

Network Working Group
Internet-Draft
Intended status: Experimental
Expires: September 7, 2019

TCI
March 6, 2019

Fake Server Name Indication
draft-belyavskiy-fakesni-02

Abstract

The document provides a specification of the Fake Server Name Indication. Being implemented, the Fake SNI specification provides a way to cheat the monitoring solutions without providing any additional information to external observers.

Status of This Memo

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

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

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

This Internet-Draft will expire on September 7, 2019.

Copyright Notice

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

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

1. Introduction

Many DPI solutions use SNI information as a criterion to filter connection to various sites. Though Encrypted SNI makes impossible to read the SNI value, there is information [1] that absence of SNI looks suspicious itself and all communications are blocked.

This specification introduces a way to provide a value of SNI treated by TLS server as an alias to one of the names known by server but not matching the possibly suspicious hostname.

This specification does not save from DPI solutions but it provides one more loophole to cheat them.

2. Fake SNI design goals

The solution specified in this document is inspired by the design of Encrypted SNI. It is fully-compatible with current TLS specifications. As it does not make much sense to use it with TLS 1.0-1.2, where the original host name will be provided unencrypted in the certificate, in case of TLS 1.3 the certificate is delivered encrypted.

The provider publishes a name matching the target name to be provided in the clear text. This document defines a publication mechanism using DNS, but other mechanisms are also possible.

When a client wants to establish a TLS connection to a domain served by a Fake SNI-supporting provider, it replaces the value in "server_name" extension in the ClientHello with the value obtained by transport. The provider can then find out the desired name from its configuration and either establish the connection with the desired host or reject it.

3. Definitions

Original name - the hostname of service that is subject to hide.

Fake name - the hostname specified by server and sent by client to indicate intention to connect to host with original name.

4. Fake SNI indication

Fake SNI information is published in DNS via TXT RR. For example, the Fake SNI record for domain example.com may look like

```
_fakesni.example.com. 60S IN TXT "myfakerecord.com IP"
```

where IP address may be omitted. If present, it MUST match an IP address specified in A/AAAA record for the domain. Specifying IP address for a specific fake name may help in case when a service is hosted using more than one CDN.

The fake name specified in the Fake SNI RR MUST identify the original hostname it is valid for. Fake names for different hosts on the same IP address MUST be different to distinguish the original names.

5. Server behaviour

On receiving the value of known Fake SNI in the TLS ClientHello server MUST return the certificate matching the original hostname. Otherwise server SHOULD abort the connection.

6. Client behaviour

Client MAY use the Fake SNI record as fallback if connecting using ESNI is blocked. In this case client initiates normal TLS connection specifying the value from Fake SNI record in the server_name extension. If the certificate received from server does not match the original hostname, the client MUST abort the connection. Otherwise the client MUST follow the normal process of TLS handshake.

7. Operational considerations

Depending on the DPI modus operandi it may make sense to provide a valid fake name (e.g. from deep-level subdomain) resolving to the same IPs as original hostname does. If DPI tries to resolve the fake name, such behaviour will make distinguishing between real and fake names difficult.

8. Security considerations

As Fake SNI can be used in TLS 1.2, it does not provide any problems to DPI because in this case the original hostname is available in clear text in server certificate. TLS 1.3 encrypts the Certificate message, so it is RECOMMENDED to use Fake SNI together with TLS 1.3. To strengthen the protection, it's recommended to obtain _fakesni RR via DoT or DoH.

As DPI solutions are able to obtain the DNS _fakesni records as legitimate clients do, it is RECOMMENDED to set reasonable TTL values for the _fakesni records. Also it is RECOMMENDED to use such values of fake names that are syntactically correct domain names. Otherwise DPI can recognise the fake names as fake ones.

9. Current version of the specification

The current version of the specification is available at GitHub repository [2].

10. References

10.1. URIs

[1] <https://mailarchive.ietf.org/arch/msg/tls/WiT3oEh6PO96mm0z28BNMp0YgGs>

[2] <https://github.com/beldmit/fakesni/>

Author's Address

Dmitry Belyavskiy
Cryptocom LTD
Kedrova st, 14/2
Moscow 117218
RU

Email: beldmit@gmail.com

TLS
Internet-Draft
Intended status: Standards Track
Expires: July 9, 2020

A. Ghedini
Cloudflare, Inc.
V. Vasiliev
Google
January 06, 2020

TLS Certificate Compression
draft-ietf-tls-certificate-compression-10

Abstract

In TLS handshakes, certificate chains often take up the majority of the bytes transmitted.

This document describes how certificate chains can be compressed to reduce the amount of data transmitted and avoid some round trips.

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 July 9, 2020.

Copyright Notice

Copyright (c) 2020 IETF Trust and the persons identified as the document authors. All rights reserved.

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

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

Table of Contents

1. Introduction	2
2. Notational Conventions	2
3. Negotiating Certificate Compression	2
4. Compressed Certificate Message	3
5. Security Considerations	4
6. Middlebox Compatibility	5
7. IANA Considerations	5
7.1. Update of the TLS ExtensionType Registry	5
7.2. Update of the TLS HandshakeType Registry	6
7.3. Registry for Compression Algorithms	6
8. References	6
8.1. Normative References	6
8.2. Informative References	7
Appendix A. Acknowledgements	8
Authors' Addresses	8

1. Introduction

In order to reduce latency and improve performance it can be useful to reduce the amount of data exchanged during a TLS handshake.

[RFC7924] describes a mechanism that allows a client and a server to avoid transmitting certificates already shared in an earlier handshake, but it doesn't help when the client connects to a server for the first time and doesn't already have knowledge of the server's certificate chain.

This document describes a mechanism that would allow certificates to be compressed during all handshakes.

2. Notational Conventions

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. Negotiating Certificate Compression

This extension is only supported with TLS 1.3 [RFC8446] and newer; if TLS 1.2 [RFC5246] or earlier is negotiated, the peers MUST ignore this extension.

This document defines a new extension type (`compress_certificate(27)`), which can be used to signal the supported compression formats for the Certificate message to the peer. Whenever it is sent by the client as a ClientHello message extension ([RFC8446], Section 4.1.2), it indicates the support for compressed server certificates. Whenever it is sent by the server as a CertificateRequest extension ([RFC8446], Section 4.3.2), it indicates the support for compressed client certificates.

By sending a `compress_certificate` extension, the sender indicates to the peer the certificate compression algorithms it is willing to use for decompression. The "extension_data" field of this extension SHALL contain a `CertificateCompressionAlgorithms` value:

```
enum {
    zlib(1),
    brotli(2),
    zstd(3),
    (65535)
} CertificateCompressionAlgorithm;

struct {
    CertificateCompressionAlgorithm algorithms<2..2^8-2>;
} CertificateCompressionAlgorithms;
```

The `compress_certificate` extension is a unidirectional indication; no corresponding response extension is needed.

4. Compressed Certificate Message

If the peer has indicated that it supports compression, server and client MAY compress their corresponding Certificate messages (Section 4.4.2 of [RFC8446]) and send them in the form of the `CompressedCertificate` message (replacing the Certificate message).

The `CompressedCertificate` message is formed as follows:

```
struct {
    CertificateCompressionAlgorithm algorithm;
    uint24 uncompressed_length;
    opaque compressed_certificate_message<1..2^24-1>;
} CompressedCertificate;
```

`algorithm` The algorithm used to compress the certificate. The `algorithm` MUST be one of the algorithms listed in the peer's `compress_certificate` extension.

`uncompressed_length` The length of the Certificate message once it is uncompressed. If after decompression the specified length does not match the actual length, the party receiving the invalid message MUST abort the connection with the "bad_certificate" alert. The presence of this field allows the receiver to pre-allocate the buffer for the uncompressed Certificate message and to enforce limits on the message size before performing decompression.

`compressed_certificate_message` The result of applying the indicated compression algorithm to the encoded Certificate message that would have been sent if certificate compression was not in use. The compression algorithm defines how the bytes in the `compressed_certificate_message` field are converted into the Certificate message.

If the specified compression algorithm is `zlib`, then the Certificate message MUST be compressed with the ZLIB compression algorithm, as defined in [RFC1950]. If the specified compression algorithm is `brotli`, the Certificate message MUST be compressed with the Brotli compression algorithm as defined in [RFC7932]. If the specified compression algorithm is `zstd`, the Certificate message MUST be compressed with the Zstandard compression algorithm as defined in [I-D.kucherawy-rfc8478bis].

It is possible to define a certificate compression algorithm that uses a pre-shared dictionary to achieve higher compression ratio. This document does not define any such algorithms, but additional codepoints may be allocated for such use per the policy in Section 7.3.

If the received `CompressedCertificate` message cannot be decompressed, the connection MUST be terminated with the "bad_certificate" alert.

If the format of the Certificate message is altered using the `server_certificate_type` or `client_certificate_type` extensions [RFC7250], the resulting altered message is compressed instead.

5. Security Considerations

After decompression, the Certificate message MUST be processed as if it were encoded without being compressed. This way, the parsing and the verification have the same security properties as they would have in TLS normally.

In order for certificate compression to function correctly, the underlying compression algorithm MUST output the same data that was provided as input by the peer.

Since certificate chains are typically presented on a per-server name or per-user basis, a malicious application does not have control over any individual fragments in the Certificate message, meaning that they cannot leak information about the certificate by modifying the plaintext.

Implementations SHOULD bound the memory usage when decompressing the CompressedCertificate message.

Implementations MUST limit the size of the resulting decompressed chain to the specified uncompressed length, and they MUST abort the connection if the size of the output of the decompression function exceeds that limit. TLS framing imposes 16777216 byte limit on the certificate message size, and the implementations MAY impose a limit that is lower than that; in both cases, they MUST apply the same limit as if no compression were used.

While the Certificate message in TLS 1.3 is encrypted, third parties can draw inferences from the message length observed on the wire. TLS 1.3 provides a padding mechanism (discussed in Sections 5.4 and E.3 of [RFC8446]) to counteract such analysis. Certificate compression alters the length of the Certificate message, and the change in length is dependent on the actual contents of the certificate. Any padding scheme covering the Certificate message has to address compression within its design, or disable it altogether.

6. Middlebox Compatibility

It's been observed that a significant number of middleboxes intercept and try to validate the Certificate message exchanged during a TLS handshake. This means that middleboxes that don't understand the CompressedCertificate message might misbehave and drop connections that adopt certificate compression. Because of that, the extension is only supported in the versions of TLS where the certificate message is encrypted in a way that prevents middleboxes from intercepting it, that is, TLS version 1.3 [RFC8446] and higher.

7. IANA Considerations

7.1. Update of the TLS ExtensionType Registry

Create an entry, `compress_certificate(27)`, in the existing registry for ExtensionType (defined in [RFC8446]), with "TLS 1.3" column values being set to "CH, CR", and "Recommended" column being set to "Yes".

7.2. Update of the TLS HandshakeType Registry

Create an entry, `compressed_certificate(25)`, in the existing registry for HandshakeType (defined in [RFC8446]), with "DTLS-OK" column value being set to "Yes".

7.3. Registry for Compression Algorithms

This document establishes a registry of compression algorithms supported for compressing the Certificate message, titled "Certificate Compression Algorithm IDs", under the existing "Transport Layer Security (TLS) Extensions" heading.

The entries in the registry are:

Algorithm Number	Description	Reference
0	Reserved	
1	zlib	[this document]
2	brotnli	[this document]
3	zstd	[this document]
16384 to 65535	Reserved for Experimental Use	

The values in this registry shall be allocated under "IETF Review" policy for values strictly smaller than 256, under "Specification Required" policy for values 256-16383, and under "Experimental Use" otherwise (see [RFC8126] for the definition of relevant policies). Experimental Use extensions can be used both on private networks and over the open Internet.

The procedures for requesting values in the Specification Required space are specified in Section 17 of [RFC8447].

8. References

8.1. Normative References

[I-D.kucherawy-rfc8478bis]
Collet, Y. and M. Kucherawy, "Zstandard Compression and the application/zstd Media Type", draft-kucherawy-rfc8478bis-03 (work in progress), December 2019.

- [RFC1950] Deutsch, P. and J-L. Gailly, "ZLIB Compressed Data Format Specification version 3.3", RFC 1950, DOI 10.17487/RFC1950, May 1996, <<https://www.rfc-editor.org/info/rfc1950>>.
- [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>>.
- [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>>.
- [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>>.
- [RFC7932] Alakuijala, J. and Z. Szabadka, "Brotli Compressed Data Format", RFC 7932, DOI 10.17487/RFC7932, July 2016, <<https://www.rfc-editor.org/info/rfc7932>>.
- [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>>.
- [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>>.

8.2. Informative References

- [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>>.

Appendix A. Acknowledgements

Certificate compression was originally introduced in the QUIC Crypto protocol, designed by Adam Langley and Wan-Teh Chang.

This document has benefited from contributions and suggestions from David Benjamin, Ryan Hamilton, Christian Huitema, Benjamin Kaduk, Ilari Liusvaara, Piotr Sikora, Ian Swett, Martin Thomson, Sean Turner and many others.

Authors' Addresses

Alessandro Ghedini
Cloudflare, Inc.

Email: alessandro@cloudflare.com

Victor Vasiliev
Google

Email: vasilvv@google.com

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:

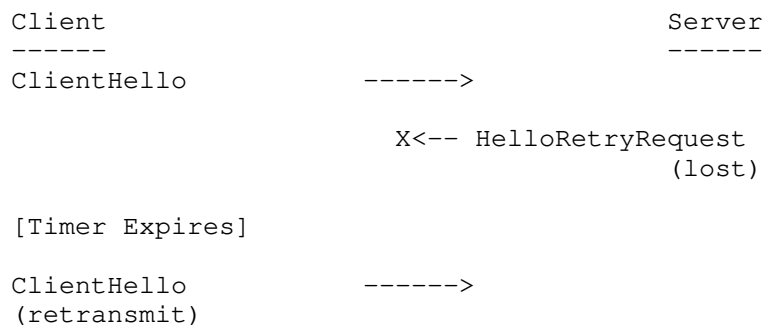


Figure 1: DTLS retransmission example

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.

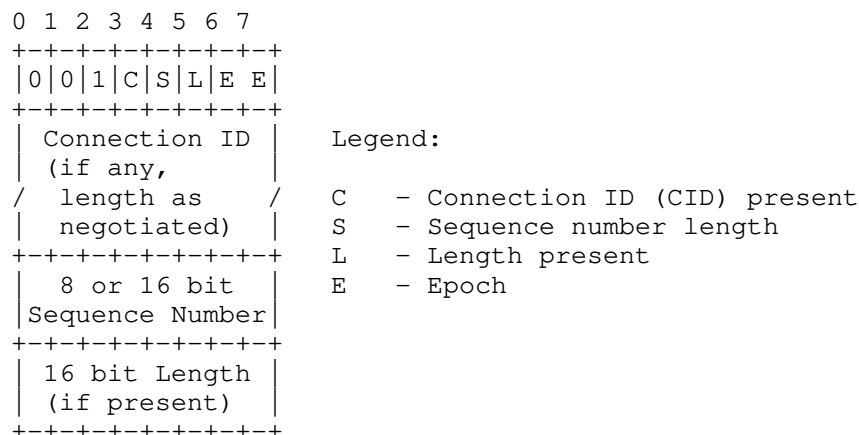


Figure 3: DTLS 1.3 Unified Header

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.

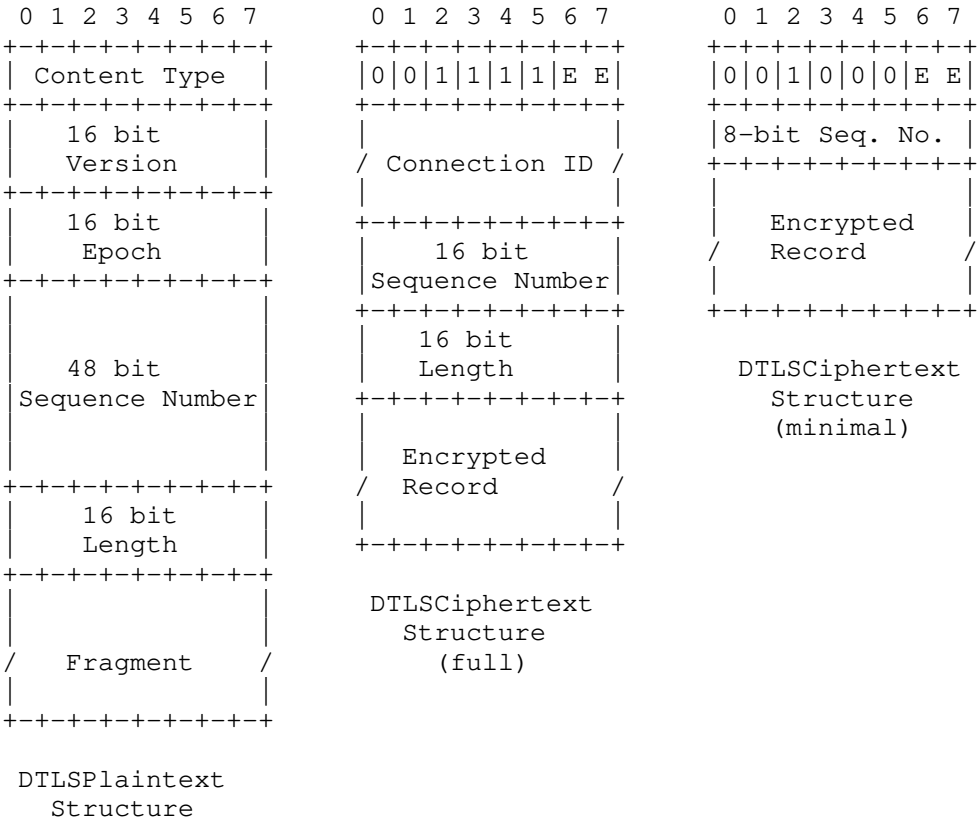


Figure 4: DTLS 1.3 Header Examples

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.

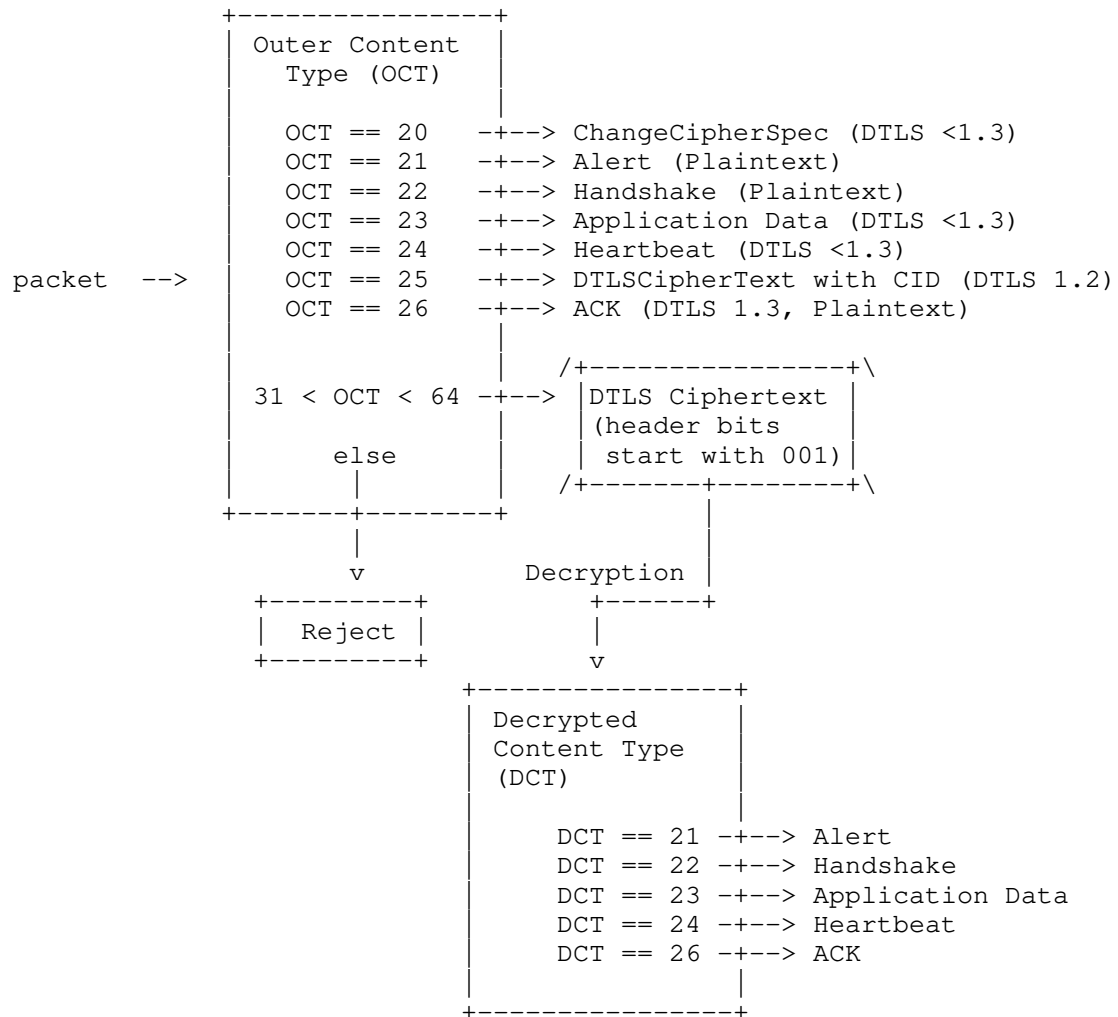


Figure 5: Demultiplexing DTLS 1.2 and DTLS 1.3 Records

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.

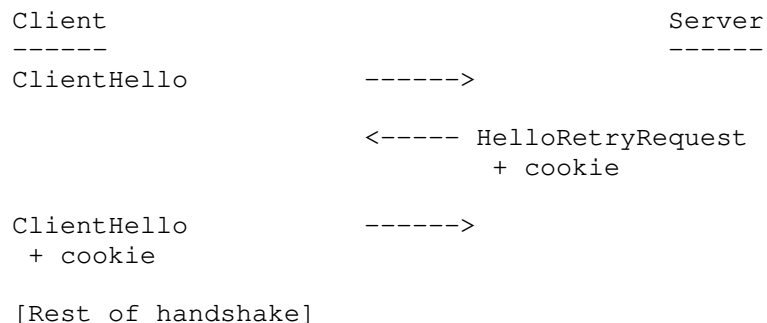


Figure 6: DTLS exchange with HelloRetryRequest containing the "cookie" extension

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

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

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

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

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

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

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

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

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

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

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

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

5.2. DTLS Handshake Message Format

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

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

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

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

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

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

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

5.3. ClientHello Message

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

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

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

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

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

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

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

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

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

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

legacy_compression_methods: Same as for TLS 1.3.

extensions: Same as for TLS 1.3.

5.4. ServerHello Message

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

5.5. Handshake Message Fragmentation and Reassembly

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

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

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

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

5.6. End Of Early Data

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

5.7. DTLS Handshake Flights

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

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

Table 1: Flight Handshake Message Combinations.

Remarks:

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

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

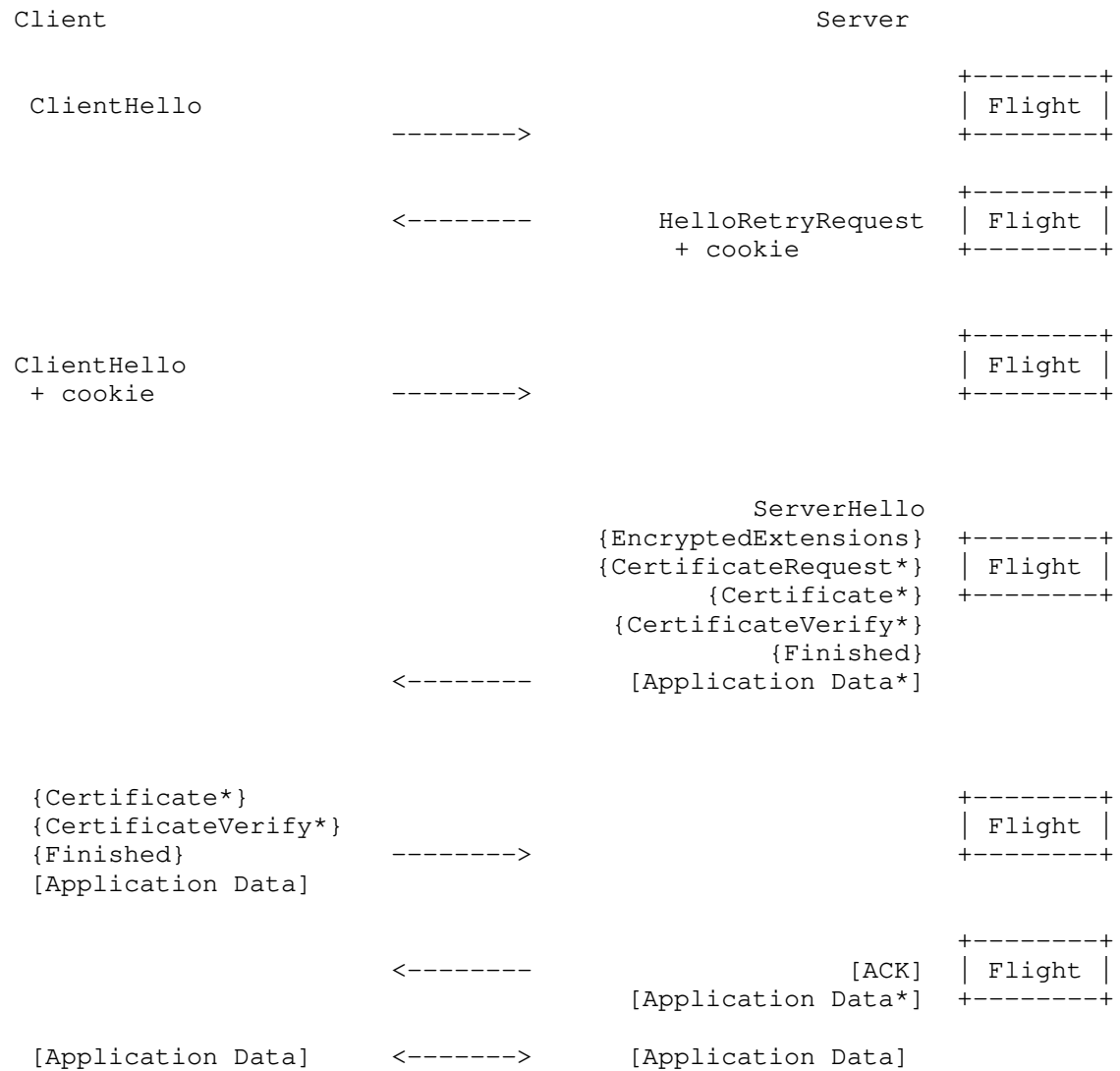


Figure 7: Message flights for a full DTLS Handshake (with cookie exchange)

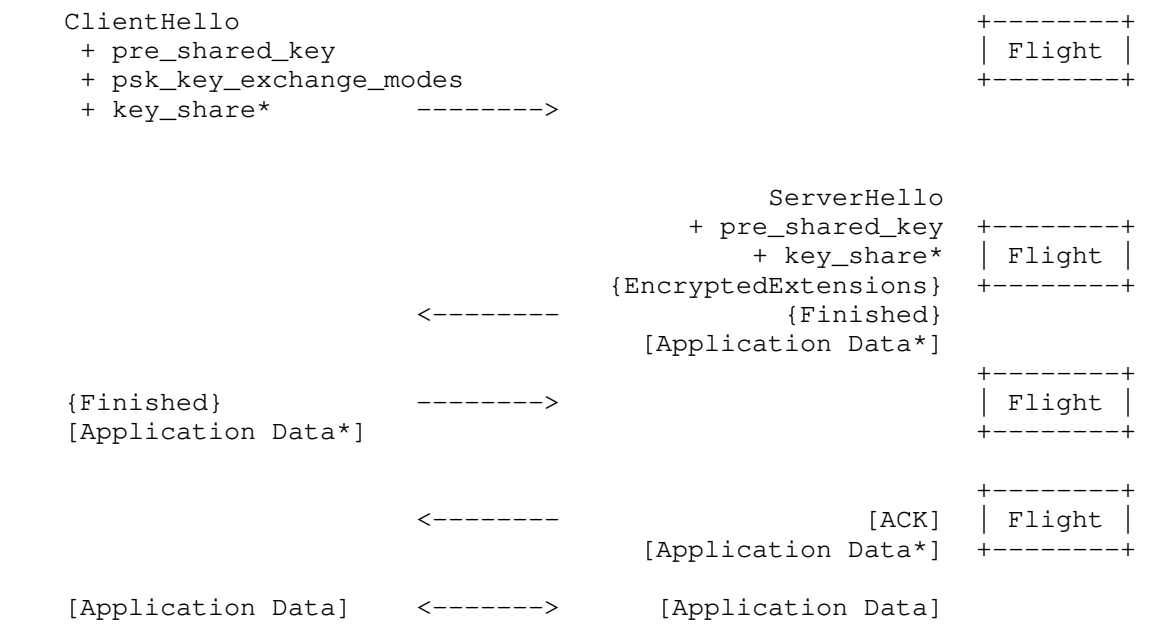


Figure 8: Message flights for resumption and PSK handshake (without cookie exchange)

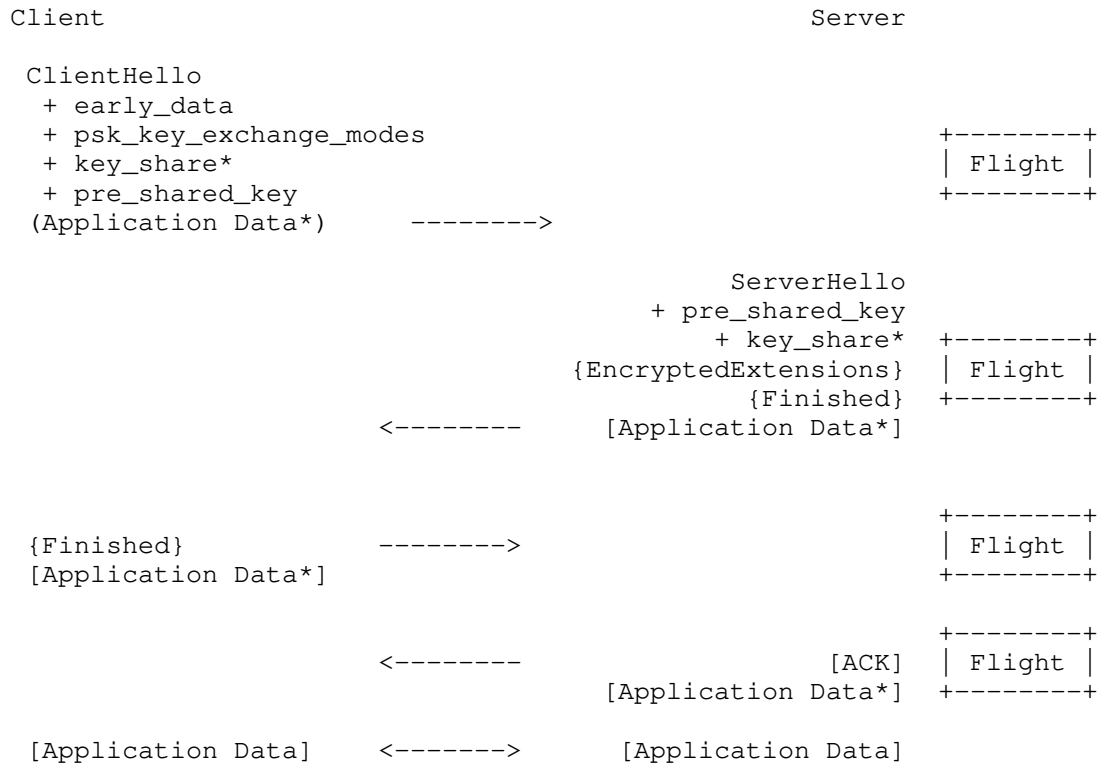
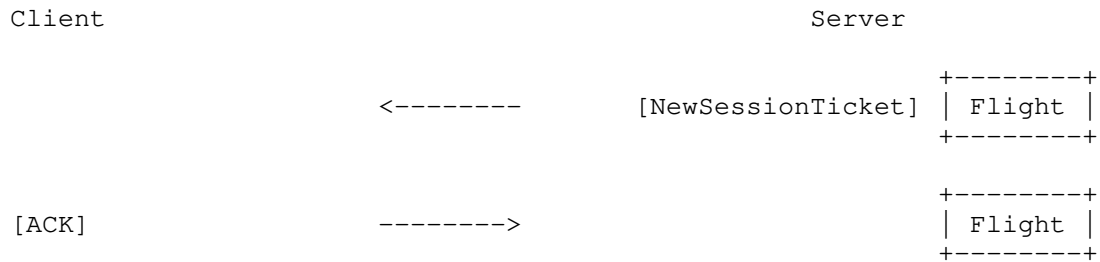


Figure 9: Message flights for the Zero-RTT handshake

Figure 10: Message flights for the `NewSessionTicket` message

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

5.8. Timeout and Retransmission

5.8.1. State Machine

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

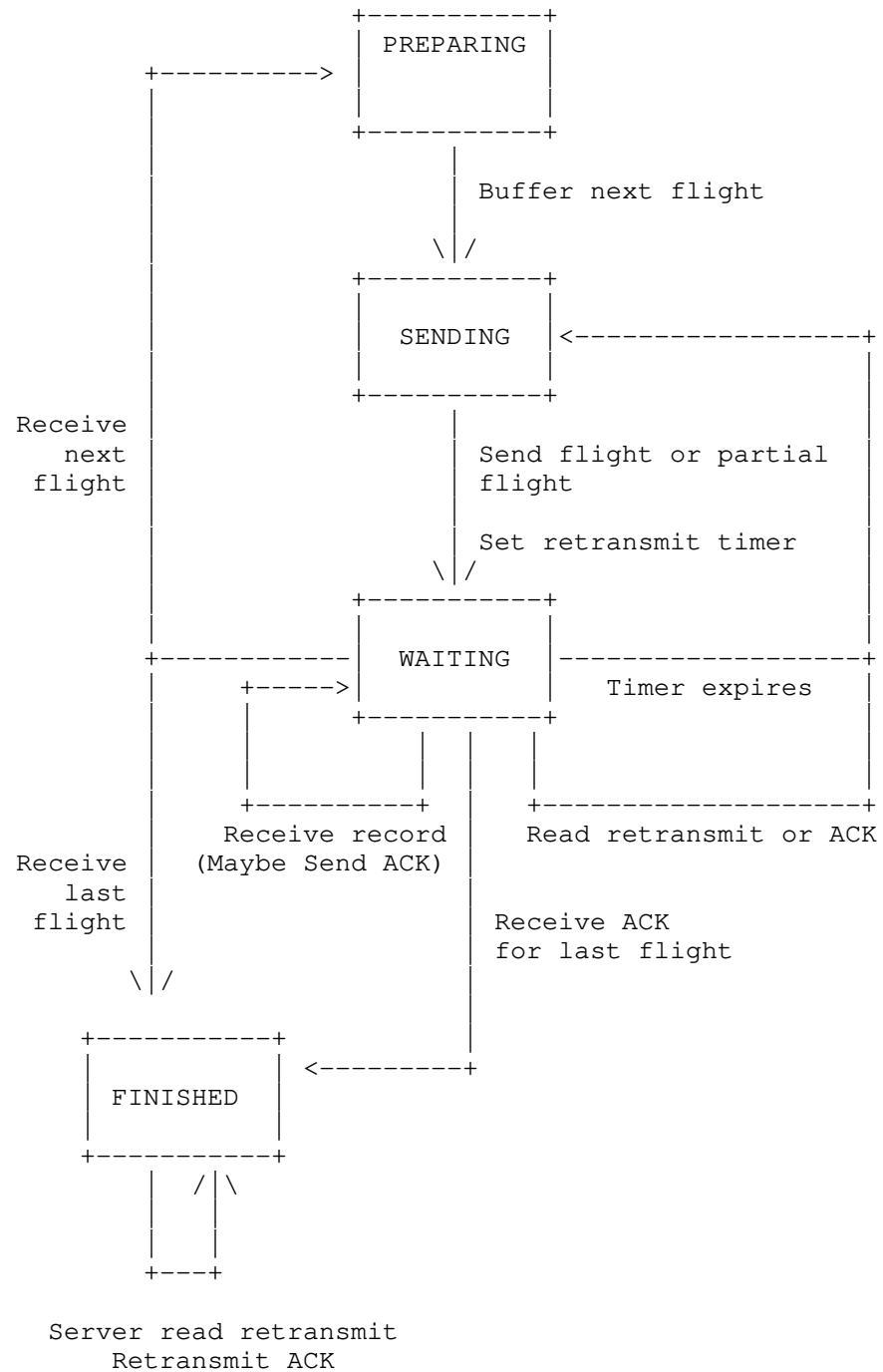


Figure 11: DTLS timeout and retransmission state machine

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

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

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

There are four ways to exit the WAITING state:

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

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

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

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

5.8.2. Timer Values

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

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

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

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

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

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

5.8.3. Large Flight Sizes

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

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

5.8.4. State machine duplication for post-handshake messages

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

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

5. Post-handshake client authentication

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

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

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

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

5.9. CertificateVerify and Finished Messages

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

5.10. Cryptographic Label Prefix

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

5.11. Alert Messages

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

5.12. Establishing New Associations with Existing Parameters

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

blind attackers from destroying associations merely by sending forged ClientHellos.

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

6. Example of Handshake with Timeout and Retransmission

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

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

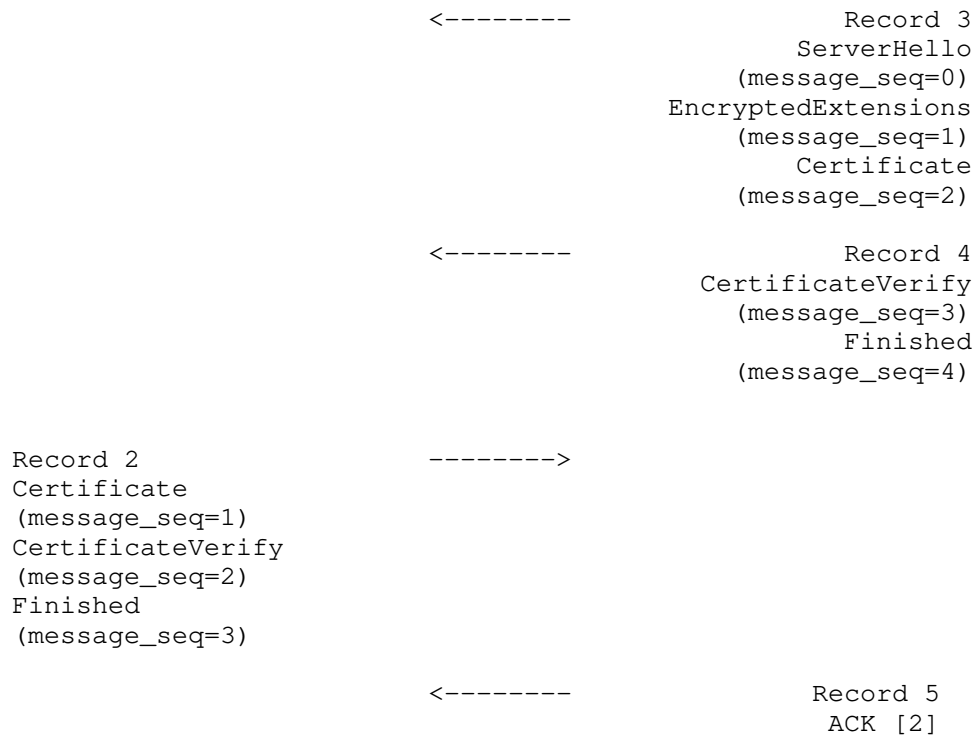


Figure 12: Example DTLS exchange illustrating message loss

6.1. Epoch Values and Rekeying

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

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

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

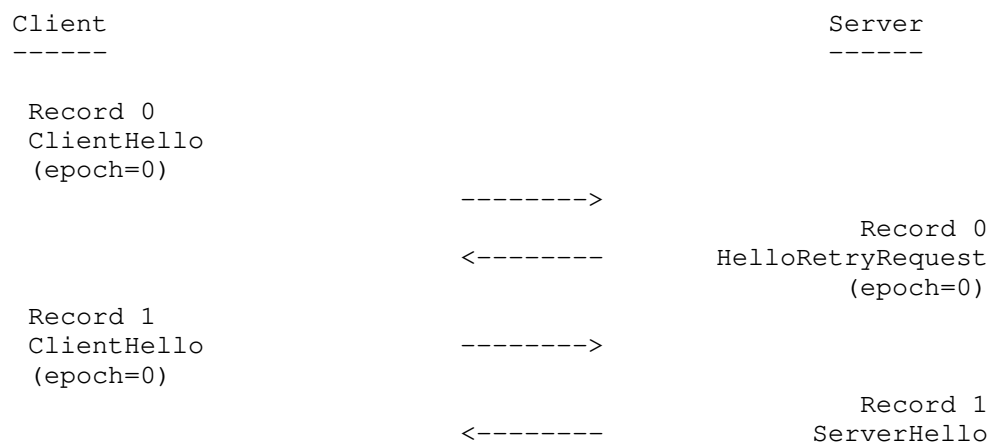
- * epoch value (1) is used for messages protected using keys derived from `client_early_traffic_secret`. Note this epoch is skipped if the client does not offer early data.
- * epoch value (2) is used for messages protected using keys derived from `[sender]_handshake_traffic_secret`. Messages transmitted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

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

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

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

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



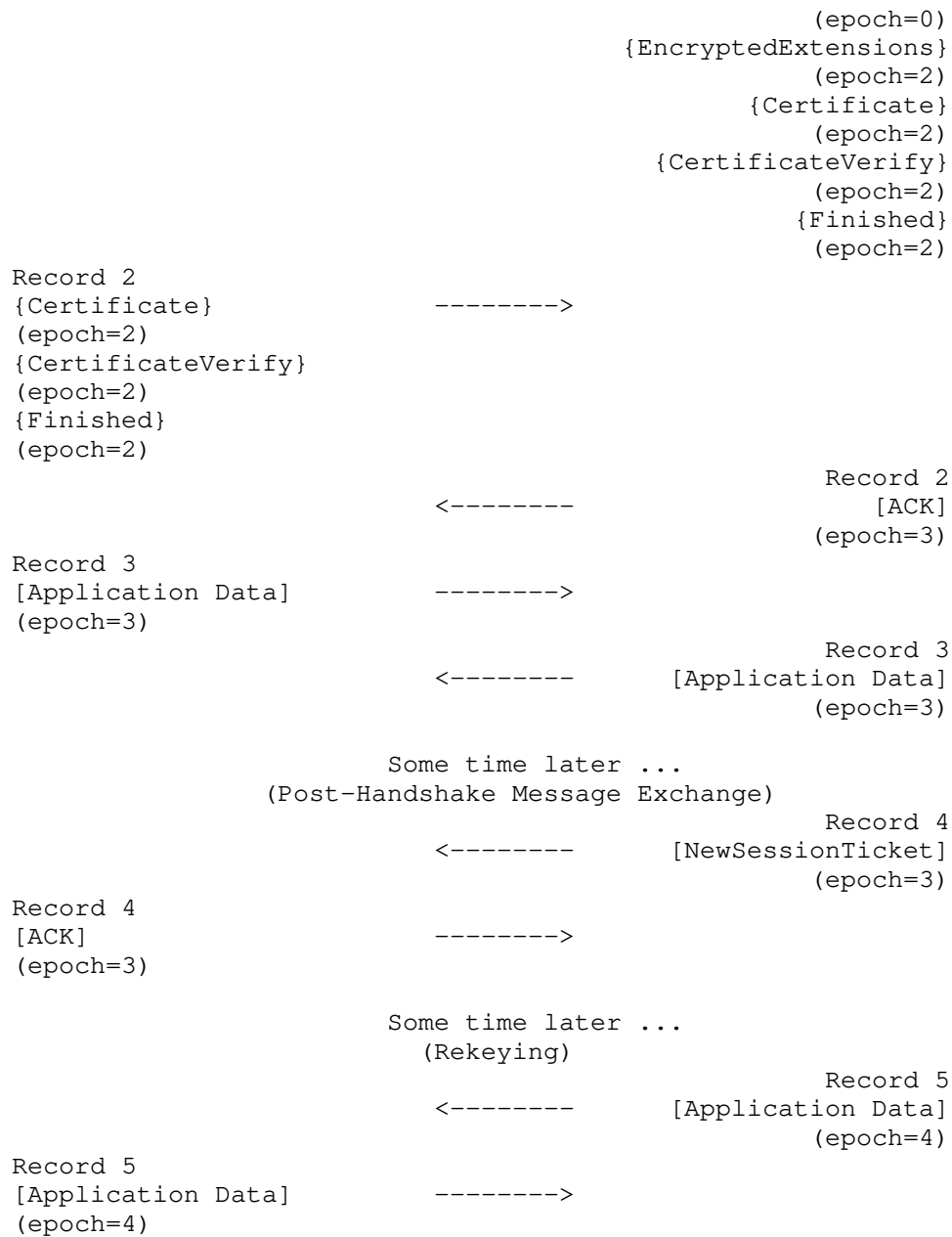


Figure 13: Example DTLS exchange with epoch information

7. ACK Message

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

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

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

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

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

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

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

7.1. Sending ACKs

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

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

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

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

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

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

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

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

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

7.2. Receiving ACKs

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

7.3. Design Rationale

ACK messages are used in two circumstances, namely :

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

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

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

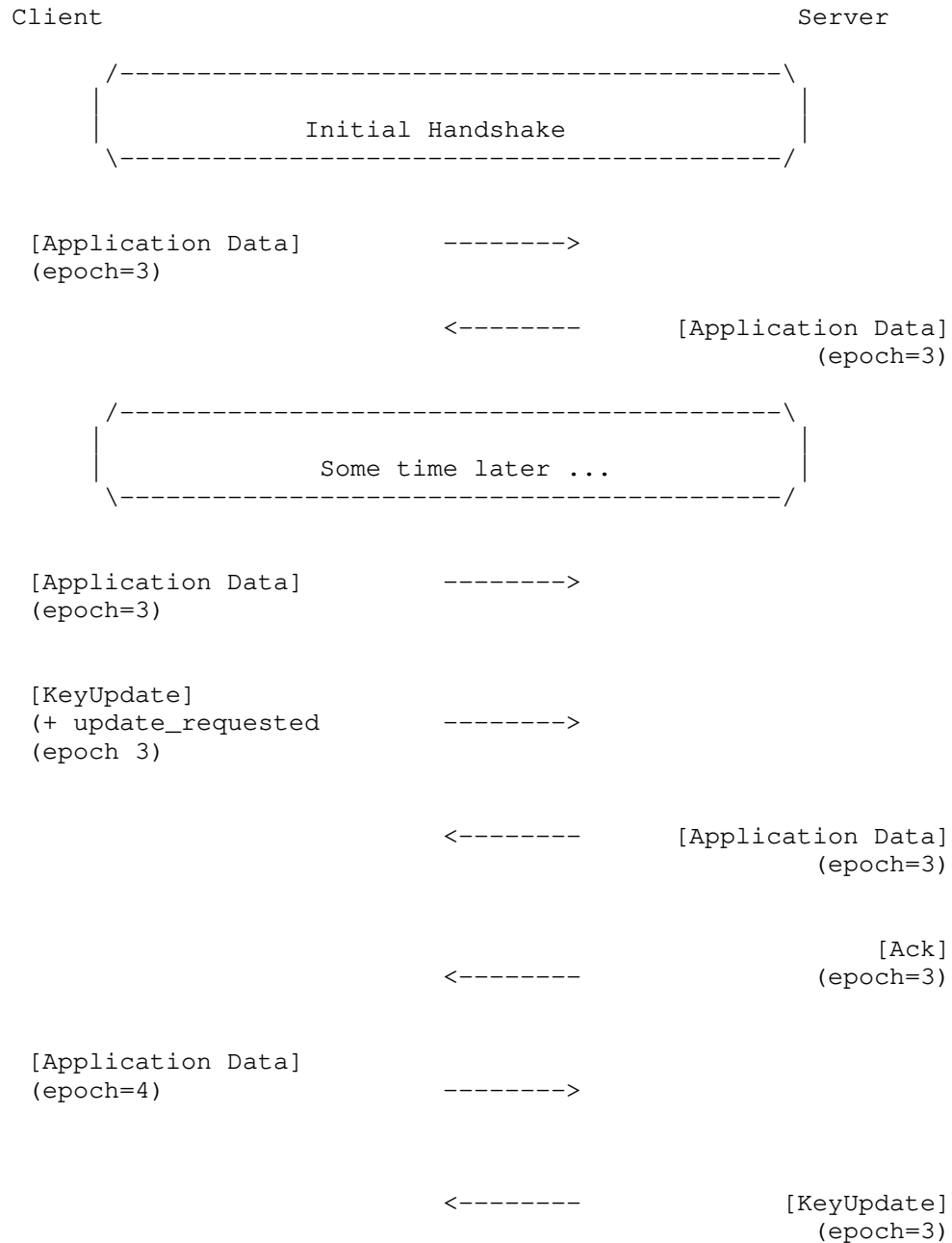
8. Key Updates

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

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

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

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



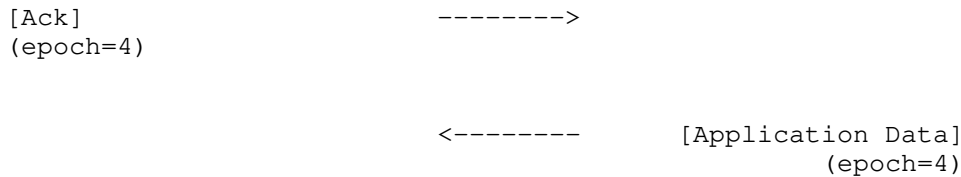


Figure 14: Example DTLS Key Update

9. Connection ID Updates

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

```

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

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

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

```

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

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

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

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

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

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

num_cids The number of CIDs desired.

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

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

9.1. Connection ID Example

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

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

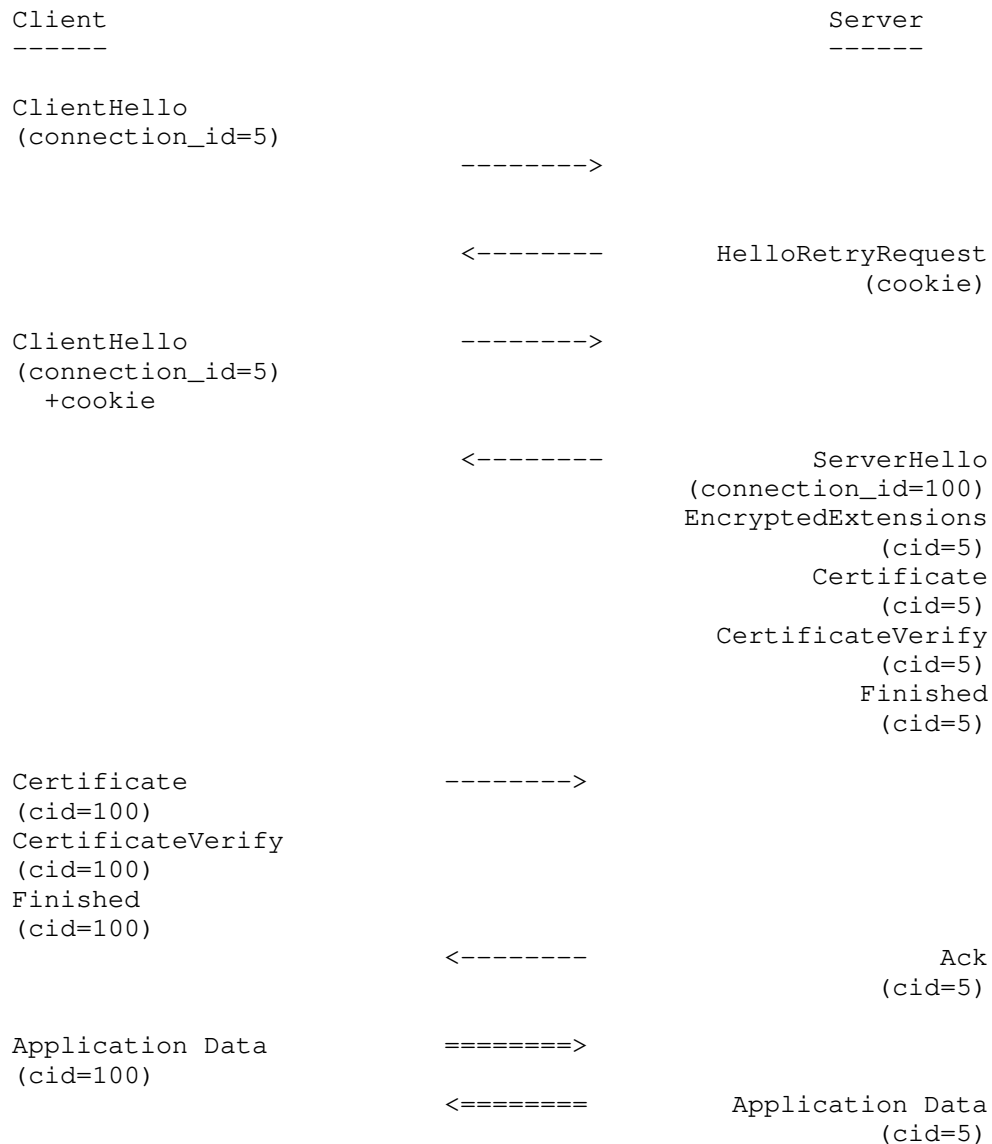


Figure 15: Example DTLS 1.3 Exchange with CIDs

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

10. Application Data Protocol

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

11. Security Considerations

Security issues are discussed primarily in [TLS13].

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

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

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

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

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

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

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

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

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

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

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

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

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

12. Changes since DTLS 1.2

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

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

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

13. Updates affecting DTLS 1.2

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

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

14. IANA Considerations

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

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

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

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

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

15. References

15.1. Normative References

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

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

15.2. Informative References

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

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

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

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

Appendix A. Protocol Data Structures and Constant Values

This section provides the normative protocol types and constants definitions.

A.1. Record Layer

```

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

```

```

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

```

```

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

```

```

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

```

Legend:

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

```

struct {
    uint16 epoch;
    uint48 sequence_number;
} RecordNumber;

```

A.2. Handshake Protocol

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

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

uint16 ProtocolVersion;
opaque Random[32];

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

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

A.3. ACKs

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

A.4. Connection ID Management

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

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

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

struct {
    uint8 num_cids;
} RequestConnectionId;
```

Appendix B. Analysis of Limits on CCM Usage

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

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

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

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

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

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

B.1. Confidentiality Limits

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

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

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

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

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

B.2. Integrity Limits

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

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

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

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

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

B.3. Limits for AEAD_AES_128_CCM_8

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

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

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

$$v \leq 2^7$$

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

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

Appendix C. Implementation Pitfalls

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

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

Appendix D. History

RFC EDITOR: PLEASE REMOVE THE THIS SECTION

(*) indicates a change that may affect interoperability.

IETF Drafts draft-42

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

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

draft-40

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

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

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

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

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

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

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

draft-32: - Editorial improvements and clarifications.

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

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

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

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

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

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

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

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

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

draft-00 - First IETF Draft

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

draft-00

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

Appendix E. Working Group Information

RFC EDITOR: PLEASE REMOVE THIS SECTION.

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

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

Appendix F. Contributors

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

In addition, we would like to thank:

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

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

Appendix G. Acknowledgements

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

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

Authors' Addresses

Eric Rescorla
RTFM, Inc.

Email: ekr@rtfm.com

Hannes Tschofenig
Arm Limited

Email: hannes.tschofenig@arm.com

Nagendra Modadugu
Google, Inc.

Email: nagendra@cs.stanford.edu

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

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

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

Abstract

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

Discussion Venues

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

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

Status of This Memo

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

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

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

This Internet-Draft will expire on 17 August 2022.

Copyright Notice

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

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

Table of Contents

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

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

1. Introduction

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

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

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

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

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

2. Conventions and Definitions

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

3. Overview

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

3.1. Topologies

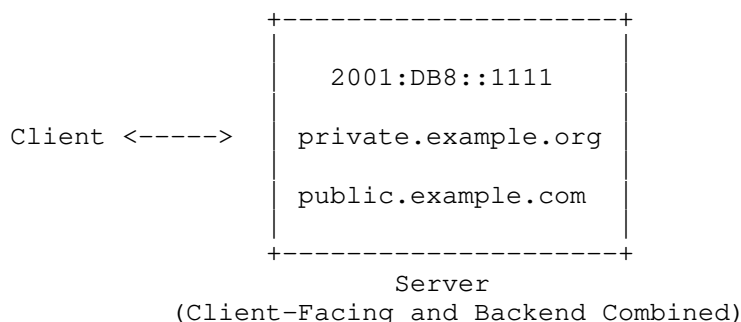


Figure 1: Shared Mode Topology

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

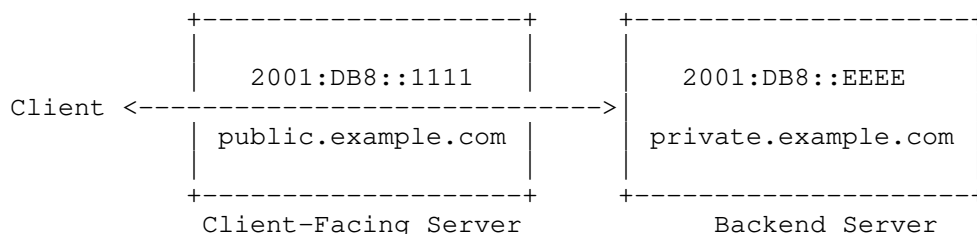


Figure 2: Split Mode Topology

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

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

3.2. Encrypted ClientHello (ECH)

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

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

The server takes one of the following actions:

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

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

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

4. Encrypted ClientHello Configuration

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

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

struct {
    HpkeKdfId kdf_id;
    HpkeAeadId aead_id;
} HpkeSymmetricCipherSuite;

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

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

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

The structure contains the following fields:

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

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

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

The ECHConfigContents structure contains the following fields:

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

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

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

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

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

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

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

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

The HpkeKeyConfig structure contains the following fields:

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

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

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

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

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

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

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

4.1. Configuration Identifiers

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

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

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

4.2. Configuration Extensions

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

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

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

5. The "encrypted_client_hello" Extension

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

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

The payload of the extension has the following structure:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

5.1. Encoding the ClientHelloInner

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

5.2. Authenticating the ClientHelloOuter

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

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

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

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

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

6. Client Behavior

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

6.1. Offering ECH

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

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

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

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

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

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

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

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

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

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

6.1.1. Encrypting the ClientHello

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

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

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

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

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

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

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

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

6.1.2. GREASE PSK

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

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

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

6.1.3. Recommended Padding Scheme

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

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

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

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

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

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

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

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

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

6.1.4. Determining ECH Acceptance

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

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

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

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

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

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

6.1.5. Handshaking with `ClientHelloInner`

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

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

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

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

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

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

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

6.1.6. Handshaking with ClientHelloOuter

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

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

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

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

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

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

6.1.7. Authenticating for the Public Name

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

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

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

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

6.2. GREASE ECH

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

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

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

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

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

7. Server Behavior

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

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

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

7.1. Client-Facing Server

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

7.1.1. Sending HelloRetryRequest

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

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

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

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

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

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

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

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

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

7.2. Backend Server

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

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

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

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

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

7.2.1. Sending HelloRetryRequest

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

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

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

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

8. Compatibility Issues

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

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

8.1. Misconfiguration and Deployment Concerns

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

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

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

8.2. Middleboxes

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

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

9. Compliance Requirements

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

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

10. Security Considerations

10.1. Security and Privacy Goals

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

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

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

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

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

10.2. Unauthenticated and Plaintext DNS

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

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

10.3. Client Tracking

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

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

10.4. Ignored Configuration Identifiers and Trial Decryption

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

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

10.5. Outer ClientHello

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

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

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

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

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

10.6. Related Privacy Leaks

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

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

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

10.7. Cookies

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

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

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

10.8. Attacks Exploiting Acceptance Confirmation

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

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

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

10.9. Comparison Against Criteria

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

10.9.1. Mitigate Cut-and-Paste Attacks

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

10.9.2. Avoid Widely Shared Secrets

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

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

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

10.9.4. Do Not Stick Out

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

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

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

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

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

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

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

10.9.5. Maintain Forward Secrecy

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

10.9.6. Enable Multi-party Security Contexts

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

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

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

10.9.7. Support Multiple Protocols

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

10.10. Padding Policy

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

10.11. Active Attack Mitigations

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

10.11.1. Client Reaction Attack Mitigation

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

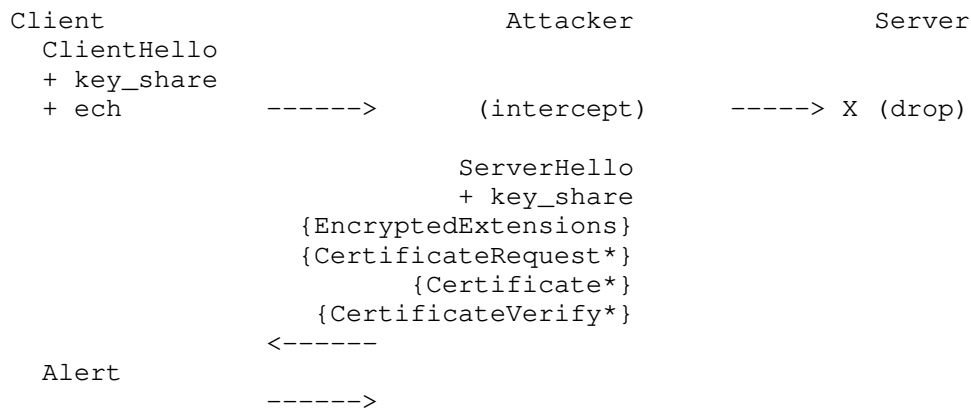


Figure 3: Client reaction attack

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

10.11.2. HelloRetryRequest Hijack Mitigation

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

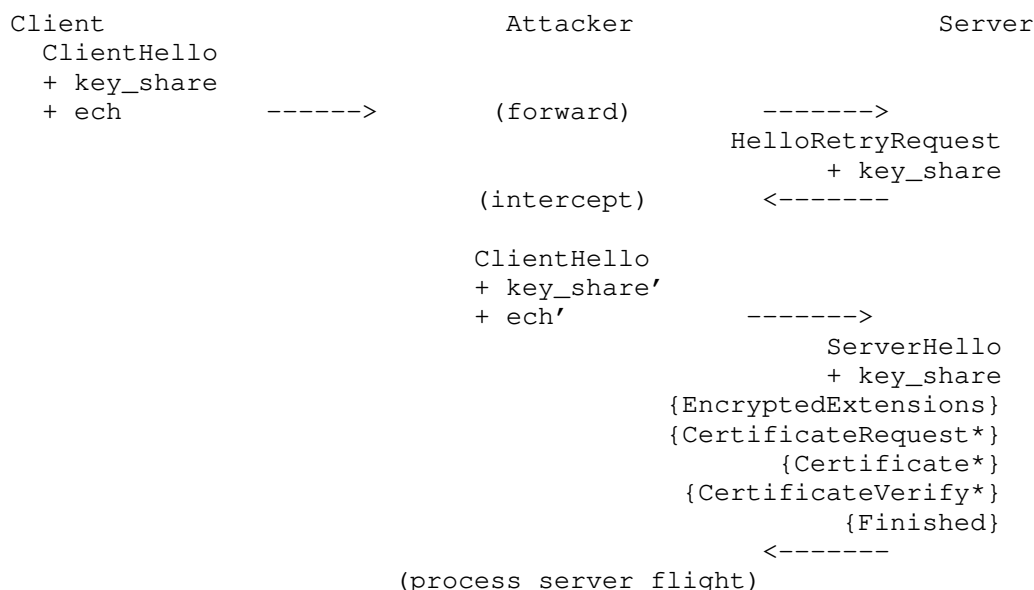


Figure 4: HelloRetryRequest hijack attack

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

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

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

10.11.3. ClientHello Malleability Mitigation

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

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

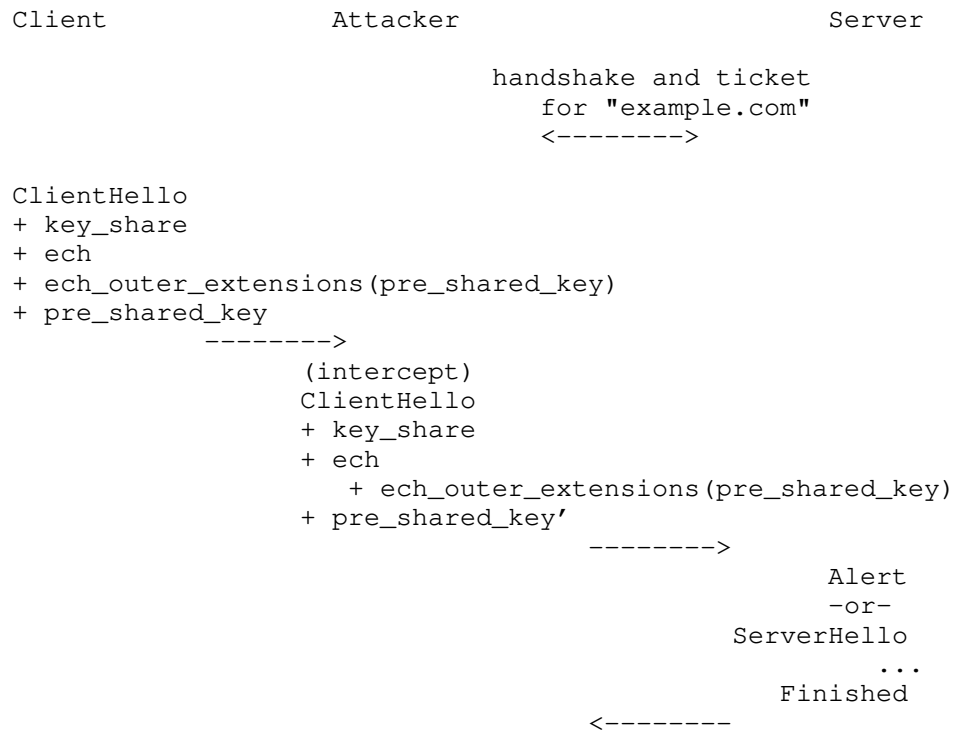


Figure 5: Message flow for malleable ClientHello

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

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

10.11.4. ClientHelloInner Packet Amplification Mitigation

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

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

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

11. IANA Considerations

11.1. Update of the TLS ExtensionType Registry

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

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

11.2. Update of the TLS Alert Registry

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

12. ECHConfig Extension Guidance

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

13. References

13.1. Normative References

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

13.2. Informative References

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

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

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

Appendix A. Alternative SNI Protection Designs

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

A.1. TLS-layer

A.1.1. TLS in Early Data

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

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

A.1.2. Combined Tickets

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

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

A.2. Application-layer

A.2.1. HTTP/2 CERTIFICATE Frames

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

Appendix B. Linear-time Outer Extension Processing

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

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

Appendix C. Acknowledgements

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

Appendix D. Change Log

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

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

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

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

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

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

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

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

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

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

Authors' Addresses

Eric Rescorla
RTFM, Inc.

Email: ekr@rtfm.com

Kazuho Oku
Fastly

Email: kazuhooku@gmail.com

Nick Sullivan
Cloudflare

Email: nick@cloudflare.com

Christopher A. Wood
Cloudflare

Email: caw@heapingbits.net

Internet Engineering Task Force
Internet-Draft
Obsoletes: 5469 7507 (if approved)

Updates: 8422 8261 7568 7562 7525 7465
7030 6750 6749 6739 6460 6614
6367 6353 6347 6176 6084 6083
6042 6012 5953 5878 5734 5456
5422 5415 5364 5281 5263 5238
5216 5158 5091 5054 5049 5024
5023 5019 5018 4992 4976 4975
4964 4851 4823 4791 4785 4744
4743 4732 4712 4681 4680 4642
4616 4582 4540 4531 4513 4497
4279 4261 4235 4217 4168 4162
4111 4097 3983 3943 3903 3887
3871 3856 3767 3749 3656 3568
3552 3501 3470 3436 3329 3261
(if approved)

Intended status: Best Current Practice
Expires: July 25, 2021

K. Moriarty
Dell EMC
S. Farrell
Trinity College Dublin
January 21, 2021

Deprecating TLSv1.0 and TLSv1.1
draft-ietf-tls-oldversions-deprecate-12

Abstract

This document, if approved, formally deprecates Transport Layer Security (TLS) versions 1.0 (RFC 2246) and 1.1 (RFC 4346). Accordingly, those documents (will be moved|have been moved) to Historic status. These versions lack support for current and recommended cryptographic algorithms and mechanisms, and various government and industry profiles of applications using TLS now mandate avoiding these old TLS versions. TLSv1.2 became the recommended version for IETF protocols in 2008, (subsequently being obsoleted by TLSv1.3 in 2018), providing sufficient time to transition away from older versions. Removing support for older versions from implementations reduces the attack surface, reduces opportunity for misconfiguration, and streamlines library and product maintenance.

This document also deprecates Datagram TLS (DTLS) version 1.0 (RFC 4347), but not DTLS version 1.2, and there is no DTLS version 1.1.

This document updates many RFCs that normatively refer to TLSv1.0 or TLSv1.1 as described herein. This document also updates the best practices for TLS usage in RFC 7525 and hence is part of BCP 195.

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 July 25, 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.

Table of Contents

1. Introduction	3
1.1. RFCs Updated	3
1.2. Terminology	5
2. Support for Deprecation	5
3. SHA-1 Usage Problematic in TLSv1.0 and TLSv1.1	6
4. Do Not Use TLSv1.0	6
5. Do Not Use TLSv1.1	7
6. Updates to RFC 7525	8
7. Operational Considerations	8
8. Security Considerations	9
9. Acknowledgements	9
10. IANA Considerations	9
11. References	9
11.1. Normative References	9

11.2. Informative References	18
Appendix A. Change Log	22
Authors' Addresses	24

1. Introduction

Transport Layer Security (TLS) versions 1.0 [RFC2246] and 1.1 [RFC4346] were superseded by TLSv1.2 [RFC5246] in 2008, which has now itself been superseded by TLSv1.3 [RFC8446]. Datagram Transport Layer Security (DTLS) version 1.0 [RFC4347] was superseded by DTLSv1.2 [RFC6347] in 2012. It is therefore timely to further deprecate TLSv1.0, TLSv1.1 and DTLSv1.0. Accordingly, those documents (will be moved|have been moved) to Historic status.

Technical reasons for deprecating these versions include:

- o They require implementation of older cipher suites that are no longer desirable for cryptographic reasons, e.g., TLSv1.0 makes TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA mandatory to implement
- o Lack of support for current recommended cipher suites, especially AEAD ciphers which are not supported prior to TLSv1.2. Note: registry entries for no-longer-desirable ciphersuites remain in the registries, but many TLS registries are being updated through [RFC8447] which indicates that such entries are not recommended by the IETF.
- o Integrity of the handshake depends on SHA-1 hash.
- o Authentication of the peers depends on SHA-1 signatures.
- o Support for four TLS protocol versions increases the likelihood of misconfiguration.
- o At least one widely-used library has plans to drop TLSv1.1 and TLSv1.0 support in upcoming releases; products using such libraries would need to use older versions of the libraries to support TLSv1.0 and TLSv1.1, which is clearly undesirable.

Deprecation of these versions is intended to assist developers as additional justification to no longer support older (D)TLS versions and to migrate to a minimum of (D)TLSv1.2. Deprecation also assists product teams with phasing out support for the older versions, to reduce the attack surface and the scope of maintenance for protocols in their offerings.

1.1. RFCs Updated

This document updates the following RFCs that normatively reference TLSv1.0 or TLSv1.1 or DTLS1.0. The update is to obsolete usage of these older versions. Fallback to these versions is prohibited through this update. Specific references to mandatory minimum protocol versions of TLSv1.0 or TLSv1.1 are replaced by TLSv1.2, and

references to minimum protocol version DTLSv1.0 are replaced by DTLSv1.2. Statements that "TLSv1.0 is the most widely deployed version and will provide the broadest interoperability" are removed without replacement.

[RFC8422] [RFC8261] [RFC7568] [RFC7562] [RFC7525] [RFC7465] [RFC7030]
[RFC6750] [RFC6749] [RFC6739] [RFC6084] [RFC6083] [RFC6367] [RFC6353]
[RFC6176] [RFC6042] [RFC6012] [RFC5878] [RFC5734] [RFC5456] [RFC5422]
[RFC5415] [RFC5364] [RFC5281] [RFC5263] [RFC5238] [RFC5216] [RFC5158]
[RFC5091] [RFC5054] [RFC5049] [RFC5024] [RFC5023] [RFC5019] [RFC5018]
[RFC4992] [RFC4976] [RFC4975] [RFC4964] [RFC4851] [RFC4823] [RFC4791]
[RFC4785] [RFC4732] [RFC4712] [RFC4681] [RFC4680] [RFC4642] [RFC4616]
[RFC4582] [RFC4540] [RFC4531] [RFC4513] [RFC4497] [RFC4279] [RFC4261]
[RFC4235] [RFC4217] [RFC4168] [RFC4162] [RFC4111] [RFC4097] [RFC3983]
[RFC3943] [RFC3903] [RFC3887] [RFC3871] [RFC3856] [RFC3767] [RFC3749]
[RFC3656] [RFC3568] [RFC3552] [RFC3501] [RFC3470] [RFC3436] [RFC3329]
[RFC3261]

The status of [RFC7562], [RFC6042], [RFC5456], [RFC5024], [RFC4540], and [RFC3656] will be updated with permission of the Independent Stream Editor.

In addition these RFCs normatively refer to TLSv1.0 or TLSv1.1 and have already been obsoleted; they are still listed here and marked as updated by this document in order to reiterate that any usage of the obsolete protocol should still use modern TLS: [RFC5953] [RFC5101] [RFC5081] [RFC5077] [RFC4934] [RFC4572] [RFC4507] [RFC4492] [RFC4366] [RFC4347] [RFC4244] [RFC4132] [RFC3920] [RFC3734] [RFC3588] [RFC3546] [RFC3489] [RFC3316]

Note that [RFC4642] has already been updated by [RFC8143], which makes an overlapping, but not quite identical, update as this document.

[RFC6614] has a requirement for TLSv1.1 or later, although only makes an informative reference to [RFC4346]. This requirement is updated to be for TLSv1.2 or later.

[RFC6460], [RFC4744], and [RFC4743] are already Historic; they are still listed here and marked as updated by this document in order to reiterate that any usage of the obsolete protocol should still use modern TLS.

This document updates DTLS [RFC6347]. [RFC6347] had allowed for negotiating the use of DTLSv1.0, which is now forbidden.

The DES and IDEA cipher suites specified in [RFC5469] were specifically removed from TLSv1.2 by [RFC5246]; since the only

versions of TLS for which their usage is defined are now Historic, RFC 5469 (will be|has been) moved to Historic as well.

The version-fallback Signaling Cipher Suite Value specified in [RFC7507] was defined to detect when a given client and server negotiate a lower version of (D)TLS than their highest shared version. TLSv1.3 ([RFC8446]) incorporates a different mechanism that achieves this purpose, via sentinel values in the ServerHello.Random field. With (D)TLS versions prior to 1.2 fully deprecated, the only way for (D)TLS implementations to negotiate a lower version than their highest shared version would be to negotiate (D)TLSv1.2 while supporting (D)TLSv1.3; supporting (D)TLSv1.3 implies support for the ServerHello.Random mechanism. Accordingly, the functionality from [RFC7507] has been superseded, and this document marks it as Obsolete.

1.2. 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.

2. Support for Deprecation

Specific details on attacks against TLSv1.0 and TLSv1.1, as well as their mitigations, are provided in [NIST800-52r2], RFC 7457 [RFC7457] and other RFCs referenced therein. Although mitigations for the current known vulnerabilities have been developed, any future issues discovered in old protocol versions might not be mitigated in older library versions when newer library versions do not support those old protocols.

NIST for example has provided the following rationale, copied with permission from [NIST800-52r2], section 1.2 "History of TLS" (with references changed for RFC formatting).

TLS 1.1, specified in [RFC4346], was developed to address weaknesses discovered in TLS 1.0, primarily in the areas of initialization vector selection and padding error processing. Initialization vectors were made explicit to prevent a certain class of attacks on the Cipher Block Chaining (CBC) mode of operation used by TLS. The handling of padding errors was altered to treat a padding error as a bad message authentication code, rather than a decryption failure. In addition, the TLS 1.1 RFC acknowledges attacks on CBC mode that rely on the time to compute the message authentication code (MAC). The TLS 1.1 specification

states that to defend against such attacks, an implementation must process records in the same manner regardless of whether padding errors exist. Further implementation considerations for CBC modes (which were not included in RFC4346 [RFC4346]) are discussed in Section 3.3.2.

TLSv1.2, specified in RFC5246 [RFC5246], made several cryptographic enhancements, particularly in the area of hash functions, with the ability to use or specify the SHA-2 family algorithms for hash, MAC, and Pseudorandom Function (PRF) computations. TLSv1.2 also adds authenticated encryption with associated data (AEAD) cipher suites.

TLSv1.3, specified in TLSv1.3 [RFC8446], represents a significant change to TLS that aims to address threats that have arisen over the years. Among the changes are a new handshake protocol, a new key derivation process that uses the HMAC-based Extract-and-Expand Key Derivation Function (HKDF), and the removal of cipher suites that use static RSA or DH key exchanges, the CBC mode of operation, or SHA-1. The list of extensions that can be used with TLSv1.3 has been reduced considerably.

3. SHA-1 Usage Problematic in TLSv1.0 and TLSv1.1

The integrity of both TLSv1.0 and TLSv1.1 depends on a running SHA-1 hash of the exchanged messages. This makes it possible to perform a downgrade attack on the handshake by an attacker able to perform 2^{77} operations, well below the acceptable modern security margin.

Similarly, the authentication of the handshake depends on signatures made using a SHA-1 hash or a not appreciably stronger concatenation of MD-5 and SHA-1 hashes, allowing the attacker to impersonate a server when it is able to break the severely weakened SHA-1 hash.

Neither TLSv1.0 nor TLSv1.1 allow the peers to select a stronger hash for signatures in the ServerKeyExchange or CertificateVerify messages, making the only upgrade path the use of a newer protocol version.

See [Bhargavan2016] for additional detail.

4. Do Not Use TLSv1.0

TLSv1.0 MUST NOT be used. Negotiation of TLSv1.0 from any version of TLS MUST NOT be permitted.

Any other version of TLS is more secure than TLSv1.0. While TLSv1.0 can be configured to prevent some types of interception, using the highest version available is preferred.

Pragmatically, clients MUST NOT send a ClientHello with ClientHello.client_version set to {03,01}. Similarly, servers MUST NOT send a ServerHello with ServerHello.server_version set to {03,01}. Any party receiving a Hello message with the protocol version set to {03,01} MUST respond with a "protocol_version" alert message and close the connection.

Historically, TLS specifications were not clear on what the record layer version number (TLSPlaintext.version) could contain when sending ClientHello. Appendix E of [RFC5246] notes that TLSPlaintext.version could be selected to maximize interoperability, though no definitive value is identified as ideal. That guidance is still applicable; therefore, TLS servers MUST accept any value {03,XX} (including {03,00}) as the record layer version number for ClientHello, but they MUST NOT negotiate TLSv1.0.

5. Do Not Use TLSv1.1

TLSv1.1 MUST NOT be used. Negotiation of TLSv1.1 from any version of TLS MUST NOT be permitted.

Pragmatically, clients MUST NOT send a ClientHello with ClientHello.client_version set to {03,02}. Similarly, servers MUST NOT send a ServerHello with ServerHello.server_version set to {03,02}. Any party receiving a Hello message with the protocol version set to {03,02} MUST respond with a "protocol_version" alert message and close the connection.

Any newer version of TLS is more secure than TLSv1.1. While TLSv1.1 can be configured to prevent some types of interception, using the highest version available is preferred. Support for TLSv1.1 is dwindling in libraries and will impact security going forward if mitigations for attacks cannot be easily addressed and supported in older libraries.

Historically, TLS specifications were not clear on what the record layer version number (TLSPlaintext.version) could contain when sending ClientHello. Appendix E of [RFC5246] notes that TLSPlaintext.version could be selected to maximize interoperability, though no definitive value is identified as ideal. That guidance is still applicable; therefore, TLS servers MUST accept any value {03,XX} (including {03,00}) as the record layer version number for ClientHello, but they MUST NOT negotiate TLSv1.1.

6. Updates to RFC 7525

RFC7525 is BCP 195, "Recommendations for Secure Use of Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)", which is the most recent best practice document for implementing TLS and was based on TLSv1.2. At the time of publication, TLSv1.0 and TLSv1.1 had not yet been deprecated. As such, BCP 195 is called out specifically to update text implementing the deprecation recommendations of this document.

This document updates [RFC7525] Section 3.1.1 changing SHOULD NOT to MUST NOT as follows:

- o Implementations MUST NOT negotiate TLS version 1.0 [RFC2246].

Rationale: TLSv1.0 (published in 1999) does not support many modern, strong cipher suites. In addition, TLSv1.0 lacks a per-record Initialization Vector (IV) for CBC-based cipher suites and does not warn against common padding errors.

- o Implementations MUST NOT negotiate TLS version 1.1 [RFC4346].

Rationale: TLSv1.1 (published in 2006) is a security improvement over TLSv1.0 but still does not support certain stronger cipher suites.

This documents updates [RFC7525] Section 3.1.2 changing SHOULD NOT to MUST NOT as follows:

- o Implementations MUST NOT negotiate DTLS version 1.0 [RFC4347], [RFC6347].

Version 1.0 of DTLS correlates to version 1.1 of TLS (see above).

7. Operational Considerations

This document is part of BCP 195, and as such reflects the understanding of the IETF (at the time of its publication) as to the best practices for TLS and DTLS usage.

Though TLSv1.1 has been obsolete since the publication of RFC 5246 in 2008, and DTLSv1.0 has been obsolete since the publication of RFC 6347 in 2012, there may remain some systems in operation that do not support (D)TLSv1.2 or higher. Adopting the practices recommended by this document for any systems that need to communicate with the aforementioned class of systems will cause failure to interoperate. However, disregarding the recommendations of this document in order to continue to interoperate with the aforementioned class of systems

incurs some amount of risk. The nature of the risks incurred by operating in contravention to the recommendations of this document are discussed in Sections 2 and 3, and knowledge of those risks should be used along with any potential mitigating factors and the risks inherent to updating the systems in question when deciding how quickly to adopt the recommendations specified in this document.

8. Security Considerations

This document deprecates two older TLS protocol versions and one older DTLS protocol version for security reasons already described. The attack surface is reduced when there are a smaller number of supported protocols and fallback options are removed.

9. Acknowledgements

Thanks to those that provided usage data, reviewed and/or improved this document, including: Michael Ackermann, David Benjamin, David Black, Deborah Brungard, Alan DeKok, Viktor Dukhovni, Julien Elie, Adrian Farrell, Gary Gapinski, Alessandro Ghedini, Peter Gutmann, Jeremy Harris, Nick Hilliard, James Hodgkinson, Russ Housley, Hubert Kario, Benjamin Kaduk, John Klensin, Watson Ladd, Eliot Lear, Ted Lemon, John Mattsson, Keith Moore, Tom Petch, Eric Mill, Yoav Nir, Andrei Popov, Michael Richardson, Eric Rescorla, Rich Salz, Mohit Sethi, Yaron Sheffer, Rob Sayre, Robert Sparks, Barbara Stark, Martin Thomson, Sean Turner, Loganaden Velvindron, and Jakub Wilk.

[[Note to RFC editor: At least Julien Elie's name above should have an accent on the first letter of the surname. Please fix that and any others needing a similar fix if you can, I'm not sure the tooling I have now allows that.]]

10. IANA Considerations

[[This memo includes no request to IANA.]]

11. References

11.1. 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>>.
- [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>>.

- [RFC3261] Rosenberg, J., Schulzrinne, H., Camarillo, G., Johnston, A., Peterson, J., Sparks, R., Handley, M., and E. Schooler, "SIP: Session Initiation Protocol", RFC 3261, DOI 10.17487/RFC3261, June 2002, <<https://www.rfc-editor.org/info/rfc3261>>.
- [RFC3329] Arkko, J., Torvinen, V., Camarillo, G., Niemi, A., and T. Haukka, "Security Mechanism Agreement for the Session Initiation Protocol (SIP)", RFC 3329, DOI 10.17487/RFC3329, January 2003, <<https://www.rfc-editor.org/info/rfc3329>>.
- [RFC3436] Jungmaier, A., Rescorla, E., and M. Tuexen, "Transport Layer Security over Stream Control Transmission Protocol", RFC 3436, DOI 10.17487/RFC3436, December 2002, <<https://www.rfc-editor.org/info/rfc3436>>.
- [RFC3470] Hollenbeck, S., Rose, M., and L. Masinter, "Guidelines for the Use of Extensible Markup Language (XML) within IETF Protocols", BCP 70, RFC 3470, DOI 10.17487/RFC3470, January 2003, <<https://www.rfc-editor.org/info/rfc3470>>.
- [RFC3501] Crispin, M., "INTERNET MESSAGE ACCESS PROTOCOL - VERSION 4rev1", RFC 3501, DOI 10.17487/RFC3501, March 2003, <<https://www.rfc-editor.org/info/rfc3501>>.
- [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>>.
- [RFC3568] Barbir, A., Cain, B., Nair, R., and O. Spatscheck, "Known Content Network (CN) Request-Routing Mechanisms", RFC 3568, DOI 10.17487/RFC3568, July 2003, <<https://www.rfc-editor.org/info/rfc3568>>.
- [RFC3656] Siemborski, R., "The Mailbox Update (MUPDATE) Distributed Mailbox Database Protocol", RFC 3656, DOI 10.17487/RFC3656, December 2003, <<https://www.rfc-editor.org/info/rfc3656>>.
- [RFC3749] Hollenbeck, S., "Transport Layer Security Protocol Compression Methods", RFC 3749, DOI 10.17487/RFC3749, May 2004, <<https://www.rfc-editor.org/info/rfc3749>>.
- [RFC3767] Farrell, S., Ed., "Securely Available Credentials Protocol", RFC 3767, DOI 10.17487/RFC3767, June 2004, <<https://www.rfc-editor.org/info/rfc3767>>.

- [RFC3856] Rosenberg, J., "A Presence Event Package for the Session Initiation Protocol (SIP)", RFC 3856, DOI 10.17487/RFC3856, August 2004, <<https://www.rfc-editor.org/info/rfc3856>>.
- [RFC3871] Jones, G., Ed., "Operational Security Requirements for Large Internet Service Provider (ISP) IP Network Infrastructure", RFC 3871, DOI 10.17487/RFC3871, September 2004, <<https://www.rfc-editor.org/info/rfc3871>>.
- [RFC3887] Hansen, T., "Message Tracking Query Protocol", RFC 3887, DOI 10.17487/RFC3887, September 2004, <<https://www.rfc-editor.org/info/rfc3887>>.
- [RFC3903] Niemi, A., Ed., "Session Initiation Protocol (SIP) Extension for Event State Publication", RFC 3903, DOI 10.17487/RFC3903, October 2004, <<https://www.rfc-editor.org/info/rfc3903>>.
- [RFC3943] Friend, R., "Transport Layer Security (TLS) Protocol Compression Using Lempel-Ziv-Stac (LZS)", RFC 3943, DOI 10.17487/RFC3943, November 2004, <<https://www.rfc-editor.org/info/rfc3943>>.
- [RFC3983] Newton, A. and M. Sanz, "Using the Internet Registry Information Service (IRIS) over the Blocks Extensible Exchange Protocol (BEEP)", RFC 3983, DOI 10.17487/RFC3983, January 2005, <<https://www.rfc-editor.org/info/rfc3983>>.
- [RFC4097] Barnes, M., Ed., "Middlebox Communications (MIDCOM) Protocol Evaluation", RFC 4097, DOI 10.17487/RFC4097, June 2005, <<https://www.rfc-editor.org/info/rfc4097>>.
- [RFC4111] Fang, L., Ed., "Security Framework for Provider-Provisioned Virtual Private Networks (PPVPNs)", RFC 4111, DOI 10.17487/RFC4111, July 2005, <<https://www.rfc-editor.org/info/rfc4111>>.
- [RFC4162] Lee, H., Yoon, J., and J. Lee, "Addition of SEED Cipher Suites to Transport Layer Security (TLS)", RFC 4162, DOI 10.17487/RFC4162, August 2005, <<https://www.rfc-editor.org/info/rfc4162>>.
- [RFC4168] Rosenberg, J., Schulzrinne, H., and G. Camarillo, "The Stream Control Transmission Protocol (SCTP) as a Transport for the Session Initiation Protocol (SIP)", RFC 4168, DOI 10.17487/RFC4168, October 2005, <<https://www.rfc-editor.org/info/rfc4168>>.

- [RFC4217] Ford-Hutchinson, P., "Securing FTP with TLS", RFC 4217, DOI 10.17487/RFC4217, October 2005, <<https://www.rfc-editor.org/info/rfc4217>>.
- [RFC4235] Rosenberg, J., Schulzrinne, H., and R. Mahy, Ed., "An INVITE-Initiated Dialog Event Package for the Session Initiation Protocol (SIP)", RFC 4235, DOI 10.17487/RFC4235, November 2005, <<https://www.rfc-editor.org/info/rfc4235>>.
- [RFC4261] Walker, J. and A. Kulkarni, Ed., "Common Open Policy Service (COPS) Over Transport Layer Security (TLS)", RFC 4261, DOI 10.17487/RFC4261, December 2005, <<https://www.rfc-editor.org/info/rfc4261>>.
- [RFC4279] Eronen, P., Ed. and H. Tschofenig, Ed., "Pre-Shared Key Ciphersuites for Transport Layer Security (TLS)", RFC 4279, DOI 10.17487/RFC4279, December 2005, <<https://www.rfc-editor.org/info/rfc4279>>.
- [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>>.
- [RFC4497] Elwell, J., Derks, F., Mourot, P., and O. Rousseau, "Interworking between the Session Initiation Protocol (SIP) and QSIG", BCP 117, RFC 4497, DOI 10.17487/RFC4497, May 2006, <<https://www.rfc-editor.org/info/rfc4497>>.
- [RFC4513] Harrison, R., Ed., "Lightweight Directory Access Protocol (LDAP): Authentication Methods and Security Mechanisms", RFC 4513, DOI 10.17487/RFC4513, June 2006, <<https://www.rfc-editor.org/info/rfc4513>>.
- [RFC4531] Zeilenga, K., "Lightweight Directory Access Protocol (LDAP) Turn Operation", RFC 4531, DOI 10.17487/RFC4531, June 2006, <<https://www.rfc-editor.org/info/rfc4531>>.
- [RFC4540] Stiemerling, M., Quittek, J., and C. Cadar, "NEC's Simple Middlebox Configuration (SIMCO) Protocol Version 3.0", RFC 4540, DOI 10.17487/RFC4540, May 2006, <<https://www.rfc-editor.org/info/rfc4540>>.
- [RFC4582] Camarillo, G., Ott, J., and K. Drage, "The Binary Floor Control Protocol (BFCP)", RFC 4582, DOI 10.17487/RFC4582, November 2006, <<https://www.rfc-editor.org/info/rfc4582>>.

- [RFC4616] Zeilenga, K., Ed., "The PLAIN Simple Authentication and Security Layer (SASL) Mechanism", RFC 4616, DOI 10.17487/RFC4616, August 2006, <<https://www.rfc-editor.org/info/rfc4616>>.
- [RFC4642] Murchison, K., Vinocur, J., and C. Newman, "Using Transport Layer Security (TLS) with Network News Transfer Protocol (NNTP)", RFC 4642, DOI 10.17487/RFC4642, October 2006, <<https://www.rfc-editor.org/info/rfc4642>>.
- [RFC4680] Santesson, S., "TLS Handshake Message for Supplemental Data", RFC 4680, DOI 10.17487/RFC4680, October 2006, <<https://www.rfc-editor.org/info/rfc4680>>.
- [RFC4681] Santesson, S., Medvinsky, A., and J. Ball, "TLS User Mapping Extension", RFC 4681, DOI 10.17487/RFC4681, October 2006, <<https://www.rfc-editor.org/info/rfc4681>>.
- [RFC4712] Siddiqui, A., Romascanu, D., Golovinsky, E., Rahman, M., and Y. Kim, "Transport Mappings for Real-time Application Quality-of-Service Monitoring (RAQMOM) Protocol Data Unit (PDU)", RFC 4712, DOI 10.17487/RFC4712, October 2006, <<https://www.rfc-editor.org/info/rfc4712>>.
- [RFC4732] Handley, M., Ed., Rescorla, E., Ed., and IAB, "Internet Denial-of-Service Considerations", RFC 4732, DOI 10.17487/RFC4732, December 2006, <<https://www.rfc-editor.org/info/rfc4732>>.
- [RFC4743] Goddard, T., "Using NETCONF over the Simple Object Access Protocol (SOAP)", RFC 4743, DOI 10.17487/RFC4743, December 2006, <<https://www.rfc-editor.org/info/rfc4743>>.
- [RFC4744] Lear, E. and K. Crozier, "Using the NETCONF Protocol over the Blocks Extensible Exchange Protocol (BEEP)", RFC 4744, DOI 10.17487/RFC4744, December 2006, <<https://www.rfc-editor.org/info/rfc4744>>.
- [RFC4785] Blumenthal, U. and P. Goel, "Pre-Shared Key (PSK) Ciphersuites with NULL Encryption for Transport Layer Security (TLS)", RFC 4785, DOI 10.17487/RFC4785, January 2007, <<https://www.rfc-editor.org/info/rfc4785>>.
- [RFC4791] Daboo, C., Desruisseaux, B., and L. Dusseault, "Calendaring Extensions to WebDAV (CalDAV)", RFC 4791, DOI 10.17487/RFC4791, March 2007, <<https://www.rfc-editor.org/info/rfc4791>>.

- [RFC4823] Harding, T. and R. Scott, "FTP Transport for Secure Peer-to-Peer Business Data Interchange over the Internet", RFC 4823, DOI 10.17487/RFC4823, April 2007, <<https://www.rfc-editor.org/info/rfc4823>>.
- [RFC4851] Cam-Winget, N., McGrew, D., Salowey, J., and H. Zhou, "The Flexible Authentication via Secure Tunneling Extensible Authentication Protocol Method (EAP-FAST)", RFC 4851, DOI 10.17487/RFC4851, May 2007, <<https://www.rfc-editor.org/info/rfc4851>>.
- [RFC4964] Allen, A., Ed., Holm, J., and T. Hallin, "The P-Answer-State Header Extension to the Session Initiation Protocol for the Open Mobile Alliance Push to Talk over Cellular", RFC 4964, DOI 10.17487/RFC4964, September 2007, <<https://www.rfc-editor.org/info/rfc4964>>.
- [RFC4975] Campbell, B., Ed., Mahy, R., Ed., and C. Jennings, Ed., "The Message Session Relay Protocol (MSRP)", RFC 4975, DOI 10.17487/RFC4975, September 2007, <<https://www.rfc-editor.org/info/rfc4975>>.
- [RFC4976] Jennings, C., Mahy, R., and A. Roach, "Relay Extensions for the Message Sessions Relay Protocol (MSRP)", RFC 4976, DOI 10.17487/RFC4976, September 2007, <<https://www.rfc-editor.org/info/rfc4976>>.
- [RFC4992] Newton, A., "XML Pipelining with Chunks for the Internet Registry Information Service", RFC 4992, DOI 10.17487/RFC4992, August 2007, <<https://www.rfc-editor.org/info/rfc4992>>.
- [RFC5018] Camarillo, G., "Connection Establishment in the Binary Floor Control Protocol (BFCP)", RFC 5018, DOI 10.17487/RFC5018, September 2007, <<https://www.rfc-editor.org/info/rfc5018>>.
- [RFC5019] Deacon, A. and R. Hurst, "The Lightweight Online Certificate Status Protocol (OCSP) Profile for High-Volume Environments", RFC 5019, DOI 10.17487/RFC5019, September 2007, <<https://www.rfc-editor.org/info/rfc5019>>.
- [RFC5023] Gregorio, J., Ed. and B. de hOra, Ed., "The Atom Publishing Protocol", RFC 5023, DOI 10.17487/RFC5023, October 2007, <<https://www.rfc-editor.org/info/rfc5023>>.

- [RFC5024] Friend, I., "ODETTE File Transfer Protocol 2.0", RFC 5024, DOI 10.17487/RFC5024, November 2007, <<https://www.rfc-editor.org/info/rfc5024>>.
- [RFC5049] Bormann, C., Liu, Z., Price, R., and G. Camarillo, Ed., "Applying Signaling Compression (SigComp) to the Session Initiation Protocol (SIP)", RFC 5049, DOI 10.17487/RFC5049, December 2007, <<https://www.rfc-editor.org/info/rfc5049>>.
- [RFC5054] Taylor, D., Wu, T., Mavrogiannopoulos, N., and T. Perrin, "Using the Secure Remote Password (SRP) Protocol for TLS Authentication", RFC 5054, DOI 10.17487/RFC5054, November 2007, <<https://www.rfc-editor.org/info/rfc5054>>.
- [RFC5091] Boyen, X. and L. Martin, "Identity-Based Cryptography Standard (IBCS) #1: Supersingular Curve Implementations of the BF and BB1 Cryptosystems", RFC 5091, DOI 10.17487/RFC5091, December 2007, <<https://www.rfc-editor.org/info/rfc5091>>.
- [RFC5158] Huston, G., "6to4 Reverse DNS Delegation Specification", RFC 5158, DOI 10.17487/RFC5158, March 2008, <<https://www.rfc-editor.org/info/rfc5158>>.
- [RFC5216] Simon, D., Aboba, B., and R. Hurst, "The EAP-TLS Authentication Protocol", RFC 5216, DOI 10.17487/RFC5216, March 2008, <<https://www.rfc-editor.org/info/rfc5216>>.
- [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>>.
- [RFC5263] Lonnfors, M., Costa-Requena, J., Leppanen, E., and H. Khartabil, "Session Initiation Protocol (SIP) Extension for Partial Notification of Presence Information", RFC 5263, DOI 10.17487/RFC5263, September 2008, <<https://www.rfc-editor.org/info/rfc5263>>.
- [RFC5281] Funk, P. and S. Blake-Wilson, "Extensible Authentication Protocol Tunneled Transport Layer Security Authenticated Protocol Version 0 (EAP-TTLSv0)", RFC 5281, DOI 10.17487/RFC5281, August 2008, <<https://www.rfc-editor.org/info/rfc5281>>.

- [RFC5364] Garcia-Martin, M. and G. Camarillo, "Extensible Markup Language (XML) Format Extension for Representing Copy Control Attributes in Resource Lists", RFC 5364, DOI 10.17487/RFC5364, October 2008, <<https://www.rfc-editor.org/info/rfc5364>>.
- [RFC5422] Cam-Winget, N., McGrew, D., Salowey, J., and H. Zhou, "Dynamic Provisioning Using Flexible Authentication via Secure Tunneling Extensible Authentication Protocol (EAP-FAST)", RFC 5422, DOI 10.17487/RFC5422, March 2009, <<https://www.rfc-editor.org/info/rfc5422>>.
- [RFC5469] Eronen, P., Ed., "DES and IDEA Cipher Suites for Transport Layer Security (TLS)", RFC 5469, DOI 10.17487/RFC5469, February 2009, <<https://www.rfc-editor.org/info/rfc5469>>.
- [RFC5734] Hollenbeck, S., "Extensible Provisioning Protocol (EPP) Transport over TCP", STD 69, RFC 5734, DOI 10.17487/RFC5734, August 2009, <<https://www.rfc-editor.org/info/rfc5734>>.
- [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/info/rfc5878>>.
- [RFC5953] Hardaker, W., "Transport Layer Security (TLS) Transport Model for the Simple Network Management Protocol (SNMP)", RFC 5953, DOI 10.17487/RFC5953, August 2010, <<https://www.rfc-editor.org/info/rfc5953>>.
- [RFC6042] Keromytis, A., "Transport Layer Security (TLS) Authorization Using KeyNote", RFC 6042, DOI 10.17487/RFC6042, October 2010, <<https://www.rfc-editor.org/info/rfc6042>>.
- [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>>.
- [RFC6353] Hardaker, W., "Transport Layer Security (TLS) Transport Model for the Simple Network Management Protocol (SNMP)", STD 78, RFC 6353, DOI 10.17487/RFC6353, July 2011, <<https://www.rfc-editor.org/info/rfc6353>>.
- [RFC6367] Kanno, S. and M. Kanda, "Addition of the Camellia Cipher Suites to Transport Layer Security (TLS)", RFC 6367, DOI 10.17487/RFC6367, September 2011, <<https://www.rfc-editor.org/info/rfc6367>>.

- [RFC6739] Schulzrinne, H. and H. Tschofenig, "Synchronizing Service Boundaries and <mapping> Elements Based on the Location-to-Service Translation (LoST) Protocol", RFC 6739, DOI 10.17487/RFC6739, October 2012, <<https://www.rfc-editor.org/info/rfc6739>>.
- [RFC6749] Hardt, D., Ed., "The OAuth 2.0 Authorization Framework", RFC 6749, DOI 10.17487/RFC6749, October 2012, <<https://www.rfc-editor.org/info/rfc6749>>.
- [RFC6750] Jones, M. and D. Hardt, "The OAuth 2.0 Authorization Framework: Bearer Token Usage", RFC 6750, DOI 10.17487/RFC6750, October 2012, <<https://www.rfc-editor.org/info/rfc6750>>.
- [RFC7030] Pritikin, M., Ed., Yee, P., Ed., and D. Harkins, Ed., "Enrollment over Secure Transport", RFC 7030, DOI 10.17487/RFC7030, October 2013, <<https://www.rfc-editor.org/info/rfc7030>>.
- [RFC7465] Popov, A., "Prohibiting RC4 Cipher Suites", RFC 7465, DOI 10.17487/RFC7465, February 2015, <<https://www.rfc-editor.org/info/rfc7465>>.
- [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>>.
- [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>>.
- [RFC7562] Thakore, D., "Transport Layer Security (TLS) Authorization Using Digital Transmission Content Protection (DTCP) Certificates", RFC 7562, DOI 10.17487/RFC7562, July 2015, <<https://www.rfc-editor.org/info/rfc7562>>.
- [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>>.
- [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>>.

- [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>>.

11.2. Informative References

- [Bhargavan2016] Bhargavan, K. and G. Leuren, "Transcript Collision Attacks: Breaking Authentication in TLS, IKE, and SSH <https://www.mitls.org/downloads/transcript-collisions.pdf>", 2016.
- [NIST800-52r2] National Institute of Standards and Technology, "NIST SP800-52r2 <https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-52r2.pdf>", August 2019.
- [RFC3316] Arkko, J., Kuijpers, G., Soliman, H., Loughney, J., and J. Wiljakka, "Internet Protocol Version 6 (IPv6) for Some Second and Third Generation Cellular Hosts", RFC 3316, DOI 10.17487/RFC3316, April 2003, <<https://www.rfc-editor.org/info/rfc3316>>.
- [RFC3489] Rosenberg, J., Weinberger, J., Huitema, C., and R. Mahy, "STUN - Simple Traversal of User Datagram Protocol (UDP) Through Network Address Translators (NATs)", RFC 3489, DOI 10.17487/RFC3489, March 2003, <<https://www.rfc-editor.org/info/rfc3489>>.
- [RFC3546] Blake-Wilson, S., Nystrom, M., Hopwood, D., Mikkelsen, J., and T. Wright, "Transport Layer Security (TLS) Extensions", RFC 3546, DOI 10.17487/RFC3546, June 2003, <<https://www.rfc-editor.org/info/rfc3546>>.
- [RFC3588] Calhoun, P., Loughney, J., Guttman, E., Zorn, G., and J. Arkko, "Diameter Base Protocol", RFC 3588, DOI 10.17487/RFC3588, September 2003, <<https://www.rfc-editor.org/info/rfc3588>>.
- [RFC3734] Hollenbeck, S., "Extensible Provisioning Protocol (EPP) Transport Over TCP", RFC 3734, DOI 10.17487/RFC3734, March 2004, <<https://www.rfc-editor.org/info/rfc3734>>.

- [RFC3920] Saint-Andre, P., Ed., "Extensible Messaging and Presence Protocol (XMPP): Core", RFC 3920, DOI 10.17487/RFC3920, October 2004, <<https://www.rfc-editor.org/info/rfc3920>>.
- [RFC4132] Moriai, S., Kato, A., and M. Kanda, "Addition of Camellia Cipher Suites to Transport Layer Security (TLS)", RFC 4132, DOI 10.17487/RFC4132, July 2005, <<https://www.rfc-editor.org/info/rfc4132>>.
- [RFC4244] Barnes, M., Ed., "An Extension to the Session Initiation Protocol (SIP) for Request History Information", RFC 4244, DOI 10.17487/RFC4244, November 2005, <<https://www.rfc-editor.org/info/rfc4244>>.
- [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>>.
- [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>>.
- [RFC4507] Salowey, J., Zhou, H., Eronen, P., and H. Tschofenig, "Transport Layer Security (TLS) Session Resumption without Server-Side State", RFC 4507, DOI 10.17487/RFC4507, May 2006, <<https://www.rfc-editor.org/info/rfc4507>>.
- [RFC4572] Lennox, J., "Connection-Oriented Media Transport over the Transport Layer Security (TLS) Protocol in the Session Description Protocol (SDP)", RFC 4572, DOI 10.17487/RFC4572, July 2006, <<https://www.rfc-editor.org/info/rfc4572>>.
- [RFC4934] Hollenbeck, S., "Extensible Provisioning Protocol (EPP) Transport Over TCP", RFC 4934, DOI 10.17487/RFC4934, May 2007, <<https://www.rfc-editor.org/info/rfc4934>>.
- [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>>.

- [RFC5081] Mavrogiannopoulos, N., "Using OpenPGP Keys for Transport Layer Security (TLS) Authentication", RFC 5081, DOI 10.17487/RFC5081, November 2007, <<https://www.rfc-editor.org/info/rfc5081>>.
- [RFC5101] Claise, B., Ed., "Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of IP Traffic Flow Information", RFC 5101, DOI 10.17487/RFC5101, January 2008, <<https://www.rfc-editor.org/info/rfc5101>>.
- [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>>.
- [RFC5415] Calhoun, P., Ed., Montemurro, M., Ed., and D. Stanley, Ed., "Control And Provisioning of Wireless Access Points (CAPWAP) Protocol Specification", RFC 5415, DOI 10.17487/RFC5415, March 2009, <<https://www.rfc-editor.org/info/rfc5415>>.
- [RFC5456] Spencer, M., Capouch, B., Guy, E., Ed., Miller, F., and K. Shumard, "IAX: Inter-Asterisk eXchange Version 2", RFC 5456, DOI 10.17487/RFC5456, February 2010, <<https://www.rfc-editor.org/info/rfc5456>>.
- [RFC6012] Salowey, J., Petch, T., Gerhards, R., and H. Feng, "Datagram Transport Layer Security (DTLS) Transport Mapping for Syslog", RFC 6012, DOI 10.17487/RFC6012, October 2010, <<https://www.rfc-editor.org/info/rfc6012>>.
- [RFC6083] Tuexen, M., Seggelmann, R., and E. Rescorla, "Datagram Transport Layer Security (DTLS) for Stream Control Transmission Protocol (SCTP)", RFC 6083, DOI 10.17487/RFC6083, January 2011, <<https://www.rfc-editor.org/info/rfc6083>>.
- [RFC6084] Fu, X., Dickmann, C., and J. Crowcroft, "General Internet Signaling Transport (GIST) over Stream Control Transmission Protocol (SCTP) and Datagram Transport Layer Security (DTLS)", RFC 6084, DOI 10.17487/RFC6084, January 2011, <<https://www.rfc-editor.org/info/rfc6084>>.
- [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>>.

- [RFC6460] Salter, M. and R. Housley, "Suite B Profile for Transport Layer Security (TLS)", RFC 6460, DOI 10.17487/RFC6460, January 2012, <<https://www.rfc-editor.org/info/rfc6460>>.
- [RFC6614] Winter, S., McCauley, M., Venaas, S., and K. Wierenga, "Transport Layer Security (TLS) Encryption for RADIUS", RFC 6614, DOI 10.17487/RFC6614, May 2012, <<https://www.rfc-editor.org/info/rfc6614>>.
- [RFC7457] Sheffer, Y., Holz, R., and P. Saint-Andre, "Summarizing Known Attacks on Transport Layer Security (TLS) and Datagram TLS (DTLS)", RFC 7457, DOI 10.17487/RFC7457, February 2015, <<https://www.rfc-editor.org/info/rfc7457>>.
- [RFC8143] Elie, J., "Using Transport Layer Security (TLS) with Network News Transfer Protocol (NNTP)", RFC 8143, DOI 10.17487/RFC8143, April 2017, <<https://www.rfc-editor.org/info/rfc8143>>.
- [RFC8261] Tuexen, M., Stewart, R., Jesup, R., and S. Loreto, "Datagram Transport Layer Security (DTLS) Encapsulation of SCTP Packets", RFC 8261, DOI 10.17487/RFC8261, November 2017, <<https://www.rfc-editor.org/info/rfc8261>>.
- [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>>.

Appendix A. Change Log

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

From draft-ietf-tls-oldversions-deprecate-11 to draft-ietf-tls-oldversions-deprecate-12 (IESG review):

- o Minor edits from IESG review comments.

From draft-ietf-tls-oldversions-deprecate-10 to draft-ietf-tls-oldversions-deprecate-11:

- o RFC 5953 was mentioned in the wrong para of section 1.1 - it has been obsoleted already.

From draft-ietf-tls-oldversions-deprecate-09 to draft-ietf-tls-oldversions-deprecate-10:

- o We missed adding change logs for a few versions, but since -09 was the one that underwent IETF last call, and there was some discussion, we figured it'd be good to mention substantive changes here.
- o Added Ben's suggested text for "operational considerations" following extensive last call discussion.
- o Re-checked the references to RFC 4347 after Tom Petch noticed we missed a couple. Added RFCs 5953 and 6353 to the list here. All others were in already.
- o Fixed various typos and ack'd those who engaged a bit in the IETF LC discussion. (If we missed you and you want to be added, or if you'd rather not be mentioned, just ping the authors.)

From draft-ietf-tls-oldversions-deprecate-05 to draft-ietf-tls-oldversions-deprecate-06:

- o Fixed "yaleman" ack.
- o Added RFC6614 to UPDATES list.
- o per preliminary AD review:
 - * Remove references from abstract
 - * s/primary technical reasons/technical reasons/
 - * Add rfc7030 to 1.1
 - * verified that all the RFCs in the (massive:-) Updates meta-data are mentioned in section 1.1 (I think appropriately;-)

From draft-ietf-tls-oldversions-deprecate-04 to draft-ietf-tls-oldversions-deprecate-05:

- o Removed references to government related deprecation statements: US, Canada, and Germany. NIST documentation rationale remains as a reference describing the relevant RFCs and justification.

From draft-ietf-tls-oldversions-deprecate-02 to draft-ietf-tls-oldversions-deprecate-03:

- o Added 8261 to updates list based on IETF-104 meeting.

From draft-ietf-tls-oldversions-deprecate-01 to draft-ietf-tls-oldversions-deprecate-02:

- o Correction: 2nd list of referenced RFCs in Section 1.1 aren't informatively referring to tls1.0/1.1
- o Remove RFC7255 from updates list - datatracker has bad data (spotted by Robert Sparks)
- o Added point about RFCs 8143 and 4642
- o Added UPDATES for RFCs that refer to 4347 and aren't OBSOLETEd
- o Added note about RFC8261 to see what WG want.

From draft-ietf-tls-oldversions-deprecate-00 to draft-ietf-tls-oldversions-deprecate-01:

- o PRs with typos and similar: so far just #1
- o PR#2 noting msft browser announced deprecation (but this was OBE as per...)
- o Implemented actions as per IETF-103 meeting:
 - * Details about which RFC's, BCP's are affected were generated using a script in the git repo: <https://github.com/tlswg/oldversions-deprecate/blob/master/nonobsnorms.sh>
 - * Removed the 'measurements' part
 - * Removed SHA-1 deprecation (section 8 of -00)

From draft-moriarty-tls-oldversions-diediedie-01 to draft-ietf-tls-oldversions-deprecate-00:

- o I-Ds became RFCs 8446/8447 (old-repo PR#4, for TLSv1.3)
- o Accepted old-repo PR#5 fixing typos

From draft-moriarty-tls-oldversions-diediedie-00 to draft-moriarty-tls-oldversions-diediedie-01:

- o Added stats sent to list so far
- o PR's #2,3
- o a few more references
- o added section on email

Authors' Addresses

Kathleen Moriarty
Dell EMC
176 South Street
Hopkinton
United States

EMail: Kathleen.Moriarty.ietf@gmail.com

Stephen Farrell
Trinity College Dublin
Dublin 2
Ireland

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

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

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

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

Abstract

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

Discussion Venues

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

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

Status of This Memo

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

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

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

This Internet-Draft will expire on 10 November 2022.

Copyright Notice

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

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

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

Table of Contents

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

1. Introduction

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

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

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

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

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

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

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

2. Conventions and Terminology

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

2.1. Change Log

RFC EDITOR PLEASE DELETE THIS SECTION.

(*) indicates changes to the wire protocol.

draft-11

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

draft-10

- * Address superficial comments
- * Add example certificate

draft-09

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

draft-08

- * Include details about the impact of signature forgery attacks

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

draft-07

- * Minor text improvements

draft-06

- * Modified IANA section, fixed nits

draft-05

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

draft-04

- * Add support for client certificates.

draft-03

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

draft-02

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

3. Solution Overview

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

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

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

CertificateRequest that indicates support for this mechanism.

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

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

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

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

3.1. Rationale

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

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

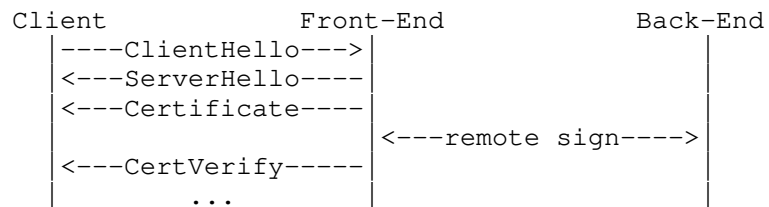
is not cryptographically protected, it is possible that a proxy certificate could be bound to another certificate with the same public key, with different X.509 parameters. Delegated credentials, which rely on a cryptographic binding between the entire certificate and the delegated credential, cannot.

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

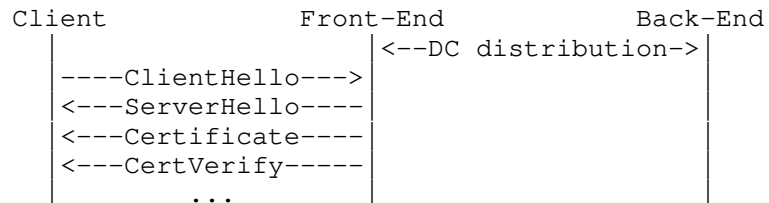
3.2. Related Work

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

Remote key signing:



Delegated Credential:



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

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

credentials.

4. Delegated Credentials

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

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

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

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

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

The DelegatedCredential has the following structure:

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

cred: The Credential structure as previously defined.

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

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

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

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

authentication.

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

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

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

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

4.1. Client and Server Behavior

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

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

4.1.1. Server Authentication

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

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

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

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

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

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

4.1.2. Client Authentication

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

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

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

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

4.1.3. Validating a Delegated Credential

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

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

in Section 4.2.

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

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

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

4.2. Certificate Requirements

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

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

```
DelegationUsage ::= NULL
```

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

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

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

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

5. Operational Considerations

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

5.1. Client Clock Skew

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

6. IANA Considerations

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

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

7. Security Considerations

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

7.1. Security of Delegated Credential's Private Key

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

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

7.2. Re-use of Delegated Credentials in Multiple Contexts

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

7.3. Revocation of Delegated Credentials

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

7.4. Interactions with Session Resumption

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

7.5. Privacy Considerations

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

7.6. The Impact of Signature Forgery Attacks

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

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

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

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

8. Acknowledgements

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

9. References

9.1. Normative References

- [I-D.ietf-tls-dtls13] Rescorla, E., Tschofenig, H., and N. Modadugu, "The Datagram Transport Layer Security (DTLS) Protocol Version 1.3", Work in Progress, Internet-Draft, draft-ietf-tls-dtls13-43, 30 April 2021, <<https://datatracker.ietf.org/doc/html/draft-ietf-tls-dtls13-43>>.
- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <<https://www.rfc-editor.org/rfc/rfc2119>>.
- [RFC5280] Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R., and W. Polk, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", RFC 5280, DOI 10.17487/RFC5280, May 2008, <<https://www.rfc-editor.org/rfc/rfc5280>>.
- [RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <<https://www.rfc-editor.org/rfc/rfc8174>>.
- [RFC8446] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <<https://www.rfc-editor.org/rfc/rfc8446>>.
- [X.680] ITU-T, "Information technology - Abstract Syntax Notation One (ASN.1): Specification of basic notation", ISO/IEC 8824-1:2015, November 2015.
- [X.690] ITU-T, "Information technology - ASN.1 encoding Rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)", ISO/IEC 8825-1:2015, November 2015.

9.2. Informative References

- [BLEI] Bleichenbacher, D., "Chosen Ciphertext Attacks against Protocols Based on RSA Encryption Standard PKCS #1", Advances in Cryptology -- CRYPTO'98, LNCS vol. 1462, pages: 1-12 , 1998.
- [KEYLESS] Sullivan, N. and D. Stebila, "An Analysis of TLS Handshake Proxying", IEEE Trustcom/BigDataSE/ISPA 2015 , 2015.
- [RFC3820] Tuecke, S., Welch, V., Engert, D., Pearlman, L., and M. Thompson, "Internet X.509 Public Key Infrastructure (PKI) Proxy Certificate Profile", RFC 3820, DOI 10.17487/RFC3820, June 2004, <<https://www.rfc-editor.org/rfc/rfc3820>>.
- [RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, DOI 10.17487/RFC5246, August 2008, <<https://www.rfc-editor.org/rfc/rfc5246>>.
- [RFC5912] Hoffman, P. and J. Schaad, "New ASN.1 Modules for the Public Key Infrastructure Using X.509 (PKIX)", RFC 5912, DOI 10.17487/RFC5912, June 2010, <<https://www.rfc-editor.org/rfc/rfc5912>>.

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

Appendix A. ASN.1 Module

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

```

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

DEFINITIONS IMPLICIT TAGS ::=
BEGIN

-- EXPORT ALL

IMPORTS

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

-- OID

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

-- EXTENSION

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

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

DelegationUsage ::= NULL

END

```

Appendix B. Example Certificate

The following certificate has the Delegated Credentials OID.

```
-----BEGIN CERTIFICATE-----
MIIFRjCCBMugAwIBAgIQDGeVB+lY0o/OecHFSJ6YnTAKBggqhkjOPQQDAzBMMQsw
CQYDVQQGEwJVUzEVMBMGA1UEChMMRGlnaUNlcnQgSW5jMSYwJAYDVQQDEx1EaWdp
Q2VydCBFQ0MgU2VjdXJlIFN1cnZlciBDQTAEFw0xOTAzMjYwMDAwMDBaFw0yMTAz
MzAxMjAwMDBaMGoxCzAJBgNVBAYTA1VTMRMwEQYDVQQIEwpDYWxpZm9ybmlhMRYw
FAYDVQQHEw1TYW4gRnJhbmNpc2NvMRkwFwYDVQQKEwBDbG91ZGZsYXJlLCBjb2Mu
MRMwEQYDVQQDEwprYzJrZG0uY29tMFkwEwYHKoZIzj0CAQYIKoZIzj0DAQcDQgAE
d4azI83Bw0fcPgfoeiZpZznwGuxjBjv++wzE0zAj8vNiUkKxOWSQiGNLn+xlWUpL
lw9djRN1rLmVmn2gb9GgdK0CA28wggNrMB8GA1UdIwQYMBaAFK0d5h/52j1PwG7o
kcuVpd0x4gqfMB0GA1UdDgQWBBSfcb7fS3fUFAYB91fRcwoDPtgtJjAjBgNVHREE
HDAaggrYzJrZG0uY29tggwqLmtjMmtkbS5jb20wDgYDVROPAQH/BAQDAgeAMB0G
A1UdJQQWMBQGCCsGAQUFBwMBBggrBgEFBQcDAjBpBgNVHR8EYjBqMC6gLKAqhiho
dHRwOi8vY3JsMy5kaWdpY2VydC5jb20vc3NjYS1lY2MtZzEuY3JsMC6gLKAqhiho
dHRwOi8vY3JsNC5kaWdpY2VydC5jb20vc3NjYS1lY2MtZzEuY3JsMEwGA1UdIARF
MEMwNwYJYIZIAyB9bAEBMCowKAYIKwYBBQUHAQEWHGh0dHBzOi8vd3d3LmRpZ21j
ZXJ0LmNvbS9DUFMwCAYGZ4EMAQICMHSGCCsGAQUFBwEBBG8wbTAKBggrBgEFBQcw
AYYYaHR0cDovL29jc3AuZGlnaWN1cnQuY29tMEUGCCsGAQUFBzAChjlodHRwOi8v
Y2FjZXJ0cy5kaWdpY2VydC5jb20vRGlnaUNlcnRFQ0NTZW1cmVTZXJ2ZXJ0QS5j
cnQwDAYDVROTAQH/BAIwADAPBgkrBgEEAYLaSywEAgUAMIIBfgYKKwYBBAHWeQIE
AgSCAW4EggFqAWgAdgC72d+8H4pxtZOUi5eqkntHOFVCqtS6BqQlQmQ2jh7RhQAA
AWm5hYJ5AAAEAwBHMEUCICiGfq+hSThRL2m8H0awoDR8OpnEHNkF0nI6nL5yYL/j
AiEAXwebGs/T6Es0YarPzoQJrVZqk+sHH/t+jrSrKd5TDjcAdgCHdb/nWXz4jEOZ
X73zbv9WjUdWnV9KtWDBtOr/XqCDDwAAAWm5hYNgAAAEAwBHMEUCIQD9OWA8KGL6
bxDKfGileHJWB0iWieRs88VgJyFag/aFDgIgQ/OsdSF9XOy1foqge0D2DM2FExuw
0JR0AGZWXoNtJzMAAgBE1GUusO7Or8RAB9io/iJA2uaCvtjLmbU/0zOWtbaBqAAA
AWm5hYHgAAAEAwBHMEUCIQc4vua1n3BqthEqpA/VBTcsNwMtAwpcuac2IhJ9wx6X
/AIgb+o00k28JQo9TMpP4vzJ3BD3HXWSNc2Zizbq7mkUQYMwCgYIKoZIzj0EAwMD
aQAwwZgIXAJsX7d0SuA8ddf/m7IWfNfs3MQfJyGkEezMJX1t6sRso5z50SS12LpXe
muGalFE2ZgIXAL+CDUF5pz7mhrAEIjQ1MqlpF9tH40dJGvYZZQ3W23cMzSkDfvlt
y5S4RfWHIIPjBw==
-----END CERTIFICATE-----
```

Authors' Addresses

Richard Barnes
Cisco
Email: rlb@ipv.sx

Subodh Iyengar
Facebook
Email: subodh@fb.com

Nick Sullivan
Cloudflare
Email: nick@cloudflare.com

Eric Rescorla
Mozilla
Email: ekr@rtfm.com

Network Working Group
Internet-Draft
Intended status: Experimental
Expires: June 25, 2020

R. Housley
Vigil Security
December 23, 2019

TLS 1.3 Extension for Certificate-based Authentication with an External
Pre-Shared Key
draft-ietf-tls-tls13-cert-with-extern-psk-07

Abstract

This document specifies a TLS 1.3 extension that allows a server to authenticate with a combination of a certificate and an external pre-shared key (PSK).

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 June 25, 2020.

Copyright Notice

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

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

1. Introduction

The TLS 1.3 [RFC8446] handshake protocol provides two mutually exclusive forms of server authentication. First, the server can be authenticated by providing a signature certificate and creating a valid digital signature to demonstrate that it possesses the corresponding private key. Second, the server can be authenticated by demonstrating that it possesses a pre-shared key (PSK) that was established by a previous handshake. A PSK that is established in this fashion is called a resumption PSK. A PSK that is established by any other means is called an external PSK. This document specifies a TLS 1.3 extension permitting certificate-based server authentication to be combined with an external PSK as an input to the TLS 1.3 key schedule.

Several implementors wanted to gain more experience with this specification before producing a standards-track RFC. As a result, this specification is being published as an Experimental RFC to enable interoperable implementations and gain deployment and operational experience.

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. Motivation and Design Rationale

The development of a large-scale quantum computer would pose a serious challenge for the cryptographic algorithms that are widely deployed today, including the digital signature algorithms that are used to authenticate the server in the TLS 1.3 handshake protocol. It is an open question whether or not it is feasible to build a large-scale quantum computer, and if so, when that might happen. However, if such a quantum computer is invented, many of the cryptographic algorithms and the security protocols that use them would become vulnerable.

The TLS 1.3 handshake protocol employs key agreement algorithms and digital signature algorithms that could be broken by the development of a large-scale quantum computer [I-D.hoffman-c2pq]. The key agreement algorithms include Diffie-Hellman (DH) [DH1977] and Elliptic Curve Diffie-Hellman (ECDH) [IEEE1363]; the digital signature algorithms include RSA [RFC8017] and Elliptic Curve Digital Signature Algorithm (ECDSA) [FIPS186]. As a result, an adversary

that stores a TLS 1.3 handshake protocol exchange today could decrypt the associated encrypted communications in the future when a large-scale quantum computer becomes available.

In the near-term, this document describes TLS 1.3 extension to protect today's communications from the future invention of a large-scale quantum computer by providing a strong external PSK as an input to the TLS 1.3 key schedule while preserving the authentication provided by the existing certificate and digital signature mechanisms.

4. Extension Overview

This section provides a brief overview of the "tls_cert_with_extern_psk" extension.

The client includes the "tls_cert_with_extern_psk" extension in the ClientHello message. The "tls_cert_with_extern_psk" extension MUST be accompanied by the "key_share", "psk_key_exchange_modes", and "pre_shared_key" extensions. The client MAY also find it useful to include the "supported_groups" extension. Since the "tls_cert_with_extern_psk" extension is intended to be used only with initial handshakes, it MUST NOT be sent alongside the "early_data" extension. These extensions are all described in Section 4.2 of [RFC8446], which also requires the "pre_shared_key" extension to be the last extension in the ClientHello message.

If the client includes both the "tls_cert_with_extern_psk" extension and the "early_data" extension, then the server MUST terminate the connection with an "illegal_parameter" alert.

If the server is willing to use one of the external PSKs listed in the "pre_shared_key" extension and perform certificate-based authentication, then the server includes the "tls_cert_with_extern_psk" extension in the ServerHello message. The "tls_cert_with_extern_psk" extension MUST be accompanied by the "key_share" and "pre_shared_key" extensions. If none of the external PSKs in the list provided by the client is acceptable to the server, then the "tls_cert_with_extern_psk" extension is omitted from the ServerHello message.

When the "tls_cert_with_extern_psk" extension is successfully negotiated, the TLS 1.3 key schedule processing includes both the selected external PSK and the (EC)DHE shared secret value. As a result, the Early Secret, Handshake Secret, and Master Secret values all depend upon the value of the selected external PSK. Of course, the Early Secret does not depend upon the (EC)DHE shared secret.

The authentication of the server and optional authentication of the client depend upon the ability to generate a signature that can be validated with the public key in their certificates. The authentication processing is not changed in any way by the selected external PSK.

Each external PSK is associated with a single hash algorithm, which is required by Section 4.2.11 of [RFC8446]. The hash algorithm MUST be set when the PSK is established, with a default of SHA-256.

5. Certificate with External PSK Extension

This section specifies the "tls_cert_with_extern_psk" extension, which MAY appear in the ClientHello message and ServerHello message. It MUST NOT appear in any other messages. The "tls_cert_with_extern_psk" extension MUST NOT appear in the ServerHello message unless the "tls_cert_with_extern_psk" extension appeared in the preceding ClientHello message. If an implementation recognizes the "tls_cert_with_extern_psk" extension and receives it in any other message, then the implementation MUST abort the handshake with an "illegal_parameter" alert.

The general extension mechanisms enable clients and servers to negotiate the use of specific extensions. Clients request extended functionality from servers with the extensions field in the ClientHello message. If the server responds with a HelloRetryRequest message, then the client sends another ClientHello message as described in Section 4.1.2 of [RFC8446], including the same "tls_cert_with_extern_psk" extension as the original ClientHello message, or aborts the handshake.

Many server extensions are carried in the EncryptedExtensions message; however, the "tls_cert_with_extern_psk" extension is carried in the ServerHello message. Successful negotiation of the "tls_cert_with_extern_psk" extension affects the key used for encryption, so it cannot be carried in the EncryptedExtensions message. Therefore, the "tls_cert_with_extern_psk" extension is only present in the ServerHello message if the server recognizes the "tls_cert_with_extern_psk" extension and the server possesses one of the external PSKs offered by the client in the "pre_shared_key" extension in the ClientHello message.

The Extension structure is defined in [RFC8446]; it is repeated here for convenience.

```
struct {  
    ExtensionType extension_type;  
    opaque extension_data<0..2^16-1>;  
} Extension;
```

The "extension_type" identifies the particular extension type, and the "extension_data" contains information specific to the particular extension type.

This document specifies the "tls_cert_with_extern_psk" extension, adding one new type to ExtensionType:

```
enum {  
    tls_cert_with_extern_psk(TBD), (65535)  
} ExtensionType;
```

The "tls_cert_with_extern_psk" extension is relevant when the client and server possess an external PSK in common that can be used as an input to the TLS 1.3 key schedule. The "tls_cert_with_extern_psk" extension is essentially a flag to use the external PSK in the key schedule, and it has the following syntax:

```
struct {  
    select (Handshake.msg_type) {  
        case client_hello: Empty;  
        case server_hello: Empty;  
    };  
} CertWithExternPSK;
```

5.1. Companion Extensions

Section 4 lists the extensions that are required to accompany the "tls_cert_with_extern_psk" extension. Most of those extensions are not impacted in any way by this specification. However, this section discusses the extensions that require additional consideration.

The "psk_key_exchange_modes" extension is defined in Section 4.2.9 of [RFC8446]. The "psk_key_exchange_modes" extension restricts the use of both the PSKs offered in this ClientHello and those that the server might supply via a subsequent NewSessionTicket. As a result, when the "psk_key_exchange_modes" extension is included in the ClientHello message, clients MUST include psk_dhe_ke mode. In

addition, clients MAY also include psk_ke mode to support a subsequent NewSessionTicket. When the "psk_key_exchange_modes" extension is included in the ServerHello message, servers MUST select the psk_dhe_ke mode for the initial handshake. Servers MUST select a key exchange mode that is listed by the client for subsequent handshakes that include the resumption PSK from the initial handshake.

The "pre_shared_key" extension is defined in Section 4.2.11 of [RFC8446]. The syntax is repeated below for convenience. All of the listed PSKs MUST be external PSKs. If a resumption PSK is listed along with the "tls_cert_with_extern_psk" extension, the server MUST abort the handshake with an "illegal_parameter" alert.

```
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;
```

The OfferedPsks contains the list of PSK identities and associated binders for the external PSKs that the client is willing to use with the server.

The identities are a list of external PSK identities that the client is willing to negotiate with the server. Each external PSK has an associated identity that is known to the client and the server; the associated identities may be known to other parties as well. In addition, the binder validation (see below) confirms that the client and server have the same key associated with the identity.

The `obfuscated_ticket_age` is not used for external PSKs. As stated in Section 4.2.11 of [RFC8446], clients SHOULD set this value to 0, and servers MUST ignore the value.

The binders are a series of HMAC [RFC2104] values, one for each external PSK offered by the client, in the same order as the identities list. The HMAC value is computed using the `binder_key`, which is derived from the external PSK, and a partial transcript of the current handshake. Generation of the `binder_key` from the external PSK is described in Section 7.1 of [RFC8446]. The partial transcript of the current handshake includes a partial ClientHello up to and including the `PreSharedKeyExtension.identities` field as described in Section 4.2.11.2 of [RFC8446].

The `selected_identity` contains the index of the external PSK identity that the server selected from the list offered by the client. As described in Section 4.2.11.2 of [RFC8446], the server MUST validate the binder value that corresponds to the selected external PSK, and if the binder does not validate, the server MUST abort the handshake with an "illegal_parameter" alert.

5.2. Authentication

When the "tls_cert_with_extern_psk" extension is successfully negotiated, authentication of the server depends upon the ability to generate a signature that can be validated with the public key in the server's certificate. This is accomplished by the server sending the Certificate and CertificateVerify messages as described in Sections 4.4.2 and 4.4.3 of [RFC8446].

TLS 1.3 does not permit the server to send a CertificateRequest message when a PSK is being used. This restriction is removed when the "tls_cert_with_extern_psk" extension is negotiated, allowing certificate-based authentication for both the client and the server. If certificate-based client authentication is desired, this is accomplished by the client sending the Certificate and CertificateVerify messages as described in Sections 4.4.2 and 4.4.3 of [RFC8446].

5.3. Keying Material

Section 7.1 of [RFC8446] specifies the TLS 1.3 Key Schedule. The successful negotiation of the "tls_cert_with_extern_psk" extension requires the key schedule processing to include both the external PSK and the (EC)DHE shared secret value.

If the client and the server have different values associated with the selected external PSK identifier, then the client and the server

will compute different values for every entry in the key schedule, which will lead to the client aborting the handshake with a "decrypt_error" alert.

6. IANA Considerations

IANA is requested to update the TLS ExtensionType Registry [IANA] to include "tls_cert_with_extern_psk" with a value of TBD and the list of messages "CH, SH" in which the "tls_cert_with_extern_psk" extension may appear.

7. Security Considerations

The Security Considerations in [RFC8446] remain relevant.

TLS 1.3 [RFC8446] does not permit the server to send a CertificateRequest message when a PSK is being used. This restriction is removed when the "tls_cert_with_extern_psk" extension is offered by the client and accepted by the server. However, TLS 1.3 does not permit an external PSK to be used in the same fashion as a resumption PSK, and this extension does not alter those restrictions. Thus, a certificate MUST NOT be used with a resumption PSK.

Implementations must protect the external pre-shared key (PSK). Compromise of the external PSK will make the encrypted session content vulnerable to the future development of a large-scale quantum computer. However, the generation, distribution, and management of the external PSKs is out of scope for this specification.

Implementers should not transmit the same content on a connection that is protected with an external PSK and a connection that is not. Doing so may allow an eavesdropper to correlate the connections, making the content vulnerable to the future invention of a large-scale quantum computer.

Implementations must generate external PSKs with a secure key management technique, such as pseudo-random generation of the key or derivation of the key from one or more other secure keys. The use of inadequate pseudo-random number generators (PRNGs) to generate external PSKs can result in little or no security. An attacker may find it much easier to reproduce the PRNG environment that produced the external PSKs and search the resulting small set of possibilities, rather than brute-force searching the whole key space. The generation of quality random numbers is difficult. [RFC4086] offers important guidance in this area.

If the external PSK is known to any party other than the client and the server, then the external PSK MUST NOT be the sole basis for authentication. The reasoning is explained in Section 4.2 of [K2016]. When this extension is used, authentication is based on certificates, not the external PSK.

In this extension, the external PSK preserves confidentiality if the (EC)DH key agreement is ever broken by cryptanalysis or the future invention of a large-scale quantum computer. As long as the attacker does not know the PSK and the key derivation algorithm remains unbroken, the attacker cannot derive the session secrets even if they are able to compute the (EC)DH shared secret. Should the attacker be able to compute the (EC)DH shared secret, the forward secrecy advantages traditionally associated with ephemeral (EC)DH keys will no longer be relevant. Although the ephemeral private keys used during a given TLS session are destroyed at the end of a session, preventing the attacker from later accessing them, these private keys would nevertheless be recoverable due to the break in the algorithm. However, a more general notion of "secrecy after key material is destroyed" would still be achievable using external PSKs, if they are managed in a way that ensures their destruction when they are no longer needed, and with the assumption that the algorithms that use the external PSKs remain quantum-safe.

TLS 1.3 key derivation makes use of the HKDF algorithm, which depends upon the HMAC [RFC2104] construction and a hash function. This extension provides the desired protection for the session secrets as long as HMAC with the selected hash function is a pseudorandom function (PRF) [GGM1986].

This specification does not require that the external PSK is known only by the client and server. The external PSK may be known to a group. Since authentication depends on the public key in a certificate, knowledge of the external PSK by other parties does not enable impersonation. Since confidentiality depends on the shared secret from (EC)DH, knowledge of the external PSK by other parties does not enable eavesdropping. However, group members can record the traffic of other members, and then decrypt it if they ever gain access to a large-scale quantum computer. Also, when many parties know the external PSK, there are many opportunities for theft of the external PSK by an attacker. Once an attacker has the external PSK, they can decrypt stored traffic if they ever gain access to a large-scale quantum computer in the same manner as a legitimate group member.

TLS 1.3 [RFC8446] takes a conservative approach to PSKs; they are bound to a specific hash function and KDF. By contrast, TLS 1.2 [RFC5246] allows PSKs to be used with any hash function and the TLS

1.2 PRF. Thus, the safest approach is to use a PSK exclusively with TLS 1.2 or exclusively with TLS 1.3. Given one PSK, one can derive a PSK for exclusive use with TLS 1.2 and derive another PSK for exclusive use with TLS 1.3 using the mechanism specified in [I-D.ietf-tls-external-psk-importer].

TLS 1.3 [RFC8446] has received careful security analysis, and the following informal reasoning shows that the addition of this extension does not introduce any security defects. This extension requires the use of certificates for authentication, but the processing of certificates is unchanged by this extension. This extension places an external PSK in the key schedule as part of the computation of the Early Secret. In the initial handshake without this extension, the Early Secret is computed as:

$$\text{Early Secret} = \text{HKDF-Extract}(0, 0)$$

With this extension, the Early Secret is computed as:

$$\text{Early Secret} = \text{HKDF-Extract}(\text{External PSK}, 0)$$

Any entropy contributed by the external PSK can only make the Early Secret better; the External PSK cannot make it worse. For these two reasons, TLS 1.3 continues to meet its security goals when this extension is used.

8. Privacy Considerations

Appendix E.6 of [RFC8446] discusses identity exposure attacks on PSKs. The guidance in this section remains relevant.

This extension makes use of external PSKs to improve resilience against attackers that gain access to a large-scale quantum computer in the future. This extension is always accompanied by the "pre_shared_key" extension to provide the PSK identities in plaintext in the ClientHello message. Passive observation of these PSK identities will aid an attacker to track users of this extension.

9. Acknowledgments

Many thanks to Liliya Akhmetzyanova, Roman Danyliw, Christian Huitema, Ben Kaduk, Geoffrey Keating, Hugo Krawczyk, Mirja Kuehlewind, Nikos Mavrogiannopoulos, Nick Sullivan, Martin Thomson, and Peter Yee for their review and comments; their efforts have improved this document.

10. References

10.1. Normative References

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

10.2. Informative References

- [DH1977] Diffie, W. and M. Hellman, "New Directions in Cryptography", IEEE Transactions on Information Theory V.IT-22 n.6, June 1977.
- [FIPS186] National Institute of Standards and Technology, "Digital Signature Standard (DSS)", Federal Information Processing Standards Publication (FIPS PUB) 186-4, July 2013.
- [GGM1986] Goldreich, O., Goldwasser, S., and S. Micali, "How to construct random functions", J. ACM 1986 (33), pp. 792-807, 1986.
- [I-D.hoffman-c2pq] Hoffman, P., "The Transition from Classical to Post-Quantum Cryptography", draft-hoffman-c2pq-06 (work in progress), November 2019.
- [I-D.ietf-tls-external-psk-importer] Benjamin, D. and C. Wood, "Importing External PSKs for TLS", draft-ietf-tls-external-psk-importer-02 (work in progress), November 2019.
- [IANA] "IANA Registry for TLS ExtensionType Values", n.d., <<https://www.iana.org/assignments/tls-extensiontype-values/tls-extensiontype-values.xhtml>>.

- [IEEE1363] Institute of Electrical and Electronics Engineers, "IEEE Standard Specifications for Public-Key Cryptography", IEEE Std 1363-2000, 2000.
- [K2016] Krawczyk, H., "A Unilateral-to-Mutual Authentication Compiler for Key Exchange (with Applications to Client Authentication in TLS 1.3)", IACR ePrint 2016/711, August 2016.
- [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>>.
- [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>>.
- [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>>.
- [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>>.

Author's Address

Russ Housley
Vigil Security, LLC
516 Dranesville Road
Herndon, VA 20170
USA

Email: housley@vigilsec.com

Network Working Group
Internet-Draft
Intended status: Informational
Expires: September 12, 2019

E. Kinnear
T. Pauly
C. Wood
Apple Inc.
March 11, 2019

TLS Client Network Address Extension
draft-kinnear-tls-client-net-address-00

Abstract

This document describes a TLS 1.3 extension that can be by clients to request their public network address from a server. This information can be used for a variety of purposes, including: NAT detection, ASN identification, and privacy-driven transport protocol features.

Status of This Memo

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

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

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

This Internet-Draft will expire on September 12, 2019.

Copyright Notice

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

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

Table of Contents

1. Introduction	2
1.1. Requirements Language	2
2. Client Network Address Use Cases	2
2.1. Connection Lifetime Optimizations	3
2.2. Privacy Stance Enhancements	3
2.3. Metric Collections	3
3. Network Address Extension	3
4. IANA Considerations	4
5. Security Considerations	4
6. Normative References	5
Authors' Addresses	5

1. Introduction

This document describes a TLS 1.3 [RFC8446] extension that can be by clients to request their public network address from a server. This has several uses, including: NAT detection, ASN identification, and privacy-driven transport protocol features. Servers that support this extension can send the perceived client address to clients. The latter may then confirm whether or not this representation matches their known public address.

Unlike the related NAT detection extension for IKE [RFC3947], clients do not send their perceived IP address to servers, even in an obfuscated form. Doing so would introduce an unwanted privacy regression for clients.

1.1. Requirements Language

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

2. Client Network Address Use Cases

Knowledge of a public client network address can serve several purposes. This extension allows clients to detect the presence of a NAT or other address-transforming proxy involved in a TLS connection. The following sections describe several uses for this information.

2.1. Connection Lifetime Optimizations

Middleboxes such as NATs typically have short lifetimes for connection state. Detecting such middleboxes may help influence client connection management logic, such as the use of keep-alive messages.

Since NATs often apply to all traffic from an endhost, detection via a TLS connection may assist other non-TLS and non-TCP connections that can be more sensitive to NAT timeouts.

2.2. Privacy Stance Enhancements

Address-transforming proxies such as NATs may improve communication privacy by masking the public IP address of clients in a session. Modulo other cleartext signals such as session identifiers, the anonymity set of a connection passing through a NAT is proportional to the number of clients serviced by the NAT. Absent NAT detection, clients cannot determine if their connections are linkable via IP-layer information, such as stable source addresses. As a result, clients cannot determine if privacy-driven policies such as never resuming TLS connections improve privacy.

If clients can detect NATs, they can make informed decisions about connection reuse. As a motivating example, consider DNS-over-TLS [RFC7858][RFC8310]. Privacy-sensitive clients may wish to use fresh connections for individual queries so as to not allow recursive resolvers the ability of building client query histories. However, in the absence of a NAT, reusing a connection does not pose a significant privacy regression since such clients are generally identifiable by their IP address.

Client network awareness may also influence privacy-driven connection migration policies, such as those prescribed by QUIC [I-D.ietf-quic-transport]. For example, if clients know they are not behind a NAT, then connection ID rotation serves little value in preventing linkability.

2.3. Metric Collections

Clients may passively use their public address discovered via TLS to identify their corresponding ASN without the use of explicit probes.

3. Network Address Extension

Servers may send the perceived client IP address to its peer using the following "network_address" extension:

```
enum {  
    network_address(TBD), (65535)  
} ExtensionType;
```

When sent by a client, this extension MUST be empty. A server which receives a non-empty `network_address` extension MUST terminate the connection with an "Illegal Parameter" alert.

Supporting servers which receive this extension may respond with a "network_address" extension, shown below, inside the `EncryptedExtensions`.

```
struct {  
    opaque address<32..255>;  
} NetworkAddress;
```

`address` The client's perceived address.

In this case, `NetworkAddress.address` carries the raw network-order byte-wise representation of the client IP address. (Since the extension is encrypted, there is no need to obfuscate the address for transit.) Clients which receive a non-empty `NetworkAddress` extension may use it to record their public IP address. Clients MUST treat empty `NetworkAddress.address` extensions as an error and send an `Illegal Parameter` alert in response.

4. IANA Considerations

IANA is requested to Create an entry, `network_address(TBD)`, in the existing registry for `ExtensionType` (defined in [RFC8446]), with "TLS 1.3" column values being set to "CH, EE", and "Recommended" column being set to "Yes".

5. Security Considerations

Since `NetworkAddress` extension contents are encrypted, this extension introduces no (known) additional security or privacy issues.

An earlier design let clients send their address to servers in an obfuscated form, e.g., by hashing the client's perceived IP address with `ClientHello.random`, so that servers could measure whether or not clients were also behind NATs. However, such obfuscation mechanisms are subject to dictionary attacks and therefore could be used by malicious on-path attackers to learn a client's true public address. Absent this information, there are no explicit signals from a single (non-resumed) TLS connection that such attackers can use to learn the client's public address.

In general, absent a mechanism to encrypt the client extensions, sending the client's perceived address in any form therefore constitutes a privacy regression.

6. Normative References

- [I-D.ietf-quic-transport]
Iyengar, J. and M. Thomson, "QUIC: A UDP-Based Multiplexed and Secure Transport", draft-ietf-quic-transport-18 (work in progress), January 2019.
- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <<https://www.rfc-editor.org/info/rfc2119>>.
- [RFC3947] Kivinen, T., Swander, B., Huttunen, A., and V. Volpe, "Negotiation of NAT-Traversal in the IKE", RFC 3947, DOI 10.17487/RFC3947, January 2005, <<https://www.rfc-editor.org/info/rfc3947>>.
- [RFC7858] Hu, Z., Zhu, L., Heidemann, J., Mankin, A., Wessels, D., and P. Hoffman, "Specification for DNS over Transport Layer Security (TLS)", RFC 7858, DOI 10.17487/RFC7858, May 2016, <<https://www.rfc-editor.org/info/rfc7858>>.
- [RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <<https://www.rfc-editor.org/info/rfc8174>>.
- [RFC8310] Dickinson, S., Gillmor, D., and T. Reddy, "Usage Profiles for DNS over TLS and DNS over DTLS", RFC 8310, DOI 10.17487/RFC8310, March 2018, <<https://www.rfc-editor.org/info/rfc8310>>.
- [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>>.

Authors' Addresses

Eric Kinnear
Apple Inc.
One Apple Park Way
Cupertino, California 95014
United States of America

Email: ekinnear@apple.com

Tommy Pauly
Apple Inc.
One Apple Park Way
Cupertino, California 95014
United States of America

Email: tpauly@apple.com

Christopher A. Wood
Apple Inc.
One Apple Park Way
Cupertino, California 95014
United States of America

Email: cawood@apple.com

Network Working Group
Internet-Draft
Intended status: Informational
Expires: August 15, 2020

D. Stebila
University of Waterloo
S. Fluhrer
Cisco Systems
S. Gueron
U. Haifa, Amazon Web Services
February 12, 2020

Hybrid key exchange in TLS 1.3
draft-stebila-tls-hybrid-design-03

Abstract

Hybrid key exchange refers to using multiple key exchange algorithms simultaneously and combining the result with the goal of providing security even if all but one of the component algorithms is broken. It is motivated by transition to post-quantum cryptography. This document provides a construction for hybrid key exchange in the Transport Layer Security (TLS) protocol version 1.3.

Discussion of this work is encouraged to happen on the TLS IETF mailing list tls@ietf.org or on the GitHub repository which contains the draft: <https://github.com/dstebila/draft-stebila-tls-hybrid-design>.

Status of This Memo

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

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

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

This Internet-Draft will expire on August 15, 2020.

Copyright Notice

Copyright (c) 2020 IETF Trust and the persons identified as the document authors. All rights reserved.

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

Table of Contents

1. Introduction	3
1.1. Revision history	3
1.2. Terminology	4
1.3. Motivation for use of hybrid key exchange	5
1.4. Scope	5
1.5. Goals	6
2. Key encapsulation mechanisms	7
3. Construction for hybrid key exchange	8
3.1. Negotiation	8
3.2. Transmitting public keys and ciphertexts	9
3.3. Shared secret calculation	10
4. Open questions	12
5. IANA Considerations	13
6. Security Considerations	13
7. Acknowledgements	14
8. References	14
8.1. Normative References	14
8.2. Informative References	14
Appendix A. Related work	18
Appendix B. Design Considerations	19
B.1. (Neg) How to negotiate hybridization and component algorithms?	21
B.1.1. Key exchange negotiation in TLS 1.3	21
B.1.2. (Neg-Ind) Negotiating component algorithms individually	21
B.1.3. (Neg-Comb) Negotiating component algorithms as a combination	22
B.1.4. Benefits and drawbacks	23
B.2. (Num) How many component algorithms to combine?	24
B.2.1. (Num-2) Two	24
B.2.2. (Num-2+) Two or more	24
B.2.3. Benefits and Drawbacks	24
B.3. (Shares) How to convey key shares?	24
B.3.1. (Shares-Concat) Concatenate key shares	25
B.3.2. (Shares-Multiple) Send multiple key shares	25
B.3.3. (Shares-Ext-Additional) Extension carrying additional	

key shares	25
B.3.4. Benefits and Drawbacks	25
B.4. (Comb) How to use keys?	26
B.4.1. (Comb-Concat) Concatenate keys	26
B.4.2. (Comb-KDF-1) KDF keys	27
B.4.3. (Comb-KDF-2) KDF keys	28
B.4.4. (Comb-XOR) XOR keys	29
B.4.5. (Comb-Chain) Chain of KDF applications for each key .	30
B.4.6. (Comb-AltInput) Second shared secret in an alternate KDF input	31
B.4.7. Benefits and Drawbacks	31
Authors' Addresses	32

1. Introduction

This document gives a construction for hybrid key exchange in TLS 1.3. The overall design approach is a simple, "concatenation"-based approach: each hybrid key exchange combination should be viewed as a single new key exchange method, negotiated and transmitted using the existing TLS 1.3 mechanisms.

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

1.1. Revision history

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

Earlier versions of this document categorized various design decisions one could make when implementing hybrid key exchange in TLS 1.3. These have been moved to the appendix of the current draft, and will be eventually be removed.

o draft-03:

- * Add requirement for KEMs to provide protection against key reuse.
- * Clarify FIPS-compliance of shared secret concatenation method.

o draft-02:

- * Design considerations from draft-00 and draft-01 are moved to the appendix.
- * A single construction is given in the main body.

- o draft-01:
 - * Add (Comb-KDF-1) (Appendix B.4.2) and (Comb-KDF-2) (Appendix B.4.3) options.
 - * Add two candidate instantiations.
- o draft-00: Initial version.

1.2. Terminology

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

- o "Traditional" algorithms: Algorithms which are widely deployed today, but which may be deprecated in the future. In the context of TLS 1.3 in 2019, examples of traditional key exchange algorithms include elliptic curve Diffie-Hellman using secp256r1 or x25519, or finite-field Diffie-Hellman.
- o "Next-generation" (or "next-gen") algorithms: Algorithms which are not yet widely deployed, but which may eventually be widely deployed. An additional facet of these algorithms may be that we have less confidence in their security due to them being relatively new or less studied. This includes "post-quantum" algorithms.

"Hybrid" key exchange, in this context, means the use of two (or more) key exchange algorithms based on different cryptographic assumptions, e.g., one traditional algorithm and one next-gen algorithm, with the purpose of the final session key being secure as long as at least one of the component key exchange algorithms remains unbroken. We use the term "component" algorithms to refer to the algorithms combined in a hybrid key exchange.

The primary motivation of this document is preparing for post-quantum algorithms. However, it is possible that public key cryptography based on alternative mathematical constructions will be required independent of the advent of a quantum computer, for example because of a cryptanalytic breakthrough. As such we opt for the more generic term "next-generation" algorithms rather than exclusively "post-quantum" algorithms.

Note that TLS 1.3 uses the phrase "groups" to refer to key exchange algorithms - for example, the "supported_groups" extension - since all key exchange algorithms in TLS 1.3 are Diffie-Hellman-based. As a result, some parts of this document will refer to data structures

or messages with the term "group" in them despite using a key exchange algorithm that is not Diffie-Hellman-based nor a group.

1.3. Motivation for use of hybrid key exchange

A hybrid key exchange algorithm allows early adopters eager for post-quantum security to have the potential of post-quantum security (possibly from a less-well-studied algorithm) while still retaining at least the security currently offered by traditional algorithms. They may even need to retain traditional algorithms due to regulatory constraints, for example FIPS compliance.

Ideally, one would not use hybrid key exchange: one would have confidence in a single algorithm and parameterization that will stand the test of time. However, this may not be the case in the face of quantum computers and cryptanalytic advances more generally.

Many (though not all) post-quantum algorithms currently under consideration are relatively new; they have not been subject to the same depth of study as RSA and finite-field or elliptic curve Diffie-Hellman, and thus the security community does not necessarily have as much confidence in their fundamental security, or the concrete security level of specific parameterizations.

Moreover, it is possible that even by the end of the NIST Post-Quantum Cryptography Standardization Project, and for a period of time thereafter, conservative users may not have full confidence in some algorithms.

As such, there may be users for whom hybrid key exchange is an appropriate step prior to an eventual transition to next-generation algorithms.

1.4. Scope

This document focuses on hybrid ephemeral key exchange in TLS 1.3 [TLS13]. It intentionally does not address:

- o Selecting which next-generation algorithms to use in TLS 1.3, nor algorithm identifiers nor encoding mechanisms for next-generation algorithms. This selection will be based on the recommendations by the Crypto Forum Research Group (CFRG), which is currently waiting for the results of the NIST Post-Quantum Cryptography Standardization Project [NIST].
- o Authentication using next-generation algorithms. If a cryptographic assumption is broken due to the advent of a quantum computer or some other cryptanalytic breakthrough, confidentiality

of information can be broken retroactively by any adversary who has passively recorded handshakes and encrypted communications. In contrast, session authentication cannot be retroactively broken.

1.5. Goals

The primary goal of a hybrid key exchange mechanism is to facilitate the establishment of a shared secret which remains secure as long as as one of the component key exchange mechanisms remains unbroken.

In addition to the primary cryptographic goal, there may be several additional goals in the context of TLS 1.3:

- o ***Backwards compatibility:** Clients and servers who are "hybrid-aware", i.e., compliant with whatever hybrid key exchange standard is developed for TLS, should remain compatible with endpoints and middle-boxes that are not hybrid-aware. The three scenarios to consider are:
 1. Hybrid-aware client, hybrid-aware server: These parties should establish a hybrid shared secret.
 2. Hybrid-aware client, non-hybrid-aware server: These parties should establish a traditional shared secret (assuming the hybrid-aware client is willing to downgrade to traditional-only).
 3. Non-hybrid-aware client, hybrid-aware server: These parties should establish a traditional shared secret (assuming the hybrid-aware server is willing to downgrade to traditional-only).

Ideally backwards compatibility should be achieved without extra round trips and without sending duplicate information; see below.

- o ***High performance:** Use of hybrid key exchange should not be prohibitively expensive in terms of computational performance. In general this will depend on the performance characteristics of the specific cryptographic algorithms used, and as such is outside the scope of this document. See [BCNS15], [CECPQ1], [FRODO] for preliminary results about performance characteristics.
- o ***Low latency:** Use of hybrid key exchange should not substantially increase the latency experienced to establish a connection. Factors affecting this may include the following.

- * The computational performance characteristics of the specific algorithms used. See above.
- * The size of messages to be transmitted. Public key and ciphertext sizes for post-quantum algorithms range from hundreds of bytes to over one hundred kilobytes, so this impact can be substantial. See [BCNS15], [FRODO] for preliminary results in a laboratory setting, and [LANGLEY] for preliminary results on more realistic networks.
- * Additional round trips added to the protocol. See below.
- o *No extra round trips:* Attempting to negotiate hybrid key exchange should not lead to extra round trips in any of the three hybrid-aware/non-hybrid-aware scenarios listed above.
- o *Minimal duplicate information:* Attempting to negotiate hybrid key exchange should not mean having to send multiple public keys of the same type.

2. Key encapsulation mechanisms

In the context of the NIST Post-Quantum Cryptography Standardization Project, key exchange algorithms are formulated as key encapsulation mechanisms (KEMs), which consist of three algorithms:

- o "KeyGen() -> (pk, sk)": A probabilistic key generation algorithm, which generates a public key "pk" and a secret key "sk".
- o "Encaps(pk) -> (ct, ss)": A probabilistic encapsulation algorithm, which takes as input a public key "pk" and outputs a ciphertext "ct" and shared secret "ss".
- o "Decaps(sk, ct) -> ss": A decapsulation algorithm, which takes as input a secret key "sk" and ciphertext "ct" and outputs a shared secret "ss", or in some cases a distinguished error value.

The main security property for KEMs is indistinguishability under adaptive chosen ciphertext attack (IND-CCA2), which means that shared secret values should be indistinguishable from random strings even given the ability to have arbitrary ciphertexts decapsulated. IND-CCA2 corresponds to security against an active attacker, and the public key / secret key pair can be treated as a long-term key or reused. A common design pattern for obtaining security under key reuse is to apply the Fujisaki-Okamoto (FO) transform [FO] or a variant thereof [HHK].

A weaker security notion is indistinguishability under chosen plaintext attack (IND-CPA), which means that the shared secret values should be indistinguishable from random strings given a copy of the public key. IND-CPA roughly corresponds to security against a passive attacker, and sometimes corresponds to one-time key exchange.

Key exchange in TLS 1.3 is phrased in terms of Diffie-Hellman key exchange in a group. DH key exchange can be modeled as a KEM, with "KeyGen" corresponding to selecting an exponent " x " as the secret key and computing the public key " g^x "; encapsulation corresponding to selecting an exponent " y ", computing the ciphertext " g^y " and the shared secret " g^{xy} ", and decapsulation as computing the shared secret " g^{xy} ". See [I-D.irtf-cfrg-hpke] for more details of such Diffie-Hellman-based key encapsulation mechanisms.

TLS 1.3 does not require that ephemeral public keys be used only in a single key exchange session; some implementations may reuse them, at the cost of limited forward secrecy. As a result, any KEM used in this document MUST explicitly be designed to be secure in the event that the public key is re-used, such as achieving IND-CCA2 security or having a transform like the Fujisaki-Okamoto transform [FO] [HHK] applied. While it is recommended that implementations avoid reuse of KEM public keys, implementations that do reuse KEM public keys MUST ensure that the number of reuses of a KEM public key abides by any bounds in the specification of the KEM or subsequent security analyses. Implementations MUST NOT reuse randomness in the generation of KEM ciphertexts.

3. Construction for hybrid key exchange

3.1. Negotiation

Each particular combination of algorithms in a hybrid key exchange will be represented as a "NamedGroup" and sent in the "supported_groups" extension. No internal structure or grammar is implied or required in the value of the identifier; they are simply opaque identifiers.

Each value representing a hybrid key exchange will correspond to an ordered pair of two algorithms. For example, a future document could specify that hybrid value 0x2000 corresponds to secp256r1+ntruhrss701, and 0x2001 corresponds to x25519+ntruhrss701. (We note that this is independent from future documents standardizing solely post-quantum key exchange methods, which would have to be assigned their own identifier.)

Specific values shall be standardized by IANA in the TLS Supported Groups registry. We suggest that values 0x2000 through 0x2EFF are

suitable for hybrid key exchange methods (the leading "2" suggesting that there are 2 algorithms), noting that 0x2A2A is reserved as a GREASE value [GREASE]. This document requests that values 0x2F00 through 0x2FFF be reserved for Private Use for hybrid key exchange.

```
enum {  
  
    /* Elliptic Curve Groups (ECDHE) */  
    secp256r1(0x0017), secp384r1(0x0018), secp521r1(0x0019),  
    x25519(0x001D), x448(0x001E),  
  
    /* Finite Field Groups (DHE) */  
    ffdhe2048(0x0100), ffdhe3072(0x0101), ffdhe4096(0x0102),  
    ffdhe6144(0x0103), ffdhe8192(0x0104),  
  
    /* Hybrid Key Exchange Methods */  
    TBD(0xTBD), ...,  
  
    /* Reserved Code Points */  
    ffdhe_private_use(0x01FC..0x01FF),  
    hybrid_private_use(0x2F00..0x2FFF),  
    ecdhe_private_use(0xFE00..0xFEFF),  
    (0xFFFF)  
} NamedGroup;
```

3.2. Transmitting public keys and ciphertexts

We take the relatively simple "concatenation approach": the messages from the two algorithms being hybridized will be concatenated together and transmitted as a single value, to avoid having to change existing data structures. However we do add structure in the concatenation procedure, specifically including length fields, so that the concatenation operation is unambiguous. Note that among the Round 2 candidates in the NIST Post-Quantum Cryptography Standardization Project, not all algorithms have fixed public key sizes; for example, the SIKE key encapsulation mechanism permits compressed or uncompressed public keys at each security level, and the compressed and uncompressed formats are interoperable.

Recall that in TLS 1.3 a KEM public key or KEM ciphertext is represented as a "KeyShareEntry":

```
struct {  
    NamedGroup group;  
    opaque key_exchange<1..2^16-1>;  
} KeyShareEntry;
```

These are transmitted in the "extension_data" fields of "KeyShareClientHello" and "KeyShareServerHello" extensions:

```
struct {  
    KeyShareEntry client_shares<0..2^16-1>;  
} KeyShareClientHello;  
  
struct {  
    KeyShareEntry server_share;  
} KeyShareServerHello;
```

The client's shares are listed in descending order of client preference; the server selects one algorithm and sends its corresponding share.

For a hybrid key exchange, the "key_exchange" field of a "KeyShareEntry" is the following data structure:

```
struct {  
    opaque key_exchange_1<1..2^16-1>;  
    opaque key_exchange_2<1..2^16-1>;  
} HybridKeyExchange
```

The order of shares in the "HybridKeyExchange" struct is the same as the order of algorithms indicated in the definition of the "NamedGroup".

For the client's share, the "key_exchange_1" and "key_exchange_2" values are the "pk" outputs of the corresponding KEMs' "KeyGen" algorithms, if that algorithm corresponds to a KEM; or the (EC)DH ephemeral key share, if that algorithm corresponds to an (EC)DH group. For the server's share, the "key_exchange_1" and "key_exchange_2" values are the "ct" outputs of the corresponding KEMs' "Encaps" algorithms, if that algorithm corresponds to a KEM; or the (EC)DH ephemeral key share, if that algorithm corresponds to an (EC)DH group.

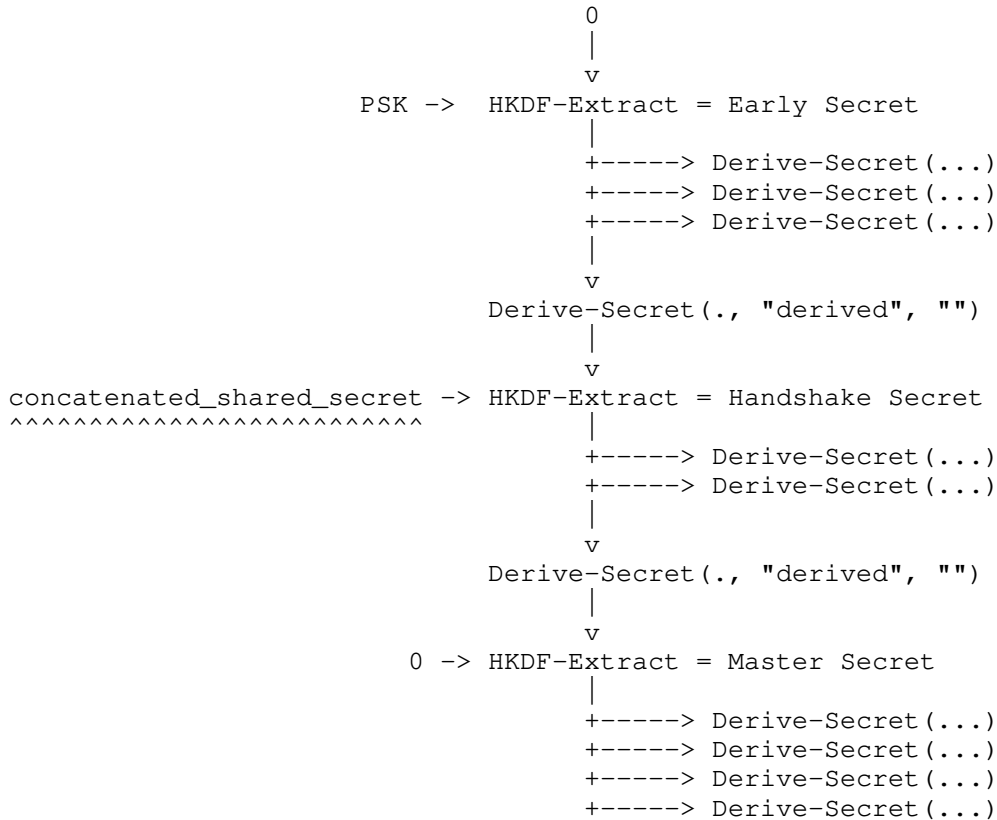
3.3. Shared secret calculation

Here we also take a simple "concatenation approach": the two shared secrets are concatenated together and used as the shared secret in the existing TLS 1.3 key schedule. In this case, we do not add any additional structure (length fields) in the concatenation procedure: among all Round 2 candidates, once the algorithm and variant are specified, the shared secret output length is fixed.

In other words, the shared secret is calculated as

```
concatenated_shared_secret = shared_secret_1 || shared_secret_2
```

and inserted into the TLS 1.3 key schedule in place of the (EC)DHE shared secret:



FIPS-compliance of shared secret concatenation. [NIST-SP-800-56C] or [NIST-SP-800-135] give NIST recommendations for key derivation methods in key exchange protocols. Some hybrid combinations may combine the shared secret from a NIST-approved algorithm (e.g., ECDH using the nistp256/secp256r1 curve) with a shared secret from a non-approved algorithm (e.g., post-quantum). Although the simple concatenation approach above is not currently an approved method in [NIST-SP-800-56C] or [NIST-SP-800-135], NIST indicated in January 2020 that a forthcoming revision of [NIST-SP-800-56C] will list simple concatenation as an approved method [NIST-FAQ].

4. Open questions

Larger public keys and/or ciphertexts. The "HybridKeyExchange" struct in Section 3.2 limits public keys and ciphertexts to $2^{16}-1$ bytes; this is bounded by the same $(2^{16}-1)$ -byte limit on the "key_exchange" field in the "KeyShareEntry" struct. Some post-quantum KEMs have larger public keys and/or ciphertexts; for example, Classic McEliece's smallest parameter set has public key size 261,120 bytes. Hence this draft can not accommodate all current NIST Round 2 candidates.

If it is desired to accommodate algorithms with public keys or ciphertexts larger than $2^{16}-1$ bytes, options include a) revising the TLS 1.3 standard to allow longer "key_exchange" fields; b) creating an alternative extension which is sufficiently large; or c) providing a reference to an external public key, e.g. a URL at which to look up the public key (along with a hash to verify).

Duplication of key shares. Concatenation of public keys in the "HybridKeyExchange" struct as described in Section 3.2 can result in sending duplicate key shares. For example, if a client wanted to offer support for two combinations, say "secp256r1+sikep503" and "x25519+sikep503", it would end up sending two sikep503 public keys, since the "KeyShareEntry" for each combination contains its own copy of a sikep503 key. This duplication may be more problematic for post-quantum algorithms which have larger public keys.

If it is desired to avoid duplication of key shares, options include a) disconnect the use of a combination for the algorithm identifier from the use of concatenation of public keys by introducing new logic and/or data structures (see Appendix B.3.2 or Appendix B.3.3); or b) provide some back reference from a later key share entry to an earlier one.

Variable-length shared secrets. The shared secret calculation in Section 3.3 directly concatenates the shared secret values of each scheme, rather than encoding them with length fields. This implicitly assumes that the length of each shared secret is fixed once the algorithm is fixed. This is the case for all Round 2 candidates.

However, if it is envisioned that this specification be used with algorithms which do not have fixed-length shared secrets (after the variant has been fixed by the algorithm identifier in the "NamedGroup" negotiation in Section 3.1), then Section 3.3 should be revised to use an unambiguous concatenation method such as the following:


```
struct {  
    opaque shared_secret_1<1..2^16-1>;  
    opaque shared_secret_2<1..2^16-1>;  
} HybridSharedSecret
```

Guidance from the working group is particularly requested on this point.

Resumption. TLS 1.3 allows for session resumption via a PSK. When a PSK is used during session establishment, an ephemeral key exchange can also be used to enhance forward secrecy. If the original key exchange was hybrid, should an ephemeral key exchange in a resumption of that original key exchange be required to use the same hybrid algorithms?

Failures. Some post-quantum key exchange algorithms have non-trivial failure rates: two honest parties may fail to agree on the same shared secret with non-negligible probability. Does a non-negligible failure rate affect the security of TLS? How should such a failure be treated operationally? What is an acceptable failure rate?

5. IANA Considerations

Identifiers for specific key exchange algorithm combinations will be defined in later documents. This document requests IANA reserve values 0x2F00..0x2FFF in the TLS Supported Groups registry for private use for hybrid key exchange methods.

6. Security Considerations

The shared secrets computed in the hybrid key exchange should be computed in a way that achieves the "hybrid" property: the resulting secret is secure as long as at least one of the component key exchange algorithms is unbroken. See [GIACON] and [BINDEL] for an investigation of these issues. Under the assumption that shared secrets are fixed length once the combination is fixed, the construction from Section 3.3 corresponds to the dual-PRF combiner of [BINDEL] which is shown to preserve security under the assumption that the hash function is a dual-PRF.

As noted in Section 2, KEMs used in this document MUST explicitly be designed to be secure in the event that the public key is re-used, such as achieving IND-CCA2 security or having a transform like the Fujisaki-Okamoto transform applied. Some IND-CPA-secure post-quantum KEMs (i.e., without countermeasures such as the FO transform) are completely insecure under public key reuse; for example, some

lattice-based IND-CPA-secure KEMs are vulnerable to attacks that recover the private key after just a few thousand samples [FLUHRER].

7. Acknowledgements

These ideas have grown from discussions with many colleagues, including Christopher Wood, Matt Campagna, Eric Crockett, authors of the various hybrid Internet-Drafts and implementations cited in this document, and members of the TLS working group. The immediate impetus for this document came from discussions with attendees at the Workshop on Post-Quantum Software in Mountain View, California, in January 2019. Martin Thomson suggested the (Comb-KDF-1) (Appendix B.4.2) approach. Daniel J. Bernstein and Tanja Lange commented on the risks of reuse of ephemeral public keys.

8. References

8.1. Normative References

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

8.2. Informative References

- [BCNS15] Bos, J., Costello, C., Naehrig, M., and D. Stebila, "Post-Quantum Key Exchange for the TLS Protocol from the Ring Learning with Errors Problem", 2015 IEEE Symposium on Security and Privacy, DOI 10.1109/sp.2015.40, May 2015.
- [BERNSTEIN] "Post-Quantum Cryptography", Springer Berlin Heidelberg book, DOI 10.1007/978-3-540-88702-7, 2009.
- [BINDEL] Bindel, N., Brendel, J., Fischlin, M., Goncalves, B., and D. Stebila, "Hybrid Key Encapsulation Mechanisms and Authenticated Key Exchange", Post-Quantum Cryptography pp. 206-226, DOI 10.1007/978-3-030-25510-7_12, 2019.
- [CECPQ1] Braithwaite, M., "Experimenting with Post-Quantum Cryptography", July 2016, <<https://security.googleblog.com/2016/07/experimenting-with-post-quantum.html>>.
- [CECPQ2] Langley, A., "CECPQ2", December 2018, <<https://www.imperialviolet.org/2018/12/12/cecpq2.html>>.

- [DFGS15] Dowling, B., Fischlin, M., Guenther, F., and D. Stebila, "A Cryptographic Analysis of the TLS 1.3 Handshake Protocol Candidates", Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security - CCS '15, DOI 10.1145/2810103.2813653, 2015.
- [DODIS] Dodis, Y. and J. Katz, "Chosen-Ciphertext Security of Multiple Encryption", Theory of Cryptography pp. 188-209, DOI 10.1007/978-3-540-30576-7_11, 2005.
- [DOWLING] Dowling, B., "Provable Security of Internet Protocols", Queensland University of Technology dissertation, DOI 10.5204/thesis.eprints.108960, n.d..
- [ETSI] Campagna, M., Ed. and . others, "Quantum safe cryptography and security: An introduction, benefits, enablers and challengers", ETSI White Paper No. 8 , June 2015, <<https://www.etsi.org/images/files/ETSIWhitePapers/QuantumSafeWhitepaper.pdf>>.
- [EVEN] Even, S. and O. Goldreich, "On the Power of Cascade Ciphers", Advances in Cryptology pp. 43-50, DOI 10.1007/978-1-4684-4730-9_4, 1984.
- [EXTERN-PSK] Housley, R., "TLS 1.3 Extension for Certificate-based Authentication with an External Pre-Shared Key", draft-ietf-tls-tls13-cert-with-extern-psk-07 (work in progress), December 2019.
- [FLUHRER] Fluhrer, S., "Cryptanalysis of ring-LWE based key exchange with key share reuse", Cryptology ePrint Archive, Report 2016/085 , January 2016, <<https://eprint.iacr.org/2016/085>>.
- [FO] Fujisaki, E. and T. Okamoto, "Secure Integration of Asymmetric and Symmetric Encryption Schemes", Journal of Cryptology Vol. 26, pp. 80-101, DOI 10.1007/s00145-011-9114-1, December 2011.
- [FRODO] Bos, J., Costello, C., Ducas, L., Mironov, I., Naehrig, M., Nikolaenko, V., Raghunathan, A., and D. Stebila, "Frodo", Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security - CCS'16, DOI 10.1145/2976749.2978425, 2016.

- [GIACON] Giacon, F., Heuer, F., and B. Poettering, "KEM Combiners", Public-Key Cryptography – PKC 2018 pp. 190–218, DOI 10.1007/978-3-319-76578-5_7, 2018.
- [GREASE] Benjamin, D., "Applying GREASE to TLS Extensibility", draft-ietf-tls-grease-04 (work in progress), August 2019.
- [HARNIK] Harnik, D., Kilian, J., Naor, M., Reingold, O., and A. Rosen, "On Robust Combiners for Oblivious Transfer and Other Primitives", Lecture Notes in Computer Science pp. 96–113, DOI 10.1007/11426639_6, 2005.
- [HHK] Hofheinz, D., Hoewelmanns, K., and E. Kiltz, "A Modular Analysis of the Fujisaki-Okamoto Transformation", Theory of Cryptography pp. 341–371, DOI 10.1007/978-3-319-70500-2_12, 2017.
- [HOFFMAN] Hoffman, P., "The Transition from Classical to Post-Quantum Cryptography", draft-hoffman-c2pq-06 (work in progress), November 2019.
- [I-D.irtf-cfrg-hpke]
Barnes, R. and K. Bhargavan, "Hybrid Public Key Encryption", draft-irtf-cfrg-hpke-02 (work in progress), November 2019.
- [IKE-HYBRID]
Tjhai, C., Tomlinson, M., grbartle@cisco.com, g., Fluhrer, S., Geest, D., Garcia-Morchon, O., and V. Smyslov, "Framework to Integrate Post-quantum Key Exchanges into Internet Key Exchange Protocol Version 2 (IKEv2)", draft-tjhai-ipsecme-hybrid-qske-ikev2-04 (work in progress), July 2019.
- [IKE-PSK] Fluhrer, S., Kampanakis, P., McGrew, D., and V. Smyslov, "Mixing Preshared Keys in IKEv2 for Post-quantum Security", draft-ietf-ipsecme-qr-ikev2-11 (work in progress), January 2020.
- [KIEFER] Kiefer, F. and K. Kwiatkowski, "Hybrid ECDHE-SIDH Key Exchange for TLS", draft-kiefer-tls-ecdhe-sidh-00 (work in progress), November 2018.
- [KPW13] Krawczyk, H., Paterson, K., and H. Wee, "On the Security of the TLS Protocol: A Systematic Analysis", Advances in Cryptology – CRYPTO 2013 pp. 429–448, DOI 10.1007/978-3-642-40041-4_24, 2013.

- [LANGLEY] Langley, A., "Post-quantum confidentiality for TLS", April 2018, <<https://www.imperialviolet.org/2018/04/11/pqconftls.html>>.
- [NIELSEN] Nielsen, M. and I. Chuang, "Quantum Computation and Quantum Information", Cambridge University Press , 2000.
- [NIST] National Institute of Standards and Technology (NIST), "Post-Quantum Cryptography", n.d., <<https://www.nist.gov/pqcrypto>>.
- [NIST-FAQ] National Institute of Standards and Technology (NIST), "Post-Quantum Cryptography - FAQs", January 2020, <<https://csrc.nist.gov/Projects/post-quantum-cryptography/faqs>>.
- [NIST-SP-800-135] National Institute of Standards and Technology (NIST), "Recommendation for Existing Application-Specific Key Derivation Functions", December 2011, <<https://doi.org/10.6028/NIST.SP.800-135r1>>.
- [NIST-SP-800-56C] National Institute of Standards and Technology (NIST), "Recommendation for Key-Derivation Methods in Key-Establishment Schemes", April 2018, <<https://doi.org/10.6028/NIST.SP.800-56Cr1>>.
- [OQS-102] Open Quantum Safe Project, "OQS-OpenSSL-1-0-2_stable", November 2018, <https://github.com/open-quantum-safe/openssl/tree/OQS-OpenSSL_1_0_2-stable>.
- [OQS-111] Open Quantum Safe Project, "OQS-OpenSSL-1-1-1_stable", November 2018, <https://github.com/open-quantum-safe/openssl/tree/OQS-OpenSSL_1_1_1-stable>.
- [SCHANCK] Schanck, J. and D. Stebila, "A Transport Layer Security (TLS) Extension For Establishing An Additional Shared Secret", draft-schanck-tls-additional-keyshare-00 (work in progress), April 2017.
- [WHYTE12] Schanck, J., Whyte, W., and Z. Zhang, "Quantum-Safe Hybrid (QSH) Ciphersuite for Transport Layer Security (TLS) version 1.2", draft-whyte-qsh-tls12-02 (work in progress), July 2016.

- [WHYTE13] Whyte, W., Zhang, Z., Fluhrer, S., and O. Garcia-Morchon, "Quantum-Safe Hybrid (QSH) Key Exchange for Transport Layer Security (TLS) version 1.3", draft-whyte-qsh-tls13-06 (work in progress), October 2017.
- [XMSS] Huelsing, A., Butin, D., Gazdag, S., Rijneveld, J., and A. Mohaisen, "XMSS: eXtended Merkle Signature Scheme", RFC 8391, DOI 10.17487/RFC8391, May 2018, <<https://www.rfc-editor.org/info/rfc8391>>.
- [ZHANG] Zhang, R., Hanaoka, G., Shikata, J., and H. Imai, "On the Security of Multiple Encryption or CCA-security+CCA-security=CCA-security?", Public Key Cryptography - PKC 2004 pp. 360-374, DOI 10.1007/978-3-540-24632-9_26, 2004.

Appendix A. Related work

Quantum computing and post-quantum cryptography in general are outside the scope of this document. For a general introduction to quantum computing, see a standard textbook such as [NIELSEN]. For an overview of post-quantum cryptography as of 2009, see [BERNSTEIN]. For the current status of the NIST Post-Quantum Cryptography Standardization Project, see [NIST]. For additional perspectives on the general transition from classical to post-quantum cryptography, see for example [ETSI] and [HOFFMAN], among others.

There have been several Internet-Drafts describing mechanisms for embedding post-quantum and/or hybrid key exchange in TLS:

- o Internet-Drafts for TLS 1.2: [WHYTE12]
- o Internet-Drafts for TLS 1.3: [KIEFER], [SCHANCK], [WHYTE13]

There have been several prototype implementations for post-quantum and/or hybrid key exchange in TLS:

- o Experimental implementations in TLS 1.2: [BCNS15], [CECPQ1], [FRODO], [OQS-102]
- o Experimental implementations in TLS 1.3: [CECPQ2], [OQS-111]

These experimental implementations have taken an ad hoc approach and not attempted to implement one of the drafts listed above.

Unrelated to post-quantum but still related to the issue of combining multiple types of keying material in TLS is the use of pre-shared keys, especially the recent TLS working group document on including an external pre-shared key [EXTERN-PSK].

Considering other IETF standards, there is work on post-quantum preshared keys in IKEv2 [IKE-PSK] and a framework for hybrid key exchange in IKEv2 [IKE-HYBRID]. The XMSS hash-based signature scheme has been published as an informational RFC by the IRTF [XMSS].

In the academic literature, [EVEN] initiated the study of combining multiple symmetric encryption schemes; [ZHANG], [DODIS], and [HARNIK] examined combining multiple public key encryption schemes, and [HARNIK] coined the term "robust combiner" to refer to a compiler that constructs a hybrid scheme from individual schemes while preserving security properties. [GIACON] and [BINDEL] examined combining multiple key encapsulation mechanisms.

Appendix B. Design Considerations

This appendix discusses choices one could make along four distinct axes when integrating hybrid key exchange into TLS 1.3:

1. How to negotiate the use of hybridization in general and component algorithms specifically?
2. How many component algorithms can be combined?
3. How should multiple key shares (public keys / ciphertexts) be conveyed?
4. How should multiple shared secrets be combined?

The construction in the main body illustrates one selection along each of these axes. The remainder of this appendix outlines various options we have identified for each of these choices. Immediately below we provide a summary list. Options are labelled with a short code in parentheses to provide easy cross-referencing.

1. (Neg) (Appendix B.1) How to negotiate the use of hybridization in general and component algorithms specifically?
 - * (Neg-Ind) (Appendix B.1.2) Negotiating component algorithms individually
 - + (Neg-Ind-1) (Appendix B.1.2.1) Traditional algorithms in "ClientHello" "supported_groups" extension, next-gen algorithms in another extension
 - + (Neg-Ind-2) (Appendix B.1.2.2) Both types of algorithms in "supported_groups" with external mapping to tradition/next-gen.

- + (Neg-Ind-3) (Appendix B.1.2.3) Both types of algorithms in "supported_groups" separated by a delimiter.
 - * (Neg-Comb) (Appendix B.1.3) Negotiating component algorithms as a combination
 - + (Neg-Comb-1) (Appendix B.1.3.1) Standardize "NamedGroup" identifiers for each desired combination.
 - + (Neg-Comb-2) (Appendix B.1.3.2) Use placeholder identifiers in "supported_groups" with an extension defining the combination corresponding to each placeholder.
 - + (Neg-Comb-3) (Appendix B.1.3.3) List combinations by inserting grouping delimiters into "supported_groups" list.
2. (Num) (Appendix B.2) How many component algorithms can be combined?
- * (Num-2) (Appendix B.2.1) Two.
 - * (Num-2+) (Appendix B.2.2) Two or more.
3. (Shares) (Appendix B.3) How should multiple key shares (public keys / ciphertexts) be conveyed?
- * (Shares-Concat) (Appendix B.3.1) Concatenate each combination of key shares.
 - * (Shares-Multiple) (Appendix B.3.2) Send individual key shares for each algorithm.
 - * (Shares-Ext-Additional) (Appendix B.3.3) Use an extension to convey key shares for component algorithms.
4. (Comb) (Appendix B.4) How should multiple shared secrets be combined?
- * (Comb-Concat) (Appendix B.4.1) Concatenate the shared secrets then use directly in the TLS 1.3 key schedule.
 - * (Comb-KDF-1) (Appendix B.4.2) and (Comb-KDF-2) (Appendix B.4.3) KDF the shared secrets together, then use the output in the TLS 1.3 key schedule.
 - * (Comb-XOR) (Appendix B.4.4) XOR the shared secrets then use directly in the TLS 1.3 key schedule.

- * (Comb-Chain) (Appendix B.4.5) Extend the TLS 1.3 key schedule so that there is a stage of the key schedule for each shared secret.
- * (Comb-AltInput) (Appendix B.4.6) Use the second shared secret in an alternate (otherwise unused) input in the TLS 1.3 key schedule.

B.1. (Neg) How to negotiate hybridization and component algorithms?

B.1.1. Key exchange negotiation in TLS 1.3

Recall that in TLS 1.3, the key exchange mechanism is negotiated via the "supported_groups" extension. The "NamedGroup" enum is a list of standardized groups for Diffie-Hellman key exchange, such as "secp256r1", "x25519", and "ffdhe2048".

The client, in its "ClientHello" message, lists its supported mechanisms in the "supported_groups" extension. The client also optionally includes the public key of one or more of these groups in the "key_share" extension as a guess of which mechanisms the server might accept in hopes of reducing the number of round trips.

If the server is willing to use one of the client's requested mechanisms, it responds with a "key_share" extension containing its public key for the desired mechanism.

If the server is not willing to use any of the client's requested mechanisms, the server responds with a "HelloRetryRequest" message that includes an extension indicating its preferred mechanism.

B.1.2. (Neg-Ind) Negotiating component algorithms individually

In these three approaches, the parties negotiate which traditional algorithm and which next-gen algorithm to use independently. The "NamedGroup" enum is extended to include algorithm identifiers for each next-gen algorithm.

B.1.2.1. (Neg-Ind-1)

The client advertises two lists to the server: one list containing its supported traditional mechanisms (e.g. via the existing "ClientHello" "supported_groups" extension), and a second list containing its supported next-generation mechanisms (e.g., via an additional "ClientHello" extension). A server could then select one algorithm from the traditional list, and one algorithm from the next-generation list. (This is the approach in [SCHANCK].)

B.1.2.2. (Neg-Ind-2)

The client advertises a single list to the server which contains both its traditional and next-generation mechanisms (e.g., all in the existing "ClientHello" "supported_groups" extension), but with some external table provides a standardized mapping of those mechanisms as either "traditional" or "next-generation". A server could then select two algorithms from this list, one from each category.

B.1.2.3. (Neg-Ind-3)

The client advertises a single list to the server delimited into sublists: one for its traditional mechanisms and one for its next-generation mechanisms, all in the existing "ClientHello" "supported_groups" extension, with a special code point serving as a delimiter between the two lists. For example, "supported_groups = secp256r1, x25519, delimiter, nextgen1, nextgen4".

B.1.3. (Neg-Comb) Negotiating component algorithms as a combination

In these three approaches, combinations of key exchange mechanisms appear as a single monolithic block; the parties negotiate which of several combinations they wish to use.

B.1.3.1. (Neg-Comb-1)

The "NamedGroup" enum is extended to include algorithm identifiers for each *combination* of algorithms desired by the working group. There is no "internal structure" to the algorithm identifiers for each combination, they are simply new code points assigned arbitrarily. The client includes any desired combinations in its "ClientHello" "supported_groups" list, and the server picks one of these. This is the approach in [KIEFER] and [OQS-111].

B.1.3.2. (Neg-Comb-2)

The "NamedGroup" enum is extended to include algorithm identifiers for each next-gen algorithm. Some additional field/extension is used to convey which combinations the parties wish to use. For example, in [WHYTE13], there are distinguished "NamedGroup" called "hybrid_marker 0", "hybrid_marker 1", "hybrid_marker 2", etc. This is complemented by a "HybridExtension" which contains mappings for each numbered "hybrid_marker" to the set of component key exchange algorithms (2 or more) for that proposed combination.

B.1.3.3. (Neg-Comb-3)

The client lists combinations in "supported_groups" list, using a special delimiter to indicate combinations. For example, "supported_groups = combo_delimiter, secp256r1, nextgen1, combo_delimiter, secp256r1, nextgen4, standalone_delimiter, secp256r1, x25519" would indicate that the client's highest preference is the combination secp256r1+nextgen1, the next highest preference is the combination secp256r1+nextgen4, then the single algorithm secp256r1, then the single algorithm x25519. A hybrid-aware server would be able to parse these; a hybrid-unaware server would see "unknown, secp256r1, unknown, unknown, secp256r1, unknown, unknown, secp256r1, x25519", which it would be able to process, although there is the potential that every "projection" of a hybrid list that is tolerable to a client does not result in list that is tolerable to the client.

B.1.4. Benefits and drawbacks

Combinatorial explosion. (Neg-Comb-1) (Appendix B.1.3.1) requires new identifiers to be defined for each desired combination. The other 4 options in this section do not.

Extensions. (Neg-Ind-1) (Appendix B.1.2.1) and (Neg-Comb-2) (Appendix B.1.3.2) require new extensions to be defined. The other options in this section do not.

New logic. All options in this section except (Neg-Comb-1) (Appendix B.1.3.1) require new logic to process negotiation.

Matching security levels. (Neg-Ind-1) (Appendix B.1.2.1), (Neg-Ind-2) (Appendix B.1.2.2), (Neg-Ind-3) (Appendix B.1.2.3), and (Neg-Comb-2) (Appendix B.1.3.2) allow algorithms of different claimed security level from their corresponding lists to be combined. For example, this could result in combining ECDH secp256r1 (classical security level 128) with NewHope-1024 (classical security level 256). Implementations dissatisfied with a mismatched security levels must either accept this mismatch or attempt to renegotiate. (Neg-Ind-1) (Appendix B.1.2.1), (Neg-Ind-2) (Appendix B.1.2.2), and (Neg-Ind-3) (Appendix B.1.2.3) give control over the combination to the server; (Neg-Comb-2) (Appendix B.1.3.2) gives control over the combination to the client. (Neg-Comb-1) (Appendix B.1.3.1) only allows standardized combinations, which could be set by TLS working group to have matching security (provided security estimates do not evolve separately).

Backwards-compability. TLS 1.3-compliant hybrid-unaware servers should ignore unrecognized elements in "supported_groups" (Neg-Ind-2)

(Appendix B.1.2.2), (Neg-Ind-3) (Appendix B.1.2.3), (Neg-Comb-1) (Appendix B.1.3.1), (Neg-Comb-2) (Appendix B.1.3.2) and unrecognized "ClientHello" extensions (Neg-Ind-1) (Appendix B.1.2.1), (Neg-Comb-2) (Appendix B.1.3.2). In (Neg-Ind-3) (Appendix B.1.2.3) and (Neg-Comb-3) (Appendix B.1.3.3), a server that is hybrid-unaware will ignore the delimiters in "supported_groups", and thus might try to negotiate an algorithm individually that is only meant to be used in combination; depending on how such an implementation is coded, it may also encounter bugs when the same element appears multiple times in the list.

B.2. (Num) How many component algorithms to combine?

B.2.1. (Num-2) Two

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

B.2.2. (Num-2+) Two or more

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

B.2.3. Benefits and Drawbacks

Restricting the number of component algorithms that can be hybridized to two substantially reduces the generality required. On the other hand, some adopters may want to further reduce risk by employing multiple next-gen algorithms built on different cryptographic assumptions.

B.3. (Shares) How to convey key shares?

In ECDH ephemeral key exchange, the client sends its ephemeral public key in the "key_share" extension of the "ClientHello" message, and the server sends its ephemeral public key in the "key_share" extension of the "ServerHello" message.

For a general key encapsulation mechanism used for ephemeral key exchange, we imagine that that client generates a fresh KEM public key / secret pair for each connection, sends it to the client, and the server responds with a KEM ciphertext. For simplicity and consistency with TLS 1.3 terminology, we will refer to both of these types of objects as "key shares".

In hybrid key exchange, we have to decide how to convey the client's two (or more) key shares, and the server's two (or more) key shares.

B.3.1. (Shares-Concat) Concatenate key shares

The client concatenates the bytes representing its two key shares and uses this directly as the "key_exchange" value in a "KeyShareEntry" in its "key_share" extension. The server does the same thing. Note that the "key_exchange" value can be an octet string of length at most $2^{16}-1$. This is the approach taken in [KIEFER], [OQS-111], and [WHYTE13].

B.3.2. (Shares-Multiple) Send multiple key shares

The client sends multiple key shares directly in the "client_shares" vectors of the "ClientHello" "key_share" extension. The server does the same. (Note that while the existing "KeyShareClientHello" struct allows for multiple key share entries, the existing "KeyShareServerHello" only permits a single key share entry, so some modification would be required to use this approach for the server to send multiple key shares.)

B.3.3. (Shares-Ext-Additional) Extension carrying additional key shares

The client sends the key share for its traditional algorithm in the original "key_share" extension of the "ClientHello" message, and the key share for its next-gen algorithm in some additional extension in the "ClientHello" message. The server does the same thing. This is the approach taken in [SCHANCK].

B.3.4. Benefits and Drawbacks

Backwards compatibility. (Shares-Multiple) (Appendix B.3.2) is fully backwards compatible with non-hybrid-aware servers. (Shares-Ext-Additional) (Appendix B.3.3) is backwards compatible with non-hybrid-aware servers provided they ignore unrecognized extensions. (Shares-Concat) (Appendix B.3.1) is backwards-compatible with non-hybrid aware servers, but may result in duplication / additional round trips (see below).

Duplication versus additional round trips. If a client wants to offer multiple key shares for multiple combinations in order to avoid retry requests, then the client may ended up sending a key share for one algorithm multiple times when using (Shares-Ext-Additional) (Appendix B.3.3) and (Shares-Concat) (Appendix B.3.1). (For example, if the client wants to send an ECDH-secp256r1 + McEliece123 key share, and an ECDH-secp256r1 + NewHope1024 key share, then the same ECDH public key may be sent twice. If the client also wants to offer a traditional ECDH-only key share for non-hybrid-aware implementations and avoid retry requests, then that same ECDH public

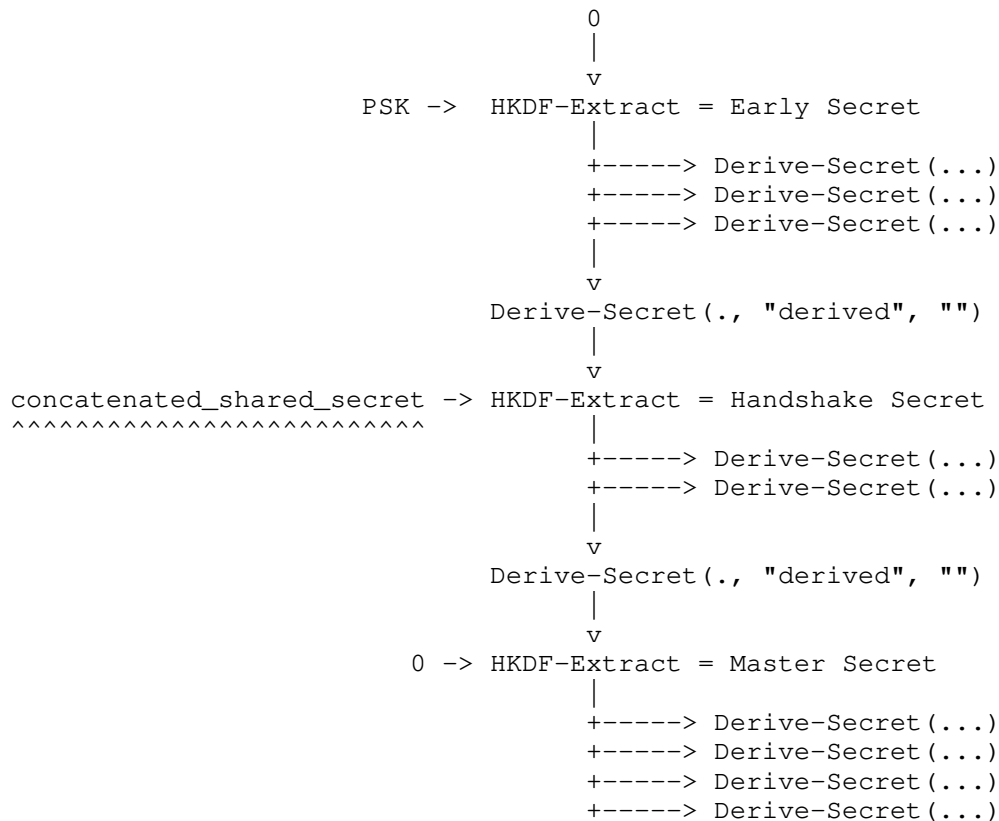
key may be sent another time.) (Shares-Multiple) (Appendix B.3.2)
does not result in duplicate key shares.

B.4. (Comb) How to use keys?

Each component key exchange algorithm establishes a shared secret. These shared secrets must be combined in some way that achieves the "hybrid" property: the resulting secret is secure as long as at least one of the component key exchange algorithms is unbroken.

B.4.1. (Comb-Concat) Concatenate keys

Each party concatenates the shared secrets established by each component algorithm in an agreed-upon order, then feeds that through the TLS key schedule. In the context of TLS 1.3, this would mean using the concatenated shared secret in place of the (EC)DHE input to the second call to "HKDF-Extract" in the TLS 1.3 key schedule:



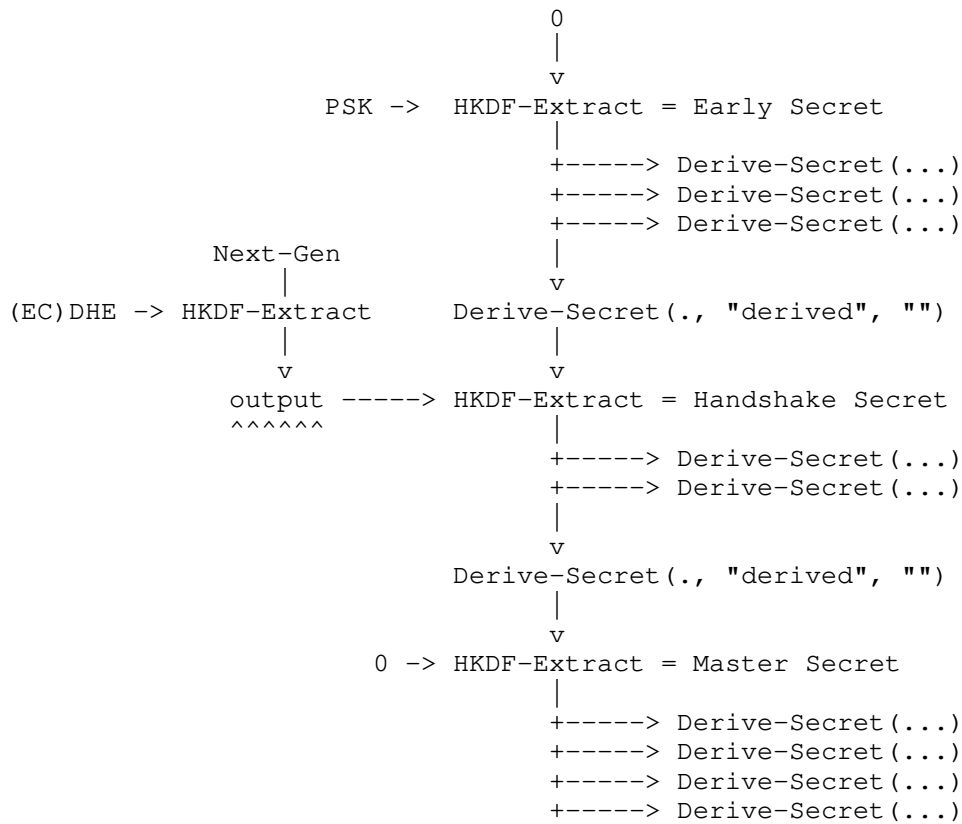
This is the approach used in [KIEFER], [OQS-111], and [WHYTE13].

[GIACON] analyzes the security of applying a KDF to concatenated KEM shared secrets, but their analysis does not exactly apply here since the transcript of ciphertexts is included in the KDF application (though it should follow relatively straightforwardly).

[BINDEL] analyzes the security of the (Comb-Concat) approach as abstracted in their "dualPRF" combiner. They show that, if the component KEMs are IND-CPA-secure (or IND-CCA-secure), then the values output by "Derive-Secret" are IND-CPA-secure (respectively, IND-CCA-secure). An important aspect of their analysis is that each ciphertext is input to the final PRF calls; this holds for TLS 1.3 since the "Derive-Secret" calls that derive output keys (application traffic secrets, and exporter and resumption master secrets) include the transcript hash as input.

B.4.2. (Comb-KDF-1) KDF keys

Each party feeds the shared secrets established by each component algorithm in an agreed-upon order into a KDF, then feeds that through the TLS key schedule. In the context of TLS 1.3, this would mean first applying "HKDF-Extract" to the shared secrets, then using the output in place of the (EC)DHE input to the second call to "HKDF-Extract" in the TLS 1.3 key schedule:

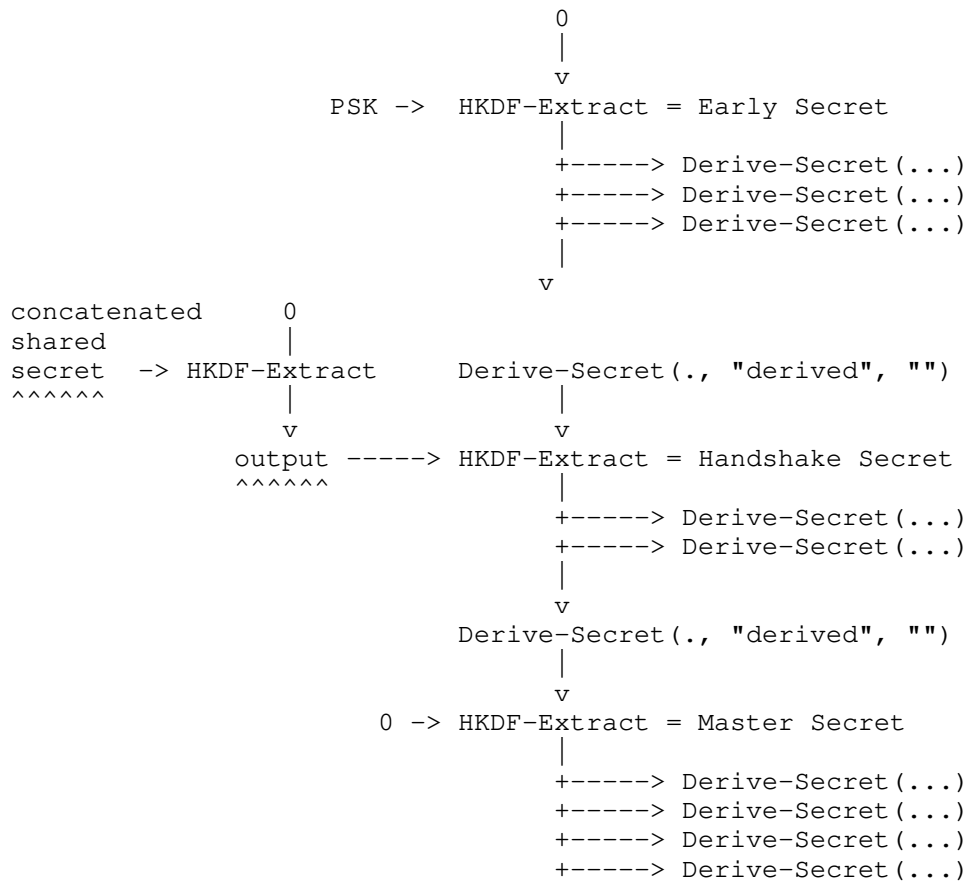


B.4.3. (Comb-KDF-2) KDF keys

Each party concatenates the shared secrets established by each component algorithm in an agreed-upon order then feeds that into a KDF, then feeds the result through the TLS key schedule.

Compared with (Comb-KDF-1) (Appendix B.4.2), this method concatenates the (2 or more) shared secrets prior to input to the KDF, whereas (Comb-KDF-1) puts the (exactly 2) shared secrets in the two different input slots to the KDF.

Compared with (Comb-Concat) (Appendix B.4.1), this method has an extract KDF application. While this adds computational overhead, this may provide a cleaner abstraction of the hybridization mechanism for the purposes of formal security analysis.



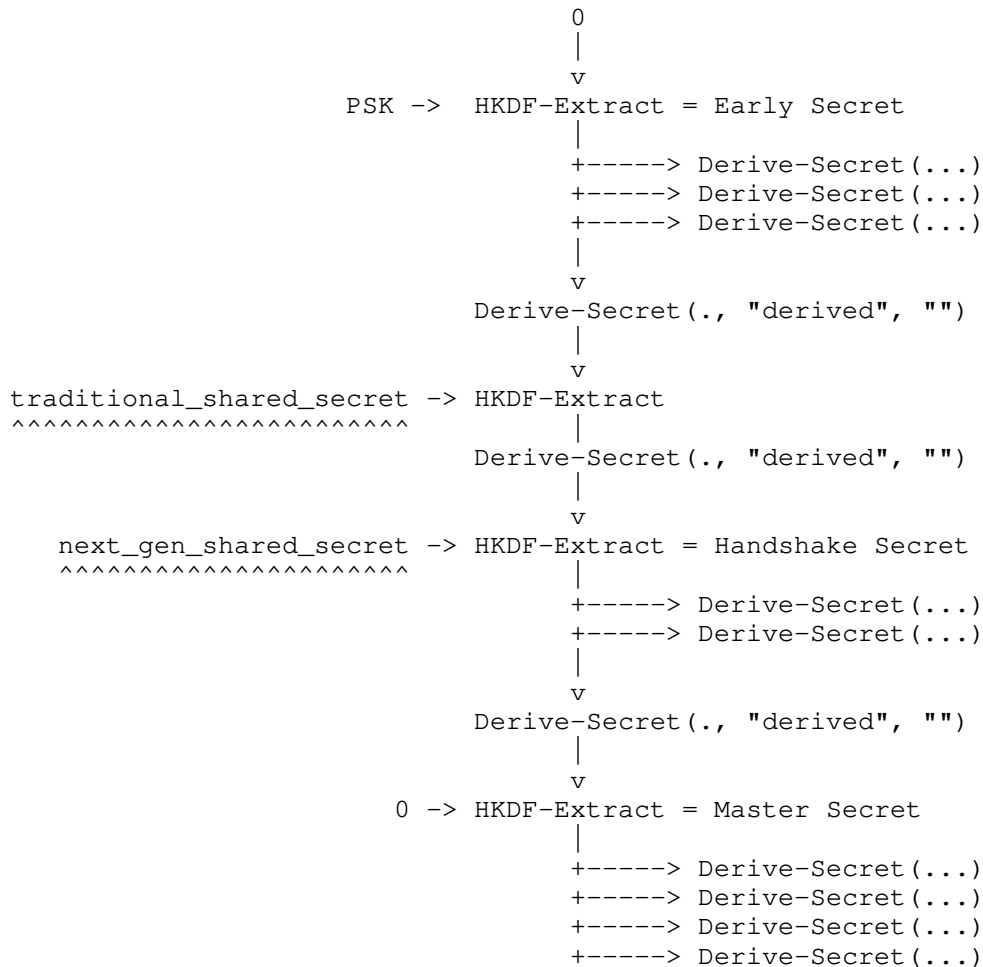
B.4.4. (Comb-XOR) XOR keys

Each party XORs the shared secrets established by each component algorithm (possibly after padding secrets of different lengths), then feeds that through the TLS key schedule. In the context of TLS 1.3, this would mean using the XORed shared secret in place of the (EC)DHE input to the second call to "HKDF-Extract" in the TLS 1.3 key schedule.

[GIACON] analyzes the security of applying a KDF to the XORed KEM shared secrets, but their analysis does not quite apply here since the transcript of ciphertexts is included in the KDF application (though it should follow relatively straightforwardly).

B.4.5. (Comb-Chain) Chain of KDF applications for each key

Each party applies a chain of key derivation functions to the shared secrets established by each component algorithm in an agreed-upon order; roughly speaking: $F(k_1 || F(k_2))$. In the context of TLS 1.3, this would mean extending the key schedule to have one round of the key schedule applied for each component algorithm's shared secret:



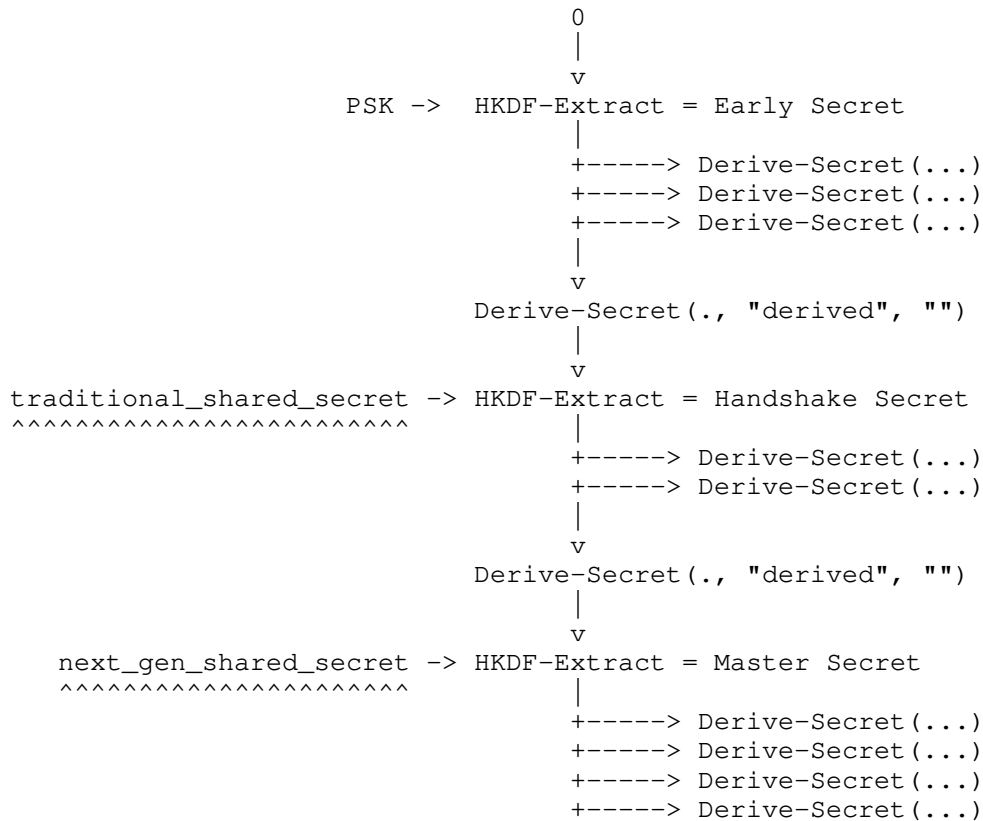
This is the approach used in [SCHANCK].

[BINDEL] analyzes the security of this approach as abstracted in their nested dual-PRF "N" combiner, showing a similar result as for the dualPRF combiner that it preserves IND-CPA (or IND-CCA) security.

Again their analysis depends on each ciphertext being input to the final PRF ("Derive-Secret") calls, which holds for TLS 1.3.

B.4.6. (Comb-AltInput) Second shared secret in an alternate KDF input

In the context of TLS 1.3, the next-generation shared secret is used in place of a currently unused input in the TLS 1.3 key schedule, namely replacing the "0" "IKM" input to the final "HKDF-Extract":



This approach is not taken in any of the known post-quantum/hybrid TLS drafts. However, it bears some similarities to the approach for using external PSKs in [EXTERN-PSK].

B.4.7. Benefits and Drawbacks

New logic. While (Comb-Concat) (Appendix B.4.1), (Comb-KDF-1) (Appendix B.4.2), and (Comb-KDF-2) (Appendix B.4.3) require new logic to compute the concatenated shared secret, this value can then be used by the TLS 1.3 key schedule without changes to the key schedule

logic. In contrast, (Comb-Chain) (Appendix B.4.5) requires the TLS 1.3 key schedule to be extended for each extra component algorithm.

Philosophical. The TLS 1.3 key schedule already applies a new stage for different types of keying material (PSK versus (EC)DHE), so (Comb-Chain) (Appendix B.4.5) continues that approach.

Efficiency. (Comb-KDF-1) (Appendix B.4.2), (Comb-KDF-2) (Appendix B.4.3), and (Comb-Chain) (Appendix B.4.5) increase the number of KDF applications for each component algorithm, whereas (Comb-Concat) (Appendix B.4.1) and (Comb-AltInput) (Appendix B.4.6) keep the number of KDF applications the same (though with potentially longer inputs).

Extensibility. (Comb-AltInput) (Appendix B.4.6) changes the use of an existing input, which might conflict with other future changes to the use of the input.

More than 2 component algorithms. The techniques in (Comb-Concat) (Appendix B.4.1) and (Comb-Chain) (Appendix B.4.5) can naturally accommodate more than 2 component shared secrets since there is no distinction to how each shared secret is treated. (Comb-AltInput) (Appendix B.4.6) would have to make some distinct, since the 2 component shared secrets are used in different ways; for example, the first shared secret is used as the "IKM" input in the 2nd "HKDF-Extract" call, and all subsequent shared secrets are concatenated to be used as the "IKM" input in the 3rd "HKDF-Extract" call.

Authors' Addresses

Douglas Stebila
University of Waterloo

Email: dstebila@uwaterloo.ca

Scott Fluhrer
Cisco Systems

Email: sfluhrer@cisco.com

Shay Gueron
University of Haifa and Amazon Web Services

Email: shay.gueron@gmail.com

Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: 26 August 2021

N. Sullivan
Cloudflare
H. Krawczyk
IBM Research
O. Friel
R. Barnes
Cisco
22 February 2021

OPAQUE with TLS 1.3
draft-sullivan-tls-opaque-01

Abstract

This document describes two mechanisms for enabling the use of the OPAQUE password-authenticated key exchange in TLS 1.3.

Status of This Memo

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

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

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

This Internet-Draft will expire on 26 August 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.

Table of Contents

1. Introduction	2
2. Conventions and Definitions	3
3. OPAQUE	3
4. Password Registration	4
5. Password Authentication	4
5.1. TLS Extensions	5
6. Use of extensions in TLS handshake flows	7
6.1. OPAQUE-KEX	7
6.2. OPAQUE-Sign	8
7. Integration into Exported Authenticators	9
8. Summary of properties	10
9. Privacy considerations	10
10. Security Considerations	10
11. IANA Considerations	11
12. References	11
12.1. Normative References	11
12.2. Informative References	11
Appendix A. Acknowledgments	12
Authors' Addresses	12

1. Introduction

Note that this draft has not received significant security review and should not be the basis for production systems.

OPAQUE [opaque-paper] is a mutual authentication method that enables the establishment of an authenticated cryptographic key between a client and server based on a user's password, without ever exposing the password to servers or other entities other than the client machine and without relying on a Public Key Infrastructure (PKI). OPAQUE leverages a primitive called a Strong symmetric Password Authenticated Key Exchange (Strong aPAKE) to provide desirable properties including resistance to pre-computation attacks in the event of a server compromise.

In some cases, it is desirable to combine password-based authentication with traditional PKI-based authentication as a defense-in-depth measure. For example, in the case of IoT devices, it may be useful to validate that both parties were issued a certificate from a certain manufacturer. Another desirable property for password-based authentication systems is the ability to hide the client's identity from the network. This document describes the use of OPAQUE in TLS 1.3 [TLS13] both as part of the TLS handshake and post-handshake facilitated by Exported Authenticators [I-D.ietf-tls-exported-authenticator], how the different approaches

satisfy the above properties and the trade-offs associated with each design.

The in-handshake instantiations of OPAQUE can be used to authenticate a TLS handshake with a password alone, or in conjunction with certificate-based (mutual) authentication but does not provide identity hiding for the client. The Exported Authenticators instantiation of OPAQUE provides client identity hiding by default and allows the application to do password authentication at any time during the connection, but requires PKI authentication for the initial handshake and application-layer semantics to be defined for transporting authentication messages.

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.

3. OPAQUE

OPAQUE [opaque-paper] is a Strong Asymmetric Password-Authenticated Key Exchange (Strong aPAKE) built on an oblivious pseudo-random function (OPRF) and authenticated key exchange protocol that is secure against key-compromise impersonation (KCI) attacks. Unlike previous PAKE methods such as SRP [RFC2945] and SPAKE-2 [I-D.irtf-cfrg-spake2], which require a public salt value, a Strong aPAKE leverages the OPRF private key as salt, making it resistant to pre-computation attacks on the password database stored on the server.

TLS 1.3 provides a KCI-secure key agreement algorithm suitable for use with OPAQUE. This document describes two instantiations of OPAQUE in TLS 1.3: one based on digital signatures, called OPAQUE-Sign, and one on Diffie-Hellman key agreement, called OPAQUE-KEX.

OPAQUE consists of two distinct phases: password registration and authentication. We will describe the mechanisms for password registration in this document but it is assumed to have been done outside of a TLS connection. During password registration, the client and server establish a shared set of parameters for future authentication and two private-public key pairs are generated, one for the client and one for the server. The server keeps its private key and stores an encapsulated copy of the client's key pair along with its own public key in an "envelope" that is encrypted with the result of the OPRF operation. Note that it is possible for the

server to use the same key for multiple clients. It may be necessary to permit multiple simultaneous server keys in the even of a key rollover. The client does not store any state nor any PKI information.

In OPAQUE-Sign, the key pairs generated at password registration time are digital signature keys. These signature keys are used in place of certificate keys for both server and client authentication in a TLS handshake. Client authentication is technically optional, though in practice is almost universally required. OPAQUE-Sign cannot be used alongside certificate-based handshake authentication. This instantiation can also be leveraged to do part of a post-handshake authentication using Exported Authenticators [I-D.ietf-tls-exported-authenticator] given an established TLS connection protected with certificate-based authentication.

In OPAQUE-KEX, the key pairs are Diffie-Hellman keys and are used to establish a shared secret that is fed into the key schedule for the handshake. The handshake continues to use Certificate-based authentication and establishes the shared key using Diffie-Hellman. This instantiations is best suited to use cases in which both password and certificate-based authentication are needed during the initial handshake, which is useful in some scenarios. There is no unilateral authentication in this context, mutual authentication is demonstrated explicitly through the finished messages.

4. Password Registration

Password registration is run between a client U and a server S. It is assumed that U can authenticate S during this registration phase (this is the only part in OPAQUE that requires some form of authenticated channel, either physical, out-of-band, PKI-based, etc.) During this phase, clients run the registration flow in [I-D.irtf-cfrg-opaque] using a specific OPAQUE configuration consisting of a tuple (OPRF, Hash, MHF, AKE). The specific AKE is not used during registration. It is only used during login.

During this phase, a specific OPAQUE configuration is chosen, which consists of a tuple (OPRF, Hash, MHF, AKE). See [I-D.irtf-cfrg-opaque] for details about configuration parameters. In this context, AKE is either OPAQUE-Sign or OPAQUE-KEX.

5. Password Authentication

Password authentication integrates TLS into OPAQUE in such a way that clients prove knowledge of a password to servers. In this section, we describe TLS extensions that support this integration for both OPAQUE-KEX and OPAQUE-Sign.

5.1. TLS Extensions

We define several TLS extensions to signal support for OPAQUE and transport the parameters. The extensions used here have a similar structure to those described in Usage of PAKE with TLS 1.3 [I-D.barnes-tls-pake]. The permitted messages that these extensions are allowed and the expected protocol flows are described below.

First, this document specifies extensions used to convey OPAQUE client and server messages, called "opaque_client_auth" and "opaque_server_auth" respectively.

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

The "opaque_client_auth" extension contains a "PAKEClientAuthExtension" struct and can only be included in the "CertificateRequest" and "Certificate" messages.

```
struct {  
    opaque identity<0..2^16-1>;  
} PAKEClientAuthExtension;
```

The "opaque_server_auth" extension contains a "PAKEServerAuthExtension" struct and can only be included in the "ClientHello", "EncryptedExtensions", "CertificateRequest" and "Certificate" messages, depending on the type.

```
struct {
    opaque idU<0..2^16-1>;
    CredentialRequest request;
} PAKEShareClient;

struct {
    opaque idS<0..2^16-1>;
    CredentialResponse response;
} PAKEShareServer;

struct {
    select (Handshake.msg_type) {
        ClientHello:
            PAKEShareClient client_shares<0..2^16-1>;
            OPAQUETYPE types<0..2^16-1>;
        EncryptedExtensions, Certificate:
            PAKEShareServer server_share;
            OPAQUETYPE type;
    }
} PAKEServerAuthExtension;
```

This document also defines the following set of types;

```
enum {
    OPAQUE-Sign(1),
    OPAQUE-KEX(2),
} OPAQUETYPE;
```

Servers use PAKEShareClient.idU to index the user's record on the server and create the PAKEShareServer.response. The types field indicates the set of supported auth types by the client. PAKEShareClient.request and PAKEShareServer.response, of type CredentialRequest and CredentialResponse, respectively, are defined in [I-D.irtf-cfrg-opaque].

This document also describes a new CertificateEntry structure that corresponds to an authentication via a signature derived using OPAQUE. This structure serves as a placeholder for the PAKEServerAuthExtension extension.

```
struct {
    select (certificate_type) {
        case OPAQUESign:
            /* Defined in this document */
            opaque null<0>

        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;
```

We request that IANA add an additional type to the "TLS Certificate Types" registry for this OPAQUESign.

Support for the OPAQUESign Certificate type for server authentication can be negotiated using the `server_certificate_type` [RFC7250] and the Certificate type for client authentication can be negotiated using the `client_certificate_type` extension [RFC7250].

Note that there needs to be a change to the `client_certificate_type` row in the IANA "TLS ExtensionType Values" table to allow `client_certificate_type` extension to be used as an extension to the `CertificateRequest` message.

6. Use of extensions in TLS handshake flows

6.1. OPAQUE-KEX

In this mode, OPAQUE private keys are used for key agreement algorithm and the result is fed into the TLS key schedule. Password validation is confirmed by the validation of the finished message. These modes can be used in conjunction with optional Certificate-based authentication.

It should be noted that since the identity of the client it is not encrypted as it is sent as an extension to the `ClientHello`. This may present a privacy problem unless a mechanism like Encrypted Client Hello [ECH] is created to protect it.

Upon receiving a `PAKEServerAuth` extension, the server checks to see if it has a matching record for this identity. If the record does not exist, the handshake is aborted with a "illegal_parameter" alert. If the record does exist, but the key type of the record does not

match any of the `supported_groups` sent in the `key_share` extension of the `ClientHello`, an HRR is sent containing the set of valid key types that it found records for.

Given a matching `key_share` and an identity with a matching `supported_group`, the server returns its `PAKEServerAuth` as an extension to its `EncryptedExtensions`. Both parties then derive a shared OPAQUE key as follows:

```
U computes
  K = H(g^y ^ PrivU || PubU ^ x || PubS ^ PrivU || IdU || IdS )
S computes
  K = H(g^x ^ PrivS || PubS ^ y || PubU ^ PrivS || IdU || IdS )
```

`IdU`, `IdS` represent the identities of user (sent as identity in `PAKEShareClient`) and server (`Certificate` message). `H` is the HKDF function agreed upon in the TLS handshake.

The result, `K`, is then added as an input to the Master Secret in place of the 0 value defined in TLS 1.3. Specifically,

```
0 -> HKDF-Extract = Master Secret
```

becomes

```
K -> HKDF-Extract = Master Secret
```

In this construction, the finished messages cannot be validated unless the OPAQUE computation was done correctly on both sides, authenticating both client and server.

6.2. OPAQUE-Sign

In this modes of operation, the OPAQUE private keys are used for digital signatures and are used to define a new `Certificate` type and `CertificateVerify` algorithm. Like the OPAQUE-KEX instantiations above, the identity of the client is sent in the clear in the client's first flight unless a mechanism like `Encrypted Client Hello` [ECH] is created to protect it.

Upon receiving a `PAKEServerAuth` extension, the server checks to see if it has a matching record for this identity. If the record does not exist, the handshake is aborted with a TBD error message. If the record does exist, but the key type of the record does not match any of the `supported_signatures` sent in the the `ClientHello`, the handshake must be aborted with a `"illegal_parameter"` error.

We define a new Certificate message type for an OPAQUE-Sign authenticated handshake.

```
enum {  
    X509(0),  
    RawPublicKey(2),  
    OPAQUE-Sign(3),  
    (255)  
} CertificateType;
```

Certificates of this type have CertificateEntry structs of the form:

```
struct {  
    Extension extensions<0..2^16-1>;  
} CertificateEntry;
```

Given a matching signature_scheme and an identity with a matching key type, the server returns a certificate message with type OPAQUE-Sign with PAKEServerAuth as an extension. The private key used in the CertificateVerify message is set to the private key used during account registration, and the client verifies it using the server public key contained in the client's envelope.

It is RECOMMENDED that the server includes a CertificateRequest message with a PAKEClientAuth and the identity originally sent in the PAKEServerAuth extension from the client hello. On receiving a CertificateRequest message with a PAKEClientAuth extension, the client returns a CertificateVerify message signed by PrivC which is validated by the server using PubC.

7. Integration into Exported Authenticators

Neither of the above mechanisms provides privacy for the user during the authentication phase, as the user id is sent in the clear. Additionally, OPAQUE-Sign has the drawback that it cannot be used in conjunction with certificate-based authentication.

It is possible to address both the privacy concerns and the requirement for certificate-based authentication by using OPAQUE-Sign in an Exported Authenticator [I-D.ietf-tls-exported-authenticator] flow, since exported authenticators are sent over a secure channel that is typically established with certificate-based authentication. Using Exported Authenticators for OPAQUE has the additional benefit that it can be triggered at any time after a TLS session has been established, which better fits modern web-based authentication mechanism.

The ClientHello contains PAKEServerAuth, PAKEClientAuth with empty identity values to indicate support for these mechanisms.

1. Client creates Authenticator Request with CR extension PAKEServerAuth.
2. Server creates Exported Authenticator with OPAQUE-Sign (PAKEServerAuth) and CertificateVerify (signed with the OPAQUE private key).

If the server would like to then establish mutual authentication, it can do the following:

1. Server creates Authenticator Request with CH extension PAKEClientAuth (identity)
2. Client creates Exported Authenticator with OPAQUE-Sign Certificate and CertificateVerify (signed with user private key derived from the envelope).

Support for Exported Authenticators is negotiated at the application layer.

8. Summary of properties

Variant \ Property	Identity hiding	Certificate auth	Server-only auth	Post-handshake auth	Minimum round trips
OPAQUE-Sign with EA	yes	yes	yes	yes	2-RTT
OPAQUE-Sign	no	no	yes	no	1-RTT
OPAQUE-KEX	no	no	no	no	1-RTT

Table 1

9. Privacy considerations

TBD: cleartext identity, etc

10. Security Considerations

TODO: protecting against user enumeration

11. IANA Considerations

- * Existing IANA references have not been updated yet to point to this document.

IANA is asked to register a new value in the "TLS Certificate Types" registry of Transport Layer Security (TLS) Extensions (TLS-Certificate-Types-Registry), as follows:

- * Value: 4 Description: OPAQUE Authentication Reference: This RFC

Correction request: The client_certificate_type row in the IANA TLS ExtensionType Values table to allow client_certificate_type extension to be used as an extension to the CertificateRequest message.

12. References

12.1. Normative References

- [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>>.
- [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>>.
- [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>>.
- [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>>.

12.2. Informative References

- [ECH] Rescorla, E., Oku, K., Sullivan, N., and C. A. Wood, "TLS Encrypted Client Hello", Work in Progress, Internet-Draft, draft-ietf-tls-esni-09, 16 December 2020, <<https://www.ietf.org/archive/id/draft-ietf-tls-esni-09.txt>>.
- [I-D.barnes-tls-pake] Barnes, R. and O. Friel, "Usage of PAKE with TLS 1.3", Work in Progress, Internet-Draft, draft-barnes-tls-pake-04, 16 July 2018, <<https://www.ietf.org/archive/id/draft-barnes-tls-pake-04.txt>>.
- [I-D.irtf-cfrg-opaque] Krawczyk, H., Lewi, K., and C. A. Wood, "The OPAQUE Asymmetric PAKE Protocol", Work in Progress, Internet-Draft, draft-irtf-cfrg-opaque-03, 21 February 2021, <<https://www.ietf.org/archive/id/draft-irtf-cfrg-opaque-03.txt>>.
- [I-D.irtf-cfrg-spake2] Ladd, W. and B. Kaduk, "SPAKE2, a PAKE", Work in Progress, Internet-Draft, draft-irtf-cfrg-spake2-18, 17 January 2021, <<https://www.ietf.org/archive/id/draft-irtf-cfrg-spake2-18.txt>>.
- [opaque-paper] Xu, J., "OPAQUE: An Asymmetric PAKE Protocol Secure Against Pre-Computation Attacks", 2018.
- [RFC2945] Wu, T., "The SRP Authentication and Key Exchange System", RFC 2945, DOI 10.17487/RFC2945, September 2000, <<https://www.rfc-editor.org/info/rfc2945>>.
- [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>>.

Appendix A. Acknowledgments

Authors' Addresses

Nick Sullivan
Cloudflare

Email: nick@cloudflare.com

Hugo Krawczyk
IBM Research

Email: hugo@ee.technion.ac.il

Owen Friel
Cisco

Email: ofriel@cisco.com

Richard Barnes
Cisco

Email: rlb@ipv.sx

tls
Internet-Draft
Intended status: Experimental
Expires: September 2, 2019

E. Sy
University of Hamburg
March 01, 2019

TLS Resumption across Server Name Indications for TLS 1.3
draft-sy-tls-resumption-group-00

Abstract

This document defines a mechanism for resuming a TLS 1.3 session across different Server Name Indications.

Status of This Memo

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

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

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

This Internet-Draft will expire on September 2, 2019.

Copyright Notice

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

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

Table of Contents

1. Introduction	2
2. Conventions and Definitions	3
3. Overview on Resumptions across SNI values	3
4. The "resumption_group" Extension	3
4.1. Client Behavior	3
4.2. Server Behavior	4
5. Expectations on Certificates	5
6. Compatibility Issues with Middleboxes	5
7. Security Considerations	5
8. IANA Considerations	5
9. References	5
9.1. Normative References	5
9.2. Informative References	6
Acknowledgments	6
Author's Address	6

1. Introduction

Most web transactions are short transfers that are significantly delayed by the TLS connection establishment. To accelerate the connection establishment, TLS 1.3 [RFC8446] and its predecessors provide session resumption mechanisms. They abbreviate the TLS handshake based on a shared secret exchanged during a prior TLS session between client and server. In total, these resumption handshakes significantly reduce computational overhead for cryptographic operations and save up to one round-trip compared to the full TLS connection establishment.

TLS 1.3 [RFC8446] allows resumption handshakes across Server Name Indications (SNIs) when they share the same TLS certificate. However, TLS 1.3 recommends not to use TLS resumptions across SNIs to avoid loosing a single-use ticket in case of a failed resumption attempt. This practice requires costly full TLS connection establishments in situations where a performance-optimized resumption handshake across SNI values would be possible. To illustrate this performance limitation, we describe the common situation of a redirected web request. We assume that the hostname example.com redirects to www.example.com and both hostnames are operated by the same entity and use the same certificate for their authentication. A client requesting www.example.com via this redirect requires two full TLS handshakes following the recommendation of TLS 1.3 [RFC8446]. Using resumption across SNI values, the later full handshake can be converted to a performance-optimized resumed handshake. A comprehensive study of the performance benefits of resumptions across SNI values for popular websites can be found in [PERF].

This document defines a mechanism to inform the client in between which SNI values TLS resumptions are supported. This information enables the client to use resumption across SNI values only in situations where the chance of a successful resumption handshake is high.

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.

3. Overview on Resumptions across SNI values

When a client wants to form a TLS connection to a server, it indicates support for the "resumption_group" extension in the ClientHello message. To signal its support for this extension type, the server returns the "resumption_group" extension with an empty data field.

The client is now aware, that all SNI values for which the presented server certificate is valid, form a TLS resumption group. Thus, resumption tickets issued by a group member are designated to be used to establish resumed connections to any member of the same group.

4. The "resumption_group" Extension

This extension carries no data as defined in the following ResumptionGroup structure:

```
struct {  
} ResumptionGroup;
```

4.1. Client Behavior

To indicate support for the "resumption_group" extension, the client sends this extension type within the initial ClientHello message to the server.

Upon receiving the server's response, the client checks whether the "resumption_group" extension is present in the extension list of the server's CertificateEntry (see Section 4.2.2 of [RFC6066]).

If this extension type is not included in the response of the server, then the client reasons that the server is not configured to support

the "resumption_group" extension and proceeds with a normal handshake.

Otherwise, the client proceeds with a normal connection establishment and associates all retrieved resumption tickets to the corresponding resumption group. This resumption group is formed of all SNI values that are valid for the presented server certificate.

To establish a resumed connection to any SNI value included in a resumption group, the client uses a resumption ticket associated to the same group. The Client Hello of a resumed handshake MUST NOT include the "resumption_group" extension.

Tickets received during a resumed connection MUST be associated to the same resumption group of the ticket that was used during the establishment of this connection.

If a SNI value is a member of multiple resumption groups, then the client is recommended to use the freshest valid ticket for a resumption handshake. It is assumed, that fresher resumption tickets are more likely to be accepted by the respective server.

According to [RFC8446], clients MUST NOT cache tickets longer than seven days.

Note, that TLS resumption enables a server to link resumed connections to the same client. A study on the feasibility of this tracking mechanism can be found in [TRAC]. To protect the client's privacy against tracking via this mechanism, it is RECOMMENDED to cache resumption tickets only for ten minutes.

4.2. Server Behavior

Upon receiving an initial Client Hello message, the server validates if the client provided an extension of the type "resumption_group".

If the "resumption_group" extension is not listed by the client, then the server's response MUST NOT include an entry for this extension type. Otherwise, the server includes the "resumption_group" extension in the extension list of the server's CertificateEntry, to signal support for resumptions across SNI values. Subsequently, the server proceeds with a normal handshake.

This extension type does not affect the server behavior for resumed connection establishments.

5. Expectations on Certificates

This "resumption_group" extension forms the resumption group based on the SNI values that are valid for the server's certificate. To optimize the performance benefit of this extension, the server's certificate is RECOMMENDED to only include SNI values that mutually support the resumption of their TLS connections. Otherwise, the client's resumption attempt across SNI values will fail if the server does not support this practice. Note, that each failed resumption handshake uses up a single-use resumption ticket. As a result, these failed attempts might use up all cached single-use tickets, which hinders the client to establish performance-optimized resumption handshakes to legitimate SNI values.

6. Compatibility Issues with Middleboxes

[RFC8446]; Section 9.3 requires MITM proxies to remove any extensions they do not understand. If a conformant MITM proxy does not support this extension, it will remove this extension type from the Client Hello. As a result, the server reacts as if it is not supporting this extension type.

7. Security Considerations

Clients MUST only resume to a new SNI value if this SNI value is valid for the server certificate presented in the original connection. To facilitate a correct implementation of this requirement, the resumption group is identical to the list of SNI values valid for a specific server certificate. Note, that the security of TLS resumptions across different SNI values is also discussed in Section 4.6.1 of [RFC8446].

8. IANA Considerations

TODO IANA needs to be requested to create an entry, resumption_group, in the existing registry for ExtensionType (defined in [RFC8446]), with "TLS 1.3" column values being set to "CH, EE", and "Recommended" column being set to "Yes".

9. References

9.1. 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>>.

9.2. Informative References

- [PERF] Sy, E., Moennich, M., Mueller, T., Federrath, H., and M. Fischer, "Enhanced Performance for the encrypted Web through TLS Resumption across Hostnames", 2019, <<https://arxiv.org/pdf/1902.02531.pdf>>.
- [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>>.
- [TRAC] Sy, E., Burkert, C., Federrath, H., and M. Fischer, "Tracking Users across the Web via TLS Session Resumption", 2018, <<https://arxiv.org/pdf/1810.07304.pdf>>.

Acknowledgments

Tobias Mueller and Christian Burkert provided ideas for this document.

Author's Address

Erik Sy
University of Hamburg

Email: tls@erik-sy.de

TLS
Internet-Draft
Intended status: Standards Track
Expires: January 14, 2021

H. Tschofenig
M. Brossard
Arm Limited
July 13, 2020

Using CBOR Web Tokens (CWTs) in Transport Layer Security (TLS) and
Datagram Transport Layer Security (DTLS)
draft-tschofenig-tls-cwt-02

Abstract

The TLS protocol supports different credentials, including pre-shared keys, raw public keys, and X.509 certificates. For use with public key cryptography developers have to decide between raw public keys, which require out-of-band agreement and full-fledged X.509 certificates. For devices where the reduction of code size is important it is desirable to minimize the use of X.509-related libraries. With the CBOR Web Token (CWT) a structure has been defined that allows CBOR-encoded claims to be protected with CBOR Object Signing and Encryption (COSE).

This document registers a new value to the "TLS Certificate Types" sub-registry to allow TLS and DTLS to use CWTs. Conceptually, CWTs can be seen as a certificate format (when with public key cryptography) or a Kerberos ticket (when used with symmetric key cryptography).

Status of This Memo

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

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

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

This Internet-Draft will expire on January 14, 2021.

Copyright Notice

Copyright (c) 2020 IETF Trust and the persons identified as the document authors. All rights reserved.

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

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

Table of Contents

1. Introduction	2
2. Conventions and Terminology	3
3. The CWT Certificate Type	3
4. Representation and Verification the Identity of Application Services	4
5. Security and Privacy Considerations	5
6. IANA Considerations	5
7. References	5
7.1. Normative References	5
7.2. Informative References	6
Appendix A. History	8
Appendix B. Working Group Information	8
Authors' Addresses	8

1. Introduction

The CBOR Web Token (CWT) [RFC8392] was defined as the CBOR-based version of the JSON Web Token (JWT) [RFC7519]. JWT is used extensively on Web application and for use with Internet of Things

environments the believe is that a more lightweight encoding, namely CBOR, is needed. CWTs, like JWTs, contain claims and those claims are protected against modifications using COSE [RFC8152]. CWTs are flexible with regard to the use of cryptography and hence CWTs may be protected using a keyed message digest, or a digital signature. One of the claims allows keys to be included, as described in [I-D.ietf-ace-cwt-proof-of-possession]. This specification makes use of these proof-of-possession claims in CWTs. This document mandates a minimum number of claims to be present in a CWT. There may, however, be a number of additional claims present in a CWT. An example of a token with a larger number of claims is the Entity Attestation Token (EAT), which may also be encoded as a CWT.

Fundamentally, there are two types of keys that can be used with CWTs:

- Asymmetric keys: In this case a CWT contains a COSE_Key [RFC8152] representing an asymmetric public key. To protect the CWT against modifications the CWT also needs to be digitally signed.
- Symmetric keys: In this case a CWT contains the Encrypted_COSE_Key [RFC8152] structure representing a symmetric key encrypted to a key known to the recipient using COSE_Encrypt or COSE_Encrypt0. Again, to protect the CWT against modifications a keyed message digest is used.

The CWT also allows mixing symmetric and asymmetric crypto although this is less likely to be used in practice.

Exchanging CWTs in the TLS / DTLS handshake offers an alternative to the use of raw public keys and X.509 certificates. Compared to raw public keys, CWTs allow more information to be included via the use of claims. Compared to X.509 certificates CBOR offers an alternative encoding format, which may also be used by the application layer thereby potentially reducing the overall code size requirements.

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 RFC 2119 [RFC2119].

3. The CWT Certificate Type

This document defines a new value to the "TLS Certificate Types" sub-registry and the value is defined as follows.

```
/* Managed by IANA */
enum {
    X509(0),
    RawPublicKey(2),
    CWT(TBD),
    (255)
} CertificateType;

struct {
    select (certificate_type) {

        /* CWT "certificate type" defined in this document.*/
        case CWT:
            opaque cwt_data<1..2^24-1>;

        /* RawPublicKey defined in RFC 7250*/
        case RawPublicKey:
            opaque ASN.1_subjectPublicKeyInfo<1..2^24-1>;

        /* X.509 certificate defined in RFC 5246*/
        case X.509:
            opaque cert_data<1..2^24-1>;

    };

    Extension extensions<0..2^16-1>;
} CertificateEntry;
```

4. Representation and Verification the Identity of Application Services

RFC 6125 [RFC6125] provides guidance for matching identifiers used in X.509 certificates against a reference identifier, i.e. an identifier constructed from a source domain and optionally an application service type. Different types of identifiers have been defined over time, such as CN-IDs, DNS-IDs, SRV-IDs, and URI-IDs, and they may be carried in different fields inside the X.509 certificate, such as in the Common Name or in the subjectAltName extension.

For CWTs issued to servers the following rule applies: To claim conformance with this specification an implementation MUST populate the Subject claim with the value of the Server Name Indication (SNI) extension. The Subject claim is of type StringOrURI. If it is string an equality match is used between the Subject claim value and the SNI. If the value contains a URI then the URI schema must be matched against the service being requested and the remaining part of the URI is matched against the SNI in an equality match (since the SNI only defines Hostname types).

For CWTs issued to clients the application service interacting with the TLS/DTLS stack on the server side is responsible for authenticating the client. No specific rules apply but the Subject and the Audience claims are likely to be good candidates for authorization policy checks.

Note: Verification of the Not Before and the Expiration Time claims MUST be performed to determine the validity of the received CWT.

5. Security and Privacy Considerations

The security and privacy characteristics of this extension are best described in relationship to certificates (when asymmetric keys are used) and to Kerberos tickets (when symmetric keys are used) since the main difference is in the encoding.

When creating proof-of-possession keys the recommendations for state-of-the-art key sizes and algorithms have to be followed. For TLS/DTLS those algorithm recommendations can be found in [RFC7925] and [RFC7525].

CWTs without proof-of-possession keys MUST NOT be used.

When CWTs are used with TLS 1.3 [RFC8446] and DTLS 1.3 [I-D.ietf-tls-dtls13] additional privacy properties are provided since most handshake messages are encrypted.

6. IANA Considerations

IANA is requested to add a new value to the "TLS Certificate Types" sub-registry for CWTs.

7. References

7.1. Normative References

[I-D.ietf-ace-cwt-proof-of-possession]

Jones, M., Seitz, L., Selander, G., Erdtman, S., and H. Tschofenig, "Proof-of-Possession Key Semantics for CBOR Web Tokens (CWTs)", draft-ietf-ace-cwt-proof-of-possession-11 (work in progress), October 2019.

[I-D.ietf-tls-dtls13]

Rescorla, E., Tschofenig, H., and N. Modadugu, "The Datagram Transport Layer Security (DTLS) Protocol Version 1.3", draft-ietf-tls-dtls13-33 (work in progress), October 2019.

- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <<https://www.rfc-editor.org/info/rfc2119>>.
- [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>>.
- [RFC8152] Schaad, J., "CBOR Object Signing and Encryption (COSE)", RFC 8152, DOI 10.17487/RFC8152, July 2017, <<https://www.rfc-editor.org/info/rfc8152>>.
- [RFC8392] Jones, M., Wahlstroem, E., Erdtman, S., and H. Tschofenig, "CBOR Web Token (CWT)", RFC 8392, DOI 10.17487/RFC8392, May 2018, <<https://www.rfc-editor.org/info/rfc8392>>.
- [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>>.

7.2. Informative References

- [RFC6125] Saint-Andre, P. and J. Hodges, "Representation and Verification of Domain-Based Application Service Identity within Internet Public Key Infrastructure Using X.509 (PKIX) Certificates in the Context of Transport Layer Security (TLS)", RFC 6125, DOI 10.17487/RFC6125, March 2011, <<https://www.rfc-editor.org/info/rfc6125>>.
- [RFC7519] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Token (JWT)", RFC 7519, DOI 10.17487/RFC7519, May 2015, <<https://www.rfc-editor.org/info/rfc7519>>.
- [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>>.
- [RFC7925] Tschofenig, H., Ed. and T. Fossati, "Transport Layer Security (TLS) / Datagram Transport Layer Security (DTLS) Profiles for the Internet of Things", RFC 7925, DOI 10.17487/RFC7925, July 2016, <<https://www.rfc-editor.org/info/rfc7925>>.

7.3. URIs

[1] <mailto:tls@ietf.org>

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

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

Appendix A. History

RFC EDITOR: PLEASE REMOVE THE THIS SECTION

- -01: Minor editorial bugfixes and reference updates; refreshing draft
- -00: Initial version

Appendix B. Working Group Information

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

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

Authors' Addresses

Hannes Tschofenig
Arm Limited

Email: hannes.tschofenig@arm.com

Mathias Brossard
Arm Limited

Email: Mathias.Brossard@arm.com

Internet Engineering Task Force
Internet-Draft
Intended status: Standards Track
Expires: October 13, 2021

H. Wang, Ed.
Y. Yang
X. Kang
Huawei International Pte. Ltd.
Z. Cheng
Shenzhen Olym Info. Security Tech. Ltd.
M. Chen
China Mobile
April 11, 2021

Using Identity as Raw Public Key in Transport Layer Security (TLS) and
Datagram Transport Layer Security (DTLS)
draft-wang-tls-raw-public-key-with-ibc-14

Abstract

This document specifies the use of identity as a raw public key in Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS). The TLS protocol procedures are kept unchanged, but signature algorithms are extended to support Identity-based signature (IBS). A few Identity-based signature algorithms from different standard organizations are supported in the current version.

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 October 13, 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.

Table of Contents

1. Introduction	2
2. Terms	4
3. Extension of RAW Public Key to IBC-based Public Key	4
4. New Signature Algorithms for IBS	9
5. Identity Format and Key Revocation	10
6. TLS Client and Server Handshake Behavior	11
7. Examples	13
7.1. TLS Client and Server Use IBS algorithm	13
7.2. Combined Usage of Raw Public Keys and X.509 Certificates	14
8. Security Considerations	16
9. IANA Considerations	17
10. Acknowledgements	17
11. References	17
11.1. Normative References	17
11.2. Informative References	18
Authors' Addresses	19

1. Introduction

DISCLAIMER: This is a personal draft and a limited security analysis is provided.

Traditionally, TLS client and server exchange public keys endorsed by PKIX [PKIX] certificates. It is considered complicated and may cause security weaknesses with the use of PKIX certificates [Defeating-SSL]. To simplify certificates exchange, using RAW public key with TLS/DTLS has been specified in [RFC 7250] and has been included in the TLS 1.3 [RFC 8446]. Instead of transmitting a full certificate or a certificate chain in the TLS messages, only public keys are exchanged between client and server. However, using RAW public key requires out-of-band mechanisms to verify the purported public key to the claimed entity.

Recently, 3GPP has adopted the EAP as a unified authentication for 5G and included both EAP-APA' and EAP-TLS as authentication methods. It is specified in the 5G specification that EAP-TLS can be used for private networks, especially for networks with a large number of IoT devices [TS33.501]. For IoT networks, EAP-TLS with RAW public key is

particularly attractive, but binding identities with public keys might be challenging. The cost to maintain a large table for identity and public key mapping at server side incurs additional cost, e.g. devices have to pre-register to the server.

To simplify the binding between the public key and the entity, a better way could be using Identity-Based Cryptography(IBC), such as ECCSI public key specified in [RFC 6507], for authentication. Different from X.509 certificates and existing raw public keys, a public key in IBC takes the form of the entity's identity. This eliminates the necessity of binding between a public key and the entity presenting the public key.

The concept of IBC was first proposed by Adi Shamir in 1984. As a special class of public key cryptography, IBC uses a user's identity as public key, avoiding the hassle of public key certification in public key cryptosystems. IBC broadly includes IBE (Identity-based Encryption) and IBS (Identity-based Signature). For an IBC system to work, there exists a trusted third party, private key generator (PKG), which is responsible for issuing private keys to the users. A PKG has in possession a pair of Master Public Key and Master Secret Key. A private key is generated based on the user's identity by using the Master Secret key, while the Master Public key is used together with the user's identities for encryption (in case of IBE) and signature verification (in case of IBS). Another name of PKG is Key Management System (KMS), which is also used in some IBC system. In this document, the terms of PKG and KMS are interchangeable.

A number of IBE and IBS algorithms have been standardized by different standardization bodies, such as IETF, IEEE, ISO, etc. For example, IETF has specified several RFCs such as [RFC 5091], [RFC 6507] and [RFC6508] for both IBE and IBS algorithms. ISO and IEEE also have a few standards on IBC algorithms, such as IBS1, IBS2, and ChineseIBS [ISO_IEC-IBS].

RFC 7250 has specified the use of raw public key with TLS/DTLS handshake. However, supporting of IBS algorithms has not been included therein. Since IBS algorithms eliminate the binding between public keys and identities, this further simplifies the using of raw public key with TLS. Therefore, in this document, an amendment is added for supporting IBS algorithms when using raw public key.

With IBS algorithms, a PKG generates private keys for entities based on identities from requestors. Global parameters such as PKG's Master Public Key (MPK) are provisioned to both client and server. These parameters are not user specific, but PKG specific.

For a client, PKG specific parameters can be provisioned at the time PKG provisions the private key to the client. For the server, how to get the PKG specific parameters provisioned is out of the scope of this document, and it is deployment dependent.

The document is organized as follows: Section 2 defines the terms used in this document, Section 3 defines the data structure required when identity is used as raw public key. Section 4 specifies the cipher suites required to support IBS algorithm over TLS/DTLS. Section 5 explains how client and server authenticate each other when using identity as raw public key. Section 6 gives examples for using identity as raw public key over TLS/DTLS handshake procedure. Section 7 discusses the security considerations.

2. Terms

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.

3. Extension of RAW Public Key to IBC-based Public Key

To support the negotiation when using raw public between client and server, a new certificate structure is defined in RFC 7250. It is used by the client and server in the hello message exchange to indicate the types of certificates supported by each side.

When RawPublicKey type is selected for authentication, a data structure, subjectPublicKeyInfo, is used to carry the raw public key and its cryptographic algorithm. Within the subjectPublicKeyInfo structure, two fields, algorithm and subjectPublicKey, are defined. The algorithm is a data structure that specifies the cryptographic algorithm used with raw public key, which is represented by an object Identifiers (OID); and the parameters field provides necessary parameters associated with the algorithm. The subjectPublicKey field within the subjectPublicKeyInfo carries the raw public itself.

```

subjectPublicKeyInfo ::= SEQUENCE {
    algorithm          AlgorithmIdentifier,
    subjectPublicKey    BIT STRING
}

AlgorithmIdentifier ::= SEQUENCE {
    algorithm          OBJECT IDENTIFIER,
    parameters        ANY DEFINED BY algorithm OPTIONAL
}

```

Figure 1: SubjectPublicKeyInfo ASN.1 Structure

With IBS algorithm, identity is used as the raw public key, which can be converted to an BIT string and put into the subjectPublicKey field. The algorithm field in AlgorithmIdentifier structure is the object identifier of the IBS algorithm used. Specifically, for the ECCSI signature algorithm supported in this draft, the OBJECT IDENTIFIER is described with following data structure:

```

sa-eccsiWithSHA256 SIGNATURE-ALGORITHM ::= {
    IDENTIFIER id-alg-eccsi-with-sha256
    VALUE ECCSI-Sig-Value PARAMS TYPE NULL ARE absent
    HASHES { mda-sha256 }
    SMIME-CAPS { IDENTIFIED BY id-alg-eccsi-with-sha256 }
}

```

Figure 2: ECCSI Signature Algorithm ANSI.1 Structure

Beside OID, it is necessary to tell the peer the set of global parameters used by the signer. The information can be carried in the payload of the parameters field in AlgorithmIdentifier. On the other hand, when IBS algorithm is used for authentication, normally the global parameters in use are known to client and server, hence, instead of transmitting a full set of PKG public parameters, a hash value of them is transmitted, which is put in the parameters field of AlgorithmIdentifier data structure.

The data structure used to carry the hash value of public parameters is defined as follows:

```

IBSPublicParametersHash ::= SEQUENCE {
    HASHES { mda-sha256 }
}

```

Figure 3: IBS Global Parameters Hash ANSI.1 Structure

The hash value of the global parameters is generated by taking in the DER encoded PKG public parameters of each individual IBS algorithms as input. The data structure for each IBS algorithms supported in this draft are defined in the following.

For the ECCSI IBS signature algorithms, its PKG public parameters is specified in following Figure :

```

ECCSIPublicParameters ::= SEQUENCE {
    version    INTEGER { v2(2) },
    curve      OBJECT IDENTIFIER,
    hashfcn    OBJECT IDENTIFIER,
    pointP     FpPOINT,
    pointPpub  FpPOINT
}

FpPoint ::= SEQUENCE {
    x INTEGER,
    y INTEGER
}

```

Figure 4: ECCSI Global Parameters ANSI.1 Structure

The structure to carry the ISO-IBS1/ISO-IBS2 PKG public parameters are the same and is specified in following Figure :

```

ISOIBSPublicParameters ::= SEQUENCE {
    version    INTEGER { v3(3) },
    curve      OBJECT IDENTIFIER,
    hashfcn    OBJECT IDENTIFIER,
    pairing    PAIRING OPTIONAL,
    p          INTEGER OPTIONAL,
    q          INTEGER OPTIONAL,
    pointP     FpPoint,
    pointPpub  FpPoint
}

PAIRING ::= ENUMERATED{
    weil (1)  --Weil pairing
    tate (2)  --Tate pairing
    optimalAte (3) --Optimal Ate pairing
}

```

Figure 5: ISO-IBS1/IBS2 Global Parameters ANSI.1 Structure

The structure to carry the ISO-SM9 PKG public parameters is specified in following Figure :

```

SM9PublicParameters ::= SEQUENCE {
    version    INTEGER { v3(3) },
    curve      OBJECT IDENTIFIER,
    hashfcn    OBJECT IDENTIFIER,
    pairing    PAIRING OPTIONAL,
    p          INTEGER OPTIONAL,
    q          INTEGER OPTIONAL,
    pointP2    FpxPoint,
    pointP2pub FpxPoint,
    v          FpxElement
}

FpxPoint ::= CHOICE{
    fpPoint FpPoint,
    fp2Point [2] EXPLICIT Fp2Point,
}

Fp2Point ::= SEQUENCE{
    x Fp2Element,
    y Fp2Element
}

Fp2Element ::= SEQUENCE{
    a INTEGER,
    b INTEGER
}

FpxElement ::= CHOICE{
    fp2Elemt Fp2Element,
    fp12Elemt Fp12Element,
}

Fp12Element ::= SEQUENCE{
    a Fp6Element,
    b Fp6Element
}

Fp6Element ::= SEQUENCE{
    a Fp2Element,
    b Fp2Element,
    c Fp2Element
}

```

Figure 6: ISO-ChineseIBS Global Parameters ANSI.1 Structure

For ECCSIPublicParameters data structure, pointP shall be G in RFC 6507 and pointPpub shall be KPAK in RFC 6507. For

ISOIBSPublicParameters data structure, pointP and pointPpub shall be the same as defined in RFC 5091, and the pairing field shall be weil (1) or tate (2). The pairing field in SM9PublicParameters should be optimalAte (3) and the choice of v should be determined by the curve identifier. For example, for supersingular curves [RFC 5901], v shall be of type Fp2Element and for BN curves or BLS12-curves [FST10], v shall be of type Fp12Element.

To support IBS algorithm over TLS protocol, a data structure for signature value need to be defined.

Data structure for ECCSI is defined as follows(based RFC 6507):

```
ECCSI-Sig-Value ::= SEQUENCE {
    r INTEGER,
    s INTEGER,
    PVT OCTET STRING
}
```

Figure 7: ECCSI Signature Value ANSI.1 Structure

where PVT (as defined in RFC 6507) is encoded as 0x04 || x-coordinate of [v]G || y-coordinate of [v]G.

Data structure for ISO-IBS1 is defined as follows:

```
ISO-IBS1-Sig-Value ::= SEQUENCE {
    r INTEGER,
    s ECPoint
}
```

Figure 8: ISO-IBS1 Signature Value ANSI.1 Structure

Data structure for ISO-IBS2 is defined as follows:

```
ISO-IBS2-Sig-Value ::= SEQUENCE {
    r INTEGER,
    s ECPoint
}
```

Figure 9: ISO-IBS2 Signature Value ANSI.1 Structure

Data structure for ISO-ChineseIBS (SM9) is defined as follows:

```

SM9-Sig-Value ::= SEQUENCE {
    r INTEGER,
    s ECPPoint
}

```

Figure 10: ISO-ChineseIBS Signature Value ANSI.1 Structure

The definition of ECPPoint can be found in section 2.2 of RFC 5480.

To use a signature algorithm with TLS, OID for the signature algorithm need be provided. For ECCSI algorithm, an OID has been assigned by IANA recently. The following table shows the basic information needed for the ECCSI signature algorithm to be used for TLS.

Key Type	Document	OID
ISO/IEC 14888-3 IBS-1	ISO/IEC 14888-3: IBS-1 mechanism	1.0.14888.3.0.7
ISO/IEC 14888-3 IBS-2	ISO/IEC 14888-3: IBS-2 mechanism	1.0.14888.3.0.8
ISO/IEC 14888-3 ChineseIBS (SM9)	ISO/IEC 14888-3: ChineseIBS mechanism	1.2.156.10197.1.302.1
Elliptic Curve-Based Signatureless For Identity-based Encryption (ECCSI)	Section 5.2 in RFC 6507	1.3.6.1.5.5.7.6.29

Table 1: Algorithm Object Identifiers

4. New Signature Algorithms for IBS

To using identity as raw public key, new signature algorithms corresponding to the IBS need to be defined. With TLS 1.3, the value for signature algorithm is defined in the SignatureScheme. This document specifies how to support IBS algorithm. As a result, the SignatureScheme data structure has to be amended by including the presented IBS algorithms.


```
enum {  
    ...  
  
    /* IBS ECCSI signature algorithm */  
    eccsi_sha256 (0x0704),  
    iso_ibs1 (0x0705),  
    iso_ibs2 (0x0706),  
    iso_chinese_ibs (0x0707),  
  
    /* Reserved Code Points */  
    private_use (0xFE00..0xFFFF),  
    (0xFFFF)  
} SignatureScheme;
```

Figure 11: Include IBS in KeyExchangeAlgorithm

Note: The signature algorithm of eccsi_sha256 is defined in RFC6507.

Note: Other IBS signature algorithms can be added in the future.

5. Identity Format and Key Revocation

With the raw public scheme proposed in TLS 1.3 [RFC 8446], the server maintains a whitelist to bind raw public key and identity. When a raw public key is revoked, then the server removes the binding record from the whitelist. On the other hand, when using IBS algorithms, it is not necessary to maintain a whitelist at the server's side. Instead, the server can simply maintain a blacklist, which is much shorter than the whitelist. However, if we simply use the identifier as a raw public key, the revocation list may keep on increasing with the time going on. Hence, to prevent the revocation list from increasing continuously, it is recommended to include a timestamp for the automatic expiration of key material. With the timestamp included in the identifier, i.e. the raw public key, server can remove the revoked raw public key from the revocation list when it is expired.

Based on the above analysis, it is necessary to include an expiration time in the identifiers for the purpose of public key management. Therefore, in this draft, we recommend both client and server take following format for the identifiers used for TLS session setup:

```
Identifier ::= SEQUENCE {  
    version INTEGER {v1 (1)},  
    identity String,  
    expiration UTCTime  
}
```

Figure 12: Identifier Format ANSI.1 Structure

Both the client and server should check the validity of the expiration field of the raw public key before verify the signature. If the expiration time is invalid, the client or the server should abort the handshake procedure.

The identities of client or server shall be unique within the domain managed by one PKG. There are many different identities domains such as email address, telephone number, Network Access Identifier (NAI), International Mobile Subscriber Identity (IMSI) etc. It is up to network operators' choice to determine which name domain the device and server take.

6. TLS Client and Server Handshake Behavior

When RAW public is used with IBS for TLS, signature and hash algorithms are negotiated during the handshake.

The handshake between the TLS client and server follows the procedures defined in [RFC 8446], but with the support of the new signature algorithms specific to the IBS algorithms. The high-level message exchange in the following figure shows the TLS handshake using raw public keys, where the `client_certificate_type` and `server_certificate_type` extensions added to the client and server hello messages (see Section 4 of [RFC 7250]).

```

client_hello,
+key_share
+signature_algorithms
client_certificate_type,
server_certificate_type  ->

                                <-  server_hello,
                                + key_share
                                {EncryptedExtensions}
                                {client_certificate_type}
                                {server_certificate_type}
                                {Certificate}
                                {CertificateVerify}
                                {CertificateRequest}
                                {Finished}
                                [Application Data]

{Certificate}
{CertificateVerify}
{Finished}          ----->
[Application Data] <-----> [Application Data]

```

Figure 13: Basic Raw Public Key TLS Exchange

The client hello message tells the server the types of certificate or raw public key supported by the client, and also the certificate types that the client expects to receive from the server. When raw public with IBS algorithm from the server is supported by the client, the client includes desired IBS signature algorithm in the client hello message based on the order of client preference.

After receiving the client hello message, the server determines the client and server certificate types for handshakes. When the selected certificate type is RAW public key and IBS is the chosen signature algorithm, the server uses the SubjectPublicKeyInfo structure to carry the raw public key, OID for IBS algorithm and public parameters or the hash value of public parameters. Assuming that ECCSI is selected, the ECCSIPublicParameters data structure is used to carry global public parameters. With this information, the client knows the signature algorithm and the public parameters that should be used to verify the signature. The signature value is in the CertificateVerify message and the format of signature value is specified by the selected IBS algorithm. The data structures for PKG public parameters and signature values have been specified in the previous section of this document.

When the sever specifies that RAW public key should be used by the client to authenticate with the server, the client_certificate_type in the server hello is set to RawPublicKey. Besides that, the server

also sends Certificate Request, indicating that client should use some specific signature and hash algorithms. When IBS is chosen as signature algorithm, the server need to indicate the required IBS signature algorithms in the signature_algorithm extension within the CertificateRequest.

After receiving the server hello, the client checks the CertificateRequest for signature algorithms. If the client wants to use an IBS algorithm for signature, then the signature algorithm it intended to use must be in the list of supported signature algorithms specified by the server. Assume the IBS algorithm supported by the client is in the list, then the client responds with the IBS signature algorithm and PKG information with SubjectPublicKeyInfo structure in the certificate structure and provide signatures in the certificate verify message. The format of signature in the CertificateVerify message should be specified by each individual signature algorithm.

The server verifies the signature based on the chosen IBS algorithm and the relevant PKG parameters specified by the client.

7. Examples

In the following, examples of handshake exchange using IBS algorithm under RawPublicKey are illustrated.

7.1. TLS Client and Server Use IBS algorithm

In this example, both the TLS client and server use ECCSI for authentication, and they are restricted in that they can only process ECCSI signature algorithm. As a result, the TLS client sets both the server_certificate_type and the client_certificate_type extensions to be raw public key; in addition, the client sets the signature algorithm in the client hello message to be eccsi_sha256.

When the TLS server receives the client hello, it processes the message. Since it has an ECCSI raw public key from the PKG, it indicates in (2) that it agrees to use ECCSI and provides an ECCSI key by placing the SubjectPublicKeyInfo structure into the Certificate payload back to the client (3), including the OID, the identity of the server, ServerID, which is the public key of the server also, and the hash value of PKG public parameters. The client_certificate_type in (4) indicates that the TLS server accepts raw public key. The TLS server demands client authentication, and therefore includes a certificate_request(5), which requires the client to use eccsi_sha256 for signature. A signature value based on the eccsi_sha256 algorithm is carried in the CertificateVerify (6). The client, which has an ECCSI key, returns its ECCSI public key in

the Certificate payload to the server (7), which includes an OID for the ECCSI signature algorithm, the PKGInfo for KMS parameters, and identity of the client, ClientID, which is the public key of client also. The client also includes a signature value, ECCSI-Sig-Value, in the CertificateVerify (8) message.

When client/server receives PKG public parameters from peer, it should decide whether these parameters are acceptable or not. An example way to make decision is that a whitelist of acceptable PKG public parameters are stored locally at client/server. They can simply make a decision based on the white list stored locally.

```

client_hello,
+key_share                                     //(1)
signature_algorithm = (eccsi_sha256)         //(1)
client_certificate_type=(RawPublicKey)        //(1)
server_certificate_type=(RawPublicKey)        //(1)
->
<- server_hello,
+ key_share
{ server_certificate_type = RawPublicKey} //(2)
{certificate=((1.3.6.1.5.5.7.6.29, hash
value of ECCSIPublicParameters),
serverID)}                                     //(3)
{client_certificate_type = RawPublicKey}       //(4)
{certificate_request = (eccsi_sha256)}         //(5)
{CertificateVerify = {ECCSI-Sig-Value}         //(6)
{Finishaed}

{Certificate=(
(1.3.6.1.5.5.7.6.29,
hash value of ECCSIPublicParameters),
ClientID)}                                     //(7)
{CertificateVerify = {ECCSI-Sig-Value}} //(8)
{Finished }
[Applicateion Data] ---->
[Application Data] <----> [Application Data]

```

Figure 14: Basic Raw Public Key TLS Exchange

7.2. Combined Usage of Raw Public Keys and X.509 Certificates

This example combines the uses of an ECCSI key and an X.509 certificate. The TLS client uses an ECCSI key for client authentication, and the TLS server provides an X.509 certificate for server authentication.

The exchange starts with the client indicating its ability to process a raw public key, or an X.509 certificate, if provided by the server. It prefers a raw public key with ECCSI signature algorithm since `eccsi_sha256` precedes the `ecdsa_secp256r1_sha256`. Furthermore, the client indicates that it has a ECCSI-based raw public key for client-side authentication. The client also indicates that it supports the server using either ECCSI or `ecdsa_secp256r1_sha256` for the certificate signature. This further indicates that the server can use `ecdsa_secp256r1_sha256` to sign the message.

With the received `client_hello`, the server chooses to provide its X.509 certificate in (3) and indicates that choice in (2). For client authentication, the server indicates in (4) that it has selected the raw public key format and requests an ECCSI certificate from the client in (4) and (5). The TLS client provides an ECCSI certificate in (6) and signature value after receiving and processing the TLS server hello message.

```
client_hello,
+key_share
signature_algorithms =(eccsi_sha256,
                        ecdsa_secp256r1_sha256)    //(1)
signature_algorithms_cert = (
eccsi_sha256, ecdsa_secp256r1_sha256)    //(1)
{client_certificate_type=
(RawPublicKey)}                          //(1)
{server_certificate_type=
(RawPublicKey, X.509)}                   //(1)
->
<-  server_hello,
    +key_share
    {server_certificate_type=X.509}          //(2)
    {Certificate = (x.509 certificate)}      //(3)
    {client_certificate_type = (RawPublicKey)}//(4)
    {CertificateRequest} = (eccsi_sha256)}  //(5)
    {CertificateVerify}
    {Finished}

certificate=(
(1.3.6.1.5.5.7.6.29,
ECCSIPublicParameters),
ClientID),                               //(6)
{CertificateVerify =
(ECCSI-Sig-Value)}                      //(7)
{ Finished }
[Applicateion Data] ---->
[Application Data] <---> [Application Data]
```

Figure 15: Basic Raw Public Key TLS Exchange

Handshake for other IBS algorithms can be completed similarly by including different data structures for public parameters and signature values respectively.

8. Security Considerations

Using IBS-based raw public key in TLS/DTLS does not change the message flows of TLS, hence, for the most part, the security considerations involved in using the Transport Layer Security protocol with raw public key also apply here. The additional security of the resulting protocol rests on the security of the used IBS algorithms.

IBS signature algorithm has been standardized for ten years and has been adopted in real applications. However, we would like to point out the differences between IBS signature algorithm and the existing raw public key based algorithms: the private key of IBS used for signature generation is generated by the PKG centre, while the private key for the existing raw public key algorithms can be generated locally. Therefore, IBS mechanism may face a security risk of private key disclosure due to improper management of KMS system. The entity using IBS with TLS protocol shall be aware the above risk and an enforced key management system shall be adopted by the organization.

When using IBS algorithm, key escrow is an concern as the private key of user or devices normally is generated by PKG. PKG in the system which could generate each device's private key. However, when IBS is used in TLS1.3, passive attacks to recover the session key is not possible. Actively man-in-the-middle attack by replacing exchanged DH tokens and signatures would certainly leave traces even transiently. Similarly, a PKG could impersonate an entity to conduct a TLS session, just as the KMS in the symmetric key solution, but forensic traces could be also collected in this situation. It would be hugely risky for a PKG, which would usually be a trusted party, to launch such attacks. If such an attack is caught in red-handed, no one would trust the PKG's service anymore.

Another worry of using IBS is about the compromising of PKG. The PKG could become operationally compromised and an attacker may obtain master secrets of a PKG. However, this security risk can be solved by protect the PKG with HSM, which is often used by CA to protect the root signing key.

Private key compromising is one security risk that need to be considered when using public key technology. When using raw public key with IBS algorithm, as we have suggested in this document, a revocation list shall be maintained at the server side. At the same

time, a timestamp shall be included in the public key to prevent the revocation list from keeping on increasing. With the revocation list, the server can prevent following attacks:

- 1) when a device use a revoked identifier for authentication, which has not expired yet, then the server can reject the TLS session by checking the revocation list maintained at the server-side. As it is on the list, then the server aborts the TLS handshake.
- 2) When a device using a identifier which has been expired, the server can simply verify the timestamp contained in the identifier and abort the handshake procedure immediately.
- 3) If the attacker changes the timestamp within the identifier, then it will cause signature verification error when the server verify the signature contained in the signature_verify from client.

9. IANA Considerations

IANA has assigned 4 code points from the TLS SignatureScheme registry for the four IBS algorithms used in this document. The code points are listed as follows:

- eccsi_sha256
- iso_ibs1
- iso_ibs2
- iso_chinese_ibs

For all of these entries the Recommended field should be N, and the Reference field should be this document.

10. Acknowledgements

11. References

11.1. Normative References

- [PKIX] "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List(CRL) Profile", June 2008.
- [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>>.

- [RFC2434] "Guidelines for Writing an IANA Consideration Section in RFCs", October 1998.
- [RFC5091] Boyen, X. and L. Martin, "Identity-Based Cryptography Standard (IBCS) #1: Supersingular Curve Implementations of the BF and BB1 Cryptosystems", RFC 5091, DOI 10.17487/RFC5091, December 2007, <<https://www.rfc-editor.org/info/rfc5091>>.
- [RFC5480] Turner, S., Brown, D., Yiu, K., Housley, R., and T. Polk, "Elliptic Curve Cryptography Subject Public Key Information", RFC 5480, DOI 10.17487/RFC5480, March 2009, <<https://www.rfc-editor.org/info/rfc5480>>.
- [RFC6507] Groves, M., "Elliptic Curve-Based Certificateless Signatures for Identity-Based Encryption (ECCSI)", RFC 6507, DOI 10.17487/RFC6507, February 2012, <<https://www.rfc-editor.org/info/rfc6507>>.
- [RFC6508] Groves, M., "Sakai-Kasahara Key Encryption (SAKKE)", RFC 6508, DOI 10.17487/RFC6508, February 2012, <<https://www.rfc-editor.org/info/rfc6508>>.
- [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>>.
- [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>>.
- [RFC8216] Pantos, R., Ed. and W. May, "HTTP Live Streaming", RFC 8216, DOI 10.17487/RFC8216, August 2017, <<https://www.rfc-editor.org/info/rfc8216>>.
- [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>>.

11.2. Informative References

- [Defeating-SSL] "New Tricks for Defeating SSL in Practice", Feb 2009, <<http://www.blackhat.com/presentations/bh-dc-09/Marlinspike/BlackHat-DC-09-Marlinspike-Defeating-SSL.pdf>>.

[FST10] "A Taxonomy of Pairing-Friendly Elliptic Curves. Journal of Cryptology. Volume 23, Issue 2, pp 224-280," Apr 2010.

[ISO_IEC-IBS]
"ISO/IEC 14888-3:2018 IT Security techniques - Digital signatures with appendix - Part 3: Discrete logarithm based mechanisms", Sept 2019.

[TS33.501]
"TS 33.501, Security architecture and procedures for 5G System, v.g.0.0", Sept 2019,
<https://www.3gpp.org/ftp/Specs/archive/33_series/33.501/>.

Authors' Addresses

Haiguang Wang (editor)
Huawei International Pte. Ltd.
9 North Buona Vista Dr, #13-01
Singapore 138589
SG

Phone: +65 6825 4200
Email: wang.haiguang1@huawei.com

Yanjiang Yang
Huawei International Pte. Ltd.
9 North Buona Vista Dr, #13-01
Singapore 138589
SG

Phone: +65 6825 4200
Email: yang.yanjiang@huawei.com

Xin Kang
Huawei International Pte. Ltd.
9 North Buona Vista Dr, #13-01
Singapore 138589
SG

Phone: +65 6825 4200
Email: xin.kang@huawei.com

Zhaohui Cheng
Shenzhen Olym Info. Security Tech. Ltd.
Futong Haowangjiao Podium Building
Shenzhen, Guang Dong Province 518101
CN

Phone: +86 755 8618 2108
Email: chenzh@myibc.net

Meiling Chen
China Mobile
32, Xuanwumen West
BeiJing, BeiJing 100053
China

Email: chenmeiling@chinamobile.com

tls
Internet-Draft
Intended status: Experimental
Expires: September 12, 2019

D. Benjamin
Google, LLC.
C. Wood
Apple, Inc.
March 11, 2019

Importing External PSKs for TLS
draft-wood-tls-external-psk-importer-01

Abstract

This document describes an interface for importing external PSK (Pre-Shared Key) into TLS.

Status of This Memo

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

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

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

This Internet-Draft will expire on September 12, 2019.

Copyright Notice

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

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

Table of Contents

1. Introduction	2
2. Conventions and Definitions	2
3. Overview	3
3.1. Terminology	3
4. Key Import	3
5. Deprecating Hash Functions	5
6. Backwards Compatibility	5
7. Security Considerations	5
8. Privacy Considerations	5
9. IANA Considerations	6
10. References	6
10.1. Normative References	6
10.2. Informative References	7
Appendix A. Acknowledgements	7
Authors' Addresses	7

1. Introduction

TLS 1.3 [RFC8446] supports pre-shared key (PSK) resumption, wherein PSKs can be established via session tickets from prior connections or externally via some out-of-band mechanism. The protocol mandates that each PSK only be used with a single hash function. This was done to simplify protocol analysis. TLS 1.2, in contrast, has no such requirement, as a PSK may be used with any hash algorithm and the TLS 1.2 PRF. This means that external PSKs could possibly be re-used in two different contexts with the same hash functions during key derivation. Moreover, it requires external PSKs to be provisioned for specific hash functions.

To mitigate these problems, external PSKs can be bound to a specific hash function when used in TLS 1.3, even if they are associated with a different KDF (and hash function) when provisioned. This document specifies an interface by which external PSKs may be imported for use in a TLS 1.3 connection to achieve this goal. In particular, it describes how KDF-bound PSKs can be differentiated by different hash algorithms to produce a set of candidate PSKs, each of which are bound to a specific hash function. This expands what would normally have been a single PSK identity into a set of PSK identities. However, it requires no change to the TLS 1.3 key schedule.

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.

3. Overview

Intuitively, key importers mirror the concept of key exporters in TLS in that they diversify a key based on some contextual information before use in a connection. In contrast to key exporters, wherein differentiation is done via an explicit label and context string, the key importer defined herein uses a label and set of hash algorithms to differentiate an external PSK into one or more PSKs for use.

Imported keys do not require negotiation for use, as a client and server will not agree upon identities if not imported correctly. Thus, importers induce no protocol changes with the exception of expanding the set of PSK identities sent on the wire.

3.1. Terminology

- o External PSK (EPSK): A PSK established or provisioned out-of-band, i.e., not from a TLS connection, which is a tuple of (Base Key, External Identity, KDF). The associated KDF (and hash function) may be undefined.
- o Base Key: The secret value of an EPSK.
- o External Identity: The identity of an EPSK.
- o Imported Identity: The identity of a PSK as sent on the wire.

4. Key Import

A key importer takes as input an EPSK with external identity 'external_identity' and base key 'epsk', as defined in Section 3.1, along with an optional label, and transforms it into a set of PSKs and imported identities for use in a connection based on supported HashAlgorithms. In particular, for each supported HashAlgorithm 'hash', the importer constructs an ImportedIdentity structure as follows:

```
struct {  
    opaque external_identity<1...2^16-1>;  
    opaque label<0..2^8-1>;  
    HashAlgorithm hash;  
} ImportedIdentity;
```

[[TODO: An alternative design might combine label and hash into the same field so that future protocols which don't have a notion of HashAlgorithm don't need this field.]]

ImportedIdentity.label MUST be bound to the protocol for which the key is imported. Thus, TLS 1.3 and QUICv1 [I-D.ietf-quic-transport] MUST use "tls13" as the label. Similarly, TLS 1.2 and all prior TLS versions should use "tls12" as ImportedIdentity.label, as well as SHA256 as ImportedIdentity.hash. Note that this means future versions of TLS will increase the number of PSKs derived from an external PSK.

A unique and imported PSK (IPSK) with base key 'ipskx' bound to this identity is then computed as follows:

```
epskx = HKDF-Extract(0, epsk)
ipskx = HKDF-Expand-Label(epskx, "derived psk",
                          Hash(ImportedIdentity), Hash.length)
```

[[TODO: The length of ipskx MUST match that of the corresponding and supported ciphersuites.]]

The hash function used for HKDF [RFC5869] is that which is associated with the external PSK. It is not bound to ImportedIdentity.hash. If no hash function is specified, SHA-256 MUST be used. Differentiating epsk by ImportedIdentity.hash ensures that each imported PSK is only used with at most one hash function, thus satisfying the requirements in [RFC8446]. Endpoints MUST import and derive an ipsk for each hash function used by each ciphersuite they support. For example, importing a key for TLS_AES_128_GCM_SHA256 and TLS_AES_256_GCM_SHA384 would yield two PSKs, one for SHA256 and another for SHA384. In contrast, if TLS_AES_128_GCM_SHA256 and TLS_CHACHA20_POLY1305_SHA256 are supported, only one derived key is necessary.

The resulting IPSK base key 'ipskx' is then used as the binder key in TLS 1.3 with identity ImportedIdentity. With knowledge of the supported hash functions, one may import PSKs before the start of a connection.

EPSKs may be imported for early data use if they are bound to protocol settings and configurations that would otherwise be required for early data with normal (ticket-based PSK) resumption. Minimally, that means ALPN, QUIC transport settings, etc., must be provisioned alongside these EPSKs.

5. Deprecating Hash Functions

If a client or server wish to deprecate a hash function and no longer use it for TLS 1.3, they may remove this hash function from the set of hashes used during while importing keys. This does not affect the KDF operation used to derive concrete PSKs.

6. Backwards Compatibility

Recall that TLS 1.2 permits computing the TLS PRF with any hash algorithm and PSK. Thus, an external PSK may be used with the same KDF (and underlying HMAC hash algorithm) as TLS 1.3 with importers. However, critically, the derived PSK will not be the same since the importer differentiates the PSK via the identity and hash function. Thus, PSKs imported for TLS 1.3 are distinct from those used in TLS 1.2, and thereby avoid cross-protocol collisions.

7. Security Considerations

This is a WIP draft and has not yet seen significant security analysis.

8. Privacy Considerations

DISCLAIMER: This section contains a sketch of a design for protecting external PSK identities. It is not meant to be implementable as written.

External PSK identities are typically static by design so that endpoints may use them to lookup keying material. For some systems and use cases, this identity may become a persistent tracking identifier. One mitigation to this problem is encryption. Future drafts may specify a way for encrypting PSK identities using a mechanism similar to that of the Encrypted SNI proposal [I-D.ietf-tls-esni]. Another approach is to replace the identity with an unpredictable or "obfuscated" value derived from the corresponding PSK. One such proposal, derived from a design outlined in [I-D.ietf-dnssd-privacy], is as follows. Let `ipskx` be the imported PSK with identity `ImportedIdentity`, and `N` be a unique nonce of length equal to that of `ImportedIdentity.hash`. With these values, construct the following "obfuscated" identity:

```
struct {
    opaque nonce[hash.length];
    opaque obfuscated_identity<1..2^16-1>;
    HashAlgorithm hash;
} ObfuscatedIdentity;
```


ObfuscatedIdentity.nonce carries N,
ObfuscatedIdentity.obfuscated_identity carries HMAC(ipskx, N), where
HMAC is computed with ImportedIdentity.hash, and
ObfuscatedIdentity.hash is ImportedIdentity.hash.

Upon receipt of such an obfuscated identity, a peer must lookup the
corresponding PSK by exhaustively trying to compute
ObfuscatedIdentity.obfuscated_identity using ObfuscatedIdentity.nonce
and each of its known imported PSKs. If N is chosen in a predictable
fashion, e.g., as a timestamp, it may be possible for peers to
precompute these obfuscated identities to ease the burden of trial
decryption.

9. IANA Considerations

This document makes no IANA requests.

10. References

10.1. Normative References

- [I-D.ietf-quic-transport]
Iyengar, J. and M. Thomson, "QUIC: A UDP-Based Multiplexed
and Secure Transport", draft-ietf-quic-transport-18 (work
in progress), January 2019.
- [RFC1035] Mockapetris, P., "Domain names - implementation and
specification", STD 13, RFC 1035, DOI 10.17487/RFC1035,
November 1987, <<https://www.rfc-editor.org/info/rfc1035>>.
- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate
Requirement Levels", BCP 14, RFC 2119,
DOI 10.17487/RFC2119, March 1997,
<<https://www.rfc-editor.org/info/rfc2119>>.
- [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>>.
- [RFC6234] Eastlake 3rd, D. and T. Hansen, "US Secure Hash Algorithms
(SHA and SHA-based HMAC and HKDF)", RFC 6234,
DOI 10.17487/RFC6234, May 2011,
<<https://www.rfc-editor.org/info/rfc6234>>.
- [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>>.

10.2. Informative References

[I-D.ietf-dnssd-privacy]
Huitema, C. and D. Kaiser, "Privacy Extensions for DNS-SD", draft-ietf-dnssd-privacy-05 (work in progress), October 2018.

[I-D.ietf-tls-esni]
Rescorla, E., Oku, K., Sullivan, N., and C. Wood, "Encrypted Server Name Indication for TLS 1.3", draft-ietf-tls-esni-03 (work in progress), March 2019.

Appendix A. Acknowledgements

The authors thank Eric Rescorla and Martin Thomson for discussions that led to the production of this document, as well as Christian Huitema for input regarding privacy considerations of external PSKs.

Authors' Addresses

David Benjamin
Google, LLC.

Email: davidben@google.com

Christopher A. Wood
Apple, Inc.

Email: cawood@apple.com