deterministic signature function. This improves randomness from broken or otherwise subverted random number generators.

C.2. Certificates and Authentication

Implementations are responsible for verifying the integrity of certificates and should generally support certificate revocation messages. Absent a specific indication from an application profile, certificates should always be verified to ensure proper signing by a trusted certificate authority (CA). The selection and addition of trust anchors should be done very carefully. Users should be able to view information about the certificate and trust anchor. Applications SHOULD also enforce minimum and maximum key sizes. For example, certification paths containing keys or signatures weaker than 2048-bit RSA or 224-bit ECDSA are not appropriate for secure applications.

C.3. Implementation Pitfalls

Implementation experience has shown that certain parts of earlier TLS specifications are not easy to understand and have been a source of interoperability and security problems. Many of these areas have been clarified in this document but this appendix contains a short list of the most important things that require special attention from implementors.

TLS protocol issues:

- * Do you correctly handle handshake messages that are fragmented to multiple TLS records (see Section 5.1)? Do you correctly handle corner cases like a ClientHello that is split into several small fragments? Do you fragment handshake messages that exceed the

maximum fragment size? In particular, the Certificate and CertificateRequest handshake messages can be large enough to require fragmentation. Certificate compression as defined in [RFC8879] can be used to reduce the risk of fragmentation.

- * Do you ignore the TLS record layer version number in all unencrypted TLS records (see Appendix E)?
- * Have you ensured that all support for SSL, RC4, EXPORT ciphers, and MD5 (via the "signature_algorithms" extension) is completely removed from all possible configurations that support TLS 1.3 or later, and that attempts to use these obsolete capabilities fail correctly? (see Appendix E)?
- * Do you handle TLS extensions in ClientHellos correctly, including unknown extensions?
- * When the server has requested a client certificate but no suitable certificate is available, do you correctly send an empty Certificate message, instead of omitting the whole message (see Section 4.4.2)?
- * When processing the plaintext fragment produced by AEAD-Decrypt and scanning from the end for the ContentType, do you avoid scanning past the start of the cleartext in the event that the peer has sent a malformed plaintext of all zeros?
- * Do you properly ignore unrecognized cipher suites (Section 4.1.2), hello extensions (Section 4.2), named groups (Section 4.2.7), key shares (Section 4.2.8), supported versions (Section 4.2.1), and signature algorithms (Section 4.2.3) in the ClientHello?
- * As a server, do you send a HelloRetryRequest to clients which support a compatible (EC)DHE group but do not predict it in the "key_share" extension? As a client, do you correctly handle a HelloRetryRequest from the server?

Cryptographic details:

- * What countermeasures do you use to prevent timing attacks [TIMING]?
- * When using Diffie-Hellman key exchange, do you correctly preserve leading zero bytes in the negotiated key (see Section 7.4.1)?
- * Does your TLS client check that the Diffie-Hellman parameters sent by the server are acceptable (see Section 4.2.8.1)?

- * Do you use a strong and, most importantly, properly seeded random number generator (see Appendix C.1) when generating Diffie-Hellman private values, the ECDSA "k" parameter, and other security-critical values? It is RECOMMENDED that implementations implement "deterministic ECDSA" as specified in [RFC6979]. Note that purely deterministic ECC signatures such as deterministic ECDSA and EdDSA may be vulnerable to certain side-channel and fault injection attacks in easily accessible IoT devices.
- * Do you zero-pad Diffie-Hellman public key values and shared secrets to the group size (see Section 4.2.8.1 and Section 7.4.1)?
- * Do you verify signatures after making them, to protect against RSA-CRT key leaks [FW15]?

C.4. Client Tracking Prevention

Clients SHOULD NOT reuse a ticket for multiple connections. Reuse of a ticket allows passive observers to correlate different connections. Servers that issue tickets SHOULD offer at least as many tickets as the number of connections that a client might use; for example, a web browser using HTTP/1.1 [RFC7230] might open six connections to a server. Servers SHOULD issue new tickets with every connection. This ensures that clients are always able to use a new ticket when creating a new connection.

Offering a ticket to a server additionally allows the server to correlate different connections. This is possible independent of ticket reuse. Client applications SHOULD NOT offer tickets across connections that are meant to be uncorrelated. For example, [FETCH] defines network partition keys to separate cache lookups in web browsers.

C.5. Unauthenticated Operation

Previous versions of TLS offered explicitly unauthenticated cipher suites based on anonymous Diffie-Hellman. These modes have been deprecated in TLS 1.3. However, it is still possible to negotiate parameters that do not provide verifiable server authentication by several methods, including:

- * Raw public keys [RFC7250].
- * Using a public key contained in a certificate but without validation of the certificate chain or any of its contents.

Either technique used alone is vulnerable to man-in-the-middle attacks and therefore unsafe for general use. However, it is also possible to bind such connections to an external authentication mechanism via out-of-band validation of the server's public key, trust on first use, or a mechanism such as channel bindings (though the channel bindings described in [RFC5929] are not defined for TLS 1.3). If no such mechanism is used, then the connection has no protection against active man-in-the-middle attack; applications **MUST NOT** use TLS in such a way absent explicit configuration or a specific application profile.

Appendix D. Updates to TLS 1.2

To align with the names used this document, the following terms from [RFC5246] are renamed:

- * The master secret, computed in Section 8.1 of [RFC5246], is renamed to the main secret. It is referred to as `main_secret` in formulas and structures, instead of `master_secret`. However, the label parameter to the PRF function is left unchanged for compatibility.
- * The premaster secret is renamed to the preliminary secret. It is referred to as `preliminary_secret` in formulas and structures, instead of `pre_master_secret`.
- * The `PreMasterSecret` and `EncryptedPreMasterSecret` structures, defined in Section 7.4.7.1 of [RFC5246], are renamed to `PreliminarySecret` and `EncryptedPreliminarySecret`, respectively.

Correspondingly, the extension defined in [RFC7627] is renamed to the "Extended Main Secret" extension. The extension code point is renamed to `"extended_main_secret"`. The label parameter to the PRF function in Section 4 of [RFC7627] is left unchanged for compatibility.

Appendix E. Backward Compatibility

The TLS protocol provides a built-in mechanism for version negotiation between endpoints potentially supporting different versions of TLS.

TLS 1.x and SSL 3.0 use compatible `ClientHello` messages. Servers can also handle clients trying to use future versions of TLS as long as the `ClientHello` format remains compatible and there is at least one protocol version supported by both the client and the server.

Prior versions of TLS used the record layer version number (TLSPlaintext.legacy_record_version and TLSCiphertext.legacy_record_version) for various purposes. As of TLS 1.3, this field is deprecated. The value of TLSPlaintext.legacy_record_version MUST be ignored by all implementations. The value of TLSCiphertext.legacy_record_version is included in the additional data for deprotection but MAY otherwise be ignored or MAY be validated to match the fixed constant value. Version negotiation is performed using only the handshake versions (ClientHello.legacy_version and ServerHello.legacy_version, as well as the ClientHello, HelloRetryRequest, and ServerHello "supported_versions" extensions). In order to maximize interoperability with older endpoints, implementations that negotiate the use of TLS 1.0-1.2 SHOULD set the record layer version number to the negotiated version for the ServerHello and all records thereafter.

For maximum compatibility with previously non-standard behavior and misconfigured deployments, all implementations SHOULD support validation of certification paths based on the expectations in this document, even when handling prior TLS versions' handshakes (see Section 4.4.2.2).

TLS 1.2 and prior supported an "Extended Main Secret" [RFC7627] extension which digested large parts of the handshake transcript into the secret and derived keys. Note this extension was renamed in Appendix D. Because TLS 1.3 always hashes in the transcript up to the server Finished, implementations which support both TLS 1.3 and earlier versions SHOULD indicate the use of the Extended Main Secret extension in their APIs whenever TLS 1.3 is used.

E.1. Negotiating with an Older Server

A TLS 1.3 client who wishes to negotiate with servers that do not support TLS 1.3 will send a normal TLS 1.3 ClientHello containing 0x0303 (TLS 1.2) in ClientHello.legacy_version but with the correct version(s) in the "supported_versions" extension. If the server does not support TLS 1.3, it will respond with a ServerHello containing an older version number. If the client agrees to use this version, the negotiation will proceed as appropriate for the negotiated protocol. A client using a ticket for resumption SHOULD initiate the connection using the version that was previously negotiated.

Note that 0-RTT data is not compatible with older servers and SHOULD NOT be sent absent knowledge that the server supports TLS 1.3. See Appendix E.3.

If the version chosen by the server is not supported by the client (or is not acceptable), the client MUST abort the handshake with a "protocol_version" alert.

Some legacy server implementations are known to not implement the TLS specification properly and might abort connections upon encountering TLS extensions or versions which they are not aware of. Interoperability with buggy servers is a complex topic beyond the scope of this document. Multiple connection attempts may be required in order to negotiate a backward-compatible connection; however, this practice is vulnerable to downgrade attacks and is NOT RECOMMENDED.

E.2. Negotiating with an Older Client

A TLS server can also receive a ClientHello indicating a version number smaller than its highest supported version. If the "supported_versions" extension is present, the server MUST negotiate using that extension as described in Section 4.2.1. If the "supported_versions" extension is not present, the server MUST negotiate the minimum of ClientHello.legacy_version and TLS 1.2. For example, if the server supports TLS 1.0, 1.1, and 1.2, and legacy_version is TLS 1.0, the server will proceed with a TLS 1.0 ServerHello. If the "supported_versions" extension is absent and the server only supports versions greater than ClientHello.legacy_version, the server MUST abort the handshake with a "protocol_version" alert.

Note that earlier versions of TLS did not clearly specify the record layer version number value in all cases (TLSPlaintext.legacy_record_version). Servers will receive various TLS 1.x versions in this field, but its value MUST always be ignored.

E.3. 0-RTT Backward Compatibility

0-RTT data is not compatible with older servers. An older server will respond to the ClientHello with an older ServerHello, but it will not correctly skip the 0-RTT data and will fail to complete the handshake. This can cause issues when a client attempts to use 0-RTT, particularly against multi-server deployments. For example, a deployment could deploy TLS 1.3 gradually with some servers implementing TLS 1.3 and some implementing TLS 1.2, or a TLS 1.3 deployment could be downgraded to TLS 1.2.

A client that attempts to send 0-RTT data MUST fail a connection if it receives a ServerHello with TLS 1.2 or older. It can then retry the connection with 0-RTT disabled. To avoid a downgrade attack, the client SHOULD NOT disable TLS 1.3, only 0-RTT.

To avoid this error condition, multi-server deployments SHOULD ensure a uniform and stable deployment of TLS 1.3 without 0-RTT prior to enabling 0-RTT.

E.4. Middlebox Compatibility Mode

Field measurements [Ben17a] [Ben17b] [Res17a] [Res17b] have found that a significant number of middleboxes misbehave when a TLS client/server pair negotiates TLS 1.3. Implementations can increase the chance of making connections through those middleboxes by making the TLS 1.3 handshake look more like a TLS 1.2 handshake:

- * The client always provides a non-empty session ID in the ClientHello, as described in the legacy_session_id section of Section 4.1.2.
- * If not offering early data, the client sends a dummy change_cipher_spec record (see the third paragraph of Section 5) immediately before its second flight. This may either be before its second ClientHello or before its encrypted handshake flight. If offering early data, the record is placed immediately after the first ClientHello.
- * The server sends a dummy change_cipher_spec record immediately after its first handshake message. This may either be after a ServerHello or a HelloRetryRequest.

When put together, these changes make the TLS 1.3 handshake resemble TLS 1.2 session resumption, which improves the chance of successfully connecting through middleboxes. This "compatibility mode" is partially negotiated: the client can opt to provide a session ID or not, and the server has to echo it. Either side can send change_cipher_spec at any time during the handshake, as they must be ignored by the peer, but if the client sends a non-empty session ID, the server MUST send the change_cipher_spec as described in this appendix.

E.5. Security Restrictions Related to Backward Compatibility

Implementations negotiating the use of older versions of TLS SHOULD prefer forward secret and AEAD cipher suites, when available.

The security of RC4 cipher suites is considered insufficient for the reasons cited in [RFC7465]. Implementations MUST NOT offer or negotiate RC4 cipher suites for any version of TLS for any reason.

Old versions of TLS permitted the use of very low strength ciphers. Ciphers with a strength less than 112 bits MUST NOT be offered or negotiated for any version of TLS for any reason.

The security of SSL 2.0 [SSL2], SSL 3.0 [RFC6101], TLS 1.0 [RFC2246], and TLS 1.1 [RFC4346] are considered insufficient for the reasons enumerated in [RFC6176], [RFC7568], and [RFC8996] and they MUST NOT be negotiated for any reason.

Implementations MUST NOT send an SSL version 2.0 compatible CLIENT-HELLO. Implementations MUST NOT negotiate TLS 1.3 or later using an SSL version 2.0 compatible CLIENT-HELLO. Implementations are NOT RECOMMENDED to accept an SSL version 2.0 compatible CLIENT-HELLO in order to negotiate older versions of TLS.

Implementations MUST NOT send a ClientHello.legacy_version or ServerHello.legacy_version set to 0x0300 or less. Any endpoint receiving a Hello message with ClientHello.legacy_version or ServerHello.legacy_version set to 0x0300 MUST abort the handshake with a "protocol_version" alert.

Implementations MUST NOT send any records with a version less than 0x0300. Implementations SHOULD NOT accept any records with a version less than 0x0300 (but may inadvertently do so if the record version number is ignored completely).

Implementations MUST NOT use the Truncated HMAC extension, defined in Section 7 of [RFC6066], as it is not applicable to AEAD algorithms and has been shown to be insecure in some scenarios.

Appendix F. Overview of Security Properties

A complete security analysis of TLS is outside the scope of this document. In this appendix, we provide an informal description of the desired properties as well as references to more detailed work in the research literature which provides more formal definitions.

We cover properties of the handshake separately from those of the record layer.

F.1. Handshake

The TLS handshake is an Authenticated Key Exchange (AKE) protocol which is intended to provide both one-way authenticated (server-only) and mutually authenticated (client and server) functionality. At the completion of the handshake, each side outputs its view of the following values:

- * A set of "session keys" (the various secrets derived from the main secret) from which can be derived a set of working keys.
- * A set of cryptographic parameters (algorithms, etc.).
- * The identities of the communicating parties.

We assume the attacker to be an active network attacker, which means it has complete control over the network used to communicate between the parties [RFC3552]. Even under these conditions, the handshake should provide the properties listed below. Note that these properties are not necessarily independent, but reflect the protocol consumers' needs.

Establishing the same session keys: The handshake needs to output the same set of session keys on both sides of the handshake, provided that it completes successfully on each endpoint (see [CK01]; Definition 1, part 1).

Secrecy of the session keys: The shared session keys should be known only to the communicating parties and not to the attacker (see [CK01]; Definition 1, part 2). Note that in a unilaterally authenticated connection, the attacker can establish its own session keys with the server, but those session keys are distinct from those established by the client.

Peer Authentication: The client's view of the peer identity should reflect the server's identity. If the client is authenticated, the server's view of the peer identity should match the client's identity.

Uniqueness of the session keys: Any two distinct handshakes should produce distinct, unrelated session keys. Individual session keys produced by a handshake should also be distinct and independent.

Downgrade Protection: The cryptographic parameters should be the same on both sides and should be the same as if the peers had been communicating in the absence of an attack (see [BBFGKZ16]; Definitions 8 and 9).

Forward secret with respect to long-term keys: If the long-term keying material (in this case the signature keys in certificate-based authentication modes or the external/resumption PSK in PSK with (EC)DHE modes) is compromised after the handshake is complete, this does not compromise the security of the session key (see [DOW92]), as long as the session key itself has been erased. The forward secrecy property is not satisfied when PSK is used in the "psk_ke" PskKeyExchangeMode.

Key Compromise Impersonation (KCI) resistance: In a mutually authenticated connection with certificates, compromising the long-term secret of one actor should not break that actor's authentication of their peer in the given connection (see [HGFS15]). For example, if a client's signature key is compromised, it should not be possible to impersonate arbitrary servers to that client in subsequent handshakes.

Protection of endpoint identities: The server's identity (certificate) should be protected against passive attackers. The client's identity (certificate) should be protected against both passive and active attackers. This property does not hold for cipher suites without confidentiality; while this specification does not define any such cipher suites, other documents may do so.

Informally, the signature-based modes of TLS 1.3 provide for the establishment of a unique, secret, shared key established by an (EC)DHE key exchange and authenticated by the server's signature over the handshake transcript, as well as tied to the server's identity by a MAC. If the client is authenticated by a certificate, it also signs over the handshake transcript and provides a MAC tied to both identities. [SIGMA] describes the design and analysis of this type of key exchange protocol. If fresh (EC)DHE keys are used for each connection, then the output keys are forward secret.

The external PSK and resumption PSK bootstrap from a long-term shared secret into a unique per-connection set of short-term session keys. This secret may have been established in a previous handshake. If PSK with (EC)DHE key establishment is used, these session keys will also be forward secret. The resumption PSK has been designed so that the resumption secret computed by connection N and needed to form connection N+1 is separate from the traffic keys used by connection N, thus providing forward secrecy between the connections. In addition, if multiple tickets are established on the same connection, they are associated with different keys, so compromise of the PSK associated with one ticket does not lead to the compromise of connections established with PSKs associated with other tickets. This property is most interesting if tickets are stored in a database (and so can be deleted) rather than if they are self-encrypted.

Forward secrecy limits the effect of key leakage in one direction (compromise of a key at time T2 does not compromise some key at time T1 where $T1 < T2$). Protection in the other direction (compromise at time T1 does not compromise keys at time T2) can be achieved by rerunning EC(DHE). If a long-term authentication key has been compromised, a full handshake with EC(DHE) gives protection against passive attackers. If the `resumption_master_secret` has been compromised, a resumption handshake with EC(DHE) gives protection

against passive attackers and a full handshake with EC(DHE) gives protection against active attackers. If a traffic secret has been compromised, any handshake with EC(DHE) gives protection against active attackers. Using the terms in [RFC7624], forward secrecy without rerunning EC(DHE) does not stop an attacker from doing static key exfiltration. After key exfiltration of `application_traffic_secret_N`, an attacker can e.g., passively eavesdrop on all future data sent on the connection including data encrypted with `application_traffic_secret_N+1`, `application_traffic_secret_N+2`, etc. Frequently rerunning EC(DHE) forces an attacker to do dynamic key exfiltration (or content exfiltration).

The PSK binder value forms a binding between a PSK and the current handshake, as well as between the session where the PSK was established and the current session. This binding transitively includes the original handshake transcript, because that transcript is digested into the values which produce the resumption secret. This requires that both the KDF used to produce the resumption secret and the MAC used to compute the binder be collision resistant. See Appendix F.1.1 for more on this. Note: The binder does not cover the binder values from other PSKs, though they are included in the Finished MAC.

Note: TLS does not currently permit the server to send a `certificate_request` message in non-certificate-based handshakes (e.g., PSK). If this restriction were to be relaxed in future, the client's signature would not cover the server's certificate directly. However, if the PSK was established through a `NewSessionTicket`, the client's signature would transitively cover the server's certificate through the PSK binder. [PSK-FINISHED] describes a concrete attack on constructions that do not bind to the server's certificate (see also [Kraw16]). It is unsafe to use certificate-based client authentication when the client might potentially share the same PSK/key-id pair with two different endpoints. Implementations MUST NOT combine external PSKs with certificate-based authentication of either the client or server. Future specifications MAY provide an extension to permit this.

If an exporter is used, then it produces values which are unique and secret (because they are generated from a unique session key). Exporters computed with different labels and contexts are computationally independent, so it is not feasible to compute one from another or the session secret from the exported value. Note: Exporters can produce arbitrary-length values; if exporters are to be used as channel bindings, the exported value MUST be large enough to provide collision resistance. The exporters provided in TLS 1.3 are derived from the same Handshake Contexts as the early traffic keys

and the application traffic keys, respectively, and thus have similar security properties. Note that they do not include the client's certificate; future applications which wish to bind to the client's certificate may need to define a new exporter that includes the full handshake transcript.

For all handshake modes, the Finished MAC (and, where present, the signature) prevents downgrade attacks. In addition, the use of certain bytes in the random nonces as described in Section 4.1.3 allows the detection of downgrade to previous TLS versions. See [BBFGKZ16] for more details on TLS 1.3 and downgrade.

As soon as the client and the server have exchanged enough information to establish shared keys, the remainder of the handshake is encrypted, thus providing protection against passive attackers, even if the computed shared key is not authenticated. Because the server authenticates before the client, the client can ensure that if it authenticates to the server, it only reveals its identity to an authenticated server. Note that implementations must use the provided record-padding mechanism during the handshake to avoid leaking information about the identities due to length. The client's proposed PSK identities are not encrypted, nor is the one that the server selects.

F.1.1. Key Derivation and HKDF

Key derivation in TLS 1.3 uses HKDF as defined in [RFC5869] and its two components, HKDF-Extract and HKDF-Expand. The full rationale for the HKDF construction can be found in [Kraw10] and the rationale for the way it is used in TLS 1.3 in [KW16]. Throughout this document, each application of HKDF-Extract is followed by one or more invocations of HKDF-Expand. This ordering should always be followed (including in future revisions of this document); in particular, one SHOULD NOT use an output of HKDF-Extract as an input to another application of HKDF-Extract without an HKDF-Expand in between. Multiple applications of HKDF-Expand to some of the same inputs are allowed as long as these are differentiated via the key and/or the labels.

Note that HKDF-Expand implements a pseudorandom function (PRF) with both inputs and outputs of variable length. In some of the uses of HKDF in this document (e.g., for generating exporters and the `resumption_secret`), it is necessary that the application of HKDF-Expand be collision resistant; namely, it should be infeasible to find two different inputs to HKDF-Expand that output the same value. This requires the underlying hash function to be collision resistant and the output length from HKDF-Expand to be of size at least 256 bits (or as much as needed for the hash function to prevent finding collisions).

F.1.2. Certificate-Based Client Authentication

A client that has sent certificate-based authentication data to a server, either during the handshake or in post-handshake authentication, cannot be sure whether the server afterwards considers the client to be authenticated or not. If the client needs to determine if the server considers the connection to be unilaterally or mutually authenticated, this has to be provisioned by the application layer. See [CHHSV17] for details. In addition, the analysis of post-handshake authentication from [Kraw16] shows that the client identified by the certificate sent in the post-handshake phase possesses the traffic key. This party is therefore the client that participated in the original handshake or one to whom the original client delegated the traffic key (assuming that the traffic key has not been compromised).

F.1.3. 0-RTT

The 0-RTT mode of operation generally provides security properties similar to those of 1-RTT data, with the two exceptions that the 0-RTT encryption keys do not provide full forward secrecy and that the server is not able to guarantee uniqueness of the handshake (non-replayability) without keeping potentially undue amounts of state. See Section 8 for mechanisms to limit the exposure to replay.

F.1.4. Exporter Independence

The `exporter_secret` and `early_exporter_secret` are derived to be independent of the traffic keys and therefore do not represent a threat to the security of traffic encrypted with those keys. However, because these secrets can be used to compute any exporter value, they SHOULD be erased as soon as possible. If the total set of exporter labels is known, then implementations SHOULD pre-compute the inner Derive-Secret stage of the exporter computation for all those labels, then erase the `[early_]exporter_secret`, followed by each inner values as soon as it is known that it will not be needed again.

F.1.5. Post-Compromise Security

TLS does not provide security for handshakes which take place after the peer's long-term secret (signature key or external PSK) is compromised. It therefore does not provide post-compromise security [CCG16], sometimes also referred to as backwards or future secrecy. This is in contrast to KCI resistance, which describes the security guarantees that a party has after its own long-term secret has been compromised.

F.1.6. External References

The reader should refer to the following references for analysis of the TLS handshake: [DFGS15], [CHSV16], [DFGS16], [KW16], [Kraw16], [FGSW16], [LXZFH16], [FG17], and [BBK17].

F.2. Record Layer

The record layer depends on the handshake producing strong traffic secrets which can be used to derive bidirectional encryption keys and nonces. Assuming that is true, and the keys are used for no more data than indicated in Section 5.5, then the record layer should provide the following guarantees:

Confidentiality: An attacker should not be able to determine the plaintext contents of a given record.

Integrity: An attacker should not be able to craft a new record which is different from an existing record which will be accepted by the receiver.

Order protection/non-replayability: An attacker should not be able to cause the receiver to accept a record which it has already accepted or cause the receiver to accept record N+1 without having first processed record N.

Length concealment: Given a record with a given external length, the attacker should not be able to determine the amount of the record that is content versus padding.

Forward secrecy after key change: If the traffic key update mechanism described in Section 4.6.3 has been used and the previous generation key is deleted, an attacker who compromises the endpoint should not be able to decrypt traffic encrypted with the old key.

Informally, TLS 1.3 provides these properties by AEAD-protecting the plaintext with a strong key. AEAD encryption [RFC5116] provides confidentiality and integrity for the data. Non-replayability is provided by using a separate nonce for each record, with the nonce being derived from the record sequence number (Section 5.3), with the sequence number being maintained independently at both sides; thus records which are delivered out of order result in AEAD deprotection failures. In order to prevent mass cryptanalysis when the same plaintext is repeatedly encrypted by different users under the same key (as is commonly the case for HTTP), the nonce is formed by mixing the sequence number with a secret per-connection initialization vector derived along with the traffic keys. See [BT16] for analysis of this construction.

The rekeying technique in TLS 1.3 (see Section 7.2) follows the construction of the serial generator as discussed in [REKEY], which shows that rekeying can allow keys to be used for a larger number of encryptions than without rekeying. This relies on the security of the HKDF-Expand-Label function as a pseudorandom function (PRF). In addition, as long as this function is truly one way, it is not possible to compute traffic keys from prior to a key change (forward secrecy).

TLS does not provide security for data which is communicated on a connection after a traffic secret of that connection is compromised. That is, TLS does not provide post-compromise security/future secrecy/backward secrecy with respect to the traffic secret. Indeed, an attacker who learns a traffic secret can compute all future traffic secrets on that connection. Systems which want such guarantees need to do a fresh handshake and establish a new connection with an (EC)DHE exchange.

F.2.1. External References

The reader should refer to the following references for analysis of the TLS record layer: [BMMRT15], [BT16], [BDFKPPRSZZ16], [BBK17], and [PS18].

F.3. Traffic Analysis

TLS is susceptible to a variety of traffic analysis attacks based on observing the length and timing of encrypted packets [CLINIC] [HCJC16]. This is particularly easy when there is a small set of possible messages to be distinguished, such as for a video server hosting a fixed corpus of content, but still provides usable information even in more complicated scenarios.

TLS does not provide any specific defenses against this form of attack but does include a padding mechanism for use by applications: The plaintext protected by the AEAD function consists of content plus variable-length padding, which allows the application to produce arbitrary-length encrypted records as well as padding-only cover traffic to conceal the difference between periods of transmission and periods of silence. Because the padding is encrypted alongside the actual content, an attacker cannot directly determine the length of the padding, but may be able to measure it indirectly by the use of timing channels exposed during record processing (i.e., seeing how long it takes to process a record or trickling in records to see which ones elicit a response from the server). In general, it is not known how to remove all of these channels because even a constant-time padding removal function will likely feed the content into data-dependent functions. At minimum, a fully constant-time server or client would require close cooperation with the application-layer protocol implementation, including making that higher-level protocol constant time.

Note: Robust traffic analysis defenses will likely lead to inferior performance due to delays in transmitting packets and increased traffic volume.

F.4. Side Channel Attacks

In general, TLS does not have specific defenses against side-channel attacks (i.e., those which attack the communications via secondary channels such as timing), leaving those to the implementation of the relevant cryptographic primitives. However, certain features of TLS are designed to make it easier to write side-channel resistant code:

- * Unlike previous versions of TLS which used a composite MAC-then-encrypt structure, TLS 1.3 only uses AEAD algorithms, allowing implementations to use self-contained constant-time implementations of those primitives.
- * TLS uses a uniform "bad_record_mac" alert for all decryption errors, which is intended to prevent an attacker from gaining piecewise insight into portions of the message. Additional resistance is provided by terminating the connection on such errors; a new connection will have different cryptographic material, preventing attacks against the cryptographic primitives that require multiple trials.

Information leakage through side channels can occur at layers above TLS, in application protocols and the applications that use them. Resistance to side-channel attacks depends on applications and application protocols separately ensuring that confidential information is not inadvertently leaked.

F.5. Replay Attacks on 0-RTT

Replayable 0-RTT data presents a number of security threats to TLS-using applications, unless those applications are specifically engineered to be safe under replay (minimally, this means idempotent, but in many cases may also require other stronger conditions, such as constant-time response). Potential attacks include:

- * Duplication of actions which cause side effects (e.g., purchasing an item or transferring money) to be duplicated, thus harming the site or the user.
- * Attackers can store and replay 0-RTT messages in order to reorder them with respect to other messages (e.g., moving a delete to after a create).
- * Exploiting cache timing behavior to discover the content of 0-RTT messages by replaying a 0-RTT message to a different cache node and then using a separate connection to measure request latency, to see if the two requests address the same resource.

If data can be replayed a large number of times, additional attacks become possible, such as making repeated measurements of the speed of cryptographic operations. In addition, they may be able to overload rate-limiting systems. For a further description of these attacks, see [Mac17].

Ultimately, servers have the responsibility to protect themselves against attacks employing 0-RTT data replication. The mechanisms described in Section 8 are intended to prevent replay at the TLS layer but do not provide complete protection against receiving multiple copies of client data. TLS 1.3 falls back to the 1-RTT handshake when the server does not have any information about the client, e.g., because it is in a different cluster which does not share state or because the ticket has been deleted as described in Section 8.1. If the application-layer protocol retransmits data in this setting, then it is possible for an attacker to induce message duplication by sending the ClientHello to both the original cluster (which processes the data immediately) and another cluster which will fall back to 1-RTT and process the data upon application-layer replay. The scale of this attack is limited by the client's willingness to retry transactions and therefore only allows a limited amount of duplication, with each copy appearing as a new connection at the server.

If implemented correctly, the mechanisms described in Section 8.1 and Section 8.2 prevent a replayed ClientHello and its associated 0-RTT data from being accepted multiple times by any cluster with consistent state; for servers which limit the use of 0-RTT to one cluster for a single ticket, then a given ClientHello and its associated 0-RTT data will only be accepted once. However, if state is not completely consistent, then an attacker might be able to have multiple copies of the data be accepted during the replication window. Because clients do not know the exact details of server behavior, they **MUST NOT** send messages in early data which are not safe to have replayed and which they would not be willing to retry across multiple 1-RTT connections.

Application protocols **MUST NOT** use 0-RTT data without a profile that defines its use. That profile needs to identify which messages or interactions are safe to use with 0-RTT and how to handle the situation when the server rejects 0-RTT and falls back to 1-RTT.

In addition, to avoid accidental misuse, TLS implementations **MUST NOT** enable 0-RTT (either sending or accepting) unless specifically requested by the application and **MUST NOT** automatically resend 0-RTT data if it is rejected by the server unless instructed by the application. Server-side applications may wish to implement special processing for 0-RTT data for some kinds of application traffic (e.g., abort the connection, request that data be resent at the application layer, or delay processing until the handshake completes). In order to allow applications to implement this kind of processing, TLS implementations **MUST** provide a way for the application to determine if the handshake has completed.

F.5.1. Replay and Exporters

Replays of the ClientHello produce the same early exporter, thus requiring additional care by applications which use these exporters. In particular, if these exporters are used as an authentication channel binding (e.g., by signing the output of the exporter) an attacker who compromises the PSK can transplant authenticators between connections without compromising the authentication key.

In addition, the early exporter SHOULD NOT be used to generate server-to-client encryption keys because that would entail the reuse of those keys. This parallels the use of the early application traffic keys only in the client-to-server direction.

F.6. PSK Identity Exposure

Because implementations respond to an invalid PSK binder by aborting the handshake, it may be possible for an attacker to verify whether a given PSK identity is valid. Specifically, if a server accepts both external-PSK and certificate-based handshakes, a valid PSK identity will result in a failed handshake, whereas an invalid identity will just be skipped and result in a successful certificate handshake. Servers which solely support PSK handshakes may be able to resist this form of attack by treating the cases where there is no valid PSK identity and where there is an identity but it has an invalid binder identically.

F.7. Sharing PSKs

TLS 1.3 takes a conservative approach to PSKs by binding them to a specific KDF. By contrast, TLS 1.2 allows PSKs to be used with any hash function and the TLS 1.2 PRF. Thus, any PSK which is used with both TLS 1.2 and TLS 1.3 must be used with only one hash in TLS 1.3, which is less than optimal if users want to provision a single PSK. The constructions in TLS 1.2 and TLS 1.3 are different, although they are both based on HMAC. While there is no known way in which the same PSK might produce related output in both versions, only limited analysis has been done. Implementations can ensure safety from cross-protocol related output by not reusing PSKs between TLS 1.3 and TLS 1.2.

F.8. Attacks on Static RSA

Although TLS 1.3 does not use RSA key transport and so is not directly susceptible to Bleichenbacher-type attacks [Blei98] if TLS 1.3 servers also support static RSA in the context of previous versions of TLS, then it may be possible to impersonate the server for TLS 1.3 connections [JSS15]. TLS 1.3 implementations can prevent this attack by disabling support for static RSA across all versions of TLS. In principle, implementations might also be able to separate certificates with different keyUsage bits for static RSA decryption and RSA signature, but this technique relies on clients refusing to accept signatures using keys in certificates that do not have the digitalSignature bit set, and many clients do not enforce this restriction.

Appendix G. Changes Since -00

[[RFC EDITOR: Please remove in final RFC.]]

- * Update TLS 1.2 terminology
- * Specify "certificate-based" client authentication
- * Clarify that privacy guarantees don't apply when you have null encryption
- * Shorten some names
- * Address tracking implications of resumption

Contributors

Martin Abadi
University of California, Santa Cruz
abadi@cs.ucsc.edu

Christopher Allen
(co-editor of TLS 1.0)
Alacrity Ventures
ChristopherA@AlacrityManagement.com

Nimrod Aviram
Tel Aviv University
nimrod.aviram@gmail.com

Richard Barnes
Cisco
rlb@ipv.sx

Steven M. Bellovin
Columbia University
smb@cs.columbia.edu

David Benjamin
Google
davidben@google.com

Benjamin Beurdouche
INRIA & Microsoft Research
benjamin.beurdouche@ens.fr

Karthikeyan Bhargavan
(editor of [RFC7627])
INRIA
karthikeyan.bhargavan@inria.fr

Simon Blake-Wilson
(co-author of [RFC4492])
BCI
sblakewilson@bcisse.com

Nelson Bolyard
(co-author of [RFC4492])
Sun Microsystems, Inc.
nelson@bolyard.com

Ran Canetti
IBM
canetti@watson.ibm.com

Matt Caswell
OpenSSL
matt@openssl.org

Stephen Checkoway
University of Illinois at Chicago
sfc@uic.edu

Pete Chown
Skygate Technology Ltd
pc@skygate.co.uk

Katriel Cohn-Gordon
University of Oxford
me@katriel.co.uk

Cas Cremers

University of Oxford
cas.cremers@cs.ox.ac.uk

Antoine Delignat-Lavaud
(co-author of [RFC7627])
INRIA
antdl@microsoft.com

Tim Dierks
(co-author of TLS 1.0, co-editor of TLS 1.1 and 1.2)
Independent
tim@dierks.org

Roelof DuToit
Symantec Corporation
roelof_dutoit@symantec.com

Taher Elgamal
Securify
taher@securify.com

Pasi Eronen
Nokia
pasi.eronen@nokia.com

Cedric Fournet
Microsoft
fournet@microsoft.com

Anil Gangolli
anil@busybuddha.org

David M. Garrett
dave@nulldereference.com

Illya Gerasymchuk
Independent
illya@iluxonchik.me

Alessandro Ghedini
Cloudflare Inc.
alessandro@cloudflare.com

Daniel Kahn Gillmor
ACLU
dkg@fifthhorseman.net

Matthew Green

Johns Hopkins University
mgreen@cs.jhu.edu

Jens Guballa
ETAS
jens.guballa@etas.com

Felix Guenther
TU Darmstadt
mail@felixguenther.info

Vipul Gupta
(co-author of [RFC4492])
Sun Microsystems Laboratories
vipul.gupta@sun.com

Chris Hawk
(co-author of [RFC4492])
Corriente Networks LLC
chris@corriente.net

Kipp Hickman

Alfred Hoenes

David Hopwood
Independent Consultant
david.hopwood@blueyonder.co.uk

Marko Horvat
MPI-SWS
mhorvat@mpi-sws.org

Jonathan Hoyland
Royal Holloway, University of London
jonathan.hoyland@gmail.com

Subodh Iyengar
Facebook
subodh@fb.com

Benjamin Kaduk
Akamai Technologies
kaduk@mit.edu

Hubert Kario
Red Hat Inc.
hkario@redhat.com

Phil Karlton
(co-author of SSL 3.0)

Leon Klingele
Independent
mail@leonklingele.de

Paul Kocher
(co-author of SSL 3.0)
Cryptography Research
paul@cryptography.com

Hugo Krawczyk
IBM
hugokraw@us.ibm.com

Adam Langley
(co-author of [RFC7627])
Google
agl@google.com

Olivier Levillain
ANSSI
olivier.levillain@ssi.gouv.fr

Xiaoyin Liu
University of North Carolina at Chapel Hill
xiaoyin.l@outlook.com

Ilari Liusvaara
Independent
ilariliusvaara@welho.com

Atul Luykx
K.U. Leuven
atul.luykx@kuleuven.be

Colm MacCarthaigh
Amazon Web Services
colm@allcosts.net

Carl Mehner
USAA
carl.mehner@usaa.com

Jan Mikkelsen
Transactionware
janm@transactionware.com

Bodo Moeller
(co-author of [RFC4492])
Google
bodo@acm.org

Kyle Nekritz
Facebook
knekritz@fb.com

Erik Nygren
Akamai Technologies
erik+ietf@nygren.org

Magnus Nystrom
Microsoft
mnystrom@microsoft.com

Kazuho Oku
DeNA Co., Ltd.
kazuhooku@gmail.com

Kenny Paterson
Royal Holloway, University of London
kenny.paterson@rhul.ac.uk

Christopher Patton
University of Florida
cjpatton@ufl.edu

Alfredo Pironti
(co-author of [RFC7627])
INRIA
alfredo.pironti@inria.fr

Andrei Popov
Microsoft
andrei.popov@microsoft.com

Marsh Ray
(co-author of [RFC7627])
Microsoft
maray@microsoft.com

Robert Relyea
Netscape Communications
relyea@netscape.com

Kyle Rose

Akamai Technologies
krose@krose.org

Jim Roskind
Amazon
jroskind@amazon.com

Michael Sabin

Joe Salowey
Tableau Software
joe@salowey.net

Rich Salz
Akamai
rsalz@akamai.com

David Schinazi
Apple Inc.
dschinazi@apple.com

Sam Scott
Royal Holloway, University of London
me@samjs.co.uk

Thomas Shrimpton
University of Florida
teshrim@ufl.edu

Dan Simon
Microsoft, Inc.
dansimon@microsoft.com

Brian Smith
Independent
brian@briansmith.org

Brian Sniffen
Akamai Technologies
ietf@bts.evenmere.org

Nick Sullivan
Cloudflare Inc.
nick@cloudflare.com

Bjoern Tackmann
University of California, San Diego
btackmann@eng.ucsd.edu

Tim Taubert
Mozilla
ttaubert@mozilla.com

Martin Thomson
Mozilla
mt@mozilla.com

Hannes Tschofenig
Arm Limited
Hannes.Tschofenig@arm.com

Sean Turner
sn3rd
sean@sn3rd.com

Steven Valdez
Google
svaldez@google.com

Filippo Valsorda
Cloudflare Inc.
filippo@cloudflare.com

Thyla van der Merwe
Royal Holloway, University of London
tjvdmerwe@gmail.com

Victor Vasiliev
Google
vasilvv@google.com

Hoeteck Wee
Ecole Normale Supérieure, Paris
hoeteck@alum.mit.edu

Tom Weinstein

David Wong
NCC Group
david.wong@nccgroup.trust

Christopher A. Wood
Apple Inc.
cawood@apple.com

Tim Wright
Vodafone

timothy.wright@vodafone.com

Peter Wu
Independent
peter@lekensteyn.nl

Kazu Yamamoto
Internet Initiative Japan Inc.
kazu@iiij.ad.jp

Author's Address

Eric Rescorla
Mozilla
Email: ekr@rtfm.com

TLS
Internet-Draft
Intended status: Standards Track
Expires: 8 September 2022

Y. Nir
Dell Technologies
7 March 2022

A Flags Extension for TLS 1.3
draft-ietf-tls-tlsflags-09

Abstract

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

Status of This Memo

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

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

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

This Internet-Draft will expire on 8 September 2022.

Copyright Notice

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

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

Table of Contents

1. Introduction	2
1.1. Requirements and Other Notation	3
2. The tls_flags Extension	3
3. Rules for The Flags Extension	4
3.1. Interaction with the 0-RTT Handshake	5
4. IANA Considerations	5
5. Security Considerations	6
6. Acknowledgements	7
7. References	7
7.1. Normative References	7
7.2. Informative References	7
Appendix A. Change Log	8
Author's Address	9

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].
- * The post_handshake_auth extension from the TLS 1.3 base document indicates that the client is willing to perform post-handshake authentication.

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.

None of the current proposed extensions allow for indication of support in ServerHello (SH), EncryptedExtensions (EE), Certificate (CT), or HelloRetryRequest (HRR) without first being indicated in ClientHello (CH). Similarly, none of the current proposed extensions allow for an indication of support in the client-side Certificate (CT) message without first being indicated in the server's CertificateRequest (CR) message. This restriction is enforced by the rules in Section 3.

1.1. Requirements and Other Notation

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

The term "flag extension" is used to denote an extension where the extension_data field is always zero-length in a particular context, and the presence of the extension denotes either support for some feature or the intent to use that feature.

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

2. The tls_flags Extension

This document defines the following extension code point:

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

This document also defines the data for this extension as a variable-length bit string, allowing for the encoding of up to 2040 features.

```
struct {  
    opaque flags<1..255>;  
} FlagExtensions;
```

The FlagExtensions field contains 8 flags in each octet. The length of the extension is the minimal length that allows it to encode all of the present flags. Within each octet, the bits are packed such that the first bit is the least significant bit and the eighth bit is

the most significant. Using zero-based indexing, the first octet holds flags 0-7, the second octet holds bits 8-15 and so on. For example, if we want to encode only flag number zero, the FlagExtension field will be 1 octet long, that is encoded as follows:

```
00000001
```

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

```
00100010
```

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

```
00101000 00000000 10000000
```

An implementation that receives an all-zero value for this extension or a value that contains trailing zero bytes MUST generate a fatal `illegal_parameter` alert.

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.

3. Rules for The Flags Extension

Any TLS implementation that intends to propose or indicate support for a flag extension SHALL send this extension with the relevant bits set. It MUST NOT send this extension empty -- with a length of zero.

This specification does not require every flag extension to be acknowledged. Acknowledging a flag extension is typically needed to inform the peer proposing the extension that the other side understands and supports the extension, but some extensions do not require this acknowledgement.

A flag proposed by the client in ClientHello (CH) that requires acknowledgement SHOULD be acknowledged in either ServerHello (SH), in EncryptedExtensions (EE), in Certificate (CT), or in HelloRetryRequest (HRR) as the corresponding flag document specifies. Similarly, a flag proposed by the server in the CertificateRequest (CR) message that requires acknowledgement SHOULD be acknowledged in the client's Certificate (CT) message. A flag proposed by the server in the NewSessionTicket (NST) message is never acknowledged as there is not client-side response message.

Multiple flags can be proposed or acknowledged in the same extension.

In all of the above cases, a flag MUST NOT be acknowledged in SH, EE, CT, or HRR without first having been proposed in CH or CR. Unsolicited flags may appear only in CH, CR, and NST. An endpoint that receives an unsolicited flag in another message (HRR, SH, EE, or CT) MUST generate a fatal `illegal_parameter` alert.

A client that supports this extension and at least one flag extension SHALL send this extension with the flags field having bits set only for those extensions that it intends to set. It MUST NOT send this extension with a length of zero.

An implementation that receives an invalid `tls_flags` extension MUST terminate the TLS handshake with a fatal `illegal_parameter` alert.

3.1. Interaction with the 0-RTT Handshake

The 0-RTT handshake, defined in section 2.3 of [RFC8446], has a ClientHello message, a ServerHello message, and an EncryptedExtensions message. Those can include the `tls_flags` extension just as they can in a regular handshake.

Future flag extensions MUST define their interaction with 0-RTT, just as other extensions are required to.

4. 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,HRR,EE,CR,CT,NST
- * The DTLS-Only value should be N
- * 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 2039. 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 Section 4.6 of [RFC8126] with the exception of flags numbered from 0-15, which follow the "Standards Action" policy (Section 4.9 of [RFC8126]). Designated expert(s) are advised to follow the advice in Section 17 of [RFC8447] when reviewing registration requests.

The initial contents of the registry shall be one entry, as follows:

- * Value shall be 8
- * Flag Name shall be `resumption_across_names`
- * Message shall be NST
- * Recommended shall be set to no (N)
- * The reference shall be the RFC-to-be [I-D.ietf-tls-cross-sni-resumption].

5. 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.

Extension authors should be aware that acknowledging flags in a `tls_flags` extension of the `ServerHello` and `HelloRetryRequest` messages expose this response to passive observers. Unless there is a special reason to place the response in the `ServerHello`, most flags should go in other (encrypted) messages.

6. 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.

Improvement to the encoding were suggested by Ilari Liusvaara, who also asked for a better explanation of the semantics of missing extensions.

Useful comments received from Martin Thomson, including the suggestion to eliminate the option to have the server send unsolicited flag types and the rules for where unsolicited flags can appear.

7. References

7.1. Normative References

- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <<https://www.rfc-editor.org/info/rfc2119>>.
- [RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <<https://www.rfc-editor.org/info/rfc8174>>.
- [RFC8446] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <<https://www.rfc-editor.org/info/rfc8446>>.
- [RFC8447] Salowey, J. and S. Turner, "IANA Registry Updates for TLS and DTLS", RFC 8447, DOI 10.17487/RFC8447, August 2018, <<https://www.rfc-editor.org/info/rfc8447>>.

7.2. Informative References

[I-D.ietf-tls-cross-sni-resumption]

Vasiliev, V., "Transport Layer Security (TLS) Resumption across Server Names", Work in Progress, Internet-Draft, draft-ietf-tls-cross-sni-resumption-02, 5 December 2021, <<https://www.ietf.org/archive/id/draft-ietf-tls-cross-sni-resumption-02.txt>>.

[I-D.ietf-tls-tls13-cert-with-extern-psk]

Housley, R., "TLS 1.3 Extension for Certificate-Based Authentication with an External Pre-Shared Key", Work in Progress, Internet-Draft, draft-ietf-tls-tls13-cert-with-extern-psk-07, 23 December 2019, <<https://www.ietf.org/archive/id/draft-ietf-tls-tls13-cert-with-extern-psk-07.txt>>.

[I-D.sy-tls-resumption-group]

Sy, E., "TLS Resumption across Server Name Indications for TLS 1.3", Work in Progress, Internet-Draft, draft-sy-tls-resumption-group-00, 1 March 2019, <<https://www.ietf.org/archive/id/draft-sy-tls-resumption-group-00.txt>>.

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

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

Appendix A. Change Log

RFC EDITOR: PLEASE REMOVE THIS SECTION AS IT IS ONLY MEANT TO AID THE WORKING GROUP IN TRACKING CHANGES TO THIS DOCUMENT.

draft-ietf-tls-tlsflags-02 set the maximum number of flags to 2048, and added guidance for the IANA experts.

draft-ietf-tls-tlsflags-01 allows server-only flags and allows the client to send an empty extension. Also modified the packing order of the bits.

draft-ietf-tls-tlsflags-00 had the same text as draft-nir-tls-tlsflags-02, and was re-submitted as a working group document following the adoption call.

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
Dell Technologies
9 Andrei Sakharov St
Haifa 3190500
Israel
Email: ynir.ietf@gmail.com

TLS WG
Internet-Draft
Obsoletes: 8447 (if approved)
Updates: 3749, 5077, 4680, 5246, 5705, 5878,
6520, 7301 (if approved)
Intended status: Standards Track
Expires: 5 June 2022

J. Salowey
Salesforce
S. Turner
sn3rd
2 December 2021

IANA Registry Updates for TLS and DTLS
draft-salowey-tls-rfc8447bis-01

Abstract

This document describes a number of changes to TLS and DTLS IANA registries that range from adding notes to the registry all the way to changing the registration policy. These changes were mostly motivated by WG review of the TLS- and DTLS-related registries undertaken as part of the TLS 1.3 development process.

This document obsoletes RFC8447 and updates the following RFCs: 3749, 5077, 4680, 5246, 5705, 5878, 6520, 7301.

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 5 June 2022.

Copyright Notice

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

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

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

Table of Contents

1. Introduction	2
2. Terminology	3
3. Adding "TLS" to Registry Names	3
4. Aligning with RFC 8126	3
5. Adding "Recommended" Column	4
6. Session Ticket TLS Extension	4
7. TLS ExtensionType Values	5
8. TLS Cipher Suites Registry	8
9. TLS Supported Groups	11
10. TLS ClientCertificateType Identifiers	12
11. New Session Ticket TLS Handshake Message Type	12
12. TLS Exporter Labels Registry	13
13. Adding Missing Item to TLS Alerts Registry	14
14. TLS Certificate Types	14
15. Orphaned Registries	15
16. Additional Notes	16
17. Designated Expert Pool	17
18. Security Considerations	17
19. IANA Considerations	18
20. References	18
20.1. Normative References	18
20.2. Informative References	19
Authors' Addresses	20

1. Introduction

This document instructs IANA to make changes to a number of the IANA registries related to Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS). These changes were almost entirely motivated by the development of TLS 1.3 [I-D.ietf-tls-tls13].

The changes introduced by this document range from simple, e.g., adding notes, to complex, e.g., changing a registry's registration policy. Instead of listing the changes and their rationale here in the introduction, each section provides rationale for the proposed change(s).

This document proposes no changes to the registration policies for TLS Alerts [RFC8446], TLS ContentType [RFC8446], TLS HandshakeType [RFC8446], and TLS Certificate Status Types [RFC6961] registries; the

existing policies (Standards Action for the first three; IETF Review for the last), are appropriate for these one-byte code points because of their scarcity.

2. 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.

3. Adding "TLS" to Registry Names

For consistency amongst TLS registries, IANA [SHALL prepend/has prepended] "TLS" to the following registries:

- * Application-Layer Protocol Negotiation (ALPN) Protocol IDs [RFC7301],
- * ExtensionType Values,
- * Heartbeat Message Types [RFC6520], and
- * Heartbeat Modes [RFC6520].

IANA [SHALL update/has updated] the reference for these four registries to also refer to this document. The remainder of this document will use the registry names with the "TLS" prefix.

4. Aligning with RFC 8126

Many of the TLS-related IANA registries had the registration procedure "IETF Consensus", which was changed to "IETF Review" by [RFC8126]. To align with the new terminology, IANA [SHALL update/has updated] the following registries to "IETF Review":

- * TLS Authorization Data Formats [RFC4680]
- * TLS Supplemental Data Formats (SupplementalDataType) [RFC5878]

This is not a universal change, as some registries originally defined with "IETF Consensus" are undergoing other changes either as a result of this document or [RFC8422].

IANA [SHALL update/has updated] the reference for these two registries to also refer to this document.

5. Adding "Recommended" Column

The instructions in this document update the Recommended column, originally added in [RFC8447] to add a third value, "D", indicating that a value is "Discouraged". The permitted values are:

- * Y: Indicates that the IETF has consensus that the item is RECOMMENDED. This only means that the associated mechanism is fit for the purpose for which it was defined. Careful reading of the documentation for the mechanism is necessary to understand the applicability of that mechanism. The IETF could recommend mechanisms that have limited applicability, but will provide applicability statements that describe any limitations of the mechanism or necessary constraints on its use.
- * N: Indicates that the item has not been evaluated by the IETF and that the IETF has made no statement about the suitability of the associated mechanism. This does not necessarily mean that the mechanism is flawed, only that no consensus exists. The IETF might have consensus to leave an items marked as "N" on the basis of it having limited applicability or usage constraints.
- * D: Indicates that the item is discouraged and SHOULD NOT or MUST NOT be used. This marking could be used to identify mechanisms that might result in problems if they are used, such as a weak cryptographic algorithm or a mechanism that might cause interoperability problems in deployment.

Setting the Recommended item to "Y" or "D" or changing a item whose current value is "Y" or "D" requires standards action. Not all items defined in standards track documents need to be marked as Recommended. Changing the Recommended status of a standards track item requires standards action.

[Note: the registries in the rest of the document will need to have the recommended column updated appropriately, specifically to deprecate MD5 and SHA-1, etc.]

6. Session Ticket TLS Extension

The nomenclature for the registry entries in the TLS ExtensionType Values registry correspond to the presentation language field name except for entry 35. To ensure that the values in the registry are consistently identified in the registry, IANA:

- * [SHALL rename/has renamed] entry 35 to "session_ticket (renamed from "SessionTicket TLS")" [RFC5077].

- * [SHALL add/has added] a reference to this document in the "Reference" column for entry 35.

7. TLS ExtensionType Values

Experience has shown that the IETF Review registry policy for TLS extensions was too strict. Based on WG consensus, the decision was taken to change the registration policy to Specification Required [RFC8126] while reserving a small part of the code space for private use. Therefore, IANA [SHALL update/has updated] the TLS ExtensionType Values registry as follows:

- * Changed the registry policy to:

Values with the first byte in the range 0-254 (decimal) are assigned via Specification Required [RFC8126]. Values with the first byte 255 (decimal) are reserved for Private Use [RFC8126].

- * Updated the "Reference" to also refer to this document.

See Section 17 for additional information about the designated expert pool.

Despite wanting to "loosen" the registration policies for TLS extensions, it is still useful to indicate in the IANA registry which extensions the WG recommends be supported. Therefore, IANA [SHALL update/has updated] the TLS ExtensionType Values registry as follows:

- * Add a "Recommended" column with the contents as listed below. This table has been generated by marking Standards Track RFCs as "Y" and all others as "N". The "Recommended" column is assigned a value of "N" unless explicitly requested, and adding a value with a "Recommended" value of "Y" requires Standards Action [RFC8126]. IESG Approval is REQUIRED for a Y->N transition.

Extension	Recommended
server_name	Y
max_fragment_length	N
client_certificate_url	Y
trusted_ca_keys	Y
truncated_hmac	Y

status_request	Y	
+-----+-----+		
user_mapping	Y	
+-----+-----+		
client_authz	N	
+-----+-----+		
server_authz	N	
+-----+-----+		
cert_type	N	
+-----+-----+		
supported_groups	Y	
+-----+-----+		
ec_point_formats	Y	
+-----+-----+		
srp	N	
+-----+-----+		
signature_algorithms	Y	
+-----+-----+		
use_srtp	Y	
+-----+-----+		
heartbeat	Y	
+-----+-----+		
application_layer_protocol_negotiation	Y	
+-----+-----+		
status_request_v2	Y	
+-----+-----+		
signed_certificate_timestamp	N	
+-----+-----+		
client_certificate_type	Y	
+-----+-----+		
server_certificate_type	Y	
+-----+-----+		
padding	Y	
+-----+-----+		
encrypt_then_mac	Y	
+-----+-----+		
extended_master_secret	Y	
+-----+-----+		
cached_info	Y	
+-----+-----+		
session_ticket	Y	
+-----+-----+		
renegotiation_info	Y	
+-----+-----+		

Table 1

IANA [SHALL update/has added] the following notes:

Note: The role of the designated expert is described in [RFC8447]. The designated expert [RFC8126] ensures that the specification is publicly available. It is sufficient to have an Internet-Draft (that is posted and never published as an RFC) or a document from another standards body, industry consortium, university site, etc. The expert may provide more in-depth reviews, but their approval should not be taken as an endorsement of the extension.

Note: As specified in [RFC8126], assignments made in the Private Use space are not generally useful for broad interoperability. It is the responsibility of those making use of the Private Use range to ensure that no conflicts occur (within the intended scope of use). For widespread experiments, temporary reservations are available.

Note: If an item is not marked as "Recommended", it does not necessarily mean that it is flawed; rather, it indicates that the item either has not been through the IETF consensus process, has limited applicability, or is intended only for specific use cases.

The extensions added by [RFC8446] are omitted from the above table; additionally, token_binding is omitted, since [I-D.ietf-tokbind-negotiation] specifies the value of the "Recommended" column as for this extension.

[RFC8446] also uses the TLS ExtensionType Values registry originally created in [RFC4366]. The following text is from [RFC8446] and is included here to ensure alignment between these specifications.

- * IANA [SHALL update/has updated] this registry to include the "key_share", "pre_shared_key", "psk_key_exchange_modes", "early_data", "cookie", "supported_versions", "certificate_authorities", "oid_filters", "post_handshake_auth", and "signature_algorithms_cert", extensions with the values defined in [RFC8446] and the "Recommended" value of "Y".
- * IANA [SHALL update/has updated] this registry to include a "TLS 1.3" column that lists the messages in which the extension may appear. This column [SHALL be/has been] initially populated from the table in Section 4.2 of [RFC8446] with any extension not listed there marked as "-" to indicate that it is not used by TLS 1.3.

8. TLS Cipher Suites Registry

Experience has shown that the IETF Consensus registry policy for TLS Cipher Suites was too strict. Based on WG consensus, the decision was taken to change the TLS Cipher Suites registry's registration policy to Specification Required [RFC8126] while reserving a small part of the code space for experimental and private use. Therefore, IANA [SHALL update/has updated] the TLS Cipher Suites registry's policy as follows:

Values with the first byte in the range 0-254 (decimal) are assigned via Specification Required {{RFC8126}} . Values with the first byte 255 (decimal) are reserved for Private Use {{RFC8126}} .

See Section 17 for additional information about the designated expert pool.

The TLS Cipher Suites registry has grown significantly and will continue to do so. To better guide those not intimately involved in TLS, IANA [shall update/has updated] the TLS Cipher Suites registry as follows:

[The following text needs to be update to reflect the new recommended policy]

- * Added a "Recommended" column to the TLS Cipher Suites registry. The cipher suites that follow in the two tables are marked as "Y". All other cipher suites are marked as "N". The "Recommended" column is assigned a value of "N" unless explicitly requested, and adding a value with a "Recommended" value of "Y" requires Standards Action [RFC8126]. IESG Approval is REQUIRED for a Y->N transition.

The cipher suites that follow are Standards Track server-authenticated (and optionally client-authenticated) cipher suites that are currently available in TLS 1.2.

RFC EDITOR: The previous paragraph is for document reviewers and is not meant for the registry.

Cipher Suite Name	Value
TLS_DHE_RSA_WITH_AES_128_GCM_SHA256	{0x00,0x9E}
TLS_DHE_RSA_WITH_AES_256_GCM_SHA384	{0x00,0x9F}
TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256	{0xC0,0x2B}
TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384	{0xC0,0x2C}
TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256	{0xC0,0x2F}
TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384	{0xC0,0x30}
TLS_DHE_RSA_WITH_AES_128_CCM	{0xC0,0x9E}
TLS_DHE_RSA_WITH_AES_256_CCM	{0xC0,0x9F}
TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305_SHA256	{0xCC,0xA8}
TLS_ECDHE_ECDSA_WITH_CHACHA20_POLY1305_SHA256	{0xCC,0xA9}
TLS_DHE_RSA_WITH_CHACHA20_POLY1305_SHA256	{0xCC,0xAA}

The cipher suites that follow are Standards Track ephemeral pre-shared key cipher suites that are available in TLS 1.2.

RFC EDITOR: The previous paragraph is for document reviewers and is not meant for the registry.

Cipher Suite Name	Value
TLS_DHE_PSK_WITH_AES_128_GCM_SHA256	{0x00,0xAA}
TLS_DHE_PSK_WITH_AES_256_GCM_SHA384	{0x00,0xAB}
TLS_DHE_PSK_WITH_AES_128_CCM	{0xC0,0xA6}
TLS_DHE_PSK_WITH_AES_256_CCM	{0xC0,0xA7}
TLS_ECDHE_PSK_WITH_AES_128_GCM_SHA256	{0xD0,0x01}
TLS_ECDHE_PSK_WITH_AES_256_GCM_SHA384	{0xD0,0x02}
TLS_ECDHE_PSK_WITH_AES_128_CCM_SHA256	{0xD0,0x05}
TLS_ECDHE_PSK_WITH_CHACHA20_POLY1305_SHA256	{0xCC,0xAC}
TLS_DHE_PSK_WITH_CHACHA20_POLY1305_SHA256	{0xCC,0xAD}

The TLS 1.3 cipher suites specified by [RFC8446] are not listed here; that document provides for their "Recommended" status.

Despite the following behavior being misguided, experience has shown that some customers use the IANA registry as a checklist against which to measure an implementation's completeness, and some implementers blindly implement cipher suites. Therefore, IANA [SHALL add/has added] the following warning to the registry:

WARNING: Cryptographic algorithms and parameters will be broken or weakened over time. Blindly implementing cipher suites listed here is not advised. Implementers and users need to check that the cryptographic algorithms listed continue to provide the expected level of security.

IANA [SHALL add/has added] the following note to ensure that those that focus on IANA registries are aware that TLS 1.3 [RFC8446] uses the same registry but defines ciphers differently:

Note: Although TLS 1.3 uses the same cipher suite space as previous versions of TLS, TLS 1.3 cipher suites are defined differently, only specifying the symmetric ciphers and hash functions, and cannot be used for TLS 1.2. Similarly, TLS 1.2 and lower cipher suite values cannot be used with TLS 1.3.

IANA [SHALL add/has added] the following notes to document the rules for populating the "Recommended" column:

Note: CCM_8 cipher suites are not marked as "Recommended". These cipher suites have a significantly truncated authentication tag that represents a security trade-off that may not be appropriate for general environments.

Note: If an item is not marked as "Recommended", it does not necessarily mean that it is flawed; rather, it indicates that the item either has not been through the IETF consensus process, has limited applicability, or is intended only for specific use cases.

IANA [SHALL add/has added] the following notes for additional information:

Note: The role of the designated expert is described in [this-RFC]. The designated expert [RFC8126] ensures that the specification is publicly available. It is sufficient to have an Internet-Draft (that is posted and never published as an RFC) or a document from another standards body, industry consortium, university site, etc. The expert may provide more in-depth reviews, but their approval should not be taken as an endorsement of the cipher suite.

Note: As specified in [RFC8126], assignments made in the Private Use space are not generally useful for broad interoperability. It is the responsibility of those making use of the Private Use range to ensure that no conflicts occur (within the intended scope of use). For widespread experiments, temporary reservations are available.

IANA [SHALL update/has updated] the reference for this registry to also refer to this document.

9. TLS Supported Groups

Similar to cipher suites, supported groups have proliferated over time, and some use the registry to measure implementations. Therefore, IANA [SHALL add/has added] a "Recommended" column with a "Y" for secp256r1, secp384r1, x25519, and x448, while all others are "N". These "Y" groups are taken from Standards Track RFCs; [RFC8422] elevates secp256r1 and secp384r1 to Standards Track. Not all groups from [RFC8422], which is Standards Track, are marked as "Y"; these groups apply to TLS 1.3 [RFC8446] and previous versions of TLS. The "Recommended" column is assigned a value of "N" unless explicitly requested, and adding a value with a "Recommended" value of "Y" requires Standards Action [RFC8126]. IESG Approval is REQUIRED for a Y->N transition.

IANA [SHALL add/has added] the following notes:

Note: If an item is not marked as "Recommended" it does not necessarily mean that it is flawed; rather, it indicates that the item either has not been through the IETF consensus process, has limited applicability, or is intended only for specific use cases.

Note: The role of the designated expert is described in [RFC8447]. The designated expert [RFC8126] ensures that the specification is publicly available. It is sufficient to have an Internet-Draft (that is posted and never published as an RFC) or a document from another standards body, industry consortium, university site, etc. The expert may provide more in-depth reviews, but their approval should not be taken as an endorsement of the supported groups.

Despite the following behavior being misguided, experience has shown that some customers use the IANA registry as a checklist against which to measure an implementation's completeness, and some implementers blindly implement supported group. Therefore, IANA [SHALL add/has added] the following warning to the registry:

WARNING: Cryptographic algorithms and parameters will be broken or weakened over time. Blindly implementing supported groups listed here is not advised. Implementers and users need to check that the cryptographic algorithms listed continue to provide the expected level of security.

IANA [SHALL update/has updated] the reference for this registry to also refer to this document.

The value 0 (0x0000) has been marked as reserved.

10. TLS ClientCertificateType Identifiers

Experience has shown that the IETF Consensus registry policy for TLS ClientCertificateType Identifiers is too strict. Based on WG consensus, the decision was taken to change the registration policy to Specification Required [RFC8126] while reserving some of the code space for Standards Track usage and a small part of the code space for private use. Therefore, IANA has updated the TLS ClientCertificateType Identifiers registry's policy as follows:

Values in the range 0-63 are assigned via Standards Action.
Values 64-223 are assigned via Specification Required [RFC8126].
Values 224-255 are reserved for Private Use.

See Section 17 for additional information about the designated expert pool.

IANA [SHALL add/has added] the following notes:

Note: The role of the designated expert is described in [this-RFC]. The designated expert [RFC8126] ensures that the specification is publicly available. It is sufficient to have an Internet-Draft (that is posted and never published as an RFC) or a document from another standards body, industry consortium, university site, etc. The expert may provide more in-depth reviews, but their approval should not be taken as an endorsement of the identifier.

Note: As specified in [RFC8126], assignments made in the Private Use space are not generally useful for broad interoperability. It is the responsibility of those making use of the Private Use range to ensure that no conflicts occur (within the intended scope of use). For widespread experiments, temporary reservations are available.

11. New Session Ticket TLS Handshake Message Type

To align with TLS implementations and to align the naming nomenclature with other Handshake message types, IANA:

- * [SHALL rename/has renamed] entry 4 in the TLS HandshakeType registry to "new_session_ticket (renamed from NewSessionTicket)" [RFC5077].
- * [SHALL add/has added] a reference to this document in the "Reference" column for entry 4 in the TLS HandshakeType registry.

12. TLS Exporter Labels Registry

To aid those reviewers who start with the IANA registry, IANA [SHALL add/has added]:

- * The following note to the TLS Exporter Labels registry:

Note: [RFC5705] defines keying material exporters for TLS in terms of the TLS PRF. [RFC8446] replaced the PRF with HKDF, thus requiring a new construction. The exporter interface remains the same; however, the value is computed differently.

- * A "Recommended" column to the TLS Exporter Labels registry. The table that follows has been generated by marking Standards Track RFCs as "Y" and all others as "N". The "Recommended" column is assigned a value of "N" unless explicitly requested, and adding a value with a "Recommended" value of "Y" requires Standards Action [RFC8126]. IESG Approval is REQUIRED for a Y->N transition.

Exporter Value	Recommended
client finished	Y
server finished	Y
master secret	Y
key expansion	Y
client EAP encryption	Y
ttls keying material	N
ttls challenge	N
EXTRACTOR-dtls_srtp	Y
EXPORTER_DTLS_OVER_SCTP	Y
EXPORTER: teap session key seed	Y

To provide additional information for the designated experts, IANA [SHALL add/has added] the following notes:

Note: The role of the designated expert is described in [RFC8447] . The designated expert [RFC8126] ensures that the specification is publicly available. It is sufficient to have an Internet-Draft (that is posted and never published as an RFC) or a document from another standards body, industry consortium, university site, etc. The expert may provide more in-depth reviews, but their approval should not be taken as an endorsement of the exporter label. The expert also verifies that the label is a string consisting of printable ASCII characters beginning with "EXPORTER". IANA MUST also verify that one label is not a prefix of any other label. For example, labels "key" or "master secretary" are forbidden.

Note: If an item is not marked as "Recommended", it does not

necessarily mean that it is flawed; rather, it indicates that the item either has not been through the IETF consensus process, has limited applicability, or is intended only for specific use cases.

IANA [SHALL update/has updated] the reference for this registry to also refer to this document.

13. Adding Missing Item to TLS Alerts Registry

IANA [SHALL add/has added] the following entry to the TLS Alerts registry; the entry was omitted from the IANA instructions in [RFC7301]:

```
120    no_application_protocol    Y    [RFC7301][RFC8447]
```

14. TLS Certificate Types

Experience has shown that the IETF Consensus registry policy for TLS Certificate Types is too strict. Based on WG consensus, the decision was taken to change registration policy to Specification Required [RFC8126] while reserving a small part of the code space for private use. Therefore, IANA [SHALL change/has changed] the TLS Certificate Types registry as follows:

- * Changed the registry policy to:

Values in the range 0-223 (decimal) are assigned via Specification Required [RFC8126]. Values in the range 224-255 (decimal) are reserved for Private Use [RFC8126].

- * Added a "Recommended" column to the registry. X.509 and Raw Public Key are "Y". All others are "N". The "Recommended" column is assigned a value of "N" unless explicitly requested, and adding a value with a "Recommended" value of "Y" requires Standards Action [RFC8126]. IESG Approval is REQUIRED for a Y->N transition.

See Section 17 for additional information about the designated expert pool.

IANA [SHALL add/has added] the following note:

Note: The role of the designated expert is described in [this-RFC].

The designated expert [RFC8126] ensures that the specification is publicly available. It is sufficient to have an Internet-Draft (that is posted and never published as an RFC) or a document from another standards body, industry consortium, university site, etc. The expert may provide more in-depth reviews, but their approval should not be taken as an endorsement of the certificate type.

Note: If an item is not marked as "Recommended", it does not necessarily mean that it is flawed; rather, it indicates that the item either has not been through the IETF consensus process, has limited applicability, or is intended only for specific use cases.

IANA [SHALL update/has updated] the reference for this registry to also refer this document.

15. Orphaned Registries

To make it clear that (D)TLS 1.3 has orphaned certain registries (i.e., they are only applicable to version of (D)TLS protocol versions prior to 1.3), IANA:

- * [SHALL add/has added] the following to the TLS Compression Method Identifiers registry [RFC3749]:

Note: Value 0 (NULL) is the only value in this registry applicable to (D)TLS protocol version 1.3 or later.

- * [SHALL add/has added] the following to the TLS HashAlgorithm [RFC5246] and TLS SignatureAlgorithm registries [RFC5246]:

Note: The values in this registry are only applicable to (D)TLS protocol versions prior to 1.3. (D)TLS 1.3 and later versions' values are registered in the TLS SignatureScheme registry.

- * [SHALL update/has updated] the "Reference" field in the TLS Compression Method Identifiers, TLS HashAlgorithm and TLS SignatureAlgorithm registries to also refer to this document.
- * [SHALL update/has updated] the TLS HashAlgorithm registry to list values 7 and 9-223 as "Reserved" and the TLS SignatureAlgorithm registry to list values 4-6 and 9-223 as "Reserved".
- * has added the following to the TLS ClientCertificateType Identifiers registry [RFC5246]:

Note: The values in this registry are only applicable to (D)TLS protocol versions prior to 1.3.

Despite the fact that the TLS HashAlgorithm and SignatureAlgorithm registries are orphaned, it is still important to warn implementers of pre-TLS1.3 implementations about the dangers of blindly implementing cryptographic algorithms. Therefore, IANA has added the following warning to the TLS HashAlgorithm and SignatureAlgorithm registries:

WARNING: Cryptographic algorithms and parameters will be broken or weakened over time. Blindly implementing the cryptographic algorithms listed here is not advised. Implementers and users need to check that the cryptographic algorithms listed continue to provide the expected level of security.

16. Additional Notes

IANA has added the following warning and note to the TLS SignatureScheme registry:

WARNING: Cryptographic algorithms and parameters will be broken or weakened over time. Blindly implementing signature schemes listed here is not advised. Implementers and users need to check that the cryptographic algorithms listed continue to provide the expected level of security.

Note: As specified in [RFC8126], assignments made in the Private Use space are not generally useful for broad interoperability. It is the responsibility of those making use of the Private Use range to ensure that no conflicts occur (within the intended scope of use). For widespread experiments, temporary reservations are available.

IANA has added the following notes to the TLS PskKeyExchangeMode registry:

Note: If an item is not marked as "Recommended", it does not necessarily mean that it is flawed; rather, it indicates that the item either has not been through the IETF consensus process, has limited applicability, or is intended only for specific use cases.

Note: The role of the designated expert is described in RFC 8447. The designated expert [RFC8126] ensures that the specification is publicly available. It is sufficient to have an Internet-Draft (that is posted and never published as an RFC) or a document from another standards body, industry consortium, university site, etc. The expert may provide more in depth reviews, but their approval should not be taken as an endorsement of the key exchange mode.

17. Designated Expert Pool

Specification Required [RFC8126] registry requests are registered after a three-week review period on the `tls-reg-review@ietf.org` (`mailto:tls-reg-review@ietf.org`) mailing list, on the advice of one or more designated experts. However, to allow for the allocation of values prior to publication, the designated experts may approve registration once they are satisfied that such a specification will be published.

Registration requests sent to the mailing list for review SHOULD use an appropriate subject (e.g., "Request to register value in TLS bar registry").

Within the review period, the designated experts will either approve or deny the registration request, communicating this decision to the review list and IANA. Denials SHOULD include an explanation and, if applicable, suggestions as to how to make the request successful. Registration requests that are undetermined for a period longer than 21 days can be brought to the IESG's attention (using the `iesg@ietf.org` (`mailto:iesg@ietf.org`) mailing list) for resolution.

Criteria that SHOULD be applied by the designated experts includes determining whether the proposed registration duplicates existing functionality, whether it is likely to be of general applicability or useful only for a single application, and whether the registration description is clear.

IANA MUST only accept registry updates from the designated experts and SHOULD direct all requests for registration to the review mailing list.

It is suggested that multiple designated experts be appointed who are able to represent the perspectives of different applications using this specification, in order to enable broadly informed review of registration decisions. In cases where a registration decision could be perceived as creating a conflict of interest for a particular Expert, that Expert SHOULD defer to the judgment of the other Experts.

18. Security Considerations

The change to Specification Required from IETF Review lowers the amount of review provided by the WG for cipher suites and supported groups. This change reflects reality in that the WG essentially provided no cryptographic review of the cipher suites or supported groups. This was especially true of national cipher suites.

Recommended algorithms are regarded as secure for general use at the time of registration; however, cryptographic algorithms and parameters will be broken or weakened over time. It is possible that the "Recommended" status in the registry lags behind the most recent advances in cryptanalysis. Implementers and users need to check that the cryptographic algorithms listed continue to provide the expected level of security.

Designated experts ensure the specification is publicly available. They may provide more in-depth reviews. Their review should not be taken as an endorsement of the cipher suite, extension, supported group, etc.

19. IANA Considerations

This document is entirely about changes to TLS-related IANA registries.

20. References

20.1. Normative References

- [I-D.ietf-tls-tls13]
Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", Work in Progress, Internet-Draft, draft-ietf-tls-tls13-28, 20 March 2018, <<https://datatracker.ietf.org/doc/html/draft-ietf-tls-tls13-28>>.
- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <<https://www.rfc-editor.org/rfc/rfc2119>>.
- [RFC3749] Hollenbeck, S., "Transport Layer Security Protocol Compression Methods", RFC 3749, DOI 10.17487/RFC3749, May 2004, <<https://www.rfc-editor.org/rfc/rfc3749>>.
- [RFC4680] Santesson, S., "TLS Handshake Message for Supplemental Data", RFC 4680, DOI 10.17487/RFC4680, October 2006, <<https://www.rfc-editor.org/rfc/rfc4680>>.
- [RFC5077] Salowey, J., Zhou, H., Eronen, P., and H. Tschofenig, "Transport Layer Security (TLS) Session Resumption without Server-Side State", RFC 5077, DOI 10.17487/RFC5077, January 2008, <<https://www.rfc-editor.org/rfc/rfc5077>>.

- [RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, DOI 10.17487/RFC5246, August 2008, <<https://www.rfc-editor.org/rfc/rfc5246>>.
- [RFC5705] Rescorla, E., "Keying Material Exporters for Transport Layer Security (TLS)", RFC 5705, DOI 10.17487/RFC5705, March 2010, <<https://www.rfc-editor.org/rfc/rfc5705>>.
- [RFC5878] Brown, M. and R. Housley, "Transport Layer Security (TLS) Authorization Extensions", RFC 5878, DOI 10.17487/RFC5878, May 2010, <<https://www.rfc-editor.org/rfc/rfc5878>>.
- [RFC6520] Seggelmann, R., Tuexen, M., and M. Williams, "Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS) Heartbeat Extension", RFC 6520, DOI 10.17487/RFC6520, February 2012, <<https://www.rfc-editor.org/rfc/rfc6520>>.
- [RFC7301] Friedl, S., Popov, A., Langley, A., and E. Stephan, "Transport Layer Security (TLS) Application-Layer Protocol Negotiation Extension", RFC 7301, DOI 10.17487/RFC7301, July 2014, <<https://www.rfc-editor.org/rfc/rfc7301>>.
- [RFC8126] Cotton, M., Leiba, B., and T. Narten, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 8126, DOI 10.17487/RFC8126, June 2017, <<https://www.rfc-editor.org/rfc/rfc8126>>.
- [RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <<https://www.rfc-editor.org/rfc/rfc8174>>.
- [RFC8446] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <<https://www.rfc-editor.org/rfc/rfc8446>>.
- [RFC8447] Salowey, J. and S. Turner, "IANA Registry Updates for TLS and DTLS", RFC 8447, DOI 10.17487/RFC8447, August 2018, <<https://www.rfc-editor.org/rfc/rfc8447>>.

20.2. Informative References

[I-D.ietf-tokbind-negotiation]

Popov, A., Nyström, M., Balfanz, D., and A. Langley,
"Transport Layer Security (TLS) Extension for Token
Binding Protocol Negotiation", Work in Progress, Internet-
Draft, draft-ietf-tokbind-negotiation-14, 23 May 2018,
<<https://datatracker.ietf.org/doc/html/draft-ietf-tokbind-negotiation-14>>.

[RFC4366] Blake-Wilson, S., Nystrom, M., Hopwood, D., Mikkelsen, J.,
and T. Wright, "Transport Layer Security (TLS)
Extensions", RFC 4366, DOI 10.17487/RFC4366, April 2006,
<<https://www.rfc-editor.org/rfc/rfc4366>>.

[RFC6961] Pettersen, Y., "The Transport Layer Security (TLS)
Multiple Certificate Status Request Extension", RFC 6961,
DOI 10.17487/RFC6961, June 2013,
<<https://www.rfc-editor.org/rfc/rfc6961>>.

[RFC8422] Nir, Y., Josefsson, S., and M. Pegourie-Gonnard, "Elliptic
Curve Cryptography (ECC) Cipher Suites for Transport Layer
Security (TLS) Versions 1.2 and Earlier", RFC 8422,
DOI 10.17487/RFC8422, August 2018,
<<https://www.rfc-editor.org/rfc/rfc8422>>.

Authors' Addresses

Joe Salowey
Salesforce

Email: joe@salowey.net

Sean Turner
sn3rd

Email: sean@sn3rd.com