TLS Working Group
Internet-Draft
Updates: 5246, 6347 (if approved)
Intended status: Standards Track
Expires: September 4, 2018

T. Fossati
Nokia
N. Mavrogiannopoulos
RedHat

March 03, 2018

Record Header Extensions for DTLS
draft-fossati-tls-ext-header-01

Abstract

This document proposes a mechanism to extend the record header in DTLS. To that aim, the DTLS header is modified as follows: the length field is trimmed to 15 bits, and the length's top bit is given the "record header extension indicator" semantics, allowing a sender to signal that one or more record header extensions have been added to this record. We define the generic format of a record header extension and the general rules associated with its handling. Any details regarding syntax, semantics and negotiation of a specific record header extension, are left to future documents.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

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

2. Length Redefined

DTLS ([RFC6347], [I-D.ietf-tls-dtls13]) requires the size of record payloads to not exceed 2^14 bytes - plus a small amount that accounts for compression or AEAD expansion. This means that the first bit in the length field of the DTLS record header is, in fact, unused.

The proposal (Figure 1) is to shorten the length field to 15 bits and use the top bit (E) to signify the presence / absence of a record header extension.

```
    0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
   +-+-+-+-+-+-+-+-+
   |  ContentType  |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |        ProtocolVersion        |             epoch             |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |                        sequence_number                        |
   +                               +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |                               |E|            length           |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   ~              (zero or more) Extension Header(s)               ~
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |         Payload (including optional MAC and padding)          |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 1: Length redefined

Length counts the bytes of Payload and of all record header extensions that are added to this record (possibly none).

In the reminder, the top bit is called the E-bit.

3. Record Header Extension

3.1. Format

If the E-bit is asserted, then a record header extension is appended to the regular header with the following format:

```
    0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-----------+
   |M| Type  |       Length        | Value ... |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-----------+
```

Where:

- o M(ore) has the same semantics as the E-bit in the regular header - i.e.: if it is asserted then another extension header follows this one;

- o Type is a fixed length (4-bits) field that defines the way Value has to be interpreted;

- o Length is the size of Value in bytes. It uses 11 bits, therefore allowing a theoretical maximum size of 2047 bytes for any record header extension;

- o Value is the record header extension itself.

The fact that Type only allows 16 record header extension is a precise design choice: the allocation pool size is severely constrained so to raise the entry bar for any new record header extension.

3.2. Negotiation

A record header extension is allowed only if it has been negotiated via a companion DTLS extension.

An endpoint MUST NOT send a record header extension that hasn't been successfully negotiated with the receiver.

An endpoint that receives an unexpected record header extension MUST abort the session.

Record header extensions MUST NOT be sent during the initial handshake phase.

3.3. Backwards Compatibility

A legacy endpoint that receives a record header extension will interpret it as an invalid length field ([RFC6347], [I-D.ietf-tls-dtls13]) and abort the session accordingly.

Note that this is equivalent to the behaviour of an endpoint implementing this spec which receives a non-negotiated record header extension.

3.4. Use with Connection ID

A plausible use of this mechanism is with the CID extension defined in [I-D.ietf-tls-dtls-connection-id].

In that case, the companion record header extension could be defined as follows:

- Type: 0x0 (i.e., CID record header extension);

- Value: the CID itself

A DTLS 1.2 record carrying a CID "AB" would be formatted as in Figure 2:

- E=1

- Type=0x0

- Length=0x002

- Value=0x4142

```
    0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5
   +-+-+-+-+-+-+-+-+
   |  ContentType  |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |            Version            |             Epoch             |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |                        Sequence Number                        |
   +                               +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |                               |1|            Length           |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |0|  0x0  |       0x002         |            0x4142             |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |         Payload (including optional MAC and padding)          |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 2: CID header example

Note that, compared to all other possible ways to express presence/absence of a CID field within the constraints of the current header format (e.g., bumping the Version field, assigning new ContentType(s), using an invalid length), an ad hoc record header extension provides a cleaner approach that can be used with any TLS version at a reasonable cost - an overhead of 2 bytes per record.

4. Security Considerations

An on-path active attacker could try and modify an existing record header extension, insert a new record header extension in an existing session, or alter the result of the negotiation in order to add or remove arbitrary record header extensions. Given the security properties of DTLS, none of the above can be tried without being fatally noticed by the endpoints.

A passive on-path attacker could potentially extrapolate useful knowledge about endpoints from the information encoded in a record header extension (see also Section 5).

5. Privacy Considerations

The extent and consequences of metadata leakage from endpoints to path when using a certain record header extension SHALL be assessed in the document that introduces this new record header extension. If needed, the document SHALL describe the relevant risk mitigations.

6. IANA Considerations

This document defines a new IANA registry that, for each new record header extension, shall provide its Type code-point.

7. Acknowledgements

Thanks to Adam Langley and Yoav Nir for comments and discussions that have helped shaping this document.

This work is partially supported by the European Commission under Horizon 2020 grant agreement no. 688421 Measurement and Architecture for a Middleboxed Internet (MAMI). This support does not imply endorsement.

8. References

8.1. Normative References

[I-D.ietf-tls-dtls13]
Rescorla, E., Tschofenig, H., and N. Modadugu, "The Datagram Transport Layer Security (DTLS) Protocol Version 1.3", draft-ietf-tls-dtls13-22 (work in progress), November 2017.

[I-D.ietf-tls-tls13]
Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", draft-ietf-tls-tls13-25 (work in progress), March 2018.

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

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

[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>.

8.2. Informative References

[I-D.ietf-tls-dtls-connection-id]
Rescorla, E., Tschofenig, H., Fossati, T., and T. Gondrom, "The Datagram Transport Layer Security (DTLS) Connection Identifier", draft-ietf-tls-dtls-connection-id-00 (work in progress), December 2017.

Authors' Addresses

Thomas Fossati
Nokia

Email: thomas.fossati@nokia.com

Nikos Mavrogiannopoulos
RedHat

Email: nmav@redhat.com

Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: May 12, 2019

R. Housley
Vigil Security

November 8, 2018

TLS 1.3 Extension for Certificate-based Authentication with an External Pre-Shared Key

draft-housley-tls-tls13-cert-with-extern-psk-03

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

Copyright Notice

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.

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 invention 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 that could be broken by the invention of a large-scale quantum computer [I-D.hoffman-c2pq]. These algorithms include Diffie-Hellman (DH) [DH] and Elliptic Curve Diffie-Hellman (ECDH) [IEEE1363]. 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 accompanied by the "key_share", "psk_key_exchange_modes", and "pre_shared_key" extensions. The "pre_shared_key" extension MUST be the last extension in the ClientHello message, and it provides a list of external PSK identifiers that the client is willing to use with this server. Since 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 extension are all described in Section 4.2 of [RFC8446].

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 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.

The successful negotiation of the "tls_cert_with_extern_psk" extension requires the TLS 1.3 key schedule processing to include 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.

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. The hash algorithm MUST be set when the PSK is established, with a default of SHA-256 if no hash algorithm is specified during establishment.

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 "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], and it MUST include the same "tls_cert_with_extern_psk" extension as the original ClientHello message or abort 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. It 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.

To use an external PSK with certificates, clients MUST provide the "tls_cert_with_extern_psk" extension, and it MUST be accompanied by the "key_share", "psk_key_exchange_modes", and "pre_shared_key" extensions in the ClientHello. If clients offer a "tls_cert_with_extern_psk" extension without all of these other extensions, servers MUST abort the handshake. The client MAY also find it useful to include the the "supported_groups" extension. Note that Section 4.2 of [RFC8446] allows extensions to appear in any order, with the exception of the "pre_shared_key" extension, which MUST be the last extension in the ClientHello. Also, there MUST NOT be more than one instance of each extension in the ClientHello message.

The "key_share" extension is defined in Section 4.2.8 of [RFC8446].

The "psk_key_exchange_modes" extension is defined in Section 4.2.9 of [RFC8446]. The "psk_key_exchange_modes" extension restricts both the use of PSKs offered in this ClientHello and those which the server might supply via a subsequent NewSessionTicket. As a result, clients MUST include the psk_dhe_ke mode, and clients MAY also include the psk_ke mode to support a subsequent NewSessionTicket. 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 "supported_groups" extension is defined in Section 4.2.7 of [RFC8446].

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.

```
     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 identity is also referred to as an identifier or a label.)

The obfuscated_ticket_age is not used for external PSKs; clients SHOULD set this value to 0, and servers MUST ignore the value.

The binders are a series of HMAC 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 external PSK identity that the server selected from the list offered by the client. If none of the offered external PSKs in the list provided by the client are acceptable to the server, then the "tls_cert_with_extern_psk" extension MUST be omitted from the ServerHello message. The server MUST validate the binder value that corresponds to the selected external PSK as described in Section 4.2.11.2 of [RFC8446]. If the binder does not validate, the server MUST abort the handshake with an "illegal_parameter" alert. Servers SHOULD NOT attempt to validate multiple binders; rather they SHOULD select one of the offered external PSKs and validate only the binder that corresponds to that external PSK.

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 the 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].

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 termination of the connection with a "decrypt_error" alert.

6. IANA Considerations

IANA is requested to update the TLS ExtensionType Registry 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 invention of a large-scale quantum computer.

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 choose 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 searching 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.

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 with either TLS 1.2 or TLS 1.3. However, any PSK that might be used with both TLS 1.2 and TLS 1.3 must be used with only one hash function, which is the one that is bound for use in TLS 1.3. This restriction is less than optimal when users want to provision a single PSK. While the constructions used in TLS 1.2 and TLS 1.3 are both based on HMAC [RFC2104], the constructions are different, and there is no known way in which reuse of the same PSK in TLS 1.2 and TLS 1.3 that would produce related outputs.

8. Acknowledgments

Many thanks to Nikos Mavrogiannopoulos, Nick Sullivan, Martin Thomson, and Peter Yee for their review and comments; their efforts have improved this document.

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

[DH] Diffie, W. and M. Hellman, "New Directions in Cryptography", IEEE Transactions on Information Theory V.IT-22 n.6, June 1977.

[I-D.hoffman-c2pq] Hoffman, P., "The Transition from Classical to Post-Quantum Cryptography", draft-hoffman-c2pq-04 (work in progress), August 2018.

[IEEE1363] Institute of Electrical and Electronics Engineers, "IEEE Standard Specifications for Public-Key Cryptography", IEEE Std 1363-2000, 2000.

[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>.

Author's Address

Russ Housley
Vigil Security, LLC
918 Spring Knoll Drive
Herndon, VA 20170
USA

Email: housley@vigilsec.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

Copyright Notice

Table of Contents

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	brotli	[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 M. Shore
Internet-Draft Fastly
Intended status: Standards Track R. Barnes
Expires: September 22, 2018 Mozilla
S. Huque
Salesforce
W. Toorop
NLnet Labs
March 21, 2018

A DANE Record and DNSSEC Authentication Chain Extension for TLS
draft-ietf-tls-dnssec-chain-extension-07

Abstract

This draft describes a new TLS extension for transport of a DNS record set serialized with the DNSSEC signatures needed to authenticate that record set. The intent of this proposal is to allow TLS clients to perform DANE authentication of a TLS server without needing to perform additional DNS record lookups. It is not intended to be used to validate the TLS server's address records.

Status of This Memo

Copyright Notice

Table of Contents

1. Requirements Notation

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

2. Introduction

This draft describes a new TLS [RFC5246] [TLS13] extension for transport of a DNS record set serialized with the DNSSEC signatures [RFC4034] needed to authenticate that record set. The intent of this proposal is to allow TLS clients to perform DANE Authentication

[RFC6698] [RFC7671] of a TLS server without performing additional DNS record lookups and incurring the associated latency penalty. It also provides the ability to avoid potential problems with TLS clients being unable to look up DANE records because of an interfering or broken middlebox on the path between the client and a DNS server [HAMPERING]. And lastly, it allows a TLS client to validate the server's DANE (TLSA) records itself without needing access to a validating DNS resolver to which it has a secure connection.

This mechanism is useful for TLS applications that need to address the problems described above, typically web browsers or SIP/VoIP [RFC3261] and XMPP [RFC7590]. It may not be relevant for many other applications. For example, SMTP MTAs are usually located in data centers, may tolerate extra DNS lookup latency, are on servers where it is easier to provision a validating resolver, or are less likely to experience traffic interference from misconfigured middleboxes. Furthermore, SMTP MTAs usually employ Opportunistic Security [RFC7672], in which the presence of the DNS TLSA records is used to determine whether to enforce an authenticated TLS connection. Hence DANE authentication of SMTP MTAs will typically not use this mechanism.

The extension described here allows a TLS client to request that the TLS server return the DNSSEC authentication chain corresponding to its DANE record. If the server is configured for DANE authentication, then it performs the appropriate DNS queries, builds the authentication chain, and returns it to the client. The server will usually use a previously cached authentication chain, but it will need to rebuild it periodically as described in Section 5. The client then authenticates the chain using a pre-configured trust anchor.

This specification is based on Adam Langley's original proposal for serializing DNSSEC authentication chains and delivering them in an X.509 certificate extension [I-D.agl-dane-serializechain]. It modifies the approach by using wire format DNS records in the serialized data (assuming that the data will be prepared and consumed by a DNS-specific library), and by using a TLS extension to deliver the data.

As described in the DANE specification [RFC6698] [RFC7671], this procedure applies to the DANE authentication of X.509 certificates or raw public keys [RFC7250].

3. DNSSEC Authentication Chain Extension

3.1. Protocol, TLS 1.2

A client MAY include an extension of type "dnssec_chain" in the (extended) ClientHello. The "extension_data" field of this extension MUST be empty.

Servers receiving a "dnssec_chain" extension in the ClientHello and which are capable of being authenticated via DANE, return a serialized authentication chain in the extended ServerHello message using the format described below. If a server is unable to return an authentication chain, or does not wish to return an authentication chain, it does not include a dnssec_chain extension. As with all TLS extensions, if the server does not support this extension it will not return any authentication chain.

3.2. Protocol, TLS 1.3

A client MAY include an extension of type "dnssec_chain" in the ClientHello. The "extension_data" field of this extension MUST be empty.

Servers receiving a "dnssec_chain" extension in the ClientHello, and which are capable of being authenticated via DANE, return a serialized authentication chain in the extension block of the Certificate message containing the end entity certificate being validated, using the format described below.

The extension protocol behavior otherwise follows that specified for TLS version 1.2.

3.3. Raw Public Keys

[RFC7250] specifies the use of raw public keys for both server and client authentication in TLS 1.2. It points out that in cases where raw public keys are being used, code for certificate path validation is not required. However, DANE, when used in conjunction with the dnssec_chain extension, provides a mechanism for securely binding a raw public key to a named entity in the DNS, and when using DANE for authentication a raw key may be validated using a path chaining back to a DNSSEC trust root. This has the added benefit of mitigating an unknown key share attack, as described in [I-D.barnes-dane-uks], since it effectively augments the raw public key with the server's name and provides a means to commit both the server and the client to using that binding.

The UKS attack is possible in situations in which the association between a domain name and a public key is not tightly bound, as in the case in DANE in which a client either ignores the name in the certificate (as specified in [RFC7671]) or there is no attestation of trust outside of the DNS. The vulnerability arises in the following situations:

- If the client does not verify the identity in the server's certificate (as recommended in Section 5.1 of [RFC7671]), then an attacker can induce the client to accept an unintended identity for the server,

- If the client allows the use of raw public keys in TLS, then it will not receive any indication of the server's identity in the TLS channel, and is thus unable to check that the server's identity is as intended.

The mechanism for conveying DNSSEC validation chains described in this document results in a commitment by both parties, via the TLS handshake, to a validated domain name and EE key.

The mechanism for encoding DNSSEC authentication chains in a TLS extension, as described in this document, is not limited to public keys encapsulated in X.509 containers but MAY be applied to raw public keys and other representations, as well.

3.4. DNSSEC Authentication Chain Data

The "extension_data" field of the "dnssec_chain" extension MUST contain a DNSSEC Authentication Chain encoded in the following form:

```
          opaque AuthenticationChain<1..2^16-1>
```

The AuthenticationChain structure is composed of a sequence of uncompressed wire format DNS resource record sets (RRset) and corresponding signatures (RRSIG) record sets.

This sequence of native DNS wire format records enables easier generation of the data structure on the server and easier verification of the data on client by means of existing DNS library functions.

Each RRset in the chain is composed of a sequence of wire format DNS resource records. The format of the resource record is described in RFC 1035 [RFC1035], Section 3.2.1.

```
RR(i) = owner | type | class | TTL | RDATA length | RDATA
```

where RR(i) denotes the ith RR.

The resource records that make up a RRset all have the same owner, type and class, but different RDATA as specified RFC 2181 [RFC2181], Section 5. Each RRset in the sequence is followed by its associated RRsig record set. This RRset has the same owner and class as the preceding RRset, but has type RRSIG. The Type Covered field in the RDATA of the RRsigs identifies the type of the preceding RRset as described in RFC 4034 [RFC4034], Section 3. The RRsig record wire format is described in RFC 4034 [RFC4034], Section 3.1. The signature portion of the RDATA, as described in the same section, is the following:

```
signature = sign(RRSIG_RDATA | RR(1) | RR(2)... )
```

where RRSIG_RDATA is the wire format of the RRSIG RDATA fields with the Signer’s Name field in canonical form and the signature field excluded.

The first RRset in the chain MUST contain the TLSA record set being presented. However, if the owner name of the TLSA record set is an alias (CNAME or DNAME), then it MUST be preceded by the chain of alias records needed to resolve it. DNAME chains SHOULD omit unsigned CNAME records that may have been synthesized in the response from a DNS resolver. (If unsigned synthetic CNAMES are present, then the TLS client will just ignore them, as they are not necessary to validate the chain.)

The subsequent RRsets MUST contain the full set of DNS records needed to authenticate the TLSA record set from the server’s trust anchor. Typically this means a set of DNSKEY and DS RRsets that cover all zones from the target zone containing the TLSA record set to the trust anchor zone. The TLS client should be prepared to receive this set of RRsets in any order.

Names that are aliased via CNAME and/or DNAME records may involve multiple branches of the DNS tree. In this case, the authentication chain structure needs to include DS and DNSKEY record sets that cover all the necessary branches.

If the TLSA record set was synthesized by a DNS wildcard, the chain MUST include the signed NSEC or NSEC3 [RFC5155] records that prove that there was no explicit match of the TLSA record name and no closer wildcard match.

The final DNSKEY RRset in the authentication chain corresponds to the trust anchor (typically the DNS root). This trust anchor is also preconfigured in the TLS client, but including it in the response from the server permits TLS clients to use the automated trust anchor rollover mechanism defined in RFC 5011 [RFC5011] to update their configured trust anchor.

The following is an example of the records in the AuthenticationChain structure for the HTTPS server at www.example.com, where there are zone cuts at "com." and "example.com." (record data are omitted here for brevity):

```
_443._tcp.www.example.com. TLSA
RRSIG(_443._tcp.www.example.com. TLSA)
example.com. DNSKEY
RRSIG(example.com. DNSKEY)
example.com. DS
RRSIG(example.com. DS)
com. DNSKEY
RRSIG(com. DNSKEY)
com. DS
RRSIG(com. DS)
. DNSKEY
RRSIG(. DNSKEY)
```

4. Construction of Serialized Authentication Chains

This section describes a possible procedure for the server to use to build the serialized DNSSEC chain.

When the goal is to perform DANE authentication [RFC6698] [RFC7671] of the server, the DNS record set to be serialized is a TLSA record set corresponding to the server’s domain name, protocol, and port number.

The domain name of the server MUST be that included in the TLS server_name extension [RFC6066] when present. If the server_name extension is not present, or if the server does not recognize the provided name and wishes to proceed with the handshake rather than to abort the connection, the server picks one of its configured domain names associated with the server IP address to which the connection has been established.

The TLSA record to be queried is constructed by prepending the _port and _transport labels to the domain name as described in [RFC6698], where "port" is the port number associated with the TLS server. The transport is "tcp" for TLS servers, and "udp" for DTLS servers. The port number label is the left-most label, followed by the transport, followed by the base domain name.

The components of the authentication chain are typically built by starting at the target record set and its corresponding RRSIG. Then traversing the DNS tree upwards towards the trust anchor zone (normally the DNS root), for each zone cut, the DNSKEY and DS RRsets and their signatures are added. However, see Section 3.4 for specific processing needed for aliases and wildcards. If DNS response messages contain any domain names utilizing name compression [RFC1035], then they MUST be uncompressed.

Newer DNS protocol enhancements, such as the EDNS Chain Query extension [RFC7901] if supported, may offer easier ways to obtain all of the chain data in one transaction with an upstream DNSSEC aware recursive server.

5. Caching and Regeneration of the Authentication Chain

DNS records have Time To Live (TTL) parameters, and DNSSEC signatures have validity periods (specifically signature expiration times). After the TLS server constructs the serialized authentication chain, it SHOULD cache and reuse it in multiple TLS connection handshakes. However, it MUST refresh and rebuild the chain as TTLs and signature validity periods dictate. A server implementation could carefully track these parameters and requery component records in the chain correspondingly. Alternatively, it could be configured to rebuild the entire chain at some predefined periodic interval that does not exceed the DNS TTLs or signature validity periods of the component records in the chain.

6. Verification

A TLS client making use of this specification, and which receives a DNSSEC authentication chain extension from a server, MUST use this information to perform DANE authentication of the server. In order to do this, it uses the mechanism specified by the DNSSEC protocol [RFC4035] [RFC5155]. This mechanism is sometimes implemented in a DNSSEC validation engine or library.

If the authentication chain is correctly verified, the client then performs DANE authentication of the server according to the DANE TLS protocol [RFC6698] [RFC7671].

Clients MAY cache the server's validated TLSA RRset or other validated portions of the chain as an optimization to save signature verification work for future connections. The period of such caching MUST NOT exceed the TTL associated with those records. A client that possesses a validated and unexpired TLSA RRset or the full chain in its cache does not need to send the dnssec_chain extension for subsequent connections to the same TLS server. It can use the cached information to perform DANE authentication.

7. Trust Anchor Maintenance

The trust anchor may change periodically, e.g. when the operator of the trust anchor zone performs a DNSSEC key rollover. TLS clients using this specification MUST implement a mechanism to keep their trust anchors up to date. They could use the method defined in [RFC5011] to perform trust anchor updates inband in TLS, by tracking the introduction of new keys seen in the trust anchor DNSKEY RRset. However, alternative mechanisms external to TLS may also be utilized. Some operating systems may have a system-wide service to maintain and keep the root trust anchor up to date. In such cases, the TLS client application could simply reference that as its trust anchor, periodically checking whether it has changed. Some applications may prefer to implement trust anchor updates as part of their automated software updates.

8. Mandating use of this extension

Green field applications that are designed to always employ this extension, could of course unconditionally mandate its use.

If TLS applications want to mandate the use of this extension for specific servers, clients could maintain a whitelist of sites where the use of this extension is forced. The client would refuse to authenticate such servers if they failed to deliver this extension. Client applications could also employ a Trust on First Use (TOFU) like strategy, whereby they would record the fact that a server offered the extension and use that knowledge to require it for subsequent connections.

This protocol currently provides no way for a server to prove that it doesn’t have a TLSA record. Hence absent whitelists, a client misdirected to a server that has fraudulently acquired a public CA issued certificate for the real server’s name, could be induced to establish a PKIX verified connection to the rogue server that precluded DANE authentication. This could be solved by enhancing this protocol to require that servers without TLSA records need to provide a DNSSEC authentication chain that proves this (i.e. the chain includes NSEC or NSEC3 records that demonstrate either the absence of the TLSA record, or the absence of a secure delegation to the associated zone). Such an enhancement would be impossible to deploy incrementally though since it requires all TLS servers to support this protocol.

One possible way to address the threat of attackers that have fraudulently obtained valid PKIX credentials, is to use current PKIX defense mechanisms, such as checking Certificate Transparency logs to detect certificate misissuance. This may be necessary anyway, as TLS servers may support both DANE and PKIX authentication. Even TLS servers that support only DANE may be interested in detecting PKIX adversaries impersonating their service to DANE unaware TLS clients.

9. DANE and Traditional PKIX Interoperation

When DANE is being introduced incrementally into an existing PKIX environment, there may be scenarios in which DANE authentication for a server fails but PKIX succeeds, or vice versa. What happens here depends on TLS client policy. If DANE authentication fails, the client may decide to fallback to traditional PKIX authentication. In order to do so efficiently within the same TLS handshake, the TLS server needs to have provided the full X.509 certificate chain. When TLS servers only support DANE-EE or DANE-TA modes, they have the option to send a much smaller certificate chain: just the EE certificate for the former, and a short certificate chain from the DANE trust anchor to the EE certificate for the latter. If the TLS server supports both DANE and traditional PKIX, and wants to allow efficient PKIX fallback within the same handshake, they should always provide the full X.509 certificate chain.

10. Security Considerations

The security considerations of the normatively referenced RFCs all pertain to this extension. Since the server is delivering a chain of DNS records and signatures to the client, it MUST rebuild the chain in accordance with TTL and signature expiration of the chain components as described in Section 5. TLS clients need roughly accurate time in order to properly authenticate these signatures. This could be achieved by running a time synchronization protocol like NTP [RFC5905] or SNTP [RFC5905], which are already widely used today. TLS clients MUST support a mechanism to track and rollover the trust anchor key, or be able to avail themselves of a service that does this, as described in Section 7. Security considerations related to mandating the use of this extension are described in Section 8.

11. IANA Considerations

This extension requires the registration of a new value in the TLS ExtensionsType registry. The value requested from IANA is 53, and the extension should be marked "Recommended" in accordance with "IANA Registry Updates for TLS and DTLS" [TLSIANA].

12. Acknowledgments

Many thanks to Adam Langley for laying the groundwork for this extension. The original idea is his but our acknowledgment in no way implies his endorsement. This document also benefited from discussions with and review from the following people: Viktor Dukhovni, Daniel Kahn Gillmor, Jeff Hodges, Allison Mankin, Patrick McManus, Rick van Rein, Ilari Liusvaara, Eric Rescorla, Gowri Visweswaran, Duane Wessels, Nico Williams, and Paul Wouters.

13. References

13.1. Normative References

[RFC1035] Mockapetris, P., "Domain names - implementation and specification", STD 13, RFC 1035, November 1987.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC2181] Elz, R. and R. Bush, "Clarifications to the DNS Specification", RFC 2181, DOI 10.17487/RFC2181, July 1997, <http://www.rfc-editor.org/info/rfc2181>.

[RFC4034] Arends, R., Austein, R., Larson, M., Massey, D., and S. Rose, "Resource Records for the DNS Security Extensions", RFC 4034, March 2005.

[RFC4035] Arends, R., Austein, R., Larson, M., Massey, D., and S. Rose, "Protocol Modifications for the DNS Security Extensions", RFC 4035, March 2005.

[RFC5155] Laurie, B., Sisson, G., Arends, R., and D. Blacka, "DNS Security (DNSSEC) Hashed Authenticated Denial of Existence", RFC 5155, March 2008.

[RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, August 2008.

[RFC6066] Eastlake, D., "Transport Layer Security (TLS) Extensions: Extension Definitions", RFC 6066, January 2011.

[RFC6698] Hoffman, P. and J. Schlyter, "The DNS-Based Authentication of Named Entities (DANE) Transport Layer Security (TLS) Protocol: TLSA", RFC 6698, August 2012.

[RFC7671] Dukhovni, V. and W. Hardaker, "The DNS-Based Authentication of Named Entities (DANE) Protocol: Updates and Operational Guidance", RFC 7671, DOI 10.17487/RFC7671, October 2015, <http://www.rfc-editor.org/info/rfc7671>.

[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", March 2018, <https://tools.ietf.org/html/draft-ietf-tls-tls13>.

[TLSIANA] Salowey, J. and S. Turner, "IANA Registry Updates for TLS and DTLS", , <https://tools.ietf.org/html/draft-ietf-tls-iana-registry-updates>.

13.2. Informative References

[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>.

[RFC5011] StJohns, M., "Automated Updates of DNS Security (DNSSEC) Trust Anchors", STD 74, RFC 5011, September 2007.

[RFC5905] Mills, D., Martin, J., Burbank, J., and W. Kasch, "Network Time Protocol Version 4: Protocol and Algorithms Specification", RFC 5905, June 2010.

[RFC7120] Cotton, M., "Early IANA Allocation of Standards Track Code Points", BCP 100, RFC 7120, January 2014.

[RFC7250] Wouters, P., Tschofenig, H., Gilmore, J., Weiler, S., and T. Kivinen, "Using Raw Public Keys in Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)", RFC 7250, June 2014.

[RFC7590] Saint-Andre, P. and T. Alkemade, "Use of Transport Layer Security (TLS) in the Extensible Messaging and Presence Protocol (XMPP)", RFC 7590, DOI 10.17487/RFC7590, June 2015, <https://www.rfc-editor.org/info/rfc7590>.

[RFC7672] Dukhovni, V. and W. Hardaker, "SMTP Security via Opportunistic DNS-Based Authentication of Named Entities (DANE) Transport Layer Security (TLS)", RFC 7672, DOI 10.17487/RFC7672, October 2015, <http://www.rfc-editor.org/info/rfc7672>.

[RFC7901] Wouters, P., "CHAIN Query Requests in DNS", RFC 7901, DOI 10.17487/RFC7901, June 2016, <http://www.rfc-editor.org/info/rfc7901>.

[I-D.agl-dane-serializechain] Langley, A., "Serializing DNS Records with DNSSEC Authentication", draft-agl-dane-serializechain-01 (work in progress), July 2011.

[I-D.barnes-dane-uks] Barnes, R., Thomson, M., and E. Rescorla, "Unknown Key-Share Attacks on DNS-based Authentications of Named Entities (DANE)", draft-barnes-dane-uks-00 (work in progress), October 2016.

[HAMPERING] Gorjon, X. and W. Toorop, "Discovery method for a DNSSEC validating stub resolver", July 2015, <http://www.nlnetlabs.nl/downloads/publications/os3-2015-rp2-xavier-torrent-gorjon.pdf>.

Appendix A. Test vectors

The provided test vectors will authenticate the certificate used with https://example.com/, https://example.net/ and https://example.org/ at the time of writing:

```
-----BEGIN CERTIFICATE-----
MIIF8jCCBNqgAwIBAgIQDmTF+8I2reFLFyrrQceMsDANBgkqhkiG9w0BAQsFADBw
MQswCQYDVQQGEwJVUzEVMBMGA1UEChMMRGlnaUNlcnQgSW5jMRkwFwYDVQQLExB3
d3cuZGlnaWNlcnQuY29tMS8wLQYDVQQDEyZEaWdpQ2VydCBTSEEyIEhpZ2ggQXNz
dXJhbmNlIFNlcnZlciBDQTAeFw0xNTExMDMwMDAwMDBaFw0xODExMjgxMjAwMDBa
MIGlMQswCQYDVQQGEwJVUzETMBEGA1UECBMKQ2FsaWZvcm5pYTEUMBIGA1UEBxML
TG9zIEFuZ2VsZXMxPDA6BgNVBAoTM0ludGVybmV0IENvcnBvcmF0aW9uIGZvciBB
c3NpZ25lZCBOYW1lcyBhbmQgTnVtYmVyczETMBEGA1UECxMKVGVjaG5vbG9neTEY
MBYGA1UEAxMPd3d3LmV4YW1wbGUub3JnMIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8A
MIIBCgKCAQEAs0CWL2FjPiXBl61lRfvvE0KzLJmG9LWAC3bcBjgsH6NiVVo2dt6u
Xfzi5bTm7F3K7srfUBYkLO78mraM9qizrHoIeyofrV/n+pZZJauQsPjCPxMEJnRo
D8Z4KpWKX0LyDu1SputoI4nlQ/htEhtiQnuoBfNZxF7WxcxGwEsZuS1KcXIkHl5V
RJOreKFHTaXcB1qcZ/QRaBIv0yhxvK1yBTwWddT4cli6GfHcCe3xGMaSL328Fgs3
jYrvG29PueB6VJi/tbbPu6qTfwp/H1brqdjh29U52Bhb0fJkM9DWxCP/Cattcc7a
z8EXnCO+LK8vkhw/kAiJWPKx4RBvgy73nwIDAQABo4ICUDCCAkwwHwYDVR0jBBgw
FoAUUWj/kK8CB3U8zNllZGKiErhZcjswHQYDVR0OBBYEFKZPYB4fLdHn8SOgKpUW
5Oia6m5IMIGBBgNVHREEejB4gg93d3cuZXhhbXBsZS5vcmeCC2V4YW1wbGUuY29t
ggtleGFtcGxlLmVkdYILZXhhbXBsZS5uZXSCC2V4YW1wbGUub3Jngg93d3cuZXhh
bXBsZS5jb22CD3d3dy5leGFtcGxlLmVkdYIPd3d3LmV4YW1wbGUubmV0MA4GA1Ud
DwEB/wQEAwIFoDAdBgNVHSUEFjAUBggrBgEFBQcDAQYIKwYBBQUHAwIwdQYDVR0f
BG4wbDA0oDKgMIYuaHR0cDovL2NybDMuZGlnaWNlcnQuY29tL3NoYTItaGEtc2Vy
dmVyLWc0LmNybDA0oDKgMIYuaHR0cDovL2NybDQuZGlnaWNlcnQuY29tL3NoYTIt
aGEtc2VydmVyLWc0LmNybDBMBgNVHSAERTBDMDcGCWCGSAGG/WwBATAqMCgGCCsG
AQUFBwIBFhxodHRwczovL3d3dy5kaWdpY2VydC5jb20vQ1BTMAgGBmeBDAECAjCB
gwYIKwYBBQUHAQEEdzB1MCQGCCsGAQUFBzABhhhodHRwOi8vb2NzcC5kaWdpY2Vy
dC5jb20wTQYIKwYBBQUHMAKGQWh0dHA6Ly9jYWNlcnRzLmRpZ2ljZXJ0LmNvbS9E
aWdpQ2VydFNIQTJIaWdoQXNzdXJhbmNlU2VydmVyQ0EuY3J0MAwGA1UdEwEB/wQC
MAAwDQYJKoZIhvcNAQELBQADggEBAISomhGn2L0LJn5SJHuyVZ3qMIlRCIdvqe0Q
6ls+C8ctRwRO3UU3x8q8OH+2ahxlQmpzdC5al4XQzJLiLjiJ2Q1p+hub8MFiMmVP
PZjb2tZm2ipWVuMRM+zgpRVM6nVJ9F3vFfUSHOb4/JsEIUvPY+d8/Krc+kPQwLvy
ieqRbcuFjmqfyPmUv1U9QoI4TQikpw7TZU0zYZANP4C/gj4Ry48/znmUaRvy2kvI
l7gRQ21qJTK5suoiYoYNo3J9T+pXPGU7Lydz/HwW+w0DpArtAaukI8aNX4ohFUKS
wDSiIIWIWJiJGbEeIO0TIFwEVWTOnbNl/faPXpk5IRXicapqiII=
-----END CERTIFICATE-----
```

For brevity and reproducability all DNS zones involved with the test vectors are signed using keys with algorithm 13: ECDSA Curve P-256 with SHA-256.

To reflect operational practice, different zones in the examples are in different phases of rolling their signing keys:

All zones use a Key Signing Key (KSK) and Zone Signing Key (ZSK), except for the example.com and example.net zones which use a Combined Signing Key (CSK).

The root and org zones are rolling their ZSK's.

The com and org zones are rolling their KSK's.

The test vectors are DNSSEC valid in the same period as the certificate is valid, which is in between November 3 2015 and November 28 2018, with the following root trust anchor:

```
   .  IN  DS  ( 47005 13 2 2eb6e9f2480126691594d649a5a613de3052e37861634
         641bb568746f2ffc4d4 )
```

A.1. _443._tcp.www.example.com

```
   _443._tcp.www.example.com.  3600  IN  TLSA  ( 3 1 1
          c66bef6a5c1a3e78b82016e13f314f3cc5fa25b1e52aab9adb9ec5989b165
          ada )
   _443._tcp.www.example.com.  3600  IN  RRSIG  ( TLSA 13 5 3600
          20181128000000 20151103000000 1870 example.com.
          uml1DUjp5RfrXn9WtuMxEQV+ygzrONcuzsnyfOGSszwaDdkSOJ0Kndcfbb2Il
          LUV04Z+V488+Sd1jr7/21tsKA== )
   example.com.  3600  IN  DNSKEY  ( 257 3 13
          JnA1XgyJTZz+psWvbrfUWLV6ULqIJyUS2CQdhUH9VK35bslWeJpRzrlxCUs7s
          /TsSfZMaGWVvlsuieh5nHcXzA== ) ; Key ID = 1870
   example.com.  3600  IN  RRSIG  ( DNSKEY 13 2 3600
          20181128000000 20151103000000 1870 example.com.
          HujA9vQTbCxMeaYjDOCF0fYyHhajTl5xPztrp5u6P2vYV8naYQLG3zUF1gaer
          WBOagXXblaSSbYwB96LU3uSdg== )
   example.com.  900  IN  DS  ( 1870 13 2 e9b533a049798e900b5c29c90cd25a
          986e8a44f319ac3cd302bafc08f5b81e16 )
   example.com.  900  IN  RRSIG  ( DS 13 2 900 20181128000000
          20151103000000 34327 com.
          1tua9ntAqZvOnK5UztzIjN38Bqs6mJ8KAT7L4+AxevDL+z0Jft7RC1/g6Qrfa
          In1wqF4U7TvC8PYOD0U/HYtwQ== )
   com.  900  IN  DNSKEY  ( 256 3 13
          7IIE5Dol8jSMUqHTvOOiZapdEbQ9wqRxFi/zQcSdufUKLhpByvLpzSAQTqCWj
          3URIZ8L3Fa2gBLMOZUzZ1GQCw== ) ; Key ID = 34327
   com.  900  IN  DNSKEY  ( 257 3 13
          RbkcO+96XZmnp8jYIuM4lryAp3egQjSmBaSoiA7H76Tm0RLHPNPUxlVk+nQ0f
          Ic3I8xfZDNw8Wa0Pe3/g2QA/w== ) ; Key ID = 18931
   com.  900  IN  DNSKEY  ( 257 3 13
          szc7biLo5J4OHlkan1vZrF4aD4YYf+NHA/GAqdNslY9xxK9Izg68XHkqck4Rt
          DiVk37lNAQmgSlHbrGu0yOTkA== ) ; Key ID = 28809
   com.  900  IN  RRSIG  ( DNSKEY 13 1 900 20181128000000
          20151103000000 18931 com.
          lZmTBrfcRgVbqHJIfCVr6c3HUDgy3MlNSCSnrVV2S5/NmB3ZiFcvIDn0iqXPm
          7YQfvfWi6utyxBu/fSD6S1ARw== )
   com.  900  IN  RRSIG  ( DNSKEY 13 1 900 20181128000000
          20151103000000 28809 com.
          8qZOVM4X8wGt5XPWhG2HO4FAD6Kvs5eIhZUz+7DVCrZ/XMEVrMIHcm1Q+sq0s
          hm4cSivK2BxOO24PHJXoZN2Lw== )
   com.  86400  IN  DS  ( 18931 13 2 20f7a9db42d0e2042fbbb9f9ea015941202
          f9eabb94487e658c188e7bcb52115 )
   com.  86400  IN  DS  ( 28809 13 2 ad66b3276f796223aa45eda773e92c6d98e
          70643bbde681db342a9e5cf2bb380 )
   com.  86400  IN  RRSIG  ( DS 13 1 86400 20181128000000
          20151103000000 31918 .
          5KQVa0NP+6k7VEGMmeky2/Y3wIGM70Fkm0vp5NmQ6KPk8L1XMJPltcJDWGGjc
          EU3Uc4z2DUxzZyWgEDdrSOcdw== )
   .  86400  IN  DNSKEY  ( 256 3 13
          zKz+DCWkNA/vuheiVPcGqsH40U84KZAlrMRIyozj9WHzf8PsFp/oR8j8vmjjW
          P98cbte4d8NvlGLxzbUzo3+FA== ) ; Key ID = 31918
   .  86400  IN  DNSKEY  ( 256 3 13
          8wMZZ4lzHdyKZ4fv8kys/t3QMlgvEadbsbyqWrMhwddSXCZYGRrsAbPpireRW
          xbVcd1VtOrlFBcRDMTN0R0XEQ== ) ; Key ID = 2635
   .  86400  IN  DNSKEY  ( 257 3 13
          yvX+VNTUjxZiGvtr060hVbrPV9H6rVusQtF9lIxCFzbZOJxMQBFmbqlc8Xclv
          Q+gDOXnFOTsgs/frMmxyGOtRg== ) ; Key ID = 47005
   .  86400  IN  RRSIG  ( DNSKEY 13 0 86400 20181128000000
          20151103000000 47005 .
          ehAzuZD3yT0pShXkKavrMdz+DKvvFvbZ+sGRZ5iQTni+ulMzZxHQ5+kSha65B
          Y2AIUphjyWcGr6VwP3Ne74iZA== )
```

A hex dump of the wire format data of this content is:

```
   0000:  04 5f 34 34 33 04 5f 74  63 70 03 77 77 77 07 65
   0010:  78 61 6d 70 6c 65 03 63  6f 6d 00 00 34 00 01 00
   0020:  00 0e 10 00 23 03 01 01  c6 6b ef 6a 5c 1a 3e 78
   0030:  b8 20 16 e1 3f 31 4f 3c  c5 fa 25 b1 e5 2a ab 9a
   0040:  db 9e c5 98 9b 16 5a da  04 5f 34 34 33 04 5f 74
   0050:  63 70 03 77 77 77 07 65  78 61 6d 70 6c 65 03 63
   0060:  6f 6d 00 00 2e 00 01 00  00 0e 10 00 5f 00 34 0d
   0070:  05 00 00 0e 10 5b fd da  80 56 37 f9 00 07 4e 07
   0080:  65 78 61 6d 70 6c 65 03  63 6f 6d 00 ba 69 75 0d
   0090:  48 e9 e5 17 eb 5e 7f 56  b6 e3 31 11 05 7e ca 0c
   00a0:  eb 38 d7 2e ce c9 f2 7c  e1 92 b3 3c 1a 0d d9 12
   00b0:  38 9d 0a 9d d7 1f 6d bd  88 94 b5 15 d3 86 7e 57
   00c0:  8f 3c f9 27 75 8e be ff  db 5b 6c 28 07 65 78 61
   00d0:  6d 70 6c 65 03 63 6f 6d  00 00 30 00 01 00 00 0e
   00e0:  10 00 44 01 01 03 0d 26  70 35 5e 0c 89 4d 9c fe
   00f0:  a6 c5 af 6e b7 d4 58 b5  7a 50 ba 88 27 25 12 d8
   0100:  24 1d 85 41 fd 54 ad f9  6e c9 56 78 9a 51 ce b9
   0110:  71 09 4b 3b b3 f4 ec 49  f6 4c 68 65 95 be 5b 2e
   0120:  89 e8 79 9c 77 17 cc 07  65 78 61 6d 70 6c 65 03
   0130:  63 6f 6d 00 00 2e 00 01  00 00 0e 10 00 5f 00 30
   0140:  0d 02 00 00 0e 10 5b fd  da 80 56 37 f9 00 07 4e
   0150:  07 65 78 61 6d 70 6c 65  03 63 6f 6d 00 1e e8 c0
   0160:  f6 f4 13 6c 2c 4c 79 a6  23 0c e0 85 d1 f6 32 1e
   0170:  16 a3 4e 5e 71 3f 3b 6b  a7 9b ba 3f 6b d8 57 c9
   0180:  da 61 02 c6 df 35 05 d6  06 9e ad 60 4e 6a 05 d7
   0190:  6e 56 92 49 b6 30 07 de  8b 53 7b 92 76 07 65 78
   01a0:  61 6d 70 6c 65 03 63 6f  6d 00 00 2b 00 01 00 00
   01b0:  03 84 00 24 07 4e 0d 02  e9 b5 33 a0 49 79 8e 90
   01c0:  0b 5c 29 c9 0c d2 5a 98  6e 8a 44 f3 19 ac 3c d3
   01d0:  02 ba fc 08 f5 b8 1e 16  07 65 78 61 6d 70 6c 65
   01e0:  03 63 6f 6d 00 00 2e 00  01 00 00 03 84 00 57 00
   01f0:  2b 0d 02 00 00 03 84 5b  fd da 80 56 37 f9 00 86
   0200:  17 03 63 6f 6d 00 d6 db  9a f6 7b 40 a9 9b ce 9c
   0210:  ae 54 ce dc c8 8c dd fc  06 ab 3a 98 9f 0a 01 3e
   0220:  cb e3 e0 31 7a f0 cb fb  3d 09 7e de d1 0b 5f e0
   0230:  e9 0a df 68 89 f5 c2 a1  78 53 b4 ef 0b c3 d8 38
   0240:  3d 14 fc 76 2d c1 03 63  6f 6d 00 00 30 00 01 00
   0250:  00 03 84 00 44 01 00 03  0d ec 82 04 e4 3a 25 f2
   0260:  34 8c 52 a1 d3 bc e3 a2  65 aa 5d 11 b4 3d c2 a4
   0270:  71 16 2f f3 41 c4 9d b9  f5 0a 2e 1a 41 ca f2 e9
   0280:  cd 20 10 4e a0 96 8f 75  11 21 9f 0b dc 56 b6 80
   0290:  12 cc 39 95 33 67 51 90  0b 03 63 6f 6d 00 00 30
   02a0:  00 01 00 00 03 84 00 44  01 01 03 0d 45 b9 1c 3b
   02b0:  ef 7a 5d 99 a7 a7 c8 d8  22 e3 38 96 bc 80 a7 77
   02c0:  a0 42 34 a6 05 a4 a8 88  0e c7 ef a4 e6 d1 12 c7
   02d0:  3c d3 d4 c6 55 64 fa 74  34 7c 87 37 23 cc 5f 64
   02e0:  33 70 f1 66 b4 3d ed ff  83 64 00 ff 03 63 6f 6d
   02f0:  00 00 30 00 01 00 00 03  84 00 44 01 01 03 0d b3
   0300:  37 3b 6e 22 e8 e4 9e 0e  1e 59 1a 9f 5b d9 ac 5e
   0310:  1a 0f 86 18 7f e3 47 03  f1 80 a9 d3 6c 95 8f 71
   0320:  c4 af 48 ce 0e bc 5c 79  2a 72 4e 11 b4 38 95 93
   0330:  7e e5 34 04 26 81 29 47  6e b1 ae d3 23 93 90 03
   0340:  63 6f 6d 00 00 2e 00 01  00 00 03 84 00 57 00 30
   0350:  0d 01 00 00 03 84 5b fd  da 80 56 37 f9 00 49 f3
   0360:  03 63 6f 6d 00 95 99 93  06 b7 dc 46 05 5b a8 72
   0370:  48 7c 25 6b e9 cd c7 50  38 32 dc c9 4d 48 24 a7
   0380:  ad 55 76 4b 9f cd 98 1d  d9 88 57 2f 20 39 f4 8a
   0390:  a5 cf 9b b6 10 7e f7 d6  8b ab ad cb 10 6e fd f4
   03a0:  83 e9 2d 40 47 03 63 6f  6d 00 00 2e 00 01 00 00
   03b0:  03 84 00 57 00 30 0d 01  00 00 03 84 5b fd da 80
   03c0:  56 37 f9 00 70 89 03 63  6f 6d 00 f2 a6 4e 54 ce
   03d0:  17 f3 01 ad e5 73 d6 84  6d 87 3b 81 40 0f a2 af
   03e0:  b3 97 88 85 95 33 fb b0  d5 0a b6 7f 5c c1 15 ac
   03f0:  c2 07 72 6d 50 fa ca b4  b2 19 b8 71 28 af 2b 60
   0400:  71 38 ed b8 3c 72 57 a1  93 76 2f 03 63 6f 6d 00
   0410:  00 2b 00 01 00 01 51 80  00 24 49 f3 0d 02 20 f7
   0420:  a9 db 42 d0 e2 04 2f bb  b9 f9 ea 01 59 41 20 2f
   0430:  9e ab b9 44 87 e6 58 c1  88 e7 bc b5 21 15 03 63
   0440:  6f 6d 00 00 2b 00 01 00  01 51 80 00 24 70 89 0d
   0450:  02 ad 66 b3 27 6f 79 62  23 aa 45 ed a7 73 e9 2c
   0460:  6d 98 e7 06 43 bb de 68  1d b3 42 a9 e5 cf 2b b3
   0470:  80 03 63 6f 6d 00 00 2e  00 01 00 01 51 80 00 53
   0480:  00 2b 0d 01 00 01 51 80  5b fd da 80 56 37 f9 00
   0490:  7c ae 00 e4 a4 15 6b 43  4f fb a9 3b 54 41 8c 99
   04a0:  e9 32 db f6 37 c0 81 8c  ef 41 64 9b 4b e9 e4 d9
   04b0:  90 e8 a3 e4 f0 bd 57 30  93 e5 b5 c2 43 58 61 a3
   04c0:  70 45 37 51 ce 33 d8 35  31 cd 9c 96 80 40 dd ad
   04d0:  23 9c 77 00 00 30 00 01  00 01 51 80 00 44 01 00
   04e0:  03 0d cc ac fe 0c 25 a4  34 0f ef ba 17 a2 54 f7
   04f0:  06 aa c1 f8 d1 4f 38 29  90 25 ac c4 48 ca 8c e3
   0500:  f5 61 f3 7f c3 ec 16 9f  e8 47 c8 fc be 68 e3 58
   0510:  ff 7c 71 bb 5e e1 df 0d  be 51 8b c7 36 d4 ce 8d
   0520:  fe 14 00 00 30 00 01 00  01 51 80 00 44 01 00 03
   0530:  0d f3 03 19 67 89 73 1d  dc 8a 67 87 ef f2 4c ac
   0540:  fe dd d0 32 58 2f 11 a7  5b b1 bc aa 5a b3 21 c1
   0550:  d7 52 5c 26 58 19 1a ec  01 b3 e9 8a b7 91 5b 16
   0560:  d5 71 dd 55 b4 ea e5 14  17 11 0c c4 cd d1 1d 17
   0570:  11 00 00 30 00 01 00 01  51 80 00 44 01 01 03 0d
   0580:  ca f5 fe 54 d4 d4 8f 16  62 1a fb 6b d3 ad 21 55
   0590:  ba cf 57 d1 fa ad 5b ac  42 d1 7d 94 8c 42 17 36
   05a0:  d9 38 9c 4c 40 11 66 6e  a9 5c f1 77 25 bd 0f a0
   05b0:  0c e5 e7 14 e4 ec 82 cf  df ac c9 b1 c8 63 ad 46
   05c0:  00 00 2e 00 01 00 01 51  80 00 53 00 30 0d 00 00
   05d0:  01 51 80 5b fd da 80 56  37 f9 00 b7 9d 00 7a 10
   05e0:  33 b9 90 f7 c9 3d 29 4a  15 e4 29 ab eb 31 dc fe
   05f0:  0c ab ef 16 f6 d9 fa c1  91 67 98 90 4e 78 be ba
   0600:  53 33 67 11 d0 e7 e9 12  85 ae b9 05 8d 80 21 4a
   0610:  61 8f 25 9c 1a be 95 c0  fd cd 7b be 22 64
```

A.2. _25._tcp.example.com wildcard

```
   _25._tcp.example.com.  3600  IN  TLSA  ( 3 1 1
           c66bef6a5c1a3e78b82016e13f314f3cc5fa25b1e52aab9adb9ec5989b165
           ada )
   _25._tcp.example.com.  3600  IN  RRSIG  ( TLSA 13 3 3600
           20181128000000 20151103000000 1870 example.com.
           e7Q5L2x7Ca3SkSY6pRjqgtRxkEN1uYUcgyMlPp6GQ4zxAZxoO1Y1vGqxN4eNA
           +yBnlUSIJQ46KKVS5PC79Qipg== )
   *._tcp.example.com.  3600  IN  NSEC  (
            _443._tcp.www.example.com. RRSIG NSEC TLSA )
   *._tcp.example.com.  3600  IN  RRSIG  ( NSEC 13 3 3600
           20181128000000 20151103000000 1870 example.com.
           FlTtPqEPUPAQozlbt7bD9s2XIxdVPJ3nb+jK94Fxa2JsaZChH1n/DsYb5KS7J
           G5GyubhMFTLeIqwTngx6JCktg== )
   example.com.  3600  IN  DNSKEY  ( 257 3 13
           JnA1XgyJTZz+psWvbrfUWLV6ULqIJyUS2CQdhUH9VK35bslWeJpRzrlxCUs7s
           /TsSfZMaGWVvlsuieh5nHcXzA== ) ; Key ID = 1870
   example.com.  3600  IN  RRSIG  ( DNSKEY 13 2 3600
           20181128000000 20151103000000 1870 example.com.
           HujA9vQTbCxMeaYjDOCF0fYyHhajTl5xPztrp5u6P2vYV8naYQLG3zUF1gaer
           WBOagXXblaSSbYwB96LU3uSdg== )
   example.com.  900  IN  DS  ( 1870 13 2 e9b533a049798e900b5c29c90cd25a
           986e8a44f319ac3cd302bafc08f5b81e16 )
   example.com.  900  IN  RRSIG  ( DS 13 2 900 20181128000000
           20151103000000 34327 com.
           1tua9ntAqZvOnK5UztzIjN38Bqs6mJ8KAT7L4+AxevDL+z0Jft7RC1/g6Qrfa
           In1wqF4U7TvC8PYOD0U/HYtwQ== )
   com.  900  IN  DNSKEY  ( 256 3 13
           7IIE5Dol8jSMUqHTvOOiZapdEbQ9wqRxFi/zQcSdufUKLhpByvLpzSAQTqCWj
           3URIZ8L3Fa2gBLMOZUzZ1GQCw== ) ; Key ID = 34327
   com.  900  IN  DNSKEY  ( 257 3 13
           RbkcO+96XZmnp8jYIuM4lryAp3egQjSmBaSoiA7H76Tm0RLHPNPUxlVk+nQ0f
           Ic3I8xfZDNw8Wa0Pe3/g2QA/w== ) ; Key ID = 18931
   com.  900  IN  DNSKEY  ( 257 3 13
           szc7biLo5J4OHlkan1vZrF4aD4YYf+NHA/GAqdNslY9xxK9Izg68XHkqck4Rt
           DiVk37lNAQmgSlHbrGu0yOTkA== ) ; Key ID = 28809
   com.  900  IN  RRSIG  ( DNSKEY 13 1 900 20181128000000
           20151103000000 18931 com.
           lZmTBrfcRgVbqHJIfCVr6c3HUDgy3MlNSCSnrVV2S5/NmB3ZiFcvIDn0iqXPm
           7YQfvfWi6utyxBu/fSD6S1ARw== )
   com.  900  IN  RRSIG  ( DNSKEY 13 1 900 20181128000000
           20151103000000 28809 com.
           8qZOVM4X8wGt5XPWhG2HO4FAD6Kvs5eIhZUz+7DVCrZ/XMEVrMIHcm1Q+sq0s
           hm4cSivK2BxOO24PHJXoZN2Lw== )
   com.  86400  IN  DS  ( 18931 13 2 20f7a9db42d0e2042fbbb9f9ea015941202
           f9eabb94487e658c188e7bcb52115 )
   com.  86400  IN  DS  ( 28809 13 2 ad66b3276f796223aa45eda773e92c6d98e
           70643bbde681db342a9e5cf2bb380 )
   com.  86400  IN  RRSIG  ( DS 13 1 86400 20181128000000
           20151103000000 31918 .
           5KQVa0NP+6k7VEGMmeky2/Y3wIGM70Fkm0vp5NmQ6KPk8L1XMJPltcJDWGGjc
           EU3Uc4z2DUxzZyWgEDdrSOcdw== )
   .  86400  IN  DNSKEY  ( 256 3 13
           zKz+DCWkNA/vuheiVPcGqsH40U84KZAlrMRIyozj9WHzf8PsFp/oR8j8vmjjW
           P98cbte4d8NvlGLxzbUzo3+FA== ) ; Key ID = 31918
   .  86400  IN  DNSKEY  ( 256 3 13
           8wMZZ4lzHdyKZ4fv8kys/t3QMlgvEadbsbyqWrMhwddSXCZYGRrsAbPpireRW
           xbVcd1VtOrlFBcRDMTN0R0XEQ== ) ; Key ID = 2635
   .  86400  IN  DNSKEY  ( 257 3 13
           yvX+VNTUjxZiGvtr060hVbrPV9H6rVusQtF9lIxCFzbZOJxMQBFmbqlc8Xclv
           Q+gDOXnFOTsgs/frMmxyGOtRg== ) ; Key ID = 47005
   .  86400  IN  RRSIG  ( DNSKEY 13 0 86400 20181128000000
           20151103000000 47005 .
           ehAzuZD3yT0pShXkKavrMdz+DKvvFvbZ+sGRZ5iQTni+ulMzZxHQ5+kSha65B
           Y2AIUphjyWcGr6VwP3Ne74iZA== )
```

A.3. _443._tcp.www.example.org CNAME

```
   _443._tcp.www.example.org.  3600  IN  CNAME  (
            dane311.example.org. )
   _443._tcp.www.example.org.  3600  IN  RRSIG  ( CNAME 13 5 3600
            20181128000000 20151103000000 56566 example.org.
            wLQYbRNMqrXCD65GZJqwwsD0TDF2VQTklBYdYCMo+JTjqvZw1UFYmcJXmwJsL
            KezLIzSdKW6jK0LMJ3YUw3Bmw== )
   dane311.example.org.  3600  IN  TLSA  ( 3 1 1
            c66bef6a5c1a3e78b82016e13f314f3cc5fa25b1e52aab9adb9ec5989b165
            ada )
   dane311.example.org.  3600  IN  RRSIG  ( TLSA 13 3 3600
            20181128000000 20151103000000 56566 example.org.
            AllKVcpLz/9vG/xJQFwWEK0cHbjO6lI65ELWSoWxPvYJ5o8QnSbRkzfCM4lTs
            g94s5VvzMLYIbSZ1TWo2hcCdg== )
   example.org.  3600  IN  DNSKEY  ( 256 3 13
            NrbL6utGqIW1wrhhjeexdA6bMdD1lC1hj0Fnpevaa1AMyY2uy83TmoGnR996N
            UR5TlG4Zh+YPbbmUIixe4nS3w== ) ; Key ID = 56566
   example.org.  3600  IN  DNSKEY  ( 257 3 13
            uspaqp17jsMTX6AWVgmbog/3Sttz+9ANFUWLn6qKUHr0BOqRuChQWj8jyYUUr
            Wy9txxesNQ9MkO4LUrFght1LQ== ) ; Key ID = 44384
   example.org.  3600  IN  RRSIG  ( DNSKEY 13 2 3600
            20181128000000 20151103000000 44384 example.org.
            ZsQ5wl2ZvofwDq7uYlvoqEeq9byHbl59Ap4EPXdB4PpnWy2dJkIElgXCfILrU
            EUCD1aKb2SoRZe18EJ8LMVJuw== )
   example.org.  900  IN  DS  ( 44384 13 2 ec307e2efc8f0117ed96ab48a513c
            8003e1d9121f1ff11a08b4cdd348d090aa6 )
   example.org.  900  IN  RRSIG  ( DS 13 2 900 20181128000000
            20151103000000 9523 org.
            15KUWAaNkJehAUdqm46TdeGg6mVm6bVKeaWLr34FTJlfMWWij+kmA6SM/bZbq
            kZBjtMWT55XersA+llFQNQI/Q== )
   org.  900  IN  DNSKEY  ( 256 3 13
            fuLp60znhSSEr9HowILpTpyLKQdM6ixcgkTE0gqVdsLx+DSNHSc69o6fLWC0e
            HfWx7kzlBBoJB0vLrvsJtXJ6g== ) ; Key ID = 47417
   org.  900  IN  DNSKEY  ( 256 3 13
            zTHbb7JM627Bjr8CGOySUarsic91xZU3vvLJ5RjVix9YH6+iwpBXb6qfHyQHy
            mlMiAAoaoXh7BUkEBVgDVN8sQ== ) ; Key ID = 9523
   org.  900  IN  DNSKEY  ( 257 3 13
            Uf24EyNt51DMcLV+dHPInhSpmjPnqAQNUTouU+SGLu+lFRRlBetgw1bJUZNI6
            Dlger0VJTm0QuX/JVXcyGVGoQ== ) ; Key ID = 49352
   org.  900  IN  DNSKEY  ( 257 3 13
            0SZfoe8Yx+eoaGgyAGEeJax/ZBV1AuG+/smcOgRm+F6doNlgc3lddcM1MbTvJ
            HTjK6Fvy8W6yZ+cAptn8sQheg== ) ; Key ID = 12651
   org.  900  IN  RRSIG  ( DNSKEY 13 1 900 20181128000000
            20151103000000 12651 org.
            G9I7dIh5Zn2hBu8jhgnLDTXZUpnPRkOMHjl1RcyHNbvJGLIiaPRVtcJXW0Vr+
            arygWmsHrDgWz0vw2IXZr3qKw== )
   org.  900  IN  RRSIG  ( DNSKEY 13 1 900 20181128000000
            20151103000000 49352 org.
            iQmYWqUdU07Syw1Fqwx+8+hSk0w06tCGmkwdppyxUSFESumEhkOXgOv6NuIEn
            eKjwMIaLj5HFB+9WnOkzgGE5Q== )
   org.  86400  IN  DS  ( 12651 13 2 3979a51f98bbf219fcaf4a4176e766dfa8f
            9db5c24a75743eb1e704b97a9fabc )
   org.  86400  IN  DS  ( 49352 13 2 03d11a1aa114abbb8f708c3c0ff0db765fe
            f4a2f18920db5f58710dd767c293b )
   org.  86400  IN  RRSIG  ( DS 13 1 86400 20181128000000
            20151103000000 31918 .
            JGPMvEbfLoWNUELn/5cjjdRZx2CmdikbHuH6N/1BrxACWrGy05NuPvBPTEVOr
            mPFfm5SIMLLTWgxf0K0FsNHoQ== )
   .  86400  IN  DNSKEY  ( 256 3 13
            zKz+DCWkNA/vuheiVPcGqsH40U84KZAlrMRIyozj9WHzf8PsFp/oR8j8vmjjW
            P98cbte4d8NvlGLxzbUzo3+FA== ) ; Key ID = 31918
   .  86400  IN  DNSKEY  ( 256 3 13
            8wMZZ4lzHdyKZ4fv8kys/t3QMlgvEadbsbyqWrMhwddSXCZYGRrsAbPpireRW
            xbVcd1VtOrlFBcRDMTN0R0XEQ== ) ; Key ID = 2635
   .  86400  IN  DNSKEY  ( 257 3 13
            yvX+VNTUjxZiGvtr060hVbrPV9H6rVusQtF9lIxCFzbZOJxMQBFmbqlc8Xclv
            Q+gDOXnFOTsgs/frMmxyGOtRg== ) ; Key ID = 47005
   .  86400  IN  RRSIG  ( DNSKEY 13 0 86400 20181128000000
            20151103000000 47005 .
            ehAzuZD3yT0pShXkKavrMdz+DKvvFvbZ+sGRZ5iQTni+ulMzZxHQ5+kSha65B
            Y2AIUphjyWcGr6VwP3Ne74iZA== )
```

A.4. _443._tcp.www.example.net DNAME

```
   example.net.  3600  IN  DNAME  example.com.
   example.net.  3600  IN  RRSIG  ( DNAME 13 2 3600 20181128000000
           20151103000000 48085 example.net.
           +MJa5ZEmYh/kHYOhabF3ibfJ5xhJDJAA76Sugc/LFyTDJbmYW/nlYf3XLdcDh
           7lv6NfCkPuv6eCkSFGnVVvriA== )
   _443._tcp.www.example.net.  3600  IN  CNAME  (
           _443._tcp.www.example.com. )
   _443._tcp.www.example.com.  3600  IN  TLSA  ( 3 1 1
           c66bef6a5c1a3e78b82016e13f314f3cc5fa25b1e52aab9adb9ec5989b165
           ada )
   _443._tcp.www.example.com.  3600  IN  RRSIG  ( TLSA 13 5 3600
           20181128000000 20151103000000 1870 example.com.
           uml1DUjp5RfrXn9WtuMxEQV+ygzrONcuzsnyfOGSszwaDdkSOJ0Kndcfbb2Il
           LUV04Z+V488+Sd1jr7/21tsKA== )
   example.net.  3600  IN  DNSKEY  ( 257 3 13
           X9GHpJcS7bqKVEsLiVAbddHUHTZqqBbVa3mzIQmdp+5cTJk7qDazwH68Kts8d
           9MvN55HddWgsmeRhgzePz6hMg== ) ; Key ID = 48085
   example.net.  3600  IN  RRSIG  ( DNSKEY 13 2 3600
           20181128000000 20151103000000 48085 example.net.
           Qu7q2IheqxAKGnchYSvQeJuXdnBj/+wJoEmv67wemOUI6qvWWIo535w+hguUV
           mZm/W5rp3qWBGChLxxfqIK13g== )
   example.net.  900  IN  DS  ( 48085 13 2 7c1998ce683df60e2fa41460c453f
           88f463dac8cd5d074277b4a7c04502921be )
   example.net.  900  IN  RRSIG  ( DS 13 2 900 20181128000000
           20151103000000 10713 net.
           xxSlIJlpOSmrUgwR++os2SHTpRf53SO95G6FQyH5lEslnTnbZoq0p/AVrlB8q
           Qw3qmSXjRwGW3VFbkV60/tWCg== )
   net.  900  IN  DNSKEY  ( 256 3 13
           061EoQs4sBcDsPiz17vt4nFSGLmXAGguqLStOesmKNCimi4/lw/vtyfqALuLF
           JiFjtCK3HMPi8HQ1jbGEwbGCA== ) ; Key ID = 10713
   net.  900  IN  DNSKEY  ( 257 3 13
           LkNCPE+v3S4MVnsOqZFhn8n2NSwtLYOZLZjjgVsAKgu4XZncaDgq1R/7ZXRO5
           oVx2zthxuu2i+mGbRrycAaCvA== ) ; Key ID = 485
   net.  900  IN  RRSIG  ( DNSKEY 13 1 900 20181128000000
           20151103000000 485 net.
           CC494bZrtBHXImEZpe6E3h6NL0R5fRR/MEuC1f2sfC6/dlCjRwFjCy9eOKnFL
           ar4Rxbpf7dvEwqGHNTawEo6jw== )
   net.  86400  IN  DS  ( 485 13 2 ab25a2941aa7f1eb8688bb783b25587515a0c
           d8c247769b23adb13ca234d1c05 )
   net.  86400  IN  RRSIG  ( DS 13 1 86400 20181128000000
           20151103000000 31918 .
           q+G4l97pYbFgAUhzzOW5+YoFiJc5omUbe20H28AwMHOrx19BdGp/2XhKDQ5F3
           tUTNerRmklzYm+7J/XtLpGXAw== )
   .  86400  IN  DNSKEY  ( 256 3 13
           zKz+DCWkNA/vuheiVPcGqsH40U84KZAlrMRIyozj9WHzf8PsFp/oR8j8vmjjW
           P98cbte4d8NvlGLxzbUzo3+FA== ) ; Key ID = 31918
   .  86400  IN  DNSKEY  ( 256 3 13
           8wMZZ4lzHdyKZ4fv8kys/t3QMlgvEadbsbyqWrMhwddSXCZYGRrsAbPpireRW
           xbVcd1VtOrlFBcRDMTN0R0XEQ== ) ; Key ID = 2635
   .  86400  IN  DNSKEY  ( 257 3 13
           yvX+VNTUjxZiGvtr060hVbrPV9H6rVusQtF9lIxCFzbZOJxMQBFmbqlc8Xclv
           Q+gDOXnFOTsgs/frMmxyGOtRg== ) ; Key ID = 47005
   .  86400  IN  RRSIG  ( DNSKEY 13 0 86400 20181128000000
           20151103000000 47005 .
           ehAzuZD3yT0pShXkKavrMdz+DKvvFvbZ+sGRZ5iQTni+ulMzZxHQ5+kSha65B
           Y2AIUphjyWcGr6VwP3Ne74iZA== )
   example.com.  3600  IN  DNSKEY  ( 257 3 13
           JnA1XgyJTZz+psWvbrfUWLV6ULqIJyUS2CQdhUH9VK35bslWeJpRzrlxCUs7s
           /TsSfZMaGWVvlsuieh5nHcXzA== ) ; Key ID = 1870
   example.com.  3600  IN  RRSIG  ( DNSKEY 13 2 3600
           20181128000000 20151103000000 1870 example.com.
           HujA9vQTbCxMeaYjDOCF0fYyHhajTl5xPztrp5u6P2vYV8naYQLG3zUF1gaer
           WBOagXXblaSSbYwB96LU3uSdg== )
   example.com.  900  IN  DS  ( 1870 13 2 e9b533a049798e900b5c29c90cd25a
           986e8a44f319ac3cd302bafc08f5b81e16 )
   example.com.  900  IN  RRSIG  ( DS 13 2 900 20181128000000
           20151103000000 34327 com.
           1tua9ntAqZvOnK5UztzIjN38Bqs6mJ8KAT7L4+AxevDL+z0Jft7RC1/g6Qrfa
           In1wqF4U7TvC8PYOD0U/HYtwQ== )
   com.  900  IN  DNSKEY  ( 256 3 13
           7IIE5Dol8jSMUqHTvOOiZapdEbQ9wqRxFi/zQcSdufUKLhpByvLpzSAQTqCWj
           3URIZ8L3Fa2gBLMOZUzZ1GQCw== ) ; Key ID = 34327
   com.  900  IN  DNSKEY  ( 257 3 13
           RbkcO+96XZmnp8jYIuM4lryAp3egQjSmBaSoiA7H76Tm0RLHPNPUxlVk+nQ0f
           Ic3I8xfZDNw8Wa0Pe3/g2QA/w== ) ; Key ID = 18931
   com.  900  IN  DNSKEY  ( 257 3 13
           szc7biLo5J4OHlkan1vZrF4aD4YYf+NHA/GAqdNslY9xxK9Izg68XHkqck4Rt
           DiVk37lNAQmgSlHbrGu0yOTkA== ) ; Key ID = 28809
   com.  900  IN  RRSIG  ( DNSKEY 13 1 900 20181128000000
           20151103000000 18931 com.
           lZmTBrfcRgVbqHJIfCVr6c3HUDgy3MlNSCSnrVV2S5/NmB3ZiFcvIDn0iqXPm
           7YQfvfWi6utyxBu/fSD6S1ARw== )
   com.  900  IN  RRSIG  ( DNSKEY 13 1 900 20181128000000
           20151103000000 28809 com.
           8qZOVM4X8wGt5XPWhG2HO4FAD6Kvs5eIhZUz+7DVCrZ/XMEVrMIHcm1Q+sq0s
           hm4cSivK2BxOO24PHJXoZN2Lw== )
   com.  86400  IN  DS  ( 18931 13 2 20f7a9db42d0e2042fbbb9f9ea015941202
           f9eabb94487e658c188e7bcb52115 )
   com.  86400  IN  DS  ( 28809 13 2 ad66b3276f796223aa45eda773e92c6d98e
           70643bbde681db342a9e5cf2bb380 )
   com.  86400  IN  RRSIG  ( DS 13 1 86400 20181128000000
           20151103000000 31918 .
           5KQVa0NP+6k7VEGMmeky2/Y3wIGM70Fkm0vp5NmQ6KPk8L1XMJPltcJDWGGjc
           EU3Uc4z2DUxzZyWgEDdrSOcdw== )
```

Authors' Addresses

Melinda Shore
Fastly

EMail: mshore@fastly.com

Richard Barnes
Mozilla

EMail: rlb@ipv.sx

Shumon Huque
Salesforce

EMail: shuque@gmail.com

Willem Toorop
NLnet Labs

EMail: willem@nlnetlabs.nl

TLS
Internet-Draft
Updates: 6347 (if approved)
Intended status: Standards Track
Expires: May 6, 2021

E. Rescorla, Ed.
RTFM, Inc.
H. Tschofenig, Ed.
T. Fossati
Arm Limited

November 02, 2020

Connection Identifiers for DTLS 1.2
draft-ietf-tls-dtls-connection-id-08

Abstract

This document specifies the Connection ID (CID) construct for the Datagram Transport Layer Security (DTLS) protocol version 1.2.

A CID is an identifier carried in the record layer header that gives the recipient additional information for selecting the appropriate security association. In "classical" DTLS, selecting a security association of an incoming DTLS record is accomplished with the help of the 5-tuple. If the source IP address and/or source port changes during the lifetime of an ongoing DTLS session then the receiver will be unable to locate the correct security context.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

The Datagram Transport Layer Security (DTLS) [RFC6347] protocol was designed for securing connection-less transports, like UDP. DTLS, like TLS, starts with a handshake, which can be computationally demanding (particularly when public key cryptography is used). After a successful handshake, symmetric key cryptography is used to apply data origin authentication, integrity and confidentiality protection. This two-step approach allows endpoints to amortize the cost of the initial handshake across subsequent application data protection. Ideally, the second phase where application data is protected lasts over a long period of time since the established keys will only need to be updated once the key lifetime expires.

In DTLS as specified in RFC 6347, the IP address and port of the peer are used to identify the DTLS association. Unfortunately, in some cases, such as NAT rebinding, these values are insufficient. This is a particular issue in the Internet of Things when devices enter extended sleep periods to increase their battery lifetime. The NAT rebinding leads to connection failure, with the resulting cost of a new handshake.

This document defines an extension to DTLS 1.2 to add a CID to the DTLS record layer. The presence of the CID is negotiated via a DTLS extension.

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].

This document assumes familiarity with DTLS 1.2 [RFC6347].

3. The "connection_id" Extension

This document defines the "connection_id" extension, which is used in ClientHello and ServerHello messages.

The extension type is specified as follows.

```
  enum {
     connection_id(TBD1), (65535)
  } ExtensionType;
```

The extension_data field of this extension, when included in the ClientHello, MUST contain the ConnectionId structure. This structure contains the CID value the client wishes the server to use when sending messages to the client. A zero-length CID value indicates that the client is prepared to send with a CID but does not wish the server to use one when sending. Alternatively, this can be interpreted as the client wishes the server to use a zero-length CID; the result is the same.

```
  struct {
      opaque cid<0..2^8-1>;
  } ConnectionId;
```

A server willing to use CIDs will respond with a "connection_id" extension in the ServerHello, containing the CID it wishes the client to use when sending messages towards it. A zero-length value indicates that the server will send with the client's CID but does not wish the client to include a CID (or again, alternately, to use a zero-length CID).

Because each party sends the value in the "connection_id" extension it wants to receive as a CID in encrypted records, it is possible for an endpoint to use a globally constant length for such connection identifiers. This can in turn ease parsing and connection lookup, for example by having the length in question be a compile-time constant. Such implementations MUST still be able to send CIDs of different length to other parties. Implementations that want to use variable-length CIDs are responsible for constructing the CID in such a way that its length can be determined on reception. Note that there is no CID length information included in the record itself.

In DTLS 1.2, CIDs are exchanged at the beginning of the DTLS session only. There is no dedicated "CID update" message that allows new CIDs to be established mid-session, because DTLS 1.2 in general does not allow TLS 1.3-style post-handshake messages that do not themselves begin other handshakes. When a DTLS session is resumed or renegotiated, the "connection_id" extension is negotiated afresh.

If DTLS peers have not negotiated the use of CIDs then the RFC 6347-defined record format and content type MUST be used.

If DTLS peers have negotiated the use of a CIDs using the ClientHello and the ServerHello messages then the peers need to take the following steps.

The DTLS peers determine whether incoming and outgoing messages need to use the new record format, i.e., the record format containing the CID. The new record format with the the tls12_cid content type is only used once encryption is enabled. Plaintext payloads never use the new record type and the CID content type.

For sending, if a zero-length CID has been negotiated then the RFC 6347-defined record format and content type MUST be used (see Section 4.1 of [RFC6347]) else the new record layer format with the tls12_cid content type defined in Figure 3 MUST be used.

When transmitting a datagram with the tls12_cid content type, the new MAC computation defined in Section 5 MUST be used.

For receiving, if the tls12_cid content type is set, then the CID is used to look up the connection and the security association. If the tls12_cid content type is not set, then the connection and security association is looked up by the 5-tuple and a check MUST be made to determine whether the expected CID value is indeed zero length. If the check fails, then the datagram MUST be dropped.

When receiving a datagram with the tls12_cid content type, the new MAC computation defined in Section 5 MUST be used. When receiving a datagram with the RFC 6347-defined record format the MAC calculation defined in Section 4.1.2 of [RFC6347] MUST be used.

4. Record Layer Extensions

This specification defines the DTLS 1.2 record layer format and [I-D.ietf-tls-dtls13] specifies how to carry the CID in DTLS 1.3.

To allow a receiver to determine whether a record has a CID or not, connections which have negotiated this extension use a distinguished record type tls12_cid(TBD2). Use of this content type has the following three implications:

- The CID field is present and contains one or more bytes.

- The MAC calculation follows the process described in Section 5.

- The true content type is inside the encryption envelope, as described below.

Plaintext records are not impacted by this extension. Hence, the format of the DTLSPlaintext structure is left unchanged, as shown in Figure 1.

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

Figure 1: DTLS 1.2 Plaintext Record Payload.

When CIDs are being used, the content to be sent is first wrapped along with its content type and optional padding into a DTLSInnerPlaintext structure. This newly introduced structure is shown in Figure 2. The DTLSInnerPlaintext byte sequence is then encrypted. To create the DTLSCiphertext structure shown in Figure 3 the CID is added.

```
        struct {
            opaque content[length];
            ContentType real_type;
            uint8 zeros[length_of_padding];
        } DTLSInnerPlaintext;
```

Figure 2: New DTLSInnerPlaintext Payload Structure.

content Corresponds to the fragment of a given length.

real_type The content type describing the payload.

zeros An arbitrary-length run of zero-valued bytes may appear in the cleartext after the type field. This provides an opportunity for senders to pad any DTLS record by a chosen amount as long as the total stays within record size limits. See Section 5.4 of [RFC8446] for more details. (Note that the term TLSInnerPlaintext in RFC 8446 refers to DTLSInnerPlaintext in this specification.)

```
        struct {
            ContentType outer_type = tls12_cid;
            ProtocolVersion version;
            uint16 epoch;
            uint48 sequence_number;
            opaque cid[cid_length];               // New field
            uint16 length;
            opaque enc_content[DTLSCiphertext.length];
        } DTLSCiphertext;
```

Figure 3: DTLS 1.2 CID-enhanced Ciphertext Record.

outer_type The outer content type of a DTLSCiphertext record carrying a CID is always set to tls12_cid(TBD2). The real content type of the record is found in DTLSInnerPlaintext.real_type after decryption.

cid The CID value, cid_length bytes long, as agreed at the time the extension has been negotiated. Recall that (as discussed previously) each peer chooses the CID value it will receive and use to identify the connection, so an implementation can choose to always recieve CIDs of a fixed length. If, however, an implementation chooses to receive different lengths of CID, the assigned CID values must be self-delineating since there is no other mechanism available to determine what connection (and thus, what CID length) is in use.

enc_content The encrypted form of the serialized DTLSInnerPlaintext structure.

All other fields are as defined in RFC 6347.

5. Record Payload Protection

Several types of ciphers have been defined for use with TLS and DTLS and the MAC calculations for those ciphers differ slightly.

This specification modifies the MAC calculation as defined in [RFC6347] and [RFC7366], as well as the definition of the additional data used with AEAD ciphers provided in [RFC6347], for records with content type tls12_cid. The modified algorithm MUST NOT be applied to records that do not carry a CID, i.e., records with content type other than tls12_cid.

The following fields are defined in this document; all other fields are as defined in the cited documents.

cid Value of the negotiated CID (variable length).

cid_length 1 byte field indicating the length of the negotiated CID.

length_of_DTLSInnerPlaintext The length (in bytes) of the serialised DTLSInnerPlaintext (two-byte integer).
The length MUST NOT exceed 2^14.

Note "+" denotes concatenation.

5.1. Block Ciphers

The following MAC algorithm applies to block ciphers that do not use the with Encrypt-then-MAC processing described in [RFC7366].

```
    MAC(MAC_write_key, seq_num +
        tls12_cid +
        DTLSCiphertext.version +
        cid +
        cid_length +
        length_of_DTLSInnerPlaintext +
        DTLSInnerPlaintext.content +
        DTLSInnerPlaintext.real_type +
        DTLSInnerPlaintext.zeros
    )
```

5.2. Block Ciphers with Encrypt-then-MAC processing

The following MAC algorithm applies to block ciphers that use the with Encrypt-then-MAC processing described in [RFC7366].

```
    MAC(MAC_write_key, seq_num +
        tls12_cid +
        DTLSCipherText.version +
        cid +
        cid_length +
        length of (IV + DTLSCiphertext.enc_content) +
        IV +
        DTLSCiphertext.enc_content);
```

5.3. AEAD Ciphers

For ciphers utilizing authenticated encryption with additional data the following modification is made to the additional data calculation.

```
    additional_data = seq_num +
                      tls12_cid +
                      DTLSCipherText.version +
                      cid +
                      cid_length +
                      length_of_DTLSInnerPlaintext;
```

6. Peer Address Update

When a record with a CID is received that has a source address different than the one currently associated with the DTLS connection, the receiver MUST NOT replace the address it uses for sending records to its peer with the source address specified in the received datagram unless the following three conditions are met:

- The received datagram has been cryptographically verified using the DTLS record layer processing procedures.

- The received datagram is "newer" (in terms of both epoch and sequence number) than the newest datagram received. Reordered datagrams that are sent prior to a change in a peer address might otherwise cause a valid address change to be reverted. This also limits the ability of an attacker to use replayed datagrams to force a spurious address change, which could result in denial of service. An attacker might be able to succeed in changing a peer address if they are able to rewrite source addresses and if replayed packets are able to arrive before any original.

- There is a strategy for ensuring that the new peer address is able to receive and process DTLS records. No such test is defined in this specification.

The conditions above are necessary to protect against attacks that use datagrams with spoofed addresses or replayed datagrams to trigger attacks. Note that there is no requirement for use of the anti-replay window mechanism defined in Section 4.1.2.6 of DTLS 1.2. Both solutions, the "anti-replay window" or "newer" algorithm, will prevent address updates from replay attacks while the latter will only apply to peer address updates and the former applies to any application layer traffic.

Note that datagrams that pass the DTLS cryptographic verification procedures but do not trigger a change of peer address are still valid DTLS records and are still to be passed to the application.

Application protocols that implement protection against these attacks depend on being aware of changes in peer addresses so that they can engage the necessary mechanisms. When delivered such an event, an application layer-specific address validation mechanism can be triggered, for example one that is based on successful exchange of a minimal amount of ping-pong traffic with the peer. Alternatively, an DTLS-specific mechanism may be used, as described in [I-D.tschofenig-tls-dtls-rrc].

DTLS implementations MUST silently discard records with bad MACs or that are otherwise invalid.

7. Examples

Figure 4 shows an example exchange where a CID is used uni-directionally from the client to the server. To indicate that a zero-length CID is present in the "connection_id" extension we use the notation 'connection_id=empty'.

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

   ClientHello                 -------->
   (connection_id=empty)


                               <--------      HelloVerifyRequest
                                                        (cookie)

   ClientHello                 -------->
   (connection_id=empty)
   (cookie)

                                                     ServerHello
                                             (connection_id=100)
                                                     Certificate
                                               ServerKeyExchange
                                              CertificateRequest
                               <--------         ServerHelloDone

   Certificate
   ClientKeyExchange
   CertificateVerify
   [ChangeCipherSpec]
   Finished                    -------->
   <CID=100>

                                              [ChangeCipherSpec]
                               <--------                Finished


   Application Data            ========>
   <CID=100>

                               <========        Application Data

   Legend:

   <...> indicates that a connection id is used in the record layer
   (...) indicates an extension
   [...] indicates a payload other than a handshake message
```

Figure 4: Example DTLS 1.2 Exchange with CID

Note: In the example exchange the CID is included in the record layer once encryption is enabled. In DTLS 1.2 only one handshake message is encrypted, namely the Finished message. Since the example shows how to use the CID for payloads sent from the client to the server, only the record layer payloads containing the Finished message or application data include a CID.

8. Privacy Considerations

The CID replaces the previously used 5-tuple and, as such, introduces an identifier that remains persistent during the lifetime of a DTLS connection. Every identifier introduces the risk of linkability, as explained in [RFC6973].

An on-path adversary observing the DTLS protocol exchanges between the DTLS client and the DTLS server is able to link the observed payloads to all subsequent payloads carrying the same ID pair (for bi-directional communication). Without multi-homing or mobility, the use of the CID exposes the same information as the 5-tuple.

With multi-homing, a passive attacker is able to correlate the communication interaction over the two paths and the sequence number makes it possible to correlate packets across CID changes. The lack of a CID update mechanism in DTLS 1.2 makes this extension unsuitable for mobility scenarios where correlation must be considered. Deployments that use DTLS in multi-homing environments and are concerned about this aspects SHOULD refuse to use CIDs in DTLS 1.2 and switch to DTLS 1.3 where a CID update mechanism is provided and sequence number encryption is available.

The specification introduces record padding for the CID-enhanced record layer, which is a privacy feature not available with the original DTLS 1.2 specification. Padding allows to inflate the size of the ciphertext making traffic analysis more difficult. More details about record padding can be found in Section 5.4 and Appendix E.3 of RFC 8446.

Finally, endpoints can use the CID to attach arbitrary per-connection metadata to each record they receive on a given connection. This may be used as a mechanism to communicate per-connection information to on-path observers. There is no straightforward way to address this concern with CIDs that contain arbitrary values. Implementations concerned about this aspect SHOULD refuse to use CIDs.

9. Security Considerations

An on-path adversary can create reflection attacks 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.

Additionally, an attacker able to observe the data traffic exchanged between two DTLS peers is able to replay datagrams with modified IP address/port numbers.

The topic of peer address updates is discussed in Section 6.

10. IANA Considerations

IANA is requested to allocate an entry to the existing TLS "ExtensionType Values" registry, defined in [RFC5246], for connection_id(TBD1) as described in the table below. IANA is requested to add an extra column to the TLS ExtensionType Values registry to indicate whether an extension is only applicable to DTLS and to include this document as an additional reference for the registry.

Value	Extension Name	TLS 1.3	DTLS Only	Recommended	Reference
TBD1	connection_id	CH, SH	Y	N	[[This doc]]

Note: The value "N" in the Recommended column is set because this extension is intended only for specific use cases. This document describes the behavior of this extension for DTLS 1.2 only; it is not applicable to TLS, and its usage for DTLS 1.3 is described in [I-D.ietf-tls-dtls13].

IANA is requested to allocate tls12_cid(TBD2) in the "TLS ContentType Registry". The tls12_cid ContentType is only applicable to DTLS 1.2.

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>.

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

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

[RFC7366] Gutmann, P., "Encrypt-then-MAC for Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)", RFC 7366, DOI 10.17487/RFC7366, September 2014, <https://www.rfc-editor.org/info/rfc7366>.

[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

[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-dtls13-38 (work in progress), May 2020.

[I-D.tschofenig-tls-dtls-rrc]
Fossati, T. and H. Tschofenig, "Return Routability Check for DTLS 1.2 and DTLS 1.3", draft-tschofenig-tls-dtls-rrc-01 (work in progress), March 2020.

[RFC6973] Cooper, A., Tschofenig, H., Aboba, B., Peterson, J., Morris, J., Hansen, M., and R. Smith, "Privacy Considerations for Internet Protocols", RFC 6973, DOI 10.17487/RFC6973, July 2013, <https://www.rfc-editor.org/info/rfc6973>.

11.3. 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

draft-ietf-tls-dtls-connection-id-08

- RRC draft moved from normative to informative.

draft-ietf-tls-dtls-connection-id-07

- Wording changes in the security and privacy consideration and the peer address update sections.

draft-ietf-tls-dtls-connection-id-06

- Updated IANA considerations

- Enhanced security consideration section to describe a potential man-in-the-middle attack concerning address validation.

draft-ietf-tls-dtls-connection-id-05

- Restructed Section 5 "Record Payload Protection"

draft-ietf-tls-dtls-connection-id-04

- Editorial simplifications to the ’Record Layer Extensions’ and the ’Record Payload Protection’ sections.

- Added MAC calculations for block ciphers with and without Encrypt-then-MAC processing.

draft-ietf-tls-dtls-connection-id-03

- Updated list of contributors

- Updated list of contributors and acknowledgements

- Updated example

- Changed record layer design

- Changed record payload protection

- Updated introduction and security consideration section

- Author- and affiliation changes

draft-ietf-tls-dtls-connection-id-02

- Move to internal content types a la DTLS 1.3.

draft-ietf-tls-dtls-connection-id-01

- Remove 1.3 based on the WG consensus at IETF 101

draft-ietf-tls-dtls-connection-id-00

- Initial working group version (containing a solution for DTLS 1.2 and 1.3)

draft-rescorla-tls-dtls-connection-id-00

- Initial version

Appendix B. Working Group Information

RFC EDITOR: PLEASE REMOVE THE THIS SECTION

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

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

Appendix C. Contributors

Many people have contributed to this specification and we would like to thank the following individuals for their contributions:

* Yin Xinxing
  Huawei
  yinxinxing@huawei.com

* Nikos Mavrogiannopoulos
  RedHat
  nmav@redhat.com

* Tobias Gondrom
  tobias.gondrom@gondrom.org

Additionally, we would like to thank the Connection ID task force team members:

- Martin Thomson (Mozilla)

- Christian Huitema (Private Octopus Inc.)

- Jana Iyengar (Google)

- Daniel Kahn Gillmor (ACLU)

- Patrick McManus (Mozilla)

- Ian Swett (Google)

- Mark Nottingham (Fastly)

The task force team discussed various design ideas, including cryptographically generated session ids using hash chains and public key encryption, but dismissed them due to their inefficiency. The approach described in this specification is the simplest possible design that works given the limitations of DTLS 1.2. DTLS 1.3 provides better privacy features and developers are encouraged to switch to the new version of DTLS.

Finally, we want to thank the IETF TLS working group chairs, Chris Wood, Joseph Salowey, and Sean Turner, for their patience, support and feedback.

Appendix D. Acknowledgements

We would like to thank Achim Kraus for his review comments and implementation feedback.

Authors' Addresses

Eric Rescorla (editor)
RTFM, Inc.

EMail: ekr@rtfm.com

Hannes Tschofenig (editor)
Arm Limited

EMail: hannes.tschofenig@arm.com

Thomas Fossati
Arm Limited

EMail: thomas.fossati@arm.com

TLS
Internet-Draft
Obsoletes: 6347 (if approved)
Intended status: Standards Track
Expires: May 6, 2021

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

November 02, 2020

The Datagram Transport Layer Security (DTLS) Protocol Version 1.3
draft-ietf-tls-dtls13-39

Abstract

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

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

Status of This Memo

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

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

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

This Internet-Draft will expire on May 6, 2021.

Copyright Notice

Table of Contents

1. Introduction

RFC EDITOR: PLEASE REMOVE THE FOLLOWING PARAGRAPH

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

The primary goal of the TLS protocol is to 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 based on TLS 1.3 [TLS13].

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

2. Conventions and Terminology

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

The following terms are used:

- client: The endpoint initiating the DTLS connection.

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

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

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

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

- receiver: An endpoint that is receiving records.

- sender: An endpoint that is transmitting records.

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

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

- CID: Connection ID

- MSL: Maximum Segment Lifetime

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

The reader is also as 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.

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

1. TLS relies on an implicit sequence number on records. If a record is not received, then the recipient will use the wrong sequence number when attempting to remove record protection from subsequent records. DTLS solves this problem by adding sequence numbers.

2. The TLS handshake is a lock-step cryptographic handshake. Messages must be transmitted and received in a defined order; any other order is an error. DTLS handshake messages are also assigned sequence numbers to enable reassembly in the correct order in case datagrams are lost or reordered.

3. During the handshake, messages are implicitly acknowledged by other handshake messages. 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 that uses the TLS HelloRetryRequest message (see Section 5.1 for details).

3.1. Packet Loss

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

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

                       X<-- HelloRetryRequest
                                         (lost)

[Timer Expires]

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

Figure 1: DTLS retransmission example

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

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

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

3.2. Reordering

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

3.3. Message Size

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

3.4. 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 Format

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

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

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

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

Figure 3: DTLS 1.3 CipherText Header

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

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

S: The S bit (0x08) indicates the size of the sequence number. 0 means an 8-bit sequence number, 1 means 16-bit.

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

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

Connection ID: Variable length CID. The CID 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 layer header types.

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

    DTLSPlaintext
      Structure
```

Figure 4: Header Examples

The length field MAY be omitted 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.

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) is used as 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. Determining the Header Format

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

- If the first byte is alert(21), handshake(22), or ack(proposed, 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.

4.2. Sequence Number and Epoch

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

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

4.2.1. Processing Guidelines

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

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

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

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

4.2.3. Sequence Number Encryption

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

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

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

When the AEAD is based on ChaCha20, then the mask is generated by treating the first 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].

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

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

Note that sequence number encryption is only applied to the 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 PMTU estimates obtained from the record layer.

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

DTLS records 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.

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

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

4.4. PMTU Issues

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

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

- In some implementations, the application may not directly talk to the network, in which case the DTLS stack may absorb ICMP

[RFC1191] "Datagram Too Big" indications or ICMPv6 [RFC4443] "Packet Too Big" indications.

- The DTLS handshake messages can exceed the PMTU.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

4.5. Record Payload Protection

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

4.5.1. Anti-Replay

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

The received record counter for a session MUST be initialized to zero when that session is established. For each received record, the receiver MUST verify that the record contains a sequence number that does not duplicate the sequence number of any other record received during the lifetime of the session. This check SHOULD happen after deprotecting the record; otherwise the record discard might itself serve as a timing channel for the record number. Note that decompressing the records number is still a potential timing channel for the record number, though a less powerful one than whether it was deprotected.

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

The "right" edge of the window represents the highest validated sequence number value received on the session. Records that contain sequence numbers lower than the "left" edge of the window are rejected. 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 error alerts to avoid attacks where the attacker repeatedly probes the implementation to see how it responds to various types of error. Note that if DTLS is run over UDP, then any implementation which does this will be extremely susceptible to denial-of-service (DoS) attacks because UDP forgery is so easy. Thus, this practice is NOT RECOMMENDED for such transports, both to increase the reliability of DTLS service and to avoid the risk of spoofing attacks sending traffic to unrelated third parties.

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

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_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_SHA256 MUST NOT 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 packets that can fail authentication. 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 "session_id" value from the client and endpoints MUST NOT send ChangeCipherSpec messages.

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

5.1. Denial-of-Service Countermeasures

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

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

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

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

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

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

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

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

                            <----- HelloRetryRequest
                                    + cookie

      ClientHello           ------>
       + cookie

       [Rest of handshake]
```

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

The cookie extension is defined in Section 4.2.2 of [TLS13]. When sending the initial ClientHello, the client does not have a cookie yet. In this case, the cookie extension is omitted and the legacy_cookie field in the ClientHello message MUST be set to a zero length vector (i.e., a single zero 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 transcript of the HelloRetryRequest to be stored in the cookie or the internal state of the hash algorithm, since only the hash of the transcript is required for the handshake to complete.

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

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

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

DTLS servers SHOULD perform a cookie exchange whenever a new handshake is being performed. If the server is being operated in an environment where amplification is not a problem, the server MAY be configured not to perform a cookie exchange. The default SHOULD be that the exchange is performed, however. In addition, the server MAY choose not to do a cookie exchange when a session is resumed or, more generically, when the DTLS handshake uses a PSK-based key exchange.

Clients MUST be prepared to do a cookie exchange with every handshake.

If a server receives a ClientHello with an invalid cookie, it MUST NOT terminate the handshake with an "illegal_parameter" alert. This allows the client to restart the connection from scratch without a cookie.

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

5.2. DTLS Handshake Message Format

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

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

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

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 implements MUST NOT use the HelloVerifyRequest to execute a return-routability check. A dual-stack DTLS 1.2/DTLS 1.3 client MUST, however, be prepared to interact with a DTLS 1.2 server.

5.3. ClientHello Message

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

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

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

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

legacy_version: In previous versions of DTLS, this field was used for version negotiation and represented the highest version number supported by the client. Experience has shown that many servers do not properly implement version negotiation, leading to "version intolerance" in which the server rejects an otherwise acceptable ClientHello with a version number higher than it supports. In DTLS 1.3, the client indicates its version preferences in the "supported_versions" extension (see Section 4.2.1 of [TLS13]) and the legacy_version field MUST be set to {254, 253}, which was the version number for DTLS 1.2. The version fields for DTLS 1.0 and DTLS 1.2 are 0xfeff and 0xfefd (to match the wire versions) but the version field for DTLS 1.3 is 0x0304.

random: Same as for TLS 1.3.

legacy_session_id: Same as for TLS 1.3.

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

cipher_suites: Same as for TLS 1.3.

legacy_compression_methods: Same as for TLS 1.3.

extensions: Same as for TLS 1.3.

5.4. Handshake Message Fragmentation and Reassembly

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

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

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

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

5.5. End Of Early Data

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

5.6. DTLS Handshake Flights

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

```
Client                                             Server

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

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


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

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


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

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

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

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

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


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

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

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

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

```
  Client                                            Server

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

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


                                                            +-----------+
   {Finished}            -------->                          | Flight 3  |
   [Application Data*]                                      +-----------+

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

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

           Figure 8: Message flights for the Zero-RTT handshake

```
  Client                                            Server

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

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

       Figure 9: Message flights for the new session ticket message

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

5.7. Timeout and Retransmission

5.7.1. State Machine

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

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

             Server read retransmit
                 Retransmit ACK
```

Figure 10: DTLS timeout and retransmission state machine

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

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

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

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

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

peer and therefore suggests that part of one's previous flight was lost.

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

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

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

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

5.7.2. Timer Values

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

Implementations SHOULD retain the current timer value until a transmission without loss occurs, at which time the value may be reset to the initial value. After a long period of idleness, no less than 10 times the current timer value, implementations may reset the timer to the initial value.

5.7.3. 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 5.7.1. 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 have send KeyUpdate, NewConnectionId or RequestConnectionId message 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.

5.8. CertificateVerify and Finished Messages

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

5.9. Cryptographic Label Prefix

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

5.10. Alert Messages

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

5.11. Establishing New Associations with Existing Parameters

If a DTLS client-server pair is configured in such a way that repeated connections happen on the same host/port quartet, then it is possible that a client will silently abandon one connection and then initiate another with the same parameters (e.g., after a reboot). This will appear to the server as a new handshake with epoch=0. In cases where a server believes it has an existing association on a given host/port quartet and it receives an epoch=0 ClientHello, it

SHOULD proceed with a new handshake but MUST NOT destroy the existing association until the client has demonstrated reachability either by completing a cookie exchange or by completing a complete handshake including delivering a verifiable Finished message. After a correct Finished message is received, the server MUST abandon the previous association to avoid confusion between two valid associations with overlapping epochs. The reachability requirement prevents off-path/blind attackers from destroying associations merely by sending forged ClientHellos.

Note: it is not always possible to distinguish which association a given 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.

6. Example of Handshake with Timeout and Retransmission

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

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

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

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


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

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


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

    Record 2                  -------->
    Certificate
    (message_seq=1)
    CertificateVerify
    (message_seq=2)
    Finished
    (message_seq=3)

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

Figure 11: Example DTLS exchange illustrating message loss

6.1. Epoch Values and Rekeying

A recipient of a DTLS message needs to select the correct keying material in order to process an incoming message. With the possibility of message loss and re-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 payload with an epoch value for which no corresponding cipher state can be determined MUST generate a "unexpected_message" alert. For example, if a client incorrectly uses epoch value 5 when sending early application data in a 0-RTT exchange. A server will not be able to compute the appropriate keys and will therefore have to respond with an alert.

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

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

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

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

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

                            <--------             ServerHello
                                          [HelloRetryRequest]
                                                    (epoch=0)

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

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

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

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

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

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

```
                             Some time later ...
                      (Post-Handshake Message Exchange)

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

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

                             Some time later ...
                                (Rekeying)

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

Figure 12: Example DTLS exchange with epoch information

7. ACK Message

The ACK message is used by an endpoint to indicate 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). Thus, an ACK from the server would not cover both the ClientHello and the client’s Certificate. Implementations can accomplish this by clearing their ACK list upon receiving the start of the next flight.

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 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). A reasonable approach here is to set a timer for 1/4 the current retransmit timer value when the first record in the flight is received and then send an ACK when that timer expires.

In 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

2. The server’s post-handshake CertificateRequest.

ACKs SHOULD NOT be sent for these flights unless generating the responding flight takes significant time. 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 post-handshake 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 update_requested does not implicitly acknowledge that message because the KeyUpdates 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.

7.3. Design Rational

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 retransmission, as opposed to the whole flight retransmission in previous versions of DTLS. For instance in the flow shown in Figure 11 if the client does not send the ACK message when it received and processed record 1 indicating loss of record 0, the entire flight would be retransmitted. When DTLS 1.3 is used in deployments with loss 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 12, acknowledges receipt and processing of record 2 (containing the NewSessionTicket message) and if it is not sent the server will continue retransmission of the NewSessionTicket indefinitely.

8. Key Updates

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

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

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.

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. Endpoints MUST NOT have more than one NewConnectionId message outstanding.

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

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

num_cids The number of CIDs desired.

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

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

9.1. Connection ID Example

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

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

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

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


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

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

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

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

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

Figure 13: Example DTLS 1.3 Exchange with CIDs

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

10. Application Data Protocol

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

11. Security Considerations

Security issues are discussed primarily in [TLS13].

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

DTLS implementations MUST NOT update their sending address in response to packets from a different address unless they first perform some reachability test; no such test is defined in this specification. Even with such a test, an on-path adversary can also black-hole traffic or create a reflection attack against third parties because a DTLS peer has no means to distinguish a genuine address update event (for example, due to a NAT rebinding) from one that is malicious. This attack is of concern when there is a large asymmetry of request/response message sizes.

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

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

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

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

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

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

- Switching CID based on certain events, or even regularly, helps against tracking by on-path adversaries but the sequence numbers can still allow linkability. For this reason this specification defines an algorithm for encrypting sequence numbers, see Section 4.2.3. Note that sequence number encryption is used for all encrypted DTLS 1.3 records irrespective of whether a CID is used or not. Unlike the sequence number, the epoch is not encrypted. This 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 to DTLS 1.2

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

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

- Only AEAD ciphers are supported. Additional data calculation has been simplified.

- Removed support for weaker and older cryptographic algorithms

- HelloRetryRequest of TLS 1.3 used instead of HelloVerifyRequest

- More flexible ciphersuite negotiation

- New session resumption mechanism

- PSK authentication redefined

- New key derivation hierarchy utilizing a new key derivation construct

- Improved version negotiation

- Optimized record layer encoding and thereby its size

- Added CID functionality

- Sequence numbers are encrypted.

13. IANA Considerations

IANA is requested to allocate a new value in the "TLS ContentType" registry for the ACK message, defined in Section 7, with content type 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 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".

14. References

14.1. 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., and T. Fossati, "Connection Identifiers for DTLS 1.2", draft-ietf-tls-dtls-connection-id-07 (work in progress), October 2019.

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

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

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

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

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

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

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

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

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

14.2. Informative References

[AEBounds]
Luykx, A. and K. Paterson, "Limits on Authenticated Encryption Use in TLS", 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.

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

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

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

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

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

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

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

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

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

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

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

[ROBUST] Fischlin, M., Guenther, F., and C. Janson, "Robust Channels: Handling Unreliable Networks in the Record Layers of QUIC and DTLS 1.3", June 2020, <https://eprint.iacr.org/2020/718>.

14.3. URIs

[1] mailto:tls@ietf.org

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

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

Appendix A. Protocol Data Structures and Constant Values

This section provides the normative protocol types and constants definitions.

%%## Record Layer %%## Handshake Protocol %%## ACKs %%## Connection ID Management

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.

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, 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 <= 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 <= 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 <= 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. History

RFC EDITOR: PLEASE REMOVE THE THIS SECTION

IETF Drafts

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 andno 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 D. 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 [1]. Information on the group and information on how to subscribe to the list is at https://www1.ietf.org/mailman/listinfo/tls [2]

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

Appendix E. 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 Guenther
  ETH Zurich
  mail@felixguenther.info

* 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 F. Acknowledgements

We would like to thank Jonathan Hammell for his review comments.

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: December 28, 2020

N. Sullivan
Cloudflare Inc.
June 26, 2020

Exported Authenticators in TLS
draft-ietf-tls-exported-authenticator-13

Abstract

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

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

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

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

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

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

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

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

Exported Authenticators are meant to be used as a building block for application protocols. Mechanisms such as those required to advertise support and handle authentication errors are not handled at the TLS layer.

TLS (or DTLS) version 1.2 or later are REQUIRED to implement the mechanisms described in this document.

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.

3. Message Sequences

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

Client Authentication

- Server generates Authenticator Request

- Client generates Authenticator from Server’s Authenticator Request

- Server validates Client’s Authenticator

Server Authentication

- Client generates Authenticator Request
- Server generates Authenticator from Client’s Authenticator Request
- Client validates Server’s Authenticator

Spontaneous Server Authentication

- Server generates Authenticator
- Client validates Server’s Authenticator

4. Authenticator Request

The authenticator request is a structured message that can be created by either party of a TLS connection using data exported from that connection. It can be transmitted to the other party of the TLS connection at the application layer. The application layer protocol used to send the authenticator request SHOULD use TLS as its underlying transport to keep the request confidential. The application MAY use the existing TLS connection to transport the authenticator.

An authenticator request message can be constructed by either the client or the server. Server-generated authenticator requests use the CertificateRequest message from Section 4.3.2 of [TLS13]. Client-generated authenticator requests use a new message, called the ClientCertificateRequest, which uses the same structure as CertificateRequest. These messages structures are used even if the TLS connection protocol is TLS 1.2.

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

The structures are defined to be:

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

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

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

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

The uniqueness requirements of the certificate_request_context apply only to CertificateRequest and ClientCertificateRequest messages that are used as part of authenticator requests. There is no impact if the value of a certificate_request_context used in an authenticator request matches the value of a certificate_request_context in the handshake or in a post-handshake message.

5. Authenticator

The authenticator is a structured message that can be exported from either party of a TLS connection. It can be transmitted to the other party of the TLS connection at the application layer. The application layer protocol used to send the authenticator SHOULD use TLS or a protocol with comparable security properties as its underlying transport to keep the Certificate confidential. The application MAY use the existing TLS connection to transport the authenticator.

An authenticator message can be constructed by either the client or the server given an established TLS connection, an identity, such as an X.509 certificate, and a corresponding private key. Clients MUST

NOT send an authenticator without a preceding authenticator request; for servers an authenticator request is optional. For authenticators that do not correspond to authenticator requests, the certificate_request_context is chosen by the server.

5.1. Authenticator Keys

Each authenticator is computed using a Handshake Context and Finished MAC Key derived from the TLS connection. These values are derived using an exporter as described in [RFC5705] (for TLS 1.2) or Sec. 7.5 of [TLS13] (for TLS 1.3). For TLS 1.3, the exporter_master_secret MUST be used, not the early_exporter_master_secret. These values use different labels depending on the role of the sender:

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

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

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

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

5.2. Authenticator Construction

An authenticator is formed from the concatenation of TLS 1.3 [TLS13] Certificate, CertificateVerify, and Finished messages. These messages are encoded as TLS handshake messages, including length and type fields.

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

ClientHello extensions are used to determine permissible extensions in the Certificate message. This follows the general model for extensions in TLS in which extensions can only be included as part of a Certificate message if they were previously sent as part of a CertificateRequest message or ClientHello message, to ensure that the recipient will be able to process such extensions.

5.2.1. Certificate

The Certificate message contains the identity to be used for authentication, such as the end-entity certificate and any supporting certificates in the chain. This structure is defined in [TLS13], Section 4.4.2.

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

Certificates chosen in the Certificate message MUST conform to the requirements of a Certificate message in the negotiated version of TLS. In particular, the certificate chain MUST be valid for the a signature algorithms indicated by the peer in the "signature_algorithms" and "signature_algorithms_cert" extension, as described in Section 4.2.3 of [TLS13] for TLS 1.3 or the "signature_algorithms" extension from Sections 7.4.2 and 7.4.6 of [RFC5246] for TLS 1.2.

In addition to "signature_algorithms" and "signature_algorithms_cert", the "server_name" [RFC6066], "certificate_authorities" (Section 4.2.4. of [TLS13]), and "oid_filters" (Section 4.2.5. of [TLS13]) extensions are used to guide certificate selection.

Only the X.509 certificate type defined in [TLS13] is supported. Alternative certificate formats such as [RFC7250] Raw Public Keys are not supported in this version of the specification and their use in this context has not yet been analysed.

If an authenticator request was provided, the Certificate message MUST contain only extensions present in the authenticator request. Otherwise, the Certificate message MUST contain only extensions present in the TLS handshake. Unrecognized extensions in the authenticator request MUST be ignored.

5.2.2. CertificateVerify

This message is used to provide explicit proof that an endpoint possesses the private key corresponding to its identity. The definition for TLS 1.3 is:

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

The algorithm field specifies the signature algorithm used (see Section 4.2.3 of [TLS13] for the definition of this field). The signature is a digital signature using that algorithm.

The signature scheme MUST be a valid signature scheme for TLS 1.3. This excludes all RSASSA-PKCS1-v1_5 algorithms and combinations of ECDSA and hash algorithms that are not supported in TLS 1.3.

If an authenticator request is present, the signature algorithm MUST be chosen from one of the signature schemes present in the authenticator request. Otherwise, the signature algorithm used should be chosen from the "signature_algorithms" sent by the peer in the ClientHello of the TLS handshake. If there are no available signature algorithms, then no authenticator should be constructed.

The signature is computed using the chosen signature scheme over the concatenation of:

- A string that consists of octet 32 (0x20) repeated 64 times

- The context string "Exported Authenticator" (which is not NULL-terminated)

- A single 0 byte which serves as the separator

- The hashed authenticator transcript

The authenticator transcript is the hash of the concatenated Handshake Context, authenticator request (if present), and Certificate message:

```
Hash(Handshake Context || authenticator request || Certificate)
```

Where Hash is the authenticator hash defined in section 4.1. If the authenticator request is not present, it is omitted from this construction, i.e., it is zero-length.

If the party that generates the exported authenticator does so with a different connection than the party that is validating it, then the Handshake Context will not match, resulting in a CertificateVerify message that does not validate. This includes situations in which the application data is sent via TLS-terminating proxy. Given a failed CertificateVerify validation, it may be helpful for the application to confirm that both peers share the same connection using a value derived from the connection secrets before taking a user-visible action.

5.2.3. Finished

A HMAC [HMAC] over the hashed authenticator transcript, which is the concatenated Handshake Context, authenticator request (if present), Certificate, and CertificateVerify. The HMAC is computed using the authenticator hash, using the Finished MAC Key as a key.

```
Finished = HMAC(Finished MAC Key, Hash(Handshake Context ||
     authenticator request || Certificate || CertificateVerify))
```

5.2.4. Authenticator Creation

An endpoint constructs an authenticator by serializing the Certificate, CertificateVerify, and Finished as TLS handshake messages and concatenating the octets:

```
Certificate || CertificateVerify || Finished
```

An authenticator is valid if the CertificateVerify message is correctly constructed given the authenticator request (if used) and the Finished message matches the expected value. When validating an authenticator, a constant-time comparison SHOULD be used.

6. Empty Authenticator

If, given an authenticator request, the endpoint does not have an appropriate identity or does not want to return one, it constructs an authenticated refusal called an empty authenticator. This is a Finished message sent without a Certificate or CertificateVerify. This message is an HMAC over the hashed authenticator transcript with a Certificate message containing no CertificateEntries and the CertificateVerify message omitted. The HMAC is computed using the authenticator hash, using the Finished MAC Key as a key. This message is encoded as a TLS handshake message, including length and type field. It does not include TLS record layer framing.

```
Finished = HMAC(Finished MAC Key, Hash(Handshake Context ||
     authenticator request || Certificate))
```

7. API considerations

The creation and validation of both authenticator requests and authenticators SHOULD be implemented inside the TLS library even if it is possible to implement it at the application layer. TLS implementations supporting the use of exported authenticators SHOULD provide application programming interfaces by which clients and servers may request and verify exported authenticator messages.

Notwithstanding the success conditions described below, all APIs MUST fail if:

- the connection uses a TLS version of 1.1 or earlier, or
- the connection is TLS 1.2 and the extended master secret extension [RFC7627] was not negotiated

The following sections describes APIs that are considered necessary to implement exported authenticators. These are informative only.

7.1. The "request" API

The "request" API takes as input:

- certificate_request_context (from 0 to 255 bytes)
- set of extensions to include (this MUST include signature_algorithms)

It returns an authenticator request, which is a sequence of octets that comprises a CertificateRequest or ClientCertificateRequest message.

7.2. The "get context" API

The "get context" API takes as input:

- authenticator or authenticator request

It returns the certificate_request_context.

7.3. The "authenticate" API

The "authenticate" API takes as input:

- a reference to an active connection

- an identity, such as a set of certificate chains and associated extensions (OCSP, SCT, etc.)

- a signer (either the private key associated with the identity, or interface to perform private key operations) for each chain

- an authenticator request or certificate_request_context (from 0 to 255 bytes)

It returns either the exported authenticator or an empty authenticator as a sequence of octets. It is RECOMMENDED that the logic for selecting the certificates and extensions to include in the exporter is implemented in the TLS library. Implementing this in the TLS library lets the implementer take advantage of existing extension and certificate selection logic and more easily remember which extensions were sent in the ClientHello.

It is also possible to implement this API outside of the TLS library using TLS exporters. This may be preferable in cases where the application does not have access to a TLS library with these APIs or when TLS is handled independently of the application layer protocol.

7.4. The "validate" API

The "validate" API takes as input:

- a reference to an active connection

- an optional authenticator request

- an authenticator

- a function for validating a certificate chain

It returns the identity, such as the certificate chain and its extensions, and a status to indicate whether the authenticator is valid or not after applying the function for validating the certificate chain to the chain contained in the authenticator.

The API SHOULD return a failure if the certificate_request_context of the authenticator was used in a previously validated authenticator. Well-formed empty authenticators are returned as invalid.

8. IANA Considerations

8.1. Update of the TLS ExtensionType Registry

IANA is requested to update the entry for server_name(0) in the registry for ExtensionType (defined in [TLS13]) by replacing the value in the "TLS 1.3" column with the value "CH, EE, CR".

8.2. Update of the TLS Exporter Labels Registry

IANA is requested to add the following entries to the registry for Exporter Labels (defined in [RFC5705]): "EXPORTER-server authenticator handshake context", "EXPORTER-client authenticator finished key" and "EXPORTER-server authenticator finished key".

9. Security Considerations

The Certificate/Verify/Finished pattern intentionally looks like the TLS 1.3 pattern which now has been analyzed several times. For example, [SIGMAC] presents a relevant framework for analysis.

Authenticators are independent and unidirectional. There is no explicit state change inside TLS when an authenticator is either created or validated. The application in possession of a validated authenticator can rely on any semantics associated with data in the certificate_request_context.

- This property makes it difficult to formally prove that a server is jointly authoritative over multiple identities, rather than individually authoritative over each.

- There is no indication in the TLS layer about which point in time an authenticator was computed. Any feedback about the time of creation or validation of the authenticator should be tracked as part of the application layer semantics if required.

The signatures generated with this API cover the context string "Exported Authenticator" and therefore cannot be transplanted into other protocols.

10. Acknowledgements

Comments on this proposal were provided by Martin Thomson. Suggestions for Section 9 were provided by Karthikeyan Bhargavan.

11. References

11.1. Normative References

[HMAC] Krawczyk, H., Bellare, M., and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication", RFC 2104, DOI 10.17487/RFC2104, February 1997, <https://www.rfc-editor.org/info/rfc2104>.

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

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

[RFC5705] Rescorla, E., "Keying Material Exporters for Transport Layer Security (TLS)", RFC 5705, DOI 10.17487/RFC5705, March 2010, <https://www.rfc-editor.org/info/rfc5705>.

[RFC6066] Eastlake 3rd, D., "Transport Layer Security (TLS) Extensions: Extension Definitions", RFC 6066, DOI 10.17487/RFC6066, January 2011, <https://www.rfc-editor.org/info/rfc6066>.

[RFC7250] Wouters, P., Ed., Tschofenig, H., Ed., Gilmore, J., Weiler, S., and T. Kivinen, "Using Raw Public Keys in Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)", RFC 7250, DOI 10.17487/RFC7250, June 2014, <https://www.rfc-editor.org/info/rfc7250>.

[RFC7540] Belshe, M., Peon, R., and M. Thomson, Ed., "Hypertext Transfer Protocol Version 2 (HTTP/2)", RFC 7540, DOI 10.17487/RFC7540, May 2015, <https://www.rfc-editor.org/info/rfc7540>.

[RFC7627] Bhargavan, K., Ed., Delignat-Lavaud, A., Pironti, A., Langley, A., and M. Ray, "Transport Layer Security (TLS) Session Hash and Extended Master Secret Extension", RFC 7627, DOI 10.17487/RFC7627, September 2015, <https://www.rfc-editor.org/info/rfc7627>.

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

[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>.

[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>.

11.2. Informative References

[SIGMAC]   Krawczyk, H., "A Unilateral-to-Mutual Authentication Compiler for Key Exchange (with Applications to Client Authentication in TLS 1.3)", 2016, <https://eprint.iacr.org/2016/711.pdf>.

Author's Address

Nick Sullivan
Cloudflare Inc.

Email: nick@cloudflare.com

Network Working Group C. Huitema
Internet-Draft Private Octopus Inc.
Intended status: Informational E. Rescorla
Expires: April 30, 2020 RTFM, Inc.
October 28, 2019

Issues and Requirements for SNI Encryption in TLS
draft-ietf-tls-sni-encryption-09

Abstract

This draft describes the general problem of encrypting the Server Name Identification (SNI) TLS parameter. The proposed solutions hide a Hidden Service behind a fronting service, only disclosing the SNI of the fronting service to external observers. The draft lists known attacks against SNI encryption, discusses the current "co-tenancy fronting" solution, and presents requirements for future TLS layer solutions.

In practice, it may well be that no solution can meet every requirement, and that practical solutions will have to make some compromises.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

Historically, adversaries have been able to monitor the use of web services through three primary channels: looking at DNS requests, looking at IP addresses in packet headers, and looking at the data stream between user and services. These channels are getting progressively closed. A growing fraction of Internet communication is encrypted, mostly using Transport Layer Security (TLS) [RFC5246]. Progressive deployment of solutions like DNS in TLS [RFC7858] and DNS over HTTPS [RFC8484] mitigates the disclosure of DNS information. More and more services are colocated on multiplexed servers, loosening the relation between IP address and web service. For example, in virtual hosting solutions, multiple services can be hosted as co-tenants on the same server, and the IP address and port do not uniquely identify a service. In cloud or Content Delivery Networks (CDNs) solutions, a given platform hosts the services or servers servers of a lot of organization, and looking up to what netblock an IP address belongs reveals little. However, multiplexed servers rely on the Service Name Information (SNI) TLS extension [RFC6066] to direct connections to the appropriate service implementation. This protocol element is transmitted in clear text. As the other methods of monitoring get blocked, monitoring focuses on the clear text SNI. The purpose of SNI encryption is to prevent that and aid privacy.

Replacing clear text SNI transmission by an encrypted variant will improve the privacy and reliability of TLS connections, but the design of proper SNI encryption solutions is difficult. In the past, there have been multiple attempts at defining SNI encryption. These attempts have generally floundered, because the simple designs fail to mitigate several of the attacks listed in Section 3. In the absence of a TLS-level solution, the most popular approach to SNI privacy for web services is HTTP-level fronting, which we discuss in Section 4.

This document does not present the design of a solution, but provides guidelines for evaluating proposed solutions. (The review of HTTP-level solutions in Section 4 is not an endorsement of these solutions.) The need for related work on the encryption of the Application Layer Protocol Negotiation (ALPN) parameters of TLS is discussed in Section 3.7.1.

2. History of the TLS SNI extension

The SNI extension was specified in 2003 in [RFC3546] to facilitate management of "colocation servers", in which multiple services shared the same IP address. A typical example would be multiple web sites served by the same web server. The SNI extension carries the name of a specific server, enabling the TLS connection to be established with the desired server context. The current SNI extension specification can be found in [RFC6066].

The SNI specification allowed for different types of server names, though only the "hostname" variant was specified and deployed. In that variant, the SNI extension carries the domain name of the target server. The SNI extension is carried in clear text in the TLS "ClientHello" message.

2.1. Unanticipated usage of SNI information

The SNI was defined to facilitate management of servers, but the developers of middleboxes found out that they could take advantage of the information. Many examples of such usage are reviewed in [RFC8404]. Other examples came out during discussions of this draft. They include:

- Filtering or censorship of specific services for a variety of reasons.

- Content filtering by network operators or ISP blocking specific web sites in order to implement, for example, parental controls, or to prevent access to phishing or other fraudulent web sites.

- ISP assigning different QoS profiles to target services,

- Firewalls within enterprise networks blocking web sites not deemed appropriate for work, or

- Firewalls within enterprise networks exempting specific web sites from Man-In-The-Middle (MITM) inspection, such as healthcare or financial sites for which inspection would intrude on the privacy of employees.

The SNI is probably also included in the general collection of metadata by pervasive surveillance actors [RFC7258], for example to identify services used by surveillance targets.

2.2. SNI encryption timeliness

The clear-text transmission of the SNI was not flagged as a problem in the security consideration sections of [RFC3546], [RFC4366], or [RFC6066]. These specifications did not anticipate the alternative usage described in Section 2.1. One reason may be that, when these RFCs were written, the SNI information was available through a variety of other means, such as tracking IP addresses, DNS names, or server certificates.

Many deployments still allocate different IP addresses to different services, so that different services can be identified by their IP addresses. However, CDNs commonly serve a large number of services through a comparatively small number of addresses.

The SNI carries the domain name of the server, which is also sent as part of the DNS queries. Most of the SNI usage described in Section 2.1 could also be implemented by monitoring DNS traffic or controlling DNS usage. But this is changing with the advent of DNS resolvers providing services like DNS over TLS [RFC7858] or DNS over HTTPS [RFC8484].

The subjectAltName extension of type dNSName of the server certificate, or in its absence the common name component, expose the same name as the SNI. In TLS versions 1.0 [RFC2246], 1.1 [RFC4346], and 1.2 [RFC5246], servers send certificates in clear text, ensuring that there would be limited benefits in hiding the SNI. However, in TLS 1.3 [RFC8446], server certificates are encrypted in transit. Note that encryption alone is insufficient to protect server certificates; see Section 3.1 for details.

The decoupling of IP addresses and server names, deployment of DNS privacy, and protection of server certificate transmissions all contribute to user privacy in the face of an [RFC7258]-style adversary. Encrypting the SNI complements this push for privacy and make it harder to censor or otherwise provide differential treatment to specific internet services.

2.3. End-to-end alternatives

Deploying SNI encryption helps thwart most of the unanticipated SNI usages including censorship and pervasive surveillance, but it also will break or reduce the efficacy of the operational practices and techniques implemented in middle-boxes as described in Section 2.1. Most of these functions can, however, be realized by other means. For example, some DNS service providers offer customers the provision to "opt in" filtering services for parental control and phishing protection. Per-stream QoS could be provided by a combination of packet marking and end-to-end agreements. As SNI encryption becomes common, we can expect more deployment of such "end-to-end" solutions.

At the time of this writing, enterprises have the option of installing a firewall performing SNI filtering to prevent connections to certain websites. With SNI encryption this becomes ineffective. Obviously, managers could block usage of SNI encryption in enterprise computers, but this wide-scale blocking would diminish the privacy protection of traffic leaving the enterprise, which may not be desirable. Enterprise managers could rely instead on filtering software and management software deployed on the enterprise’s computers.

3. Security and Privacy Requirements for SNI Encryption

Over the past years, there have been multiple proposals to add an SNI encryption option in TLS. A review of the TLS mailing list archives shows that many of these proposals appeared promising but were rejected after security reviews identified plausible attacks. In this section, we collect a list of these known attacks.

3.1. Mitigate Replay Attacks

The simplest SNI encryption designs replace in the initial TLS exchange the clear text SNI with an encrypted value, using a key known to the multiplexed server. Regardless of the encryption used, these designs can be broken by a simple replay attack, which works as follows:

1- The user starts a TLS connection to the multiplexed server, including an encrypted SNI value.

2- The adversary observes the exchange and copies the encrypted SNI parameter.

3- The adversary starts its own connection to the multiplexed server, including in its connection parameters the encrypted SNI copied from the observed exchange.

4- The multiplexed server establishes the connection to the protected service, thus revealing the identity of the service.

One of the goals of SNI encryption is to prevent adversaries from knowing which Hidden Service the client is using. Successful replay attacks break that goal by allowing adversaries to discover that service.

3.2. Avoid Widely Shared Secrets

It is easy to think of simple schemes in which the SNI is encrypted or hashed using a shared secret. This symmetric key must be known by the multiplexed server, and by every user of the protected services. Such schemes are thus very fragile, since the compromise of a single user would compromise the entire set of users and protected services.

3.3. Prevent SNI-based Denial of Service Attacks

Encrypting the SNI may create extra load for the multiplexed server. Adversaries may mount denial of service attacks by generating random encrypted SNI values and forcing the multiplexed server to spend resources in useless decryption attempts.

It may be argued that this is not an important Denial of Service Attacks (DoS) avenue, as regular TLS connection attempts also require the server to perform a number of cryptographic operations. However, in many cases, the SNI decryption will have to be performed by a front-end component with limited resources, while the TLS operations are performed by the component dedicated to their respective services. SNI-based DoS attacks could target the front-end component.

3.4. Do not stick out

In some designs, handshakes using SNI encryption can be easily differentiated from "regular" handshakes. For example, some designs require specific extensions in the Client Hello packets, or specific values of the clear text SNI parameter. If adversaries can easily detect the use of SNI encryption, they could block it, or they could flag the users of SNI encryption for special treatment.

In the future, it might be possible to assume that a large fraction of TLS handshakes use SNI encryption. If that were the case, the detection of SNI encryption would be a lesser concern. However, we have to assume that in the near future, only a small fraction of TLS connections will use SNI encryption.

3.5. Forward Secrecy

The general concerns about forward secrecy apply to SNI encryption just as well as to regular TLS sessions. For example, some proposed designs rely on a public key of the multiplexed server to define the SNI encryption key. If the corresponding private key should be compromised, the adversaries would be able to process archival records of past connections, and retrieve the protected SNI used in these connections. These designs failed to maintain forward secrecy of SNI encryption.

3.6. Multi-Party Security Contexts

We can design solutions in which a fronting service acts as a relay to reach the protected service. Some of those solutions involve just one TLS handshake between the client and the fronting service. The master secret is verified by verifying a certificate provided by the fronting service, but not by the protected service. These solutions expose the client to a Man-In-The-Middle attack by the fronting service. Even if the client has some reasonable trust in this service, the possibility of MITM attack is troubling.

There are other classes of solutions in which the master secret is verified by verifying a certificate provided by the protected service. These solutions offer more protection against MITM attack by the fronting service. The downside is that the client will not verify the identity of the fronting service, which enables fronting server spoofing attacks such as the "honeypot" attack discussed below. Overall, end-to-end TLS to the protected service is preferable, but it is important to also provide a way to authenticate the fronting service.

The fronting service could be pressured by adversaries. By design, it could be forced to deny access to the protected service, or to divulge which client accessed it. But if MITM is possible, the adversaries would also be able to pressure the fronting service into intercepting or spoofing the communications between client and protected service.

Adversaries could also mount an attack by spoofing the fronting service. A spoofed fronting service could act as a "honeypot" for users of hidden services. At a minimum, the fake server could record the IP addresses of these users. If the SNI encryption solution places too much trust on the fronting server, the fake server could also serve fake content of its own choosing, including various forms of malware.

There are two main channels by which adversaries can conduct this attack. Adversaries can simply try to mislead users into believing that the honeypot is a valid fronting server, especially if that information is carried by word of mouth or in unprotected DNS records. Adversaries can also attempt to hijack the traffic to the regular fronting server, using, for example, spoofed DNS responses or spoofed IP level routing, combined with a spoofed certificate.

3.7. Supporting multiple protocols

The SNI encryption requirement does not stop with HTTP over TLS. Multiple other applications currently use TLS, including, for example, SMTP [RFC5246], DNS [RFC7858], IMAP [RFC8314], and XMPP [RFC7590]. These applications, too, will benefit from SNI encryption. HTTP-only methods like those described in Section 4.1 would not apply there. In fact, even for the HTTPS case, the HTTPS tunneling service described in Section 4.1 is compatible with HTTP 1.0 and HTTP 1.1, but interacts awkwardly with the multiple streams feature of HTTP/2 [RFC7540]. This points to the need for an application-agnostic solution, which would be implemented fully in the TLS layer.

3.7.1. Hiding the Application Layer Protocol Negotiation

The Application Layer Protocol Negotiation (ALPN) parameters of TLS allow implementations to negotiate the application layer protocol used on a given connection. TLS provides the ALPN values in clear text during the initial handshake. While exposing the ALPN does not create the same privacy issues as exposing the SNI, there is still a risk. For example, some networks may attempt to block applications that they do not understand, or that they wish users would not use.

In a sense, ALPN filtering could be very similar to the filtering of specific port numbers exposed in some networks. This filtering by ports has given rise to evasion tactics in which various protocols are tunneled over HTTP in order to use open ports 80 or 443. Filtering by ALPN would probably beget the same responses, in which the applications just move over HTTP, and only the HTTP ALPN values are used. Applications would not need to do that if the ALPN were hidden in the same way as the SNI.

In addition to hiding the SNI, it is thus desirable to also hide the ALPN. Of course, this implies engineering trade-offs. Using the same technique for hiding the ALPN and encrypting the SNI may result in excess complexity. It might be preferable to encrypt these independently.

3.7.2. Support other transports than TCP

The TLS handshake is also used over other transports such as UDP with both DTLS [I-D.ietf-tls-dtls13] and QUIC [I-D.ietf-quic-tls]. The requirement to encrypt the SNI applies just as well for these transports as for TLS over TCP.

This points to a requirement for SNI Encryption mechanisms to also be applicable to non-TCP transports such as DTLS or QUIC.

4. HTTP Co-Tenancy Fronting

In the absence of TLS-level SNI encryption, many sites rely on an "HTTP Co-Tenancy" solution, often refered to as Domain Fronting [domfront]. The TLS connection is established with the fronting server, and HTTP requests are then sent over that connection to the hidden service. For example, the TLS SNI could be set to "fronting.example.com", the fronting server, and HTTP requests sent over that connection could be directed to "hidden.example.com", accessing the hidden service. This solution works well in practice when the fronting server and the hidden server are "co-tenants" of the same multiplexed server.

The HTTP fronting solution can be deployed without modification to the TLS protocol, and does not require using any specific version of TLS. There are, however, a few issues regarding discovery, client implementations, trust, and applicability:

- The client has to discover that the hidden service can be accessed through the fronting server.

- The client's browser has to be directed to access the hidden service through the fronting service.

- Since the TLS connection is established with the fronting service, the client has no cryptographic proof that the content does, in fact, come from the hidden service. The solution does thus not mitigate the context sharing issues described in Section 3.6.

- Since this is an HTTP-level solution, it does not protect non-HTTP protocols as discussed in Section 3.7.

The discovery issue is common to most SNI encryption solutions. The browser issue was solved in [domfront] by implementing domain fronting as a pluggable transport for the Tor browser. The multi-protocol issue can be mitigated by using implementation of other applications over HTTP, such as for example DNS over HTTPS [RFC8484]. The trust issue, however, requires specific developments.

4.1. HTTPS Tunnels

The HTTP Fronting solution places a lot of trust in the Fronting Server. This required trust can be reduced by tunnelling HTTPS in HTTPS, which effectively treats the Fronting Server as an HTTP Proxy. In this solution, the client establishes a TLS connection to the Fronting Server, and then issues an HTTP Connect request to the Hidden Server. This will establish an end-to-end HTTPS over TLS connection between the client and the Hidden Server, mitigating the issues described in Section 3.6.

The HTTPS in HTTPS solution requires double encryption of every packet. It also requires that the fronting server decrypt and relay messages to the hidden server. Both of these requirements make the implementation onerous.

4.2. Delegation Control

Clients would see their privacy compromised if they contacted the wrong fronting server to access the hidden service, since this wrong server could disclose their access to adversaries. This requires a controlled way to indicate which fronting server is acceptable by the hidden service.

This problem is both similar and different from the "fronting server spoofing" attack described in Section 3.6. Here, the spoofing would be performed by distributing fake advice, such as "to reach hidden.example.com, use fake.example.com as a fronting server", when "fake.example.com" is under the control of an adversary.

In practice, this attack is well mitigated when the hidden service is accessed through a specialized application. The name of the fronting server can then be programmed in the code of the application. But the attack is harder to mitigate when the hidden service has to be accessed through general purpose web browsers.

There are several proposed solutions to this problem, such as creating a special form of certificate to codify the relation between fronting and hidden server, or obtaining the relation between hidden and fronting service through the DNS, possibly using DNSSEC to avoid spoofing. The experiment described in [domfront] solved the issue by integrating with the Lantern Internet circumvention circumvention tool.

We can observe that CDNs have a similar requirement. They need to convince the client that "www.example.com" can be accessed through the seemingly unrelated "cdn-node-xyz.example.net". Most CDNs have deployed DNS-based solutions to this problem. However, the CDN often holds the authoritative certificate of the origin. There is simultaneously verification of a relationship between the origin and the CDN, through the certificate, and a risk that the CDN can spoof the content from the origin.

4.3. Related work

The ORIGIN frame defined for HTTP/2 [RFC8336] can be used to flag content provided by the hidden server. Secondary certificate authentication [I-D.ietf-httpbis-http2-secondary-certs] can be used to manage authentication of hidden server content, or to perform client authentication before accessing hidden content.

5. Security Considerations

This document lists a number of attacks against SNI encryption in Section 3 and also in Section 4.2, and presents a list of requirements to mitigate these attacks. Current HTTP-based solutions described in Section 4 only meet some of these requirements. In practice, it may well be that no solution can meet every requirement, and that practical solutions will have to make some compromises.

In particular, the requirement to not stick out presented in Section 3.4 may have to be lifted, especially for proposed solutions that could quickly reach large scale deployments.

Replacing clear text SNI transmission by an encrypted variant will break or reduce the efficacy of the operational practices and techniques implemented in middle-boxes as described in Section 2.1.

As explained in Section 2.3, alternative solutions will have to be developed.

6. IANA Considerations

This draft does not require any IANA action.

7. Acknowledgements

A large part of this draft originates in discussion of SNI encryption on the TLS WG mailing list, including comments after the tunneling approach was first proposed in a message to that list: <https://mailarchive.ietf.org/arch/msg/tls/tXvdcqnogZgqmdfCugrV8M90Ftw>.

Thanks to Daniel Kahn Gillmor for a pretty detailed review of the initial draft. Thanks to Bernard Aboba, Mike Bishop, Alissa Cooper, Roman Danyliw, Stephen Farrell, Warren Kumari, Mirja Kuelewind Barry Leiba, Martin Rex, Adam Roach, Meral Shirazipour, Martin Thomson, Eric Vyncke, and employees of the UK National Cyber Security Centre for their reviews. Thanks to Chris Wood, Ben Kaduk and Sean Turner for helping publish this document.

8. Informative References

[domfront]
Fifield, D., Lan, C., Hynes, R., Wegmann, P., and V. Paxson, "Blocking-resistant communication through domain fronting", DOI 10.1515/popets-2015-0009, 2015, <https://www.bamsoftware.com/papers/fronting/>.

[I-D.ietf-httpbis-http2-secondary-certs]
Bishop, M., Sullivan, N., and M. Thomson, "Secondary Certificate Authentication in HTTP/2", draft-ietf-httpbis-http2-secondary-certs-04 (work in progress), April 2019.

[I-D.ietf-quic-tls]
Thomson, M. and S. Turner, "Using TLS to Secure QUIC", draft-ietf-quic-tls-23 (work in progress), September 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.

[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>.

[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>.

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

[RFC4366] Blake-Wilson, S., Nystrom, M., Hopwood, D., Mikkelsen, J., and T. Wright, "Transport Layer Security (TLS) Extensions", RFC 4366, DOI 10.17487/RFC4366, April 2006, <https://www.rfc-editor.org/info/rfc4366>.

[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>.

[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>.

[RFC7258] Farrell, S. and H. Tschofenig, "Pervasive Monitoring Is an Attack", BCP 188, RFC 7258, DOI 10.17487/RFC7258, May 2014, <https://www.rfc-editor.org/info/rfc7258>.

[RFC7540] Belshe, M., Peon, R., and M. Thomson, Ed., "Hypertext Transfer Protocol Version 2 (HTTP/2)", RFC 7540, DOI 10.17487/RFC7540, May 2015, <https://www.rfc-editor.org/info/rfc7540>.

[RFC7590] Saint-Andre, P. and T. Alkemade, "Use of Transport Layer Security (TLS) in the Extensible Messaging and Presence Protocol (XMPP)", RFC 7590, DOI 10.17487/RFC7590, June 2015, <https://www.rfc-editor.org/info/rfc7590>.

[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>.

[RFC8314] Moore, K. and C. Newman, "Cleartext Considered Obsolete: Use of Transport Layer Security (TLS) for Email Submission and Access", RFC 8314, DOI 10.17487/RFC8314, January 2018, <https://www.rfc-editor.org/info/rfc8314>.

[RFC8336] Nottingham, M. and E. Nygren, "The ORIGIN HTTP/2 Frame", RFC 8336, DOI 10.17487/RFC8336, March 2018, <https://www.rfc-editor.org/info/rfc8336>.

[RFC8404] Moriarty, K., Ed. and A. Morton, Ed., "Effects of Pervasive Encryption on Operators", RFC 8404, DOI 10.17487/RFC8404, July 2018, <https://www.rfc-editor.org/info/rfc8404>.

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

[RFC8484] Hoffman, P. and P. McManus, "DNS Queries over HTTPS (DoH)", RFC 8484, DOI 10.17487/RFC8484, October 2018, <https://www.rfc-editor.org/info/rfc8484>.

Authors' Addresses

Christian Huitema
Private Octopus Inc.
Friday Harbor WA 98250
U.S.A

Email: huitema@huitema.net

Eric Rescorla
RTFM, Inc.
U.S.A

Email: ekr@rtfm.com

Network Working Group
Internet-Draft
Obsoletes: 5077, 5246, 6961 (if approved)
Updates: 4492, 5705, 6066 (if approved)
Intended status: Standards Track
Expires: September 21, 2018

E. Rescorla
RTFM, Inc.

March 20, 2018

The Transport Layer Security (TLS) Protocol Version 1.3
draft-ietf-tls-tls13-28

Abstract

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

This document updates RFCs 4492, 5705, and 6066 and it obsoletes RFCs 5077, 5246, and 6961. This document also specifies new requirements for TLS 1.2 implementations.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

RFC EDITOR: PLEASE REMOVE THE FOLLOWING PARAGRAPH The source for this draft is maintained in GitHub. Suggested changes should be submitted as pull requests at https://github.com/tlswg/tls13-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 TLS is to provide a secure channel between two communicating peers; the only requirement from the underlying transport is a reliable, in-order, data stream. Specifically, the secure channel should provide the following properties:

- Authentication: The server side of the channel is always authenticated; the client side is optionally authenticated. Authentication can happen via asymmetric cryptography (e.g., RSA [RSA], ECDSA [ECDSA], EdDSA [RFC8032]) or a pre-shared key (PSK).

- Confidentiality: Data sent over the channel after establishment is only visible to the endpoints. TLS does not hide the length of the data it transmits, though endpoints are able to pad TLS records in order to obscure lengths and improve protection against traffic analysis techniques.

- Integrity: Data sent over the channel after establishment cannot be modified by attackers.

These properties should be true even in the face of an attacker who has complete control of the network, as described in [RFC3552]. See Appendix E for a more complete statement of the relevant security properties.

TLS consists of two primary components:

- A handshake protocol (Section 4) that authenticates the communicating parties, negotiates cryptographic modes and parameters, and establishes shared keying material. The handshake protocol is designed to resist tampering; an active attacker should not be able to force the peers to negotiate different parameters than they would if the connection were not under attack.

- A record protocol (Section 5) that uses the parameters established by the handshake protocol to protect traffic between the communicating peers. The record protocol divides traffic up into a series of records, each of which is independently protected using the traffic keys.

TLS is application protocol independent; higher-level protocols can layer on top of TLS transparently. The TLS standard, however, does not specify how protocols add security with TLS; how to initiate TLS handshaking and how to interpret the authentication certificates exchanged are left to the judgment of the designers and implementors of protocols that run on top of TLS.

This document defines TLS version 1.3. While TLS 1.3 is not directly compatible with previous versions, all versions of TLS incorporate a versioning mechanism which allows clients and servers to interoperably negotiate a common version if one is supported by both peers.

This document supersedes and obsoletes previous versions of TLS including version 1.2 [RFC5246]. It also obsoletes the TLS ticket mechanism defined in [RFC5077] and replaces it with the mechanism defined in Section 2.2. Section 4.2.7 updates [RFC4492] by modifying the protocol attributes used to negotiate Elliptic Curves. Because TLS 1.3 changes the way keys are derived, it updates [RFC5705] as described in Section 7.5. It also changes how OCSP messages are carried and therefore updates [RFC6066] and obsoletes [RFC6961] as described in section Section 4.4.2.1.

1.1. Conventions and Terminology

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

The following terms are used:

client: The endpoint initiating the TLS connection.

connection: A transport-layer connection between two endpoints.

endpoint: Either the client or server of the connection.

handshake: An initial negotiation between client and server that establishes the parameters of their subsequent interactions within TLS.

peer: An endpoint. When discussing a particular endpoint, "peer" refers to the endpoint that is not the primary subject of discussion.

receiver: An endpoint that is receiving records.

sender: An endpoint that is transmitting records.

server: The endpoint which did not initiate the TLS connection.

1.2. Change Log

RFC EDITOR PLEASE DELETE THIS SECTION.

(*) indicates changes to the wire protocol which may require implementations to update.

draft-28

Add a section on exposure of PSK identities.

draft-27

- SHOULD->MUST for being able to process "supported_versions" without 0x0304.
- Much editorial cleanup.

draft-26

- Clarify that you can’t negotiate pre-TLS 1.3 with supported_versions.

draft-25

- Add the header to additional data (*)
- Minor clarifications.
- IANA cleanup.

draft-24

- Require that CH2 have version 0303 (*)
- Some clarifications

draft-23

- Renumber key_share (*)
- Add a new extension and new code points to allow negotiating PSS separately for certificates and CertificateVerify (*)

- Slightly restrict when CCS must be accepted to make implementation easier.

- Document protocol invariants

- Add some text on the security of static RSA.

draft-22

- Implement changes for improved middlebox penetration (*)

- Move server_certificate_type to encrypted extensions (*)

- Allow resumption with a different SNI (*)

- Padding extension can change on HRR (*)

- Allow an empty ticket_nonce (*)

- Remove requirement to immediately respond to close_notify with close_notify (allowing half-close)

draft-21

- Add a per-ticket nonce so that each ticket is associated with a different PSK (*).

- Clarify that clients should send alerts with the handshake key if possible.

- Update state machine to show rekeying events

- Add discussion of 0-RTT and replay. Recommend that implementations implement some anti-replay mechanism.

draft-20

- Add "post_handshake_auth" extension to negotiate post-handshake authentication (*).

- Shorten labels for HKDF-Expand-Label so that we can fit within one compression block (*).

- Define how RFC 7250 works (*).

- Re-enable post-handshake client authentication even when you do PSK. The previous prohibition was editorial error.

- Remove cert_type and user_mapping, which don't work on TLS 1.3 anyway.

- Added the no_application_protocol alert from [RFC7301] to the list of extensions.

- Added discussion of traffic analysis and side channel attacks.

draft-19

- Hash context_value input to Exporters (*)

- Add an additional Derive-Secret stage to Exporters (*).

- Hash ClientHello1 in the transcript when HRR is used. This reduces the state that needs to be carried in cookies. (*)

- Restructure CertificateRequest to have the selectors in extensions. This also allowed defining a "certificate_authorities" extension which can be used by the client instead of trusted_ca_keys (*).

- Tighten record framing requirements and require checking of them (*).

- Consolidate "ticket_early_data_info" and "early_data" into a single extension (*).

- Change end_of_early_data to be a handshake message (*).

- Add pre-extract Derive-Secret stages to key schedule (*).

- Remove spurious requirement to implement "pre_shared_key".

- Clarify location of "early_data" from server (it goes in EE, as indicated by the table in S 10).

- Require peer public key validation

- Add state machine diagram.

draft-18

- Remove unnecessary resumption_psk which is the only thing expanded from the resumption master secret. (*).

- Fix signature_algorithms entry in extensions table.

- Restate rule from RFC 6066 that you can’t resume unless SNI is the same.

draft-17

- Remove 0-RTT Finished and resumption_context, and replace with a psk_binder field in the PSK itself (*)

- Restructure PSK key exchange negotiation modes (*)

- Add max_early_data_size field to TicketEarlyDataInfo (*)

- Add a 0-RTT exporter and change the transcript for the regular exporter (*)

- Merge TicketExtensions and Extensions registry. Changes ticket_early_data_info code point (*)

- Replace Client.key_shares in response to HRR (*)

- Remove redundant labels for traffic key derivation (*)

- Harmonize requirements about cipher suite matching: for resumption you need to match KDF but for 0-RTT you need whole cipher suite. This allows PSKs to actually negotiate cipher suites. (*)

- Move SCT and OCSP into Certificate.extensions (*)

- Explicitly allow non-offered extensions in NewSessionTicket

- Explicitly allow predicting client Finished for NST

- Clarify conditions for allowing 0-RTT with PSK

draft-16

- Revise version negotiation (*)

- Change RSASSA-PSS and EdDSA SignatureScheme codepoints for better backwards compatibility (*)

- Move HelloRetryRequest.selected_group to an extension (*)

- Clarify the behavior of no exporter context and make it the same as an empty context.(*)

- New KeyUpdate format that allows for requesting/not-requesting an answer. This also means changes to the key schedule to support independent updates (*)

- New certificate_required alert (*)

- Forbid CertificateRequest with 0-RTT and PSK.

- Relax requirement to check SNI for 0-RTT.

draft-15

- New negotiation syntax as discussed in Berlin (*)

- Require CertificateRequest.context to be empty during handshake (*)

- Forbid empty tickets (*)

- Forbid application data messages in between post-handshake messages from the same flight (*)

- Clean up alert guidance (*)

- Clearer guidance on what is needed for TLS 1.2.

- Guidance on 0-RTT time windows.

- Rename a bunch of fields.

- Remove old PRNG text.

- Explicitly require checking that handshake records not span key changes.

draft-14

- Allow cookies to be longer (*)

- Remove the "context" from EarlyDataIndication as it was undefined and nobody used it (*)

- Remove 0-RTT EncryptedExtensions and replace the ticket_age extension with an obfuscated version. Also necessitates a change to NewSessionTicket (*).

- Move the downgrade sentinel to the end of ServerHello.Random to accommodate tlsdate (*).

- Define ecdsa_sha1 (*).

- Allow resumption even after fatal alerts. This matches current practice.

- Remove non-closure warning alerts. Require treating unknown alerts as fatal.

- Make the rules for accepting 0-RTT less restrictive.

- Clarify 0-RTT backward-compatibility rules.

- Clarify how 0-RTT and PSK identities interact.

- Add a section describing the data limits for each cipher.

- Major editorial restructuring.

- Replace the Security Analysis section with a WIP draft.

draft-13

- Allow server to send SupportedGroups.

- Remove 0-RTT client authentication

- Remove (EC)DHE 0-RTT.

- Flesh out 0-RTT PSK mode and shrink EarlyDataIndication

- Turn PSK-resumption response into an index to save room

- Move CertificateStatus to an extension

- Extra fields in NewSessionTicket.

- Restructure key schedule and add a resumption_context value.

- Require DH public keys and secrets to be zero-padded to the size of the group.

- Remove the redundant length fields in KeyShareEntry.

- Define a cookie field for HRR.

draft-12

- Provide a list of the PSK cipher suites.

- Remove the ability for the ServerHello to have no extensions (this aligns the syntax with the text).

- Clarify that the server can send application data after its first flight (0.5 RTT data)

- Revise signature algorithm negotiation to group hash, signature algorithm, and curve together. This is backwards compatible.

- Make ticket lifetime mandatory and limit it to a week.

- Make the purpose strings lower-case. This matches how people are implementing for interop.

- Define exporters.

- Editorial cleanup

draft-11

- Port the CFRG curves & signatures work from RFC4492bis.

- Remove sequence number and version from additional_data, which is now empty.

- Reorder values in HkdfLabel.

- Add support for version anti-downgrade mechanism.

- Update IANA considerations section and relax some of the policies.

- Unify authentication modes. Add post-handshake client authentication.

- Remove early_handshake content type. Terminate 0-RTT data with an alert.

- Reset sequence number upon key change (as proposed by Fournet et al.)

draft-10

- Remove ClientCertificateTypes field from CertificateRequest and add extensions.

- Merge client and server key shares into a single extension.

draft-09

- Change to RSA-PSS signatures for handshake messages.

- Remove support for DSA.

- Update key schedule per suggestions by Hugo, Hoeteck, and Bjoern Tackmann.

- Add support for per-record padding.

- Switch to encrypted record ContentType.

- Change HKDF labeling to include protocol version and value lengths.

- Shift the final decision to abort a handshake due to incompatible certificates to the client rather than having servers abort early.

- Deprecate SHA-1 with signatures.

- Add MTI algorithms.

draft-08

- Remove support for weak and lesser used named curves.

- Remove support for MD5 and SHA-224 hashes with signatures.

- Update lists of available AEAD cipher suites and error alerts.

- Reduce maximum permitted record expansion for AEAD from 2048 to 256 octets.

- Require digital signatures even when a previous configuration is used.

- Merge EarlyDataIndication and KnownConfiguration.

- Change code point for server_configuration to avoid collision with server_hello_done.

- Relax certificate_list ordering requirement to match current practice.

draft-07

- Integration of semi-ephemeral DH proposal.

- Add initial 0-RTT support.

- Remove resumption and replace with PSK + tickets.

- Move ClientKeyShare into an extension.

- Move to HKDF.

draft-06

- Prohibit RC4 negotiation for backwards compatibility.

- Freeze & deprecate record layer version field.

- Update format of signatures with context.

- Remove explicit IV.

draft-05

- Prohibit SSL negotiation for backwards compatibility.

- Fix which MS is used for exporters.

draft-04

- Modify key computations to include session hash.

- Remove ChangeCipherSpec.

- Renumber the new handshake messages to be somewhat more consistent with existing convention and to remove a duplicate registration.

- Remove renegotiation.

- Remove point format negotiation.

draft-03

- Remove GMT time.

- Merge in support for ECC from RFC 4492 but without explicit curves.

- Remove the unnecessary length field from the AD input to AEAD ciphers.

- Rename {Client,Server}KeyExchange to {Client,Server}KeyShare.

- Add an explicit HelloRetryRequest to reject the client's.

draft-02

- Increment version number.

- Rework handshake to provide 1-RTT mode.

- Remove custom DHE groups.

- Remove support for compression.

- Remove support for static RSA and DH key exchange.

- Remove support for non-AEAD ciphers.

1.3. Major Differences from TLS 1.2

The following is a list of the major functional differences between TLS 1.2 and TLS 1.3. It is not intended to be exhaustive and there are many minor differences.

- The list of supported symmetric algorithms has been pruned of all algorithms that are considered legacy. Those that remain all use Authenticated Encryption with Associated Data (AEAD) algorithms. The ciphersuite concept has been changed to separate the authentication and key exchange mechanisms from the record protection algorithm (including secret key length) and a hash to be used with the key derivation function and HMAC.

- A 0-RTT mode was added, saving a round-trip at connection setup for some application data, at the cost of certain security properties.

- Static RSA and Diffie-Hellman cipher suites have been removed; all public-key based key exchange mechanisms now provide forward secrecy.

- All handshake messages after the ServerHello are now encrypted. The newly introduced EncryptedExtension message allows various extensions previously sent in clear in the ServerHello to also enjoy confidentiality protection from active attackers.

- The key derivation functions have been re-designed. The new design allows easier analysis by cryptographers due to their improved key separation properties. The HMAC-based Extract-and-Expand Key Derivation Function (HKDF) is used as an underlying primitive.

- The handshake state machine has been significantly restructured to be more consistent and to remove superfluous messages such as ChangeCipherSpec (except when needed for middlebox compatibility).

- Elliptic curve algorithms are now in the base spec and new signature algorithms, such as ed25519 and ed448, are included. TLS 1.3 removed point format negotiation in favor of a single point format for each curve.

- Other cryptographic improvements including the removal of compression and custom DHE groups, changing the RSA padding to use RSASSA-PSS, and the removal of DSA.

- The TLS 1.2 version negotiation mechanism has been deprecated in favor of a version list in an extension. This increases compatibility with existing servers that incorrectly implemented version negotiation.

- Session resumption with and without server-side state as well as the PSK-based ciphersuites of earlier TLS versions have been replaced by a single new PSK exchange.

- Updated references to point to the updated versions of RFCs, as appropriate (e.g., RFC 5280 rather than RFC 3280).

1.4. Updates Affecting TLS 1.2

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

- A version downgrade protection mechanism is described in Section 4.1.3.

- RSASSA-PSS signature schemes are defined in Section 4.2.3.

- The "supported_versions" ClientHello extension can be used to negotiate the version of TLS to use, in preference to the legacy_version field of the ClientHello.

- The "signature_algorithms_cert" extension allows a client to indicate which signature algorithms it can validate in X.509 certificates

Additionally, this document clarifies some compliance requirements for earlier versions of TLS; see Section 9.3.

2. Protocol Overview

The cryptographic parameters used by the secure channel are produced by the TLS handshake protocol. This sub-protocol of TLS is used by the client and server when first communicating with each other. The handshake protocol allows peers to negotiate a protocol version, select cryptographic algorithms, optionally authenticate each other, and establish shared secret keying material. Once the handshake is complete, the peers use the established keys to protect the application layer traffic.

A failure of the handshake or other protocol error triggers the termination of the connection, optionally preceded by an alert message (Section 6).

TLS supports three basic key exchange modes:

- (EC)DHE (Diffie-Hellman over either finite fields or elliptic curves)

- PSK-only

- PSK with (EC)DHE

Figure 1 below shows the basic full TLS handshake:

```
       Client                                               Server

Key  ^ ClientHello
Exch | + key_share*
     | + signature_algorithms*
     | + psk_key_exchange_modes*
     v + pre_shared_key*         -------->
                                                       ServerHello  ^ Key
                                                      + key_share*  | Exch
                                                 + pre_shared_key*  v
                                             {EncryptedExtensions}  ^  Server
                                             {CertificateRequest*}  v  Params
                                                    {Certificate*}  ^
                                              {CertificateVerify*}  | Auth
                                                        {Finished}  v
                                 <--------     [Application Data*]
     ^ {Certificate*}
Auth | {CertificateVerify*}
     v {Finished}                -------->
       [Application Data]        <------->      [Application Data]

              +  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 [sender]_application_traffic_secret_N
```

Figure 1: Message flow for full TLS Handshake

The handshake can be thought of as having three phases (indicated in the diagram above):

- Key Exchange: Establish shared keying material and select the cryptographic parameters. Everything after this phase is encrypted.

- Server Parameters: Establish other handshake parameters (whether the client is authenticated, application layer protocol support, etc.).

- Authentication: Authenticate the server (and optionally the client) and provide key confirmation and handshake integrity.

In the Key Exchange phase, the client sends the ClientHello (Section 4.1.2) message, which contains a random nonce (ClientHello.random); its offered protocol versions; a list of symmetric cipher/HKDF hash pairs; either a set of Diffie-Hellman key shares (in the "key_share" extension Section 4.2.8), a set of pre-shared key labels (in the "pre_shared_key" extension Section 4.2.11) or both; and potentially additional extensions. Additional fields and/or messages may also be present for middlebox compatibility.

The server processes the ClientHello and determines the appropriate cryptographic parameters for the connection. It then responds with its own ServerHello (Section 4.1.3), which indicates the negotiated connection parameters. The combination of the ClientHello and the ServerHello determines the shared keys. If (EC)DHE key establishment is in use, then the ServerHello contains a "key_share" extension with the server's ephemeral Diffie-Hellman share; the server's share MUST be in the same group as one of the client's shares. If PSK key establishment is in use, then the ServerHello contains a "pre_shared_key" extension indicating which of the client's offered PSKs was selected. Note that implementations can use (EC)DHE and PSK together, in which case both extensions will be supplied.

The server then sends two messages to establish the Server Parameters:

EncryptedExtensions: responses to ClientHello extensions that are not required to determine the cryptographic parameters, other than those that are specific to individual certificates. [Section 4.3.1]

CertificateRequest: if certificate-based client authentication is desired, the desired parameters for that certificate. This message is omitted if client authentication is not desired. [Section 4.3.2]

Finally, the client and server exchange Authentication messages. TLS uses the same set of messages every time that certificate-based authentication is needed. (PSK-based authentication happens as a side effect of key exchange.) Specifically:

Certificate: the certificate of the endpoint and any per-certificate extensions. This message is omitted by the server if not authenticating with a certificate and by the client if the server did not send CertificateRequest (thus indicating that the client should not authenticate with a certificate). Note that if raw public keys [RFC7250] or the cached information extension [RFC7924] are in use, then this message will not contain a certificate but rather some other value corresponding to the server's long-term key. [Section 4.4.2]

CertificateVerify: a signature over the entire handshake using the private key corresponding to the public key in the Certificate message. This message is omitted if the endpoint is not authenticating via a certificate. [Section 4.4.3]

Finished: a MAC (Message Authentication Code) over the entire handshake. This message provides key confirmation, binds the endpoint's identity to the exchanged keys, and in PSK mode also authenticates the handshake. [Section 4.4.4]

Upon receiving the server's messages, the client responds with its Authentication messages, namely Certificate and CertificateVerify (if requested), and Finished.

At this point, the handshake is complete, and the client and server derive the keying material required by the record layer to exchange application-layer data protected through authenticated encryption. Application data MUST NOT be sent prior to sending the Finished message, except as specified in [Section 2.3]. Note that while the server may send application data prior to receiving the client's Authentication messages, any data sent at that point is, of course, being sent to an unauthenticated peer.

2.1. Incorrect DHE Share

If the client has not provided a sufficient "key_share" extension (e.g., it includes only DHE or ECDHE groups unacceptable to or unsupported by the server), the server corrects the mismatch with a HelloRetryRequest and the client needs to restart the handshake with an appropriate "key_share" extension, as shown in Figure 2. If no common cryptographic parameters can be negotiated, the server MUST abort the handshake with an appropriate alert.

```
         Client                                               Server

         ClientHello
         + key_share             -------->
                                                   HelloRetryRequest
                                 <--------               + key_share
         ClientHello
         + key_share             -------->
                                                         ServerHello
                                                         + key_share
                                               {EncryptedExtensions}
                                               {CertificateRequest*}
                                                      {Certificate*}
                                                {CertificateVerify*}
                                                          {Finished}
                                 <--------       [Application Data*]
         {Certificate*}
         {CertificateVerify*}
         {Finished}              -------->
         [Application Data]      <------->        [Application Data]
```

Figure 2: Message flow for a full handshake with mismatched parameters

Note: The handshake transcript incorporates the initial ClientHello/ HelloRetryRequest exchange; it is not reset with the new ClientHello.

TLS also allows several optimized variants of the basic handshake, as described in the following sections.

2.2. Resumption and Pre-Shared Key (PSK)

Although TLS PSKs can be established out of band, PSKs can also be established in a previous connection and then used to establish a new connection ("session resumption" or "resuming" with a PSK). Once a handshake has completed, the server can send to the client a PSK identity that corresponds to a unique key derived from the initial handshake (see Section 4.6.1). The client can then use that PSK identity in future handshakes to negotiate the use of the associated PSK. If the server accepts the PSK, then the security context of the new connection is cryptographically tied to the original connection and the key derived from the initial handshake is used to bootstrap the cryptographic state instead of a full handshake. In TLS 1.2 and below, this functionality was provided by "session IDs" and "session tickets" [RFC5077]. Both mechanisms are obsoleted in TLS 1.3.

PSKs can be used with (EC)DHE key exchange in order to provide forward secrecy in combination with shared keys, or can be used alone, at the cost of losing forward secrecy for the application data.

Figure 3 shows a pair of handshakes in which the first establishes a PSK and the second uses it:

```
          Client                                               Server

   Initial Handshake:
          ClientHello
          + key_share               -------->
                                                          ServerHello
                                                          + key_share
                                                {EncryptedExtensions}
                                                {CertificateRequest*}
                                                       {Certificate*}
                                                 {CertificateVerify*}
                                                           {Finished}
                                    <--------     [Application Data*]
          {Certificate*}
          {CertificateVerify*}
          {Finished}                -------->
                                    <--------      [NewSessionTicket]
          [Application Data]        <------->      [Application Data]


   Subsequent Handshake:
          ClientHello
          + key_share*
          + pre_shared_key          -------->
                                                          ServerHello
                                                     + pre_shared_key
                                                         + key_share*
                                                {EncryptedExtensions}
                                                           {Finished}
                                    <--------     [Application Data*]
          {Finished}                -------->
          [Application Data]        <------->      [Application Data]
```

Figure 3: Message flow for resumption and PSK

As the server is authenticating via a PSK, it does not send a Certificate or a CertificateVerify message. When a client offers resumption via PSK, it SHOULD also supply a "key_share" extension to the server to allow the server to decline resumption and fall back to a full handshake, if needed. The server responds with a "pre_shared_key" extension to negotiate use of PSK key establishment and can (as shown here) respond with a "key_share" extension to do (EC)DHE key establishment, thus providing forward secrecy.

When PSKs are provisioned out of band, the PSK identity and the KDF hash algorithm to be used with the PSK MUST also be provisioned.

Note: When using an out-of-band provisioned pre-shared secret, a critical consideration is using sufficient entropy during the key generation, as discussed in [RFC4086]. Deriving a shared secret from a password or other low-entropy sources is not secure. A low-entropy secret, or password, is subject to dictionary attacks based on the PSK binder. The specified PSK authentication is not a strong password-based authenticated key exchange even when used with Diffie-Hellman key establishment. Specifically, it does not prevent an attacker that can observe the handshake from performing a brute-force attack on the password/pre-shared key.

2.3. 0-RTT Data

When clients and servers share a PSK (either obtained externally or via a previous handshake), TLS 1.3 allows clients to send data on the first flight ("early data"). The client uses the PSK to authenticate the server and to encrypt the early data.

As shown in Figure 4, the 0-RTT data is just added to the 1-RTT handshake in the first flight. The rest of the handshake uses the same messages as for a 1-RTT handshake with PSK resumption.

```
         Client                                               Server

         ClientHello
         + early_data
         + key_share*
         + psk_key_exchange_modes
         + pre_shared_key
         (Application Data*)     -------->
                                                         ServerHello
                                                    + pre_shared_key
                                                        + key_share*
                                               {EncryptedExtensions}
                                                       + early_data*
                                                          {Finished}
                                 <--------       [Application Data*]
         (EndOfEarlyData)
         {Finished}              -------->
         [Application Data]      <------->        [Application Data]

               +  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 client_early_traffic_secret.

               {} Indicates messages protected using keys
                  derived from a [sender]_handshake_traffic_secret.

               [] Indicates messages protected using keys
                  derived from [sender]_application_traffic_secret_N
```

Figure 4: Message flow for a zero round trip handshake

IMPORTANT NOTE: The security properties for 0-RTT data are weaker than those for other kinds of TLS data. Specifically:

1. This data is not forward secret, as it is encrypted solely under keys derived using the offered PSK.

2. There are no guarantees of non-replay between connections. Protection against replay for ordinary TLS 1.3 1-RTT data is provided via the server's Random value, but 0-RTT data does not depend on the ServerHello and therefore has weaker guarantees. This is especially relevant if the data is authenticated either with TLS client authentication or inside the application

protocol. The same warnings apply to any use of the early_exporter_master_secret.

0-RTT data cannot be duplicated within a connection (i.e., the server will not process the same data twice for the same connection) and an attacker will not be able to make 0-RTT data appear to be 1-RTT data (because it is protected with different keys.) Appendix E.5 contains a description of potential attacks and Section 8 describes mechanisms which the server can use to limit the impact of replay.

3. Presentation Language

This document deals with the formatting of data in an external representation. The following very basic and somewhat casually defined presentation syntax will be used.

3.1. Basic Block Size

The representation of all data items is explicitly specified. The basic data block size is one byte (i.e., 8 bits). Multiple byte data items are concatenations of bytes, from left to right, from top to bottom. From the byte stream, a multi-byte item (a numeric in the example) is formed (using C notation) by:

```
   value = (byte[0] << 8*(n-1)) | (byte[1] << 8*(n-2)) |
           ... | byte[n-1];
```

This byte ordering for multi-byte values is the commonplace network byte order or big-endian format.

3.2. Miscellaneous

Comments begin with "/*" and end with "*/".

Optional components are denoted by enclosing them in "[[ ]]" double brackets.

Single-byte entities containing uninterpreted data are of type opaque.

A type alias T' for an existing type T is defined by:

```
   T T';
```

3.3. Numbers

The basic numeric data type is an unsigned byte (uint8). All larger numeric data types are formed from fixed-length series of bytes concatenated as described in Section 3.1 and are also unsigned. The following numeric types are predefined.

```
   uint8 uint16[2];
   uint8 uint24[3];
   uint8 uint32[4];
   uint8 uint64[8];
```

All values, here and elsewhere in the specification, are transmitted in network byte (big-endian) order; the uint32 represented by the hex bytes 01 02 03 04 is equivalent to the decimal value 16909060.

3.4. Vectors

A vector (single-dimensioned array) is a stream of homogeneous data elements. The size of the vector may be specified at documentation time or left unspecified until runtime. In either case, the length declares the number of bytes, not the number of elements, in the vector. The syntax for specifying a new type, T’, that is a fixed-length vector of type T is

```
   T T’[n];
```

Here, T’ occupies n bytes in the data stream, where n is a multiple of the size of T. The length of the vector is not included in the encoded stream.

In the following example, Datum is defined to be three consecutive bytes that the protocol does not interpret, while Data is three consecutive Datum, consuming a total of nine bytes.

```
   opaque Datum[3];      /* three uninterpreted bytes */
   Datum Data[9];        /* 3 consecutive 3-byte vectors */
```

Variable-length vectors are defined by specifying a subrange of legal lengths, inclusively, using the notation <floor..ceiling>. When these are encoded, the actual length precedes the vector’s contents in the byte stream. The length will be in the form of a number consuming as many bytes as required to hold the vector’s specified maximum (ceiling) length. A variable-length vector with an actual length field of zero is referred to as an empty vector.

```
   T T’<floor..ceiling>;
```

In the following example, mandatory is a vector that must contain between 300 and 400 bytes of type opaque. It can never be empty. The actual length field consumes two bytes, a uint16, which is sufficient to represent the value 400 (see Section 3.3). Similarly, longer can represent up to 800 bytes of data, or 400 uint16 elements, and it may be empty. Its encoding will include a two-byte actual length field prepended to the vector. The length of an encoded vector must be an exact multiple of the length of a single element (e.g., a 17-byte vector of uint16 would be illegal).

```
opaque mandatory<300..400>;
      /* length field is 2 bytes, cannot be empty */
uint16 longer<0..800>;
      /* zero to 400 16-bit unsigned integers */
```

3.5. Enumerateds

An additional sparse data type is available called enum or enumerated. Each definition is a different type. Only enumerateds of the same type may be assigned or compared. Every element of an enumerated must be assigned a value, as demonstrated in the following example. Since the elements of the enumerated are not ordered, they can be assigned any unique value, in any order.

```
enum { e1(v1), e2(v2), ... , en(vn) [[, (n)]] } Te;
```

Future extensions or additions to the protocol may define new values. Implementations need to be able to parse and ignore unknown values unless the definition of the field states otherwise.

An enumerated occupies as much space in the byte stream as would its maximal defined ordinal value. The following definition would cause one byte to be used to carry fields of type Color.

```
enum { red(3), blue(5), white(7) } Color;
```

One may optionally specify a value without its associated tag to force the width definition without defining a superfluous element.

In the following example, Taste will consume two bytes in the data stream but can only assume the values 1, 2, or 4 in the current version of the protocol.

```
enum { sweet(1), sour(2), bitter(4), (32000) } Taste;
```

The names of the elements of an enumeration are scoped within the defined type. In the first example, a fully qualified reference to the second element of the enumeration would be Color.blue. Such qualification is not required if the target of the assignment is well specified.

```
   Color color = Color.blue;     /* overspecified, legal */
   Color color = blue;           /* correct, type implicit */
```

The names assigned to enumerateds do not need to be unique. The numerical value can describe a range over which the same name applies. The value includes the minimum and maximum inclusive values in that range, separated by two period characters. This is principally useful for reserving regions of the space.

```
   enum { sad(0), meh(1..254), happy(255) } Mood;
```

3.6. Constructed Types

Structure types may be constructed from primitive types for convenience. Each specification declares a new, unique type. The syntax for definition is much like that of C.

```
   struct {
       T1 f1;
       T2 f2;
       ...
       Tn fn;
   } T;
```

Fixed- and variable-length vector fields are allowed using the standard vector syntax. Structures V1 and V2 in the variants example below demonstrate this.

The fields within a structure may be qualified using the type's name, with a syntax much like that available for enumerateds. For example, T.f2 refers to the second field of the previous declaration.

3.7. Constants

Fields and variables may be assigned a fixed value using "=", as in:

```
   struct {
       T1 f1 = 8;  /* T.f1 must always be 8 */
       T2 f2;
   } T;
```

3.8. Variants

Defined structures may have variants based on some knowledge that is available within the environment. The selector must be an enumerated type that defines the possible variants the structure defines. Each arm of the select specifies the type of that variant's field and an optional field label. The mechanism by which the variant is selected at runtime is not prescribed by the presentation language.

```
struct {
    T1 f1;
    T2 f2;
    ....
    Tn fn;
    select (E) {
        case e1: Te1 [[fe1]];
        case e2: Te2 [[fe2]];
        ....
        case en: Ten [[fen]];
    };
} Tv;
```

For example:

```
enum { apple(0), orange(1) } VariantTag;

struct {
    uint16 number;
    opaque string<0..10>; /* variable length */
} V1;

struct {
    uint32 number;
    opaque string[10];    /* fixed length */
} V2;

struct {
    VariantTag type;
    select (VariantRecord.type) {
        case apple:  V1;
        case orange: V2;
    };
} VariantRecord;
```

4. Handshake Protocol

The handshake protocol is used to negotiate the security parameters of a connection. Handshake messages are supplied to the TLS record layer, where they are encapsulated within one or more TLSPlaintext or TLSCiphertext structures, which are processed and transmitted as specified by the current active connection state.

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

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

Protocol messages MUST be sent in the order defined in Section 4.4.1 and shown in the diagrams in Section 2. A peer which receives a handshake message in an unexpected order MUST abort the handshake with an "unexpected_message" alert.

New handshake message types are assigned by IANA as described in Section 11.

4.1. Key Exchange Messages

The key exchange messages are used to determine the security capabilities of the client and the server and to establish shared secrets including the traffic keys used to protect the rest of the handshake and the data.

4.1.1. Cryptographic Negotiation

In TLS, the cryptographic negotiation proceeds by the client offering the following four sets of options in its ClientHello:

- A list of cipher suites which indicates the AEAD algorithm/HKDF hash pairs which the client supports.

- A "supported_groups" (Section 4.2.7) extension which indicates the (EC)DHE groups which the client supports and a "key_share" (Section 4.2.8) extension which contains (EC)DHE shares for some or all of these groups.

- A "signature_algorithms" (Section 4.2.3) extension which indicates the signature algorithms which the client can accept.

- A "pre_shared_key" (Section 4.2.11) extension which contains a list of symmetric key identities known to the client and a "psk_key_exchange_modes" (Section 4.2.9) extension which indicates the key exchange modes that may be used with PSKs.

If the server does not select a PSK, then the first three of these options are entirely orthogonal: the server independently selects a cipher suite, an (EC)DHE group and key share for key establishment, and a signature algorithm/certificate pair to authenticate itself to the client. If there is no overlap between the received "supported_groups" and the groups supported by the server then the server MUST abort the handshake with a "handshake_failure" or an "insufficient_security" alert.

If the server selects a PSK, then it MUST also select a key establishment mode from the set indicated by client's "psk_key_exchange_modes" extension (at present, PSK alone or with (EC)DHE). Note that if the PSK can be used without (EC)DHE then non-overlap in the "supported_groups" parameters need not be fatal, as it is in the non-PSK case discussed in the previous paragraph.

If the server selects an (EC)DHE group and the client did not offer a compatible "key_share" extension in the initial ClientHello, the server MUST respond with a HelloRetryRequest (Section 4.1.4) message.

If the server successfully selects parameters and does not require a HelloRetryRequest, it indicates the selected parameters in the ServerHello as follows:

- If PSK is being used, then the server will send a "pre_shared_key" extension indicating the selected key.

- If PSK is not being used, then (EC)DHE and certificate-based authentication are always used.

- When (EC)DHE is in use, the server will also provide a "key_share" extension.

- When authenticating via a certificate, the server will send the Certificate (Section 4.4.2) and CertificateVerify (Section 4.4.3) messages. In TLS 1.3 as defined by this document, either a PSK or a certificate is always used, but not both. Future documents may define how to use them together.

If the server is unable to negotiate a supported set of parameters (i.e., there is no overlap between the client and server parameters), it MUST abort the handshake with either a "handshake_failure" or "insufficient_security" fatal alert (see Section 6).

4.1.2. Client Hello

When a client first connects to a server, it is REQUIRED to send the ClientHello as its first TLS message. The client will also send a ClientHello when the server has responded to its ClientHello with a HelloRetryRequest. In that case, the client MUST send the same ClientHello without modification, except:

- If a "key_share" extension was supplied in the HelloRetryRequest, replacing the list of shares with a list containing a single KeyShareEntry from the indicated group.

- Removing the "early_data" extension (Section 4.2.10) if one was present. Early data is not permitted after HelloRetryRequest.

- Including a "cookie" extension if one was provided in the HelloRetryRequest.

- Updating the "pre_shared_key" extension if present by recomputing the "obfuscated_ticket_age" and binder values and (optionally) removing any PSKs which are incompatible with the server’s indicated cipher suite.

- Optionally adding, removing, or changing the length of the "padding" extension [RFC7685].

- Other modifications that may be allowed by an extension defined in the future and present in the HelloRetryRequest.

Because TLS 1.3 forbids renegotiation, if a server has negotiated TLS 1.3 and receives a ClientHello at any other time, it MUST terminate the connection with an "unexpected_message" alert.

If a server established a TLS connection with a previous version of TLS and receives a TLS 1.3 ClientHello in a renegotiation, it MUST retain the previous protocol version. In particular, it MUST NOT negotiate TLS 1.3.

Structure of this message:

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

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

   struct {
       ProtocolVersion legacy_version = 0x0303;    /* TLS v1.2 */
       Random random;
       opaque legacy_session_id<0..32>;
       CipherSuite cipher_suites<2..2^16-2>;
       opaque legacy_compression_methods<1..2^8-1>;
       Extension extensions<8..2^16-1>;
   } ClientHello;
```

legacy_version In previous versions of TLS, this field was used for version negotiation and represented the highest version number supported by the client. Experience has shown that many servers do not properly implement version negotiation, leading to "version intolerance" in which the server rejects an otherwise acceptable ClientHello with a version number higher than it supports. In TLS 1.3, the client indicates its version preferences in the "supported_versions" extension (Section 4.2.1) and the legacy_version field MUST be set to 0x0303, which is the version number for TLS 1.2. (See Appendix D for details about backward compatibility.)

random 32 bytes generated by a secure random number generator. See Appendix C for additional information.

legacy_session_id Versions of TLS before TLS 1.3 supported a "session resumption" feature which has been merged with Pre-Shared

Keys in this version (see Section 2.2). A client which has a cached session ID set by a pre-TLS 1.3 server SHOULD set this field to that value. In compatibility mode (see Appendix D.4) this field MUST be non-empty, so a client not offering a pre-TLS 1.3 session MUST generate a new 32-byte value. This value need not be random but SHOULD be unpredictable to avoid implementations fixating on a specific value (also known as ossification). Otherwise, it MUST be set as a zero length vector (i.e., a single zero byte length field).

cipher_suites This is a list of the symmetric cipher options supported by the client, specifically the record protection algorithm (including secret key length) and a hash to be used with HKDF, in descending order of client preference. If the list contains cipher suites that the server does not recognize, support or wish to use, the server MUST ignore those cipher suites and process the remaining ones as usual. Values are defined in Appendix B.4. If the client is attempting a PSK key establishment, it SHOULD advertise at least one cipher suite indicating a Hash associated with the PSK.

legacy_compression_methods Versions of TLS before 1.3 supported compression with the list of supported compression methods being sent in this field. For every TLS 1.3 ClientHello, this vector MUST contain exactly one byte, set to zero, which corresponds to the "null" compression method in prior versions of TLS. If a TLS 1.3 ClientHello is received with any other value in this field, the server MUST abort the handshake with an "illegal_parameter" alert. Note that TLS 1.3 servers might receive TLS 1.2 or prior ClientHellos which contain other compression methods and (if negotiating such a prior version) MUST follow the procedures for the appropriate prior version of TLS. TLS 1.3 ClientHellos are identified as having a legacy_version of 0x0303 and a supported_versions extension present with 0x0304 as the highest version indicated therein.

extensions Clients request extended functionality from servers by sending data in the extensions field. The actual "Extension" format is defined in Section 4.2. In TLS 1.3, use of certain extensions is mandatory, as functionality is moved into extensions to preserve ClientHello compatibility with previous versions of TLS. Servers MUST ignore unrecognized extensions.

All versions of TLS allow an extensions field to optionally follow the compression_methods field. TLS 1.3 ClientHello messages always contain extensions (minimally "supported_versions", otherwise they will be interpreted as TLS 1.2 ClientHello messages). However, TLS 1.3 servers might receive ClientHello messages without an extensions field from prior versions of TLS. The presence of extensions can be detected by determining whether there are bytes following the compression_methods field at the end of the ClientHello. Note that this method of detecting optional data differs from the normal TLS method of having a variable-length field, but it is used for compatibility with TLS before extensions were defined. TLS 1.3 servers will need to perform this check first and only attempt to negotiate TLS 1.3 if the "supported_versions" extension is present. If negotiating a version of TLS prior to 1.3, a server MUST check that the message either contains no data after legacy_compression_methods or that it contains a valid extensions block with no data following. If not, then it MUST abort the handshake with a "decode_error" alert.

In the event that a client requests additional functionality using extensions, and this functionality is not supplied by the server, the client MAY abort the handshake.

After sending the ClientHello message, the client waits for a ServerHello or HelloRetryRequest message. If early data is in use, the client may transmit early application data (Section 2.3) while waiting for the next handshake message.

4.1.3. Server Hello

The server will send this message in response to a ClientHello message to proceed with the handshake if it is able to negotiate an acceptable set of handshake parameters based on the ClientHello.

Structure of this message:

```
   struct {
       ProtocolVersion legacy_version = 0x0303;    /* TLS v1.2 */
       Random random;
       opaque legacy_session_id_echo<0..32>;
       CipherSuite cipher_suite;
       uint8 legacy_compression_method = 0;
       Extension extensions<6..2^16-1>;
   } ServerHello;
```

legacy_version In previous versions of TLS, this field was used for version negotiation and represented the selected version number for the connection. Unfortunately, some middleboxes fail when presented with new values. In TLS 1.3, the TLS server indicates its version using the "supported_versions" extension (Section 4.2.1), and the legacy_version field MUST be set to 0x0303, which is the version number for TLS 1.2. (See Appendix D for details about backward compatibility.)

random 32 bytes generated by a secure random number generator. See Appendix C for additional information. The last eight bytes MUST be overwritten as described below if negotiating TLS 1.2 or TLS 1.1, but the remaining bytes MUST be random. This structure is generated by the server and MUST be generated independently of the ClientHello.random.

legacy_session_id_echo The contents of the client's legacy_session_id field. Note that this field is echoed even if the client's value corresponded to a cached pre-TLS 1.3 session which the server has chosen not to resume. A client which receives a legacy_session_id_echo field that does not match what it sent in the ClientHello MUST abort the handshake with an "illegal_parameter" alert.

cipher_suite The single cipher suite selected by the server from the list in ClientHello.cipher_suites. A client which receives a cipher suite that was not offered MUST abort the handshake with an "illegal_parameter" alert.

legacy_compression_method A single byte which MUST have the value 0.

extensions A list of extensions. The ServerHello MUST only include extensions which are required to establish the cryptographic context and negotiate the protocol version. All TLS 1.3 ServerHello messages MUST contain the "supported_versions" extension. Current ServerHello messages additionally contain either the "pre_shared_key" or "key_share" extensions, or both when using a PSK with (EC)DHE key establishment. Other extensions are sent separately in the EncryptedExtensions message.

For reasons of backward compatibility with middleboxes (see Appendix D.4) the HelloRetryRequest message uses the same structure as the ServerHello, but with Random set to the special value of the SHA-256 of "HelloRetryRequest":

```
  CF 21 AD 74 E5 9A 61 11 BE 1D 8C 02 1E 65 B8 91
  C2 A2 11 16 7A BB 8C 5E 07 9E 09 E2 C8 A8 33 9C
```

Upon receiving a message with type server_hello, implementations MUST first examine the Random value and if it matches this value, process it as described in Section 4.1.4).

TLS 1.3 has a downgrade protection mechanism embedded in the server's random value. TLS 1.3 servers which negotiate TLS 1.2 or below in response to a ClientHello MUST set the last eight bytes of their Random value specially.

If negotiating TLS 1.2, TLS 1.3 servers MUST set the last eight bytes of their Random value to the bytes:

```
44 4F 57 4E 47 52 44 01
```

If negotiating TLS 1.1 or below, TLS 1.3 servers MUST and TLS 1.2 servers SHOULD set the last eight bytes of their Random value to the bytes:

```
44 4F 57 4E 47 52 44 00
```

TLS 1.3 clients receiving a ServerHello indicating TLS 1.2 or below MUST check that the last eight bytes are not equal to either of these values. TLS 1.2 clients SHOULD also check that the last eight bytes are not equal to the second value if the ServerHello indicates TLS 1.1 or below. If a match is found, the client MUST abort the handshake with an "illegal_parameter" alert. This mechanism provides limited protection against downgrade attacks over and above what is provided by the Finished exchange: because the ServerKeyExchange, a message present in TLS 1.2 and below, includes a signature over both random values, it is not possible for an active attacker to modify the random values without detection as long as ephemeral ciphers are used. It does not provide downgrade protection when static RSA is used.

Note: This is a change from [RFC5246], so in practice many TLS 1.2 clients and servers will not behave as specified above.

A legacy TLS client performing renegotiation with TLS 1.2 or prior and which receives a TLS 1.3 ServerHello during renegotiation MUST abort the handshake with a "protocol_version" alert. Note that renegotiation is not possible when TLS 1.3 has been negotiated.

RFC EDITOR: PLEASE REMOVE THE FOLLOWING PARAGRAPH Implementations of draft versions (see Section 4.2.1.1) of this specification SHOULD NOT implement this mechanism on either client and server. A pre-RFC client connecting to RFC servers, or vice versa, will appear to downgrade to TLS 1.2. With the mechanism enabled, this will cause an interoperability failure.

4.1.4. Hello Retry Request

The server will send this message in response to a ClientHello message if it is able to find an acceptable set of parameters but the ClientHello does not contain sufficient information to proceed with the handshake. As discussed in Section 4.1.3, the HelloRetryRequest has the same format as a ServerHello message, and the legacy_version, legacy_session_id_echo, cipher_suite, and legacy_compression methods fields have the same meaning. However, for convenience we discuss HelloRetryRequest throughout this document as if it were a distinct message.

The server's extensions MUST contain "supported_versions" and otherwise the server SHOULD send only the extensions necessary for the client to generate a correct ClientHello pair. As with ServerHello, a HelloRetryRequest MUST NOT contain any extensions that were not first offered by the client in its ClientHello, with the exception of optionally the "cookie" (see Section 4.2.2) extension.

Upon receipt of a HelloRetryRequest, the client MUST check the legacy_version, legacy_session_id_echo, cipher_suite, and legacy_compression_method as specified in Section 4.1.3 and then process the extensions, starting with determining the version using "supported_versions". Clients MUST abort the handshake with an "illegal_parameter" alert if the HelloRetryRequest would not result in any change in the ClientHello. If a client receives a second HelloRetryRequest in the same connection (i.e., where the ClientHello was itself in response to a HelloRetryRequest), it MUST abort the handshake with an "unexpected_message" alert.

Otherwise, the client MUST process all extensions in the HelloRetryRequest and send a second updated ClientHello. The HelloRetryRequest extensions defined in this specification are:

- supported_versions (see Section 4.2.1)

- cookie (see Section 4.2.2)

- key_share (see Section 4.2.8)

In addition, in its updated ClientHello, the client SHOULD NOT offer any pre-shared keys associated with a hash other than that of the selected cipher suite. This allows the client to avoid having to compute partial hash transcripts for multiple hashes in the second ClientHello. A client which receives a cipher suite that was not offered MUST abort the handshake. Servers MUST ensure that they negotiate the same cipher suite when receiving a conformant updated ClientHello (if the server selects the cipher suite as the first step in the negotiation, then this will happen automatically). Upon receiving the ServerHello, clients MUST check that the cipher suite supplied in the ServerHello is the same as that in the HelloRetryRequest and otherwise abort the handshake with an "illegal_parameter" alert.

The value of selected_version in the HelloRetryRequest "supported_versions" extension MUST be retained in the ServerHello, and a client MUST abort the handshake with an "illegal_parameter" alert if the value changes.

4.2. Extensions

A number of TLS messages contain tag-length-value encoded extensions structures.

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

   enum {
       server_name(0),                             /* RFC 6066 */
       max_fragment_length(1),                     /* RFC 6066 */
       status_request(5),                          /* RFC 6066 */
       supported_groups(10),                       /* RFC 4492, 7919 */
       signature_algorithms(13),                   /* [[this document]] */
       use_srtp(14),                               /* RFC 5764 */
       heartbeat(15),                              /* RFC 6520 */
       application_layer_protocol_negotiation(16), /* RFC 7301 */
       signed_certificate_timestamp(18),           /* RFC 6962 */
       client_certificate_type(19),                /* RFC 7250 */
       server_certificate_type(20),                /* RFC 7250 */
       padding(21),                                /* RFC 7685 */
       pre_shared_key(41),                         /* [[this document]] */
       early_data(42),                             /* [[this document]] */
       supported_versions(43),                     /* [[this document]] */
       cookie(44),                                 /* [[this document]] */
       psk_key_exchange_modes(45),                 /* [[this document]] */
       certificate_authorities(47),                /* [[this document]] */
       oid_filters(48),                            /* [[this document]] */
       post_handshake_auth(49),                    /* [[this document]] */
       signature_algorithms_cert(50),              /* [[this document]] */
       key_share(51),                              /* [[this document]] */
       (65535)
   } ExtensionType;
```

Here:

- "extension_type" identifies the particular extension type.

- "extension_data" contains information specific to the particular extension type.

The list of extension types is maintained by IANA as described in Section 11.

Extensions are generally structured in a request/response fashion, though some extensions are just indications with no corresponding response. The client sends its extension requests in the ClientHello message and the server sends its extension responses in the ServerHello, EncryptedExtensions, HelloRetryRequest and Certificate messages. The server sends extension requests in the CertificateRequest message which a client MAY respond to with a Certificate message. The server MAY also send unsolicited extensions in the NewSessionTicket, though the client does not respond directly to these.

Implementations MUST NOT send extension responses if the remote endpoint did not send the corresponding extension requests, with the exception of the "cookie" extension in HelloRetryRequest. Upon receiving such an extension, an endpoint MUST abort the handshake with an "unsupported_extension" alert.

The table below indicates the messages where a given extension may appear, using the following notation: CH (ClientHello), SH (ServerHello), EE (EncryptedExtensions), CT (Certificate), CR (CertificateRequest), NST (NewSessionTicket) and HRR (HelloRetryRequest). If an implementation receives an extension which it recognizes and which is not specified for the message in which it appears it MUST abort the handshake with an "illegal_parameter" alert.

Extension	TLS 1.3
server_name [RFC6066]	CH, EE
max_fragment_length [RFC6066]	CH, EE
status_request [RFC6066]	CH, CR, CT
supported_groups [RFC7919]	CH, EE
signature_algorithms [RFC5246]	CH, CR
use_srtp [RFC5764]	CH, EE
heartbeat [RFC6520]	CH, EE
application_layer_protocol_negotiation [RFC7301]	CH, EE
signed_certificate_timestamp [RFC6962]	CH, CR, CT
client_certificate_type [RFC7250]	CH, EE
server_certificate_type [RFC7250]	CH, EE
padding [RFC7685]	CH
key_share [[this document]]	CH, SH, HRR
pre_shared_key [[this document]]	CH, SH
psk_key_exchange_modes [[this document]]	CH
early_data [[this document]]	CH, EE, NST
cookie [[this document]]	CH, HRR
supported_versions [[this document]]	CH, SH, HRR
certificate_authorities [[this document]]	CH, CR
oid_filters [[this document]]	CR
post_handshake_auth [[this document]]	CH
signature_algorithms_cert [[this document]]	CH, CR

When multiple extensions of different types are present, the extensions MAY appear in any order, with the exception of "pre_shared_key" Section 4.2.11 which MUST be the last extension in the ClientHello. There MUST NOT be more than one extension of the same type in a given extension block.

In TLS 1.3, unlike TLS 1.2, extensions are negotiated for each handshake even when in resumption-PSK mode. However, 0-RTT parameters are those negotiated in the previous handshake; mismatches may require rejecting 0-RTT (see Section 4.2.10).

There are subtle (and not so subtle) interactions that may occur in this protocol between new features and existing features which may result in a significant reduction in overall security. The following considerations should be taken into account when designing new extensions:

- Some cases where a server does not agree to an extension are error conditions (e.g., the handshake cannot continue), and some are simply refusals to support particular features. In general, error alerts should be used for the former and a field in the server extension response for the latter.

- Extensions should, as far as possible, be designed to prevent any attack that forces use (or non-use) of a particular feature by manipulation of handshake messages. This principle should be followed regardless of whether the feature is believed to cause a security problem. Often the fact that the extension fields are included in the inputs to the Finished message hashes will be sufficient, but extreme care is needed when the extension changes the meaning of messages sent in the handshake phase. Designers and implementors should be aware of the fact that until the handshake has been authenticated, active attackers can modify messages and insert, remove, or replace extensions.

4.2.1. Supported Versions

```
   struct {
       select (Handshake.msg_type) {
           case client_hello:
                ProtocolVersion versions<2..254>;

           case server_hello: /* and HelloRetryRequest */
                ProtocolVersion selected_version;
       };
   } SupportedVersions;
```

The "supported_versions" extension is used by the client to indicate which versions of TLS it supports and by the server to indicate which version it is using. The extension contains a list of supported versions in preference order, with the most preferred version first. Implementations of this specification MUST send this extension in the ClientHello containing all versions of TLS which they are prepared to negotiate (for this specification, that means minimally 0x0304, but if previous versions of TLS are allowed to be negotiated, they MUST be present as well).

If this extension is not present, servers which are compliant with this specification, and which also support TLS 1.2, MUST negotiate TLS 1.2 or prior as specified in [RFC5246], even if ClientHello.legacy_version is 0x0304 or later. Servers MAY abort the handshake upon receiving a ClientHello with legacy_version 0x0304 or later.

If this extension is present in the ClientHello, servers MUST NOT use the ClientHello.legacy_version value for version negotiation and MUST use only the "supported_versions" extension to determine client preferences. Servers MUST only select a version of TLS present in that extension and MUST ignore any unknown versions that are present in that extension. Note that this mechanism makes it possible to negotiate a version prior to TLS 1.2 if one side supports a sparse range. Implementations of TLS 1.3 which choose to support prior versions of TLS SHOULD support TLS 1.2. Servers MUST be prepared to receive ClientHellos that include this extension but do not include 0x0304 in the list of versions.

A server which negotiates a version of TLS prior to TLS 1.3 MUST set ServerHello.version and MUST NOT send the "supported_versions" extension. A server which negotiates TLS 1.3 MUST respond by sending a "supported_versions" extension containing the selected version value (0x0304). It MUST set the ServerHello.legacy_version field to 0x0303 (TLS 1.2). Clients MUST check for this extension prior to processing the rest of the ServerHello (although they will have to parse the ServerHello in order to read the extension). If this extension is present, clients MUST ignore the ServerHello.legacy_version value and MUST use only the "supported_versions" extension to determine the selected version. If the "supported_versions" extension in the ServerHello contains a version not offered by the client or contains a version prior to TLS 1.3, the client MUST abort the handshake with an "illegal_parameter" alert.

4.2.1.1. Draft Version Indicator

RFC EDITOR: PLEASE REMOVE THIS SECTION

While the eventual version indicator for the RFC version of TLS 1.3 will be 0x0304, implementations of draft versions of this specification SHOULD instead advertise 0x7f00 | draft_version in the ServerHello and HelloRetryRequest "supported_versions" extension. For instance, draft-17 would be encoded as the 0x7f11. This allows pre-RFC implementations to safely negotiate with each other, even if they would otherwise be incompatible.

4.2.2. Cookie

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

Cookies serve two primary purposes:

- Allowing the server to force the client to demonstrate reachability at their apparent network address (thus providing a measure of DoS protection). This is primarily useful for non-connection-oriented transports (see [RFC6347] for an example of this).

- Allowing the server to offload state to the client, thus allowing it to send a HelloRetryRequest without storing any state. The server can do this by storing the hash of the ClientHello in the HelloRetryRequest cookie (protected with some suitable integrity algorithm).

When sending a HelloRetryRequest, the server MAY provide a "cookie" extension to the client (this is an exception to the usual rule that the only extensions that may be sent are those that appear in the ClientHello). When sending the new ClientHello, the client MUST copy the contents of the extension received in the HelloRetryRequest into a "cookie" extension in the new ClientHello. Clients MUST NOT use cookies in their initial ClientHello in subsequent connections.

When a server is operating statelessly it may receive an unprotected record of type change_cipher_spec between the first and second ClientHello (see Section 5). Since the server is not storing any state this will appear as if it were the first message to be received. Servers operating statelessly MUST ignore these records.

4.2.3. Signature Algorithms

TLS 1.3 provides two extensions for indicating which signature algorithms may be used in digital signatures. The "signature_algorithms_cert" extension applies to signatures in certificates and the "signature_algorithms" extension, which originally appeared in TLS 1.2, applies to signatures in CertificateVerify messages. The keys found in certificates MUST also be of appropriate type for the signature algorithms they are used with. This is a particular issue for RSA keys and PSS signatures, as described below. If no "signature_algorithms_cert" extension is present, then the "signature_algorithms" extension also applies to signatures appearing in certificates. Clients which desire the server to authenticate itself via a certificate MUST send "signature_algorithms". If a server is authenticating via a certificate and the client has not sent a "signature_algorithms" extension, then the server MUST abort the handshake with a "missing_extension" alert (see Section 9.2).

The "signature_algorithms_cert" extension was added to allow implementations which supported different sets of algorithms for certificates and in TLS itself to clearly signal their capabilities. TLS 1.2 implementations SHOULD also process this extension. Implementations which have the same policy in both cases MAY omit the "signature_algorithms_cert" extension.

The "extension_data" field of these extensions contains a SignatureSchemeList value:

```
      enum {
          /* RSASSA-PKCS1-v1_5 algorithms */
          rsa_pkcs1_sha256(0x0401),
          rsa_pkcs1_sha384(0x0501),
          rsa_pkcs1_sha512(0x0601),

          /* ECDSA algorithms */
          ecdsa_secp256r1_sha256(0x0403),
          ecdsa_secp384r1_sha384(0x0503),
          ecdsa_secp521r1_sha512(0x0603),

          /* RSASSA-PSS algorithms with public key OID rsaEncryption */
          rsa_pss_rsae_sha256(0x0804),
          rsa_pss_rsae_sha384(0x0805),
          rsa_pss_rsae_sha512(0x0806),

          /* EdDSA algorithms */
          ed25519(0x0807),
          ed448(0x0808),

          /* RSASSA-PSS algorithms with public key OID RSASSA-PSS */
          rsa_pss_pss_sha256(0x0809),
          rsa_pss_pss_sha384(0x080a),
          rsa_pss_pss_sha512(0x080b),

          /* Legacy algorithms */
          rsa_pkcs1_sha1(0x0201),
          ecdsa_sha1(0x0203),

          /* Reserved Code Points */
          private_use(0xFE00..0xFFFF),
          (0xFFFF)
      } SignatureScheme;

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

Note: This enum is named "SignatureScheme" because there is already a "SignatureAlgorithm" type in TLS 1.2, which this replaces. We use the term "signature algorithm" throughout the text.

Each SignatureScheme value lists a single signature algorithm that the client is willing to verify. The values are indicated in descending order of preference. Note that a signature algorithm takes as input an arbitrary-length message, rather than a digest. Algorithms which traditionally act on a digest should be defined in TLS to first hash the input with a specified hash algorithm and then proceed as usual. The code point groups listed above have the following meanings:

RSASSA-PKCS1-v1_5 algorithms Indicates a signature algorithm using RSASSA-PKCS1-v1_5 [RFC8017] with the corresponding hash algorithm as defined in [SHS]. These values refer solely to signatures which appear in certificates (see Section 4.4.2.2) and are not defined for use in signed TLS handshake messages, although they MAY appear in "signature_algorithms" and "signature_algorithms_cert" for backward compatibility with TLS 1.2,

ECDSA algorithms Indicates a signature algorithm using ECDSA [ECDSA], the corresponding curve as defined in ANSI X9.62 [X962] and FIPS 186-4 [DSS], and the corresponding hash algorithm as defined in [SHS]. The signature is represented as a DER-encoded [X690] ECDSA-Sig-Value structure.

RSASSA-PSS RSAE algorithms Indicates a signature algorithm using RSASSA-PSS [RFC8017] with mask generation function 1. The digest used in the mask generation function and the digest being signed are both the corresponding hash algorithm as defined in [SHS]. The length of the salt MUST be equal to the length of the output of the digest algorithm. If the public key is carried in an X.509 certificate, it MUST use the rsaEncryption OID [RFC5280].

EdDSA algorithms Indicates a signature algorithm using EdDSA as defined in [RFC8032] or its successors. Note that these correspond to the "PureEdDSA" algorithms and not the "prehash" variants.

RSASSA-PSS PSS algorithms Indicates a signature algorithm using RSASSA-PSS [RFC8017] with mask generation function 1. The digest used in the mask generation function and the digest being signed are both the corresponding hash algorithm as defined in [SHS]. The length of the salt MUST be equal to the length of the digest algorithm. If the public key is carried in an X.509 certificate, it MUST use the RSASSA-PSS OID [RFC5756]. When used in certificate signatures, the algorithm parameters MUST be DER encoded. If the corresponding public key's parameters are present, then the parameters in the signature MUST be identical to those in the public key.

Legacy algorithms Indicates algorithms which are being deprecated because they use algorithms with known weaknesses, specifically SHA-1 which is used in this context with either with RSA using RSASSA-PKCS1-v1_5 or ECDSA. These values refer solely to signatures which appear in certificates (see Section 4.4.2.2) and are not defined for use in signed TLS handshake messages, although they MAY appear in "signature_algorithms" and "signature_algorithms_cert" for backward compatibility with TLS 1.2, Endpoints SHOULD NOT negotiate these algorithms but are permitted to do so solely for backward compatibility. Clients offering these values MUST list them as the lowest priority (listed after all other algorithms in SignatureSchemeList). TLS 1.3 servers MUST NOT offer a SHA-1 signed certificate unless no valid certificate chain can be produced without it (see Section 4.4.2.2).

The signatures on certificates that are self-signed or certificates that are trust anchors are not validated since they begin a certification path (see [RFC5280], Section 3.2). A certificate that begins a certification path MAY use a signature algorithm that is not advertised as being supported in the "signature_algorithms" extension.

Note that TLS 1.2 defines this extension differently. TLS 1.3 implementations willing to negotiate TLS 1.2 MUST behave in accordance with the requirements of [RFC5246] when negotiating that version. In particular:

- TLS 1.2 ClientHellos MAY omit this extension.

- In TLS 1.2, the extension contained hash/signature pairs. The pairs are encoded in two octets, so SignatureScheme values have been allocated to align with TLS 1.2's encoding. Some legacy pairs are left unallocated. These algorithms are deprecated as of TLS 1.3. They MUST NOT be offered or negotiated by any implementation. In particular, MD5 [SLOTH], SHA-224, and DSA MUST NOT be used.

- ECDSA signature schemes align with TLS 1.2's ECDSA hash/signature pairs. However, the old semantics did not constrain the signing curve. If TLS 1.2 is negotiated, implementations MUST be prepared to accept a signature that uses any curve that they advertised in the "supported_groups" extension.

- Implementations that advertise support for RSASSA-PSS (which is mandatory in TLS 1.3), MUST be prepared to accept a signature using that scheme even when TLS 1.2 is negotiated. In TLS 1.2, RSASSA-PSS is used with RSA cipher suites.

4.2.4. Certificate Authorities

The "certificate_authorities" extension is used to indicate the certificate authorities which an endpoint supports and which SHOULD be used by the receiving endpoint to guide certificate selection.

The body of the "certificate_authorities" extension consists of a CertificateAuthoritiesExtension structure.

```
   opaque DistinguishedName<1..2^16-1>;

   struct {
       DistinguishedName authorities<3..2^16-1>;
   } CertificateAuthoritiesExtension;
```

authorities A list of the distinguished names [X501] of acceptable certificate authorities, represented in DER-encoded [X690] format. These distinguished names specify a desired distinguished name for trust anchor or subordinate CA; thus, this message can be used to describe known trust anchors as well as a desired authorization space.

The client MAY send the "certificate_authorities" extension in the ClientHello message. The server MAY send it in the CertificateRequest message.

The "trusted_ca_keys" extension, which serves a similar purpose [RFC6066], but is more complicated, is not used in TLS 1.3 (although it may appear in ClientHello messages from clients which are offering prior versions of TLS).

4.2.5. OID Filters

The "oid_filters" extension allows servers to provide a set of OID/value pairs which it would like the client's certificate to match. This extension, if provided by the server, MUST only be sent in the CertificateRequest message.

```
   struct {
       opaque certificate_extension_oid<1..2^8-1>;
       opaque certificate_extension_values<0..2^16-1>;
   } OIDFilter;

   struct {
       OIDFilter filters<0..2^16-1>;
   } OIDFilterExtension;
```

filters A list of certificate extension OIDs [RFC5280] with their allowed value(s) and represented in DER-encoded [X690] format. Some certificate extension OIDs allow multiple values (e.g., Extended Key Usage). If the server has included a non-empty filters list, the client certificate included in the response MUST contain all of the specified extension OIDs that the client recognizes. For each extension OID recognized by the client, all of the specified values MUST be present in the client certificate (but the certificate MAY have other values as well). However, the client MUST ignore and skip any unrecognized certificate extension OIDs. If the client ignored some of the required certificate extension OIDs and supplied a certificate that does not satisfy the request, the server MAY at its discretion either continue the connection without client authentication, or abort the handshake with an "unsupported_certificate" alert. Any given OID MUST NOT appear more than once in the filters list.

PKIX RFCs define a variety of certificate extension OIDs and their corresponding value types. Depending on the type, matching certificate extension values are not necessarily bitwise-equal. It is expected that TLS implementations will rely on their PKI libraries to perform certificate selection using certificate extension OIDs.

This document defines matching rules for two standard certificate extensions defined in [RFC5280]:

- The Key Usage extension in a certificate matches the request when all key usage bits asserted in the request are also asserted in the Key Usage certificate extension.

- The Extended Key Usage extension in a certificate matches the request when all key purpose OIDs present in the request are also found in the Extended Key Usage certificate extension. The special anyExtendedKeyUsage OID MUST NOT be used in the request.

Separate specifications may define matching rules for other certificate extensions.

4.2.6. Post-Handshake Client Authentication

The "post_handshake_auth" extension is used to indicate that a client is willing to perform post-handshake authentication (Section 4.6.2). Servers MUST NOT send a post-handshake CertificateRequest to clients which do not offer this extension. Servers MUST NOT send this extension.

```
   struct {} PostHandshakeAuth;
```

The "extension_data" field of the "post_handshake_auth" extension is zero length.

4.2.7. Negotiated Groups

When sent by the client, the "supported_groups" extension indicates the named groups which the client supports for key exchange, ordered from most preferred to least preferred.

Note: In versions of TLS prior to TLS 1.3, this extension was named "elliptic_curves" and only contained elliptic curve groups. See [RFC4492] and [RFC7919]. This extension was also used to negotiate ECDSA curves. Signature algorithms are now negotiated independently (see Section 4.2.3).

The "extension_data" field of this extension contains a "NamedGroupList" value:

```
   enum {

       /* Elliptic Curve Groups (ECDHE) */
       secp256r1(0x0017), secp384r1(0x0018), secp521r1(0x0019),
       x25519(0x001D), x448(0x001E),

       /* Finite Field Groups (DHE) */
       ffdhe2048(0x0100), ffdhe3072(0x0101), ffdhe4096(0x0102),
       ffdhe6144(0x0103), ffdhe8192(0x0104),

       /* Reserved Code Points */
       ffdhe_private_use(0x01FC..0x01FF),
       ecdhe_private_use(0xFE00..0xFEFF),
       (0xFFFF)
   } NamedGroup;

   struct {
       NamedGroup named_group_list<2..2^16-1>;
   } NamedGroupList;
```

Elliptic Curve Groups (ECDHE) Indicates support for the corresponding named curve, defined either in FIPS 186-4 [DSS] or in [RFC7748]. Values 0xFE00 through 0xFEFF are reserved for private use.

Finite Field Groups (DHE) Indicates support of the corresponding finite field group, defined in [RFC7919]. Values 0x01FC through 0x01FF are reserved for private use.

Items in named_group_list are ordered according to the client's preferences (most preferred choice first).

As of TLS 1.3, servers are permitted to send the "supported_groups" extension to the client. Clients MUST NOT act upon any information found in "supported_groups" prior to successful completion of the handshake but MAY use the information learned from a successfully completed handshake to change what groups they use in their "key_share" extension in subsequent connections. If the server has a group it prefers to the ones in the "key_share" extension but is still willing to accept the ClientHello, it SHOULD send "supported_groups" to update the client's view of its preferences; this extension SHOULD contain all groups the server supports, regardless of whether they are currently supported by the client.

4.2.8. Key Share

The "key_share" extension contains the endpoint's cryptographic parameters.

Clients MAY send an empty client_shares vector in order to request group selection from the server at the cost of an additional round trip. (see Section 4.1.4)

```
struct {
    NamedGroup group;
    opaque key_exchange<1..2^16-1>;
} KeyShareEntry;
```

group The named group for the key being exchanged.

key_exchange Key exchange information. The contents of this field are determined by the specified group and its corresponding definition. Finite Field Diffie-Hellman [DH] parameters are described in Section 4.2.8.1; Elliptic Curve Diffie-Hellman parameters are described in Section 4.2.8.2.

In the ClientHello message, the "extension_data" field of this extension contains a "KeyShareClientHello" value:

```
struct {
    KeyShareEntry client_shares<0..2^16-1>;
} KeyShareClientHello;
```

client_shares A list of offered KeyShareEntry values in descending order of client preference.

This vector MAY be empty if the client is requesting a HelloRetryRequest. Each KeyShareEntry value MUST correspond to a group offered in the "supported_groups" extension and MUST appear in the same order. However, the values MAY be a non-contiguous subset of the "supported_groups" extension and MAY omit the most preferred groups. Such a situation could arise if the most preferred groups are new and unlikely to be supported in enough places to make pregenerating key shares for them efficient.

Clients can offer as many KeyShareEntry values as the number of supported groups it is offering, each representing a single set of key exchange parameters. For instance, a client might offer shares for several elliptic curves or multiple FFDHE groups. The key_exchange values for each KeyShareEntry MUST be generated independently. Clients MUST NOT offer multiple KeyShareEntry values for the same group. Clients MUST NOT offer any KeyShareEntry values for groups not listed in the client's "supported_groups" extension. Servers MAY check for violations of these rules and abort the handshake with an "illegal_parameter" alert if one is violated.

In a HelloRetryRequest message, the "extension_data" field of this extension contains a KeyShareHelloRetryRequest value:

```
struct {
    NamedGroup selected_group;
} KeyShareHelloRetryRequest;
```

selected_group The mutually supported group the server intends to negotiate and is requesting a retried ClientHello/KeyShare for.

Upon receipt of this extension in a HelloRetryRequest, the client MUST verify that (1) the selected_group field corresponds to a group which was provided in the "supported_groups" extension in the original ClientHello; and (2) the selected_group field does not correspond to a group which was provided in the "key_share" extension in the original ClientHello. If either of these checks fails, then the client MUST abort the handshake with an "illegal_parameter" alert. Otherwise, when sending the new ClientHello, the client MUST replace the original "key_share" extension with one containing only a new KeyShareEntry for the group indicated in the selected_group field of the triggering HelloRetryRequest.

In a ServerHello message, the "extension_data" field of this extension contains a KeyShareServerHello value:

```
struct {
    KeyShareEntry server_share;
} KeyShareServerHello;
```

server_share A single KeyShareEntry value that is in the same group as one of the client's shares.

If using (EC)DHE key establishment, servers offer exactly one KeyShareEntry in the ServerHello. This value MUST be in the same group as the KeyShareEntry value offered by the client that the server has selected for the negotiated key exchange. Servers MUST NOT send a KeyShareEntry for any group not indicated in the "supported_groups" extension and MUST NOT send a KeyShareEntry when using the "psk_ke" PskKeyExchangeMode. If using (EC)DHE key establishment, and a HelloRetryRequest containing a "key_share" extension was received by the client, the client MUST verify that the selected NamedGroup in the ServerHello is the same as that in the HelloRetryRequest. If this check fails, the client MUST abort the handshake with an "illegal_parameter" alert.

4.2.8.1. Diffie-Hellman Parameters

Diffie-Hellman [DH] parameters for both clients and servers are encoded in the opaque key_exchange field of a KeyShareEntry in a KeyShare structure. The opaque value contains the Diffie-Hellman public value (Y = g^X mod p) for the specified group (see [RFC7919] for group definitions) encoded as a big-endian integer and padded to the left with zeros to the size of p in bytes.

Note: For a given Diffie-Hellman group, the padding results in all public keys having the same length.

Peers MUST validate each other's public key Y by ensuring that 1 < Y < p-1. This check ensures that the remote peer is properly behaved and isn't forcing the local system into a small subgroup.

4.2.8.2. ECDHE Parameters

ECDHE parameters for both clients and servers are encoded in the opaque key_exchange field of a KeyShareEntry in a KeyShare structure.

For secp256r1, secp384r1 and secp521r1, the contents are the serialized value of the following struct:

```
   struct {
       uint8 legacy_form = 4;
       opaque X[coordinate_length];
       opaque Y[coordinate_length];
   } UncompressedPointRepresentation;
```

X and Y respectively are the binary representations of the x and y values in network byte order. There are no internal length markers, so each number representation occupies as many octets as implied by the curve parameters. For P-256 this means that each of X and Y use 32 octets, padded on the left by zeros if necessary. For P-384 they take 48 octets each, and for P-521 they take 66 octets each.

For the curves secp256r1, secp384r1 and secp521r1, peers MUST validate each other's public value Q by ensuring that the point is a valid point on the elliptic curve. The appropriate validation procedures are defined in Section 4.3.7 of [X962] and alternatively in Section 5.6.2.3 of [KEYAGREEMENT]. This process consists of three steps: (1) verify that Q is not the point at infinity (O), (2) verify that for Q = (x, y) both integers x and y are in the correct interval, (3) ensure that (x, y) is a correct solution to the elliptic curve equation. For these curves, implementers do not need to verify membership in the correct subgroup.

For X25519 and X448, the contents of the public value are the byte string inputs and outputs of the corresponding functions defined in [RFC7748], 32 bytes for X25519 and 56 bytes for X448.

Note: Versions of TLS prior to 1.3 permitted point format negotiation; TLS 1.3 removes this feature in favor of a single point format for each curve.

4.2.9. Pre-Shared Key Exchange Modes

In order to use PSKs, clients MUST also send a "psk_key_exchange_modes" extension. The semantics of this extension are that the client only supports the use of PSKs with these modes, which restricts both the use of PSKs offered in this ClientHello and those which the server might supply via NewSessionTicket.

A client MUST provide a "psk_key_exchange_modes" extension if it offers a "pre_shared_key" extension. If clients offer "pre_shared_key" without a "psk_key_exchange_modes" extension, servers MUST abort the handshake. Servers MUST NOT select a key exchange mode that is not listed by the client. This extension also restricts the modes for use with PSK resumption; servers SHOULD NOT send NewSessionTicket with tickets that are not compatible with the advertised modes; however, if a server does so, the impact will just be that the client's attempts at resumption fail.

The server MUST NOT send a "psk_key_exchange_modes" extension.

```
   enum { psk_ke(0), psk_dhe_ke(1), (255) } PskKeyExchangeMode;

   struct {
       PskKeyExchangeMode ke_modes<1..255>;
   } PskKeyExchangeModes;
```

psk_ke PSK-only key establishment. In this mode, the server MUST NOT supply a "key_share" value.

psk_dhe_ke PSK with (EC)DHE key establishment. In this mode, the client and server MUST supply "key_share" values as described in Section 4.2.8.

Any future values that are allocated must ensure that the transmitted protocol messages unambiguously identify which mode was selected by the server; at present, this is indicated by the presence of the "key_share" in the ServerHello.

4.2.10. Early Data Indication

When a PSK is used and early data is allowed for that PSK, the client can send application data in its first flight of messages. If the client opts to do so, it MUST supply both the "early_data" extension as well as the "pre_shared_key" extension.

The "extension_data" field of this extension contains an "EarlyDataIndication" value.

```
   struct {} Empty;

   struct {
       select (Handshake.msg_type) {
           case new_session_ticket:   uint32 max_early_data_size;
           case client_hello:         Empty;
           case encrypted_extensions: Empty;
       };
   } EarlyDataIndication;
```

See Section 4.6.1 for the use of the max_early_data_size field.

The parameters for the 0-RTT data (version, symmetric cipher suite, ALPN protocol, etc.) are those associated with the PSK in use. For externally provisioned PSKs, the associated values are those provisioned along with the key. For PSKs established via a NewSessionTicket message, the associated values are those which were negotiated in the connection which established the PSK. The PSK used to encrypt the early data MUST be the first PSK listed in the client’s "pre_shared_key" extension.

For PSKs provisioned via NewSessionTicket, a server MUST validate that the ticket age for the selected PSK identity (computed by subtracting ticket_age_add from PskIdentity.obfuscated_ticket_age modulo 2^32) is within a small tolerance of the time since the ticket was issued (see Section 8). If it is not, the server SHOULD proceed with the handshake but reject 0-RTT, and SHOULD NOT take any other action that assumes that this ClientHello is fresh.

0-RTT messages sent in the first flight have the same (encrypted) content types as messages of the same type sent in other flights (handshake and application_data) but are protected under different keys. After receiving the server's Finished message, if the server has accepted early data, an EndOfEarlyData message will be sent to indicate the key change. This message will be encrypted with the 0-RTT traffic keys.

A server which receives an "early_data" extension MUST behave in one of three ways:

- Ignore the extension and return a regular 1-RTT response. The server then skips past early data by attempting to deprotect received records using the handshake traffic key, discarding records which fail deprotection (up to the configured max_early_data_size). Once a record is deprotected successfully, it is treated as the start of the client's second flight and the the server proceeds as with an ordinary 1-RTT handshake.

- Request that the client send another ClientHello by responding with a HelloRetryRequest. A client MUST NOT include the "early_data" extension in its followup ClientHello. The server then ignores early data by skipping all records with external content type of "application_data" (indicating that they are encrypted), up to the configured max_early_data_size.

- Return its own "early_data" extension in EncryptedExtensions, indicating that it intends to process the early data. It is not possible for the server to accept only a subset of the early data messages. Even though the server sends a message accepting early data, the actual early data itself may already be in flight by the time the server generates this message.

In order to accept early data, the server MUST have accepted a PSK cipher suite and selected the first key offered in the client's "pre_shared_key" extension. In addition, it MUST verify that the following values are the same as those associated with the selected PSK:

- The TLS version number

- The selected cipher suite

- The selected ALPN [RFC7301] protocol, if any

These requirements are a superset of those needed to perform a 1-RTT handshake using the PSK in question. For externally established PSKs, the associated values are those provisioned along with the key. For PSKs established via a NewSessionTicket message, the associated values are those negotiated in the connection during which the ticket was established.

Future extensions MUST define their interaction with 0-RTT.

If any of these checks fail, the server MUST NOT respond with the extension and must discard all the first flight data using one of the first two mechanisms listed above (thus falling back to 1-RTT or 2-RTT). If the client attempts a 0-RTT handshake but the server rejects it, the server will generally not have the 0-RTT record protection keys and must instead use trial decryption (either with the 1-RTT handshake keys or by looking for a cleartext ClientHello in the case of HelloRetryRequest) to find the first non-0-RTT message.

If the server chooses to accept the "early_data" extension, then it MUST comply with the same error handling requirements specified for all records when processing early data records. Specifically, if the server fails to decrypt a 0-RTT record following an accepted "early_data" extension it MUST terminate the connection with a "bad_record_mac" alert as per Section 5.2.

If the server rejects the "early_data" extension, the client application MAY opt to retransmit the application data previously sent in early data once the handshake has been completed. Note that automatic re-transmission of early data could result in assumptions about the status of the connection being incorrect. For instance, when the negotiated connection selects a different ALPN protocol from what was used for the early data, an application might need to construct different messages. Similarly, if early data assumes anything about the connection state, it might be sent in error after the handshake completes.

A TLS implementation SHOULD NOT automatically re-send early data; applications are in a better position to decide when re-transmission is appropriate. A TLS implementation MUST NOT automatically re-send early data unless the negotiated connection selects the same ALPN protocol.

4.2.11. Pre-Shared Key Extension

The "pre_shared_key" extension is used to negotiate the identity of the pre-shared key to be used with a given handshake in association with PSK key establishment.

The "extension_data" field of this extension contains a "PreSharedKeyExtension" value:

```
struct {
    opaque identity<1..2^16-1>;
    uint32 obfuscated_ticket_age;
} PskIdentity;

opaque PskBinderEntry<32..255>;

struct {
    PskIdentity identities<7..2^16-1>;
    PskBinderEntry binders<33..2^16-1>;
} OfferedPsks;

struct {
    select (Handshake.msg_type) {
        case client_hello: OfferedPsks;
        case server_hello: uint16 selected_identity;
    };
} PreSharedKeyExtension;
```

identity A label for a key. For instance, a ticket defined in Appendix B.3.4 or a label for a pre-shared key established externally.

obfuscated_ticket_age An obfuscated version of the age of the key. Section 4.2.11.1 describes how to form this value for identities established via the NewSessionTicket message. For identities established externally an obfuscated_ticket_age of 0 SHOULD be used, and servers MUST ignore the value.

identities A list of the identities that the client is willing to negotiate with the server. If sent alongside the "early_data" extension (see Section 4.2.10), the first identity is the one used for 0-RTT data.

binders A series of HMAC values, one for each PSK offered in the "pre_shared_keys" extension and in the same order, computed as described below.

selected_identity The server's chosen identity expressed as a (0-based) index into the identities in the client's list.

Each PSK is associated with a single Hash algorithm. For PSKs established via the ticket mechanism (Section 4.6.1), this is the KDF Hash algorithm on the connection where the ticket was established. For externally established PSKs, the Hash algorithm MUST be set when the PSK is established, or default to SHA-256 if no such algorithm is defined. The server MUST ensure that it selects a compatible PSK (if any) and cipher suite.

In TLS versions prior to TLS 1.3, the Server Name Identification (SNI) value was intended to be associated with the session (Section 3 of [RFC6066]), with the server being required to enforce that the SNI value associated with the session matches the one specified in the resumption handshake. However, in reality the implementations were not consistent on which of two supplied SNI values they would use, leading to the consistency requirement being de-facto enforced by the clients. In TLS 1.3, the SNI value is always explicitly specified in the resumption handshake, and there is no need for the server to associate an SNI value with the ticket. Clients, however, SHOULD store the SNI with the PSK to fulfill the requirements of Section 4.6.1.

Implementor's note: when session resumption is the primary use case of PSKs the most straightforward way to implement the PSK/cipher suite matching requirements is to negotiate the cipher suite first and then exclude any incompatible PSKs. Any unknown PSKs (e.g., they are not in the PSK database or are encrypted with an unknown key) SHOULD simply be ignored. If no acceptable PSKs are found, the server SHOULD perform a non-PSK handshake if possible. If backwards compatibility is important, client provided, externally established PSKs SHOULD influence cipher suite selection.

Prior to accepting PSK key establishment, the server MUST validate the corresponding binder value (see Section 4.2.11.2 below). If this value is not present or does not validate, the server MUST abort the handshake. Servers SHOULD NOT attempt to validate multiple binders; rather they SHOULD select a single PSK and validate solely the binder that corresponds to that PSK. See [Section 8.2] and [Appendix E.6] for the security rationale for this requirement. In order to accept PSK key establishment, the server sends a "pre_shared_key" extension indicating the selected identity.

Clients MUST verify that the server's selected_identity is within the range supplied by the client, that the server selected a cipher suite indicating a Hash associated with the PSK and that a server "key_share" extension is present if required by the ClientHello "psk_key_exchange_modes". If these values are not consistent the client MUST abort the handshake with an "illegal_parameter" alert.

If the server supplies an "early_data" extension, the client MUST verify that the server's selected_identity is 0. If any other value is returned, the client MUST abort the handshake with an "illegal_parameter" alert.

The "pre_shared_key" extension MUST be the last extension in the ClientHello (this facilitates implementation as described below). Servers MUST check that it is the last extension and otherwise fail the handshake with an "illegal_parameter" alert.

4.2.11.1. Ticket Age

The client's view of the age of a ticket is the time since the receipt of the NewSessionTicket message. Clients MUST NOT attempt to use tickets which have ages greater than the "ticket_lifetime" value which was provided with the ticket. The "obfuscated_ticket_age" field of each PskIdentity contains an obfuscated version of the ticket age formed by taking the age in milliseconds and adding the "ticket_age_add" value that was included with the ticket (see Section 4.6.1), modulo 2^32. This addition prevents passive observers from correlating connections unless tickets are reused. Note that the "ticket_lifetime" field in the NewSessionTicket message is in seconds but the "obfuscated_ticket_age" is in milliseconds. Because ticket lifetimes are restricted to a week, 32 bits is enough to represent any plausible age, even in milliseconds.

4.2.11.2. PSK Binder

The PSK binder value forms a binding between a PSK and the current handshake, as well as a binding between the handshake in which the PSK was generated (if via a NewSessionTicket message) and the current handshake. Each entry in the binders list is computed as an HMAC over a transcript hash (see Section 4.4.1) containing a partial ClientHello up to and including the PreSharedKeyExtension.identities field. That is, it includes all of the ClientHello but not the binders list itself. The length fields for the message (including the overall length, the length of the extensions block, and the length of the "pre_shared_key" extension) are all set as if binders of the correct lengths were present.

The PskBinderEntry is computed in the same way as the Finished message (Section 4.4.4) but with the BaseKey being the binder_key derived via the key schedule from the corresponding PSK which is being offered (see Section 7.1).

If the handshake includes a HelloRetryRequest, the initial ClientHello and HelloRetryRequest are included in the transcript along with the new ClientHello. For instance, if the client sends ClientHello1, its binder will be computed over:

```
Transcript-Hash(Truncate(ClientHello1))
```

Where Truncate() removes the binders list from the ClientHello.

If the server responds with HelloRetryRequest, and the client then sends ClientHello2, its binder will be computed over:

```
Transcript-Hash(ClientHello1,
                HelloRetryRequest,
                Truncate(ClientHello2))
```

The full ClientHello1/ClientHello2 is included in all other handshake hash computations. Note that in the first flight, Truncate(ClientHello1) is hashed directly, but in the second flight, ClientHello1 is hashed and then reinjected as a "message_hash" message, as described in Section 4.4.1.

4.2.11.3. Processing Order

Clients are permitted to "stream" 0-RTT data until they receive the server's Finished, only then sending the EndOfEarlyData message, followed by the rest of the handshake. In order to avoid deadlocks, when accepting "early_data", servers MUST process the client's ClientHello and then immediately send their flight of messages, rather than waiting for the client's EndOfEarlyData message before sending its ServerHello.

4.3. Server Parameters

The next two messages from the server, EncryptedExtensions and CertificateRequest, contain information from the server that determines the rest of the handshake. These messages are encrypted with keys derived from the server_handshake_traffic_secret.

4.3.1. Encrypted Extensions

In all handshakes, the server MUST send the EncryptedExtensions message immediately after the ServerHello message. This is the first message that is encrypted under keys derived from the server_handshake_traffic_secret.

The EncryptedExtensions message contains extensions that can be protected, i.e., any which are not needed to establish the cryptographic context, but which are not associated with individual certificates. The client MUST check EncryptedExtensions for the presence of any forbidden extensions and if any are found MUST abort the handshake with an "illegal_parameter" alert.

Structure of this message:

```
   struct {
       Extension extensions<0..2^16-1>;
   } EncryptedExtensions;
```

extensions A list of extensions. For more information, see the table in Section 4.2.

4.3.2. Certificate Request

A server which is authenticating with a certificate MAY optionally request a certificate from the client. This message, if sent, MUST follow EncryptedExtensions.

Structure of this message:

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

certificate_request_context An opaque string which identifies the certificate request and which will be echoed in the client's Certificate message. The certificate_request_context MUST be unique within the scope of this connection (thus preventing replay of client CertificateVerify messages). This field SHALL be zero length unless used for the post-handshake authentication exchanges described in Section 4.6.2. When requesting post-handshake authentication, the server SHOULD make the context unpredictable to the client (e.g., by randomly generating it) in order to prevent an attacker who has temporary access to the client's private key from pre-computing valid CertificateVerify messages.

extensions A set of extensions describing the parameters of the certificate being requested. The "signature_algorithms" extension MUST be specified, and other extensions may optionally be included if defined for this message. Clients MUST ignore unrecognized extensions.

In prior versions of TLS, the CertificateRequest message carried a list of signature algorithms and certificate authorities which the server would accept. In TLS 1.3 the former is expressed by sending the "signature_algorithms" and optionally "signature_algorithms_cert" extensions. The latter is expressed by sending the "certificate_authorities" extension (see Section 4.2.4).

Servers which are authenticating with a PSK MUST NOT send the CertificateRequest message in the main handshake, though they MAY send it in post-handshake authentication (see Section 4.6.2) provided that the client has sent the "post_handshake_auth" extension (see Section 4.2.6).

4.4. Authentication Messages

As discussed in Section 2, TLS generally uses a common set of messages for authentication, key confirmation, and handshake integrity: Certificate, CertificateVerify, and Finished. (The PreSharedKey binders also perform key confirmation, in a similar fashion.) These three messages are always sent as the last messages in their handshake flight. The Certificate and CertificateVerify messages are only sent under certain circumstances, as defined below. The Finished message is always sent as part of the Authentication block. These messages are encrypted under keys derived from [sender]_handshake_traffic_secret.

The computations for the Authentication messages all uniformly take the following inputs:

- The certificate and signing key to be used.

- A Handshake Context consisting of the set of messages to be included in the transcript hash.

- A base key to be used to compute a MAC key.

Based on these inputs, the messages then contain:

Certificate The certificate to be used for authentication, and any supporting certificates in the chain. Note that certificate-based client authentication is not available in PSK (including 0-RTT) flows.

CertificateVerify A signature over the value Transcript-Hash(Handshake Context, Certificate)

Finished A MAC over the value Transcript-Hash(Handshake Context, Certificate, CertificateVerify) using a MAC key derived from the base key.

The following table defines the Handshake Context and MAC Base Key for each scenario:

Mode	Handshake Context	Base Key
Server	ClientHello ... later of EncryptedExtensions/CertificateRequest	server_handshake_traffic_secret
Client	ClientHello ... later of server Finished/EndOfEarlyData	client_handshake_traffic_secret
Post-Handshake	ClientHello ... client Finished + CertificateRequest	client_application_traffic_secret_N

4.4.1. The Transcript Hash

Many of the cryptographic computations in TLS make use of a transcript hash. This value is computed by hashing the concatenation of each included handshake message, including the handshake message header carrying the handshake message type and length fields, but not including record layer headers. I.e.,

```
 Transcript-Hash(M1, M2, ... Mn) = Hash(M1 || M2 || ... || Mn)
```

As an exception to this general rule, when the server responds to a ClientHello with a HelloRetryRequest, the value of ClientHello1 is replaced with a special synthetic handshake message of handshake type "message_hash" containing Hash(ClientHello1). I.e.,

```
Transcript-Hash(ClientHello1, HelloRetryRequest, ... Mn) =
    Hash(message_hash ||        /* Handshake type */
         00 00 Hash.length ||   /* Handshake message length (bytes) */
         Hash(ClientHello1) ||  /* Hash of ClientHello1 */
         HelloRetryRequest || ... || Mn)
```

The reason for this construction is to allow the server to do a stateless HelloRetryRequest by storing just the hash of ClientHello1 in the cookie, rather than requiring it to export the entire intermediate hash state (see Section 4.2.2).

For concreteness, the transcript hash is always taken from the following sequence of handshake messages, starting at the first ClientHello and including only those messages that were sent:

ClientHello, HelloRetryRequest, ClientHello, ServerHello, EncryptedExtensions, server CertificateRequest, server Certificate, server CertificateVerify, server Finished, EndOfEarlyData, client Certificate, client CertificateVerify, client Finished.

In general, implementations can implement the transcript by keeping a running transcript hash value based on the negotiated hash. Note, however, that subsequent post-handshake authentications do not include each other, just the messages through the end of the main handshake.

4.4.2. Certificate

This message conveys the endpoint’s certificate chain to the peer.

The server MUST send a Certificate message whenever the agreed-upon key exchange method uses certificates for authentication (this includes all key exchange methods defined in this document except PSK).

The client MUST send a Certificate message if and only if the server has requested client authentication via a CertificateRequest message (Section 4.3.2). If the server requests client authentication but no suitable certificate is available, the client MUST send a Certificate message containing no certificates (i.e., with the "certificate_list" field having length 0). A Finished message MUST be sent regardless of whether the Certificate message is empty.

Structure of this message:

```
      /* Managed by IANA */
      enum {
          X509(0),
          RawPublicKey(2),
          (255)
      } CertificateType;

      struct {
          select (certificate_type) {
              case RawPublicKey:
                /* From RFC 7250 ASN.1_subjectPublicKeyInfo */
                opaque ASN1_subjectPublicKeyInfo<1..2^24-1>;

              case X509:
                opaque cert_data<1..2^24-1>;
          };
          Extension extensions<0..2^16-1>;
      } CertificateEntry;

      struct {
          opaque certificate_request_context<0..2^8-1>;
          CertificateEntry certificate_list<0..2^24-1>;
      } Certificate;
```

certificate_request_context If this message is in response to a CertificateRequest, the value of certificate_request_context in that message. Otherwise (in the case of server authentication), this field SHALL be zero length.

certificate_list This is a sequence (chain) of CertificateEntry structures, each containing a single certificate and set of extensions.

extensions: A set of extension values for the CertificateEntry. The "Extension" format is defined in Section 4.2. Valid extensions for server certificates at present include OCSP Status extension ([RFC6066]) and SignedCertificateTimestamps ([RFC6962]); future extensions may be defined for this message as well. Extensions in the Certificate message from the server MUST correspond to ones from the ClientHello message. Extensions in the Certificate from the client MUST correspond with extensions in the CertificateRequest message from the server. If an extension applies to the entire chain, it SHOULD be included in the first CertificateEntry.

If the corresponding certificate type extension ("server_certificate_type" or "client_certificate_type") was not negotiated in Encrypted Extensions, or the X.509 certificate type was negotiated, then each CertificateEntry contains a DER-encoded X.509 certificate. The sender's certificate MUST come in the first CertificateEntry in the list. Each following certificate SHOULD directly certify the one immediately preceding it. Because certificate validation requires that trust anchors be distributed independently, a certificate that specifies a trust anchor MAY be omitted from the chain, provided that supported peers are known to possess any omitted certificates.

Note: Prior to TLS 1.3, "certificate_list" ordering required each certificate to certify the one immediately preceding it; however, some implementations allowed some flexibility. Servers sometimes send both a current and deprecated intermediate for transitional purposes, and others are simply configured incorrectly, but these cases can nonetheless be validated properly. For maximum compatibility, all implementations SHOULD be prepared to handle potentially extraneous certificates and arbitrary orderings from any TLS version, with the exception of the end-entity certificate which MUST be first.

If the RawPublicKey certificate type was negotiated, then the certificate_list MUST contain no more than one CertificateEntry, which contains an ASN1_subjectPublicKeyInfo value as defined in [RFC7250], Section 3.

The OpenPGP certificate type [RFC6091] MUST NOT be used with TLS 1.3.

The server's certificate_list MUST always be non-empty. A client will send an empty certificate_list if it does not have an appropriate certificate to send in response to the server's authentication request.

4.4.2.1. OCSP Status and SCT Extensions

[RFC6066] and [RFC6961] provide extensions to negotiate the server sending OCSP responses to the client. In TLS 1.2 and below, the server replies with an empty extension to indicate negotiation of this extension and the OCSP information is carried in a CertificateStatus message. In TLS 1.3, the server's OCSP information is carried in an extension in the CertificateEntry containing the associated certificate. Specifically: The body of the "status_request" extension from the server MUST be a CertificateStatus structure as defined in [RFC6066], which is interpreted as defined in [RFC6960].

Note: status_request_v2 extension ([RFC6961]) is deprecated. TLS 1.3 servers MUST NOT act upon its presence or information in it when processing Client Hello, in particular they MUST NOT send the status_request_v2 extension in the Encrypted Extensions, Certificate Request or the Certificate messages. TLS 1.3 servers MUST be able to process Client Hello messages that include it, as it MAY be sent by clients that wish to use it in earlier protocol versions.

A server MAY request that a client present an OCSP response with its certificate by sending an empty "status_request" extension in its CertificateRequest message. If the client opts to send an OCSP response, the body of its "status_request" extension MUST be a CertificateStatus structure as defined in [RFC6066].

Similarly, [RFC6962] provides a mechanism for a server to send a Signed Certificate Timestamp (SCT) as an extension in the ServerHello in TLS 1.2 and below. In TLS 1.3, the server's SCT information is carried in an extension in CertificateEntry.

4.4.2.2. Server Certificate Selection

The following rules apply to the certificates sent by the server:

- The certificate type MUST be X.509v3 [RFC5280], unless explicitly negotiated otherwise (e.g., [RFC7250]).

- The server's end-entity certificate's public key (and associated restrictions) MUST be compatible with the selected authentication algorithm from the client's "signature_algorithms" extension (currently RSA, ECDSA, or EdDSA).

- The certificate MUST allow the key to be used for signing (i.e., the digitalSignature bit MUST be set if the Key Usage extension is present) with a signature scheme indicated in the client's "signature_algorithms"/"signature_algorithms_cert" extensions (see Section 4.2.3).

- The "server_name" [RFC6066] and "certificate_authorities" extensions are used to guide certificate selection. As servers MAY require the presence of the "server_name" extension, clients SHOULD send this extension, when applicable.

All certificates provided by the server MUST be signed by a signature algorithm advertised by the client, if it is able to provide such a chain (see Section 4.2.3). Certificates that are self-signed or certificates that are expected to be trust anchors are not validated as part of the chain and therefore MAY be signed with any algorithm.

If the server cannot produce a certificate chain that is signed only via the indicated supported algorithms, then it SHOULD continue the handshake by sending the client a certificate chain of its choice that may include algorithms that are not known to be supported by the client. This fallback chain SHOULD NOT use the deprecated SHA-1 hash algorithm in general, but MAY do so if the client's advertisement permits it, and MUST NOT do so otherwise.

If the client cannot construct an acceptable chain using the provided certificates and decides to abort the handshake, then it MUST abort the handshake with an appropriate certificate-related alert (by default, "unsupported_certificate"; see Section 6.2 for more).

If the server has multiple certificates, it chooses one of them based on the above-mentioned criteria (in addition to other criteria, such as transport layer endpoint, local configuration and preferences).

4.4.2.3. Client Certificate Selection

The following rules apply to certificates sent by the client:

- The certificate type MUST be X.509v3 [RFC5280], unless explicitly negotiated otherwise (e.g., [RFC7250]).

- If the "certificate_authorities" extension in the CertificateRequest message was present, at least one of the certificates in the certificate chain SHOULD be issued by one of the listed CAs.

- The certificates MUST be signed using an acceptable signature algorithm, as described in Section 4.3.2. Note that this relaxes the constraints on certificate-signing algorithms found in prior versions of TLS.

- If the CertificateRequest message contained a non-empty "oid_filters" extension, the end-entity certificate MUST match the extension OIDs that are recognized by the client, as described in Section 4.2.5.

Note that, as with the server certificate, there are certificates that use algorithm combinations that cannot be currently used with TLS.

4.4.2.4. Receiving a Certificate Message

In general, detailed certificate validation procedures are out of scope for TLS (see [RFC5280]). This section provides TLS-specific requirements.

If the server supplies an empty Certificate message, the client MUST abort the handshake with a "decode_error" alert.

If the client does not send any certificates (i.e., it sends an empty Certificate message), the server MAY at its discretion either continue the handshake without client authentication, or abort the handshake with a "certificate_required" alert. Also, if some aspect of the certificate chain was unacceptable (e.g., it was not signed by a known, trusted CA), the server MAY at its discretion either continue the handshake (considering the client unauthenticated) or abort the handshake.

Any endpoint receiving any certificate which it would need to validate using any signature algorithm using an MD5 hash MUST abort the handshake with a "bad_certificate" alert. SHA-1 is deprecated and it is RECOMMENDED that any endpoint receiving any certificate which it would need to validate using any signature algorithm using a SHA-1 hash abort the handshake with a "bad_certificate" alert. For clarity, this means that endpoints MAY accept these algorithms for certificates that are self-signed or are trust anchors.

All endpoints are RECOMMENDED to transition to SHA-256 or better as soon as possible to maintain interoperability with implementations currently in the process of phasing out SHA-1 support.

Note that a certificate containing a key for one signature algorithm MAY be signed using a different signature algorithm (for instance, an RSA key signed with an ECDSA key).

4.4.3. Certificate Verify

This message is used to provide explicit proof that an endpoint possesses the private key corresponding to its certificate. The CertificateVerify message also provides integrity for the handshake up to this point. Servers MUST send this message when authenticating via a certificate. Clients MUST send this message whenever authenticating via a certificate (i.e., when the Certificate message is non-empty). When sent, this message MUST appear immediately after the Certificate message and immediately prior to the Finished message.

Structure of this message:

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

The algorithm field specifies the signature algorithm used (see Section 4.2.3 for the definition of this field). The signature is a digital signature using that algorithm. The content that is covered under the signature is the hash output as described in Section 4.4.1, namely:

```
Transcript-Hash(Handshake Context, Certificate)
```

The digital signature is then computed over the concatenation of:

- A string that consists of octet 32 (0x20) repeated 64 times

- The context string

- A single 0 byte which serves as the separator

- The content to be signed

This structure is intended to prevent an attack on previous versions of TLS in which the ServerKeyExchange format meant that attackers could obtain a signature of a message with a chosen 32-byte prefix (ClientHello.random). The initial 64-byte pad clears that prefix along with the server-controlled ServerHello.random.

The context string for a server signature is: "TLS 1.3, server CertificateVerify" The context string for a client signature is: "TLS 1.3, client CertificateVerify" It is used to provide separation between signatures made in different contexts, helping against potential cross-protocol attacks.

For example, if the transcript hash was 32 bytes of 01 (this length would make sense for SHA-256), the content covered by the digital signature for a server CertificateVerify would be:

```
2020202020202020202020202020202020202020202020202020202020202020
2020202020202020202020202020202020202020202020202020202020202020
544c5320312e332c207365727665722043657274696669636174655665726966
79
00
0101010101010101010101010101010101010101010101010101010101010101
```

On the sender side the process for computing the signature field of the CertificateVerify message takes as input:

- The content covered by the digital signature

- The private signing key corresponding to the certificate sent in the previous message

If the CertificateVerify message is sent by a server, the signature algorithm MUST be one offered in the client's "signature_algorithms" extension unless no valid certificate chain can be produced without unsupported algorithms (see Section 4.2.3).

If sent by a client, the signature algorithm used in the signature MUST be one of those present in the supported_signature_algorithms field of the "signature_algorithms" extension in the CertificateRequest message.

In addition, the signature algorithm MUST be compatible with the key in the sender's end-entity certificate. RSA signatures MUST use an RSASSA-PSS algorithm, regardless of whether RSASSA-PKCS1-v1_5 algorithms appear in "signature_algorithms". The SHA-1 algorithm MUST NOT be used in any signatures of CertificateVerify messages. All SHA-1 signature algorithms in this specification are defined solely for use in legacy certificates and are not valid for CertificateVerify signatures.

The receiver of a CertificateVerify message MUST verify the signature field. The verification process takes as input:

- The content covered by the digital signature

- The public key contained in the end-entity certificate found in the associated Certificate message.

- The digital signature received in the signature field of the CertificateVerify message

If the verification fails, the receiver MUST terminate the handshake with a "decrypt_error" alert.

4.4.4. Finished

The Finished message is the final message in the authentication block. It is essential for providing authentication of the handshake and of the computed keys.

Recipients of Finished messages MUST verify that the contents are correct and if incorrect MUST terminate the connection with a "decrypt_error" alert.

Once a side has sent its Finished message and received and validated the Finished message from its peer, it may begin to send and receive application data over the connection. There are two settings in which it is permitted to send data prior to receiving the peer's Finished:

1. Clients sending 0-RTT data as described in Section 4.2.10.

2. Servers MAY send data after sending their first flight, but because the handshake is not yet complete, they have no assurance of either the peer's identity or of its liveness (i.e., the ClientHello might have been replayed).

The key used to compute the Finished message is computed from the Base key defined in Section 4.4 using HKDF (see Section 7.1). Specifically:

```
finished_key =
    HKDF-Expand-Label(BaseKey, "finished", "", Hash.length)
```

Structure of this message:

```
struct {
    opaque verify_data[Hash.length];
} Finished;
```

The verify_data value is computed as follows:

```
verify_data =
    HMAC(finished_key,
         Transcript-Hash(Handshake Context,
                         Certificate*, CertificateVerify*))

* Only included if present.
```

HMAC [RFC2104] uses the Hash algorithm for the handshake. As noted above, the HMAC input can generally be implemented by a running hash, i.e., just the handshake hash at this point.

In previous versions of TLS, the verify_data was always 12 octets long. In TLS 1.3, it is the size of the HMAC output for the Hash used for the handshake.

Note: Alerts and any other record types are not handshake messages and are not included in the hash computations.

Any records following a Finished message MUST be encrypted under the appropriate application traffic key as described in Section 7.2. In particular, this includes any alerts sent by the server in response to client Certificate and CertificateVerify messages.

4.5. End of Early Data

```
struct {} EndOfEarlyData;
```

If the server sent an "early_data" extension, the client MUST send an EndOfEarlyData message after receiving the server Finished. If the server does not send an "early_data" extension, then the client MUST NOT send an EndOfEarlyData message. This message indicates that all 0-RTT application_data messages, if any, have been transmitted and that the following records are protected under handshake traffic keys. Servers MUST NOT send this message and clients receiving it MUST terminate the connection with an "unexpected_message" alert. This message is encrypted under keys derived from the client_early_traffic_secret.

4.6. Post-Handshake Messages

TLS also allows other messages to be sent after the main handshake. These messages use a handshake content type and are encrypted under the appropriate application traffic key.

4.6.1. New Session Ticket Message

At any time after the server has received the client Finished message, it MAY send a NewSessionTicket message. This message creates a unique association between the ticket value and a secret PSK derived from the resumption master secret (see Section 7.

The client MAY use this PSK for future handshakes by including the ticket value in the "pre_shared_key" extension in its ClientHello (Section 4.2.11). Servers MAY send multiple tickets on a single connection, either immediately after each other or after specific events (see Appendix C.4). For instance, the server might send a new ticket after post-handshake authentication in order to encapsulate the additional client authentication state. Multiple tickets are useful for clients for a variety of purposes, including:

- Opening multiple parallel HTTP connections.

- Performing connection racing across interfaces and address families via, e.g., Happy Eyeballs [RFC8305] or related techniques.

Any ticket MUST only be resumed with a cipher suite that has the same KDF hash algorithm as that used to establish the original connection.

Clients MUST only resume if the new SNI value is valid for the server certificate presented in the original session, and SHOULD only resume if the SNI value matches the one used in the original session. The latter is a performance optimization: normally, there is no reason to expect that different servers covered by a single certificate would be able to accept each other’s tickets, hence attempting resumption in that case would waste a single-use ticket. If such an indication is provided (externally or by any other means), clients MAY resume with a different SNI value.

On resumption, if reporting an SNI value to the calling application, implementations MUST use the value sent in the resumption ClientHello rather than the value sent in the previous session. Note that if a server implementation declines all PSK identities with different SNI values, these two values are always the same.

Note: Although the resumption master secret depends on the client's second flight, servers which do not request client authentication MAY compute the remainder of the transcript independently and then send a NewSessionTicket immediately upon sending its Finished rather than waiting for the client Finished. This might be appropriate in cases where the client is expected to open multiple TLS connections in parallel and would benefit from the reduced overhead of a resumption handshake, for example.

```
   struct {
       uint32 ticket_lifetime;
       uint32 ticket_age_add;
       opaque ticket_nonce<0..255>;
       opaque ticket<1..2^16-1>;
       Extension extensions<0..2^16-2>;
   } NewSessionTicket;
```

ticket_lifetime Indicates the lifetime in seconds as a 32-bit unsigned integer in network byte order from the time of ticket issuance. Servers MUST NOT use any value greater than 604800 seconds (7 days). The value of zero indicates that the ticket should be discarded immediately. Clients MUST NOT cache tickets for longer than 7 days, regardless of the ticket_lifetime, and MAY delete tickets earlier based on local policy. A server MAY treat a ticket as valid for a shorter period of time than what is stated in the ticket_lifetime.

ticket_age_add A securely generated, random 32-bit value that is used to obscure the age of the ticket that the client includes in the "pre_shared_key" extension. The client-side ticket age is added to this value modulo 2^32 to obtain the value that is transmitted by the client. The server MUST generate a fresh value for each ticket it sends.

ticket_nonce A per-ticket value that is unique across all tickets issued on this connection.

ticket The value of the ticket to be used as the PSK identity. The ticket itself is an opaque label. It MAY either be a database lookup key or a self-encrypted and self-authenticated value. Section 4 of [RFC5077] describes a recommended ticket construction mechanism.

extensions A set of extension values for the ticket. The "Extension" format is defined in Section 4.2. Clients MUST ignore unrecognized extensions.

The sole extension currently defined for NewSessionTicket is "early_data", indicating that the ticket may be used to send 0-RTT data (Section 4.2.10)). It contains the following value:

max_early_data_size The maximum amount of 0-RTT data that the client is allowed to send when using this ticket, in bytes. Only Application Data payload (i.e., plaintext but not padding or the inner content type byte) is counted. A server receiving more than max_early_data_size bytes of 0-RTT data SHOULD terminate the connection with an "unexpected_message" alert. Note that servers that reject early data due to lack of cryptographic material will be unable to differentiate padding from content, so clients SHOULD NOT depend on being able to send large quantities of padding in early data records.

The PSK associated with the ticket is computed as:

```
    HKDF-Expand-Label(resumption_master_secret,
                     "resumption", ticket_nonce, Hash.length)
```

Because the ticket_nonce value is distinct for each NewSessionTicket message, a different PSK will be derived for each ticket.

Note that in principle it is possible to continue issuing new tickets which indefinitely extend the lifetime of the keying material originally derived from an initial non-PSK handshake (which was most likely tied to the peer’s certificate). It is RECOMMENDED that implementations place limits on the total lifetime of such keying material; these limits should take into account the lifetime of the peer’s certificate, the likelihood of intervening revocation, and the time since the peer’s online CertificateVerify signature.

4.6.2. Post-Handshake Authentication

When the client has sent the "post_handshake_auth" extension (see Section 4.2.6), a server MAY request client authentication at any time after the handshake has completed by sending a CertificateRequest message. The client MUST respond with the appropriate Authentication messages (see Section 4.4). If the client chooses to authenticate, it MUST send Certificate, CertificateVerify, and Finished. If it declines, it MUST send a Certificate message containing no certificates followed by Finished. All of the client's messages for a given response MUST appear consecutively on the wire with no intervening messages of other types.

A client that receives a CertificateRequest message without having sent the "post_handshake_auth" extension MUST send an "unexpected_message" fatal alert.

Note: Because client authentication could involve prompting the user, servers MUST be prepared for some delay, including receiving an arbitrary number of other messages between sending the CertificateRequest and receiving a response. In addition, clients which receive multiple CertificateRequests in close succession MAY respond to them in a different order than they were received (the certificate_request_context value allows the server to disambiguate the responses).

4.6.3. Key and IV Update

```
enum {
    update_not_requested(0), update_requested(1), (255)
} KeyUpdateRequest;

struct {
    KeyUpdateRequest request_update;
} KeyUpdate;
```

request_update Indicates whether the recipient of the KeyUpdate should respond with its own KeyUpdate. If an implementation receives any other value, it MUST terminate the connection with an "illegal_parameter" alert.

The KeyUpdate handshake message is used to indicate that the sender is updating its sending cryptographic keys. This message can be sent by either peer after it has sent a Finished message. Implementations that receive a KeyUpdate message prior to receiving a Finished message MUST terminate the connection with an "unexpected_message" alert. After sending a KeyUpdate message, the sender SHALL send all its traffic using the next generation of keys, computed as described in Section 7.2. Upon receiving a KeyUpdate, the receiver MUST update its receiving keys.

If the request_update field is set to "update_requested" then the receiver MUST send a KeyUpdate of its own with request_update set to "update_not_requested" prior to sending its next application data record. This mechanism allows either side to force an update to the entire connection, but causes an implementation which receives multiple KeyUpdates while it is silent to respond with a single update. Note that implementations may receive an arbitrary number of messages between sending a KeyUpdate with request_update set to update_requested and receiving the peer's KeyUpdate, because those messages may already be in flight. However, because send and receive keys are derived from independent traffic secrets, retaining the receive traffic secret does not threaten the forward secrecy of data sent before the sender changed keys.

If implementations independently send their own KeyUpdates with request_update set to "update_requested", and they cross in flight, then each side will also send a response, with the result that each side increments by two generations.

Both sender and receiver MUST encrypt their KeyUpdate messages with the old keys. Additionally, both sides MUST enforce that a KeyUpdate with the old key is received before accepting any messages encrypted with the new key. Failure to do so may allow message truncation attacks.

5. Record Protocol

The TLS record protocol takes messages to be transmitted, fragments the data into manageable blocks, protects the records, and transmits the result. Received data is verified, decrypted, reassembled, and then delivered to higher-level clients.

TLS records are typed, which allows multiple higher-level protocols to be multiplexed over the same record layer. This document specifies four content types: handshake, application data, alert, and change_cipher_spec. The change_cipher_spec record is used only for compatibility purposes (see Appendix D.4).

An implementation may receive an unencrypted record of type change_cipher_spec consisting of the single byte value 0x01 at any time after the first ClientHello message has been sent or received and before the peer's Finished message has been received and MUST simply drop it without further processing. Note that this record may appear at a point at the handshake where the implementation is expecting protected records and so it is necessary to detect this condition prior to attempting to deprotect the record. An implementation which receives any other change_cipher_spec value or which receives a protected change_cipher_spec record MUST abort the handshake with an "unexpected_message" alert. A change_cipher_spec record received before the first ClientHello message or after the peer's Finished message MUST be treated as an unexpected record type (though stateless servers may not be able to distinguish these cases from allowed cases).

Implementations MUST NOT send record types not defined in this document unless negotiated by some extension. If a TLS implementation receives an unexpected record type, it MUST terminate the connection with an "unexpected_message" alert. New record content type values are assigned by IANA in the TLS Content Type Registry as described in Section 11.

5.1. Record Layer

The record layer fragments information blocks into TLSPlaintext records carrying data in chunks of 2^14 bytes or less. Message boundaries are handled differently depending on the underlying ContentType. Any future content types MUST specify appropriate rules. Note that these rules are stricter than what was enforced in TLS 1.2.

Handshake messages MAY be coalesced into a single TLSPlaintext record or fragmented across several records, provided that:

- Handshake messages MUST NOT be interleaved with other record types. That is, if a handshake message is split over two or more records, there MUST NOT be any other records between them.

- Handshake messages MUST NOT span key changes. Implementations MUST verify that all messages immediately preceding a key change align with a record boundary; if not, then they MUST terminate the connection with an "unexpected_message" alert. Because the ClientHello, EndOfEarlyData, ServerHello, Finished, and KeyUpdate messages can immediately precede a key change, implementations MUST send these messages in alignment with a record boundary.

Implementations MUST NOT send zero-length fragments of Handshake types, even if those fragments contain padding.

Alert messages (Section 6) MUST NOT be fragmented across records and multiple Alert messages MUST NOT be coalesced into a single TLSPlaintext record. In other words, a record with an Alert type MUST contain exactly one message.

Application Data messages contain data that is opaque to TLS. Application Data messages are always protected. Zero-length fragments of Application Data MAY be sent as they are potentially useful as a traffic analysis countermeasure. Application Data fragments MAY be split across multiple records or coalesced into a single record.

```
      enum {
          invalid(0),
          change_cipher_spec(20),
          alert(21),
          handshake(22),
          application_data(23),
          (255)
      } ContentType;

      struct {
          ContentType type;
          ProtocolVersion legacy_record_version;
          uint16 length;
          opaque fragment[TLSPlaintext.length];
      } TLSPlaintext;
```

type The higher-level protocol used to process the enclosed fragment.

legacy_record_version This value MUST be set to 0x0303 for all records generated by a TLS 1.3 implementation other than an initial ClientHello (i.e., one not generated after a HelloRetryRequest), where it MAY also be 0x0301 for compatibility purposes. This field is deprecated and MUST be ignored for all purposes. Previous versions of TLS would use other values in this field under some circumstances.

length The length (in bytes) of the following TLSPlaintext.fragment. The length MUST NOT exceed 2^14 bytes. An endpoint that receives a record that exceeds this length MUST terminate the connection with a "record_overflow" alert.

fragment The data being transmitted. This value is transparent and is treated as an independent block to be dealt with by the higher-level protocol specified by the type field.

This document describes TLS 1.3, which uses the version 0x0304. This version value is historical, deriving from the use of 0x0301 for TLS 1.0 and 0x0300 for SSL 3.0. In order to maximize backwards compatibility, records containing an initial ClientHello SHOULD have version 0x0301 and a record containing a second ClientHello or a ServerHello MUST have version 0x0303, reflecting TLS 1.0 and TLS 1.2 respectively. When negotiating prior versions of TLS, endpoints follow the procedure and requirements in Appendix D.

When record protection has not yet been engaged, TLSPlaintext structures are written directly onto the wire. Once record protection has started, TLSPlaintext records are protected and sent as described in the following section. Note that application data records MUST NOT be written to the wire unprotected (see Section 2 for details).

5.2. Record Payload Protection

The record protection functions translate a TLSPlaintext structure into a TLSCiphertext. The deprotection functions reverse the process. In TLS 1.3, as opposed to previous versions of TLS, all ciphers are modeled as "Authenticated Encryption with Additional Data" (AEAD) [RFC5116]. AEAD functions provide an unified encryption and authentication operation which turns plaintext into authenticated ciphertext and back again. Each encrypted record consists of a plaintext header followed by an encrypted body, which itself contains a type and optional padding.

```
struct {
    opaque content[TLSPlaintext.length];
    ContentType type;
    uint8 zeros[length_of_padding];
} TLSInnerPlaintext;

struct {
    ContentType opaque_type = application_data; /* 23 */
    ProtocolVersion legacy_record_version = 0x0303; /* TLS v1.2 */
    uint16 length;
    opaque encrypted_record[TLSCiphertext.length];
} TLSCiphertext;
```

content The TLSPLaintext.fragment value, containing the byte encoding of a handshake or an alert message, or the raw bytes of the application’s data to send.

type The TLSPlaintext.type value containing the content type of the record.

zeros An arbitrary-length run of zero-valued bytes may appear in the cleartext after the type field. This provides an opportunity for senders to pad any TLS record by a chosen amount as long as the total stays within record size limits. See Section 5.4 for more details.

opaque_type The outer opaque_type field of a TLSCiphertext record is always set to the value 23 (application_data) for outward compatibility with middleboxes accustomed to parsing previous versions of TLS. The actual content type of the record is found in TLSInnerPlaintext.type after decryption.

legacy_record_version The legacy_record_version field is always 0x0303. TLS 1.3 TLSCiphertexts are not generated until after TLS 1.3 has been negotiated, so there are no historical compatibility concerns where other values might be received. Note that the handshake protocol including the ClientHello and ServerHello messages authenticates the protocol version, so this value is redundant.

length The length (in bytes) of the following TLSCiphertext.encrypted_record, which is the sum of the lengths of the content and the padding, plus one for the inner content type, plus any expansion added by the AEAD algorithm. The length MUST NOT exceed 2^14 + 256 bytes. An endpoint that receives a record that exceeds this length MUST terminate the connection with a "record_overflow" alert.

encrypted_record The AEAD-encrypted form of the serialized TLSInnerPlaintext structure.

AEAD algorithms take as input a single key, a nonce, a plaintext, and "additional data" to be included in the authentication check, as described in Section 2.1 of [RFC5116]. The key is either the client_write_key or the server_write_key, the nonce is derived from the sequence number and the client_write_iv or server_write_iv (see Section 5.3), and the additional data input is the record header. I.e.,

```
additional_data = TLSCiphertext.opaque_type ||
                  TLSCiphertext.legacy_record_version ||
                  TLSCiphertext.length
```

The plaintext input to the AEAD algorithm is the encoded TLSInnerPlaintext structure. Derivation of traffic keys is defined in Section 7.3.

The AEAD output consists of the ciphertext output from the AEAD encryption operation. The length of the plaintext is greater than the corresponding TLSPlaintext.length due to the inclusion of TLSInnerPlaintext.type and any padding supplied by the sender. The length of the AEAD output will generally be larger than the plaintext, but by an amount that varies with the AEAD algorithm. Since the ciphers might incorporate padding, the amount of overhead could vary with different lengths of plaintext. Symbolically,

```
AEADEncrypted =
    AEAD-Encrypt(write_key, nonce, additional_data, plaintext)
```

Then the encrypted_record field of TLSCiphertext is set to AEADEncrypted.

In order to decrypt and verify, the cipher takes as input the key, nonce, additional data, and the AEADEncrypted value. The output is either the plaintext or an error indicating that the decryption failed. There is no separate integrity check. That is:

```
plaintext of encrypted_record =
    AEAD-Decrypt(peer_write_key, nonce, additional_data, AEADEncrypted)
```

If the decryption fails, the receiver MUST terminate the connection with a "bad_record_mac" alert.

An AEAD algorithm used in TLS 1.3 MUST NOT produce an expansion greater than 255 octets. An endpoint that receives a record from its peer with TLSCiphertext.length larger than 2^14 + 256 octets MUST terminate the connection with a "record_overflow" alert. This limit is derived from the maximum TLSInnerPlaintext length of 2^14 octets + 1 octet for ContentType + the maximum AEAD expansion of 255 octets.

5.3. Per-Record Nonce

A 64-bit sequence number is maintained separately for reading and writing records. The appropriate sequence number is incremented by one after reading or writing each record. Each sequence number is set to zero at the beginning of a connection and whenever the key is changed; the first record transmitted under a particular traffic key MUST use sequence number 0.

Because the size of sequence numbers is 64-bit, they should not wrap. If a TLS implementation would need to wrap a sequence number, it MUST either re-key (Section 4.6.3) or terminate the connection.

Each AEAD algorithm will specify a range of possible lengths for the per-record nonce, from N_MIN bytes to N_MAX bytes of input ([RFC5116]). The length of the TLS per-record nonce (iv_length) is set to the larger of 8 bytes and N_MIN for the AEAD algorithm (see [RFC5116] Section 4). An AEAD algorithm where N_MAX is less than 8 bytes MUST NOT be used with TLS. The per-record nonce for the AEAD construction is formed as follows:

1. The 64-bit record sequence number is encoded in network byte order and padded to the left with zeros to iv_length.

2. The padded sequence number is XORed with the static client_write_iv or server_write_iv, depending on the role.

The resulting quantity (of length iv_length) is used as the per-record nonce.

Note: This is a different construction from that in TLS 1.2, which specified a partially explicit nonce.

5.4. Record Padding

All encrypted TLS records can be padded to inflate the size of the TLSCiphertext. This allows the sender to hide the size of the traffic from an observer.

When generating a TLSCiphertext record, implementations MAY choose to pad. An unpadded record is just a record with a padding length of zero. Padding is a string of zero-valued bytes appended to the ContentType field before encryption. Implementations MUST set the padding octets to all zeros before encrypting.

Application Data records may contain a zero-length TLSInnerPlaintext.content if the sender desires. This permits generation of plausibly-sized cover traffic in contexts where the presence or absence of activity may be sensitive. Implementations MUST NOT send Handshake or Alert records that have a zero-length TLSInnerPlaintext.content; if such a message is received, the receiving implementation MUST terminate the connection with an "unexpected_message" alert.

The padding sent is automatically verified by the record protection mechanism; upon successful decryption of a TLSCiphertext.encrypted_record, the receiving implementation scans the field from the end toward the beginning until it finds a non-zero octet. This non-zero octet is the content type of the message. This padding scheme was selected because it allows padding of any encrypted TLS record by an arbitrary size (from zero up to TLS record size limits) without introducing new content types. The design also enforces all-zero padding octets, which allows for quick detection of padding errors.

Implementations MUST limit their scanning to the cleartext returned from the AEAD decryption. If a receiving implementation does not find a non-zero octet in the cleartext, it MUST terminate the connection with an "unexpected_message" alert.

The presence of padding does not change the overall record size limitations - the full encoded TLSInnerPlaintext MUST NOT exceed 2^14 + 1 octets. If the maximum fragment length is reduced, as for example by the max_fragment_length extension from [RFC6066], then the reduced limit applies to the full plaintext, including the content type and padding.

Selecting a padding policy that suggests when and how much to pad is a complex topic and is beyond the scope of this specification. If the application layer protocol on top of TLS has its own padding, it may be preferable to pad application_data TLS records within the application layer. Padding for encrypted handshake and alert TLS records must still be handled at the TLS layer, though. Later documents may define padding selection algorithms or define a padding policy request mechanism through TLS extensions or some other means.

5.5. Limits on Key Usage

There are cryptographic limits on the amount of plaintext which can be safely encrypted under a given set of keys. [AEAD-LIMITS] provides an analysis of these limits under the assumption that the underlying primitive (AES or ChaCha20) has no weaknesses. Implementations SHOULD do a key update as described in Section 4.6.3 prior to reaching these limits.

For AES-GCM, up to 2^24.5 full-size records (about 24 million) may be encrypted on a given connection while keeping a safety margin of approximately 2^-57 for Authenticated Encryption (AE) security. For ChaCha20/Poly1305, the record sequence number would wrap before the safety limit is reached.

6. Alert Protocol

One of the content types supported by the TLS record layer is the alert type. Like other messages, alert messages are encrypted as specified by the current connection state.

Alert messages convey a description of the alert and a legacy field that conveyed the severity of the message in previous versions of TLS. Alerts are divided into two classes: closure alerts and error alerts. In TLS 1.3, the severity is implicit in the type of alert being sent, and the 'level' field can safely be ignored. The "close_notify" alert is used to indicate orderly closure of one direction of the connection. Upon receiving such an alert, the TLS implementation SHOULD indicate end-of-data to the application.

Error alerts indicate abortive closure of the connection (see Section 6.2). Upon receiving an error alert, the TLS implementation SHOULD indicate an error to the application and MUST NOT allow any further data to be sent or received on the connection. Servers and clients MUST forget the secret values and keys established in failed connections, with the exception of the PSKs associated with session tickets, which SHOULD be discarded if possible.

All the alerts listed in Section 6.2 MUST be sent with AlertLevel=fatal and MUST be treated as error alerts regardless of the AlertLevel in the message. Unknown alert types MUST be treated as error alerts.

Note: TLS defines two generic alerts (see Section 6) to use upon failure to parse a message. Peers which receive a message which cannot be parsed according to the syntax (e.g., have a length extending beyond the message boundary or contain an out-of-range length) MUST terminate the connection with a "decode_error" alert. Peers which receive a message which is syntactically correct but semantically invalid (e.g., a DHE share of p - 1, or an invalid enum) MUST terminate the connection with an "illegal_parameter" alert.

```
      enum { warning(1), fatal(2), (255) } AlertLevel;

      enum {
          close_notify(0),
          unexpected_message(10),
          bad_record_mac(20),
          record_overflow(22),
          handshake_failure(40),
          bad_certificate(42),
          unsupported_certificate(43),
          certificate_revoked(44),
          certificate_expired(45),
          certificate_unknown(46),
          illegal_parameter(47),
          unknown_ca(48),
          access_denied(49),
          decode_error(50),
          decrypt_error(51),
          protocol_version(70),
          insufficient_security(71),
          internal_error(80),
          inappropriate_fallback(86),
          user_canceled(90),
          missing_extension(109),
          unsupported_extension(110),
          unrecognized_name(112),
          bad_certificate_status_response(113),
          unknown_psk_identity(115),
          certificate_required(116),
          no_application_protocol(120),
          (255)
      } AlertDescription;

      struct {
          AlertLevel level;
          AlertDescription description;
      } Alert;
```

6.1. Closure Alerts

The client and the server must share knowledge that the connection is ending in order to avoid a truncation attack.

close_notify This alert notifies the recipient that the sender will not send any more messages on this connection. Any data received after a closure alert has been received MUST be ignored.

user_canceled This alert notifies the recipient that the sender is canceling the handshake for some reason unrelated to a protocol failure. If a user cancels an operation after the handshake is complete, just closing the connection by sending a "close_notify" is more appropriate. This alert SHOULD be followed by a "close_notify". This alert generally has AlertLevel=warning.

Either party MAY initiate a close of its write side of the connection by sending a "close_notify" alert. Any data received after a closure alert has been received MUST be ignored. If a transport-level close is received prior to a "close_notify", the receiver cannot know that all the data that was sent has been received.

Each party MUST send a "close_notify" alert before closing its write side of the connection, unless it has already sent some error alert. This does not have any effect on its read side of the connection. Note that this is a change from versions of TLS prior to TLS 1.3 in which implementations were required to react to a "close_notify" by discarding pending writes and sending an immediate "close_notify" alert of their own. That previous requirement could cause truncation in the read side. Both parties need not wait to receive a "close_notify" alert before closing their read side of the connection, though doing so would introduce the possibility of truncation.

If the application protocol using TLS provides that any data may be carried over the underlying transport after the TLS connection is closed, the TLS implementation MUST receive a "close_notify" alert before indicating end-of-data to the application-layer. No part of this standard should be taken to dictate the manner in which a usage profile for TLS manages its data transport, including when connections are opened or closed.

Note: It is assumed that closing the write side of a connection reliably delivers pending data before destroying the transport.

6.2. Error Alerts

Error handling in the TLS Handshake Protocol is very simple. When an error is detected, the detecting party sends a message to its peer. Upon transmission or receipt of a fatal alert message, both parties MUST immediately close the connection.

Whenever an implementation encounters a fatal error condition, it SHOULD send an appropriate fatal alert and MUST close the connection without sending or receiving any additional data. In the rest of this specification, when the phrases "terminate the connection" and "abort the handshake" are used without a specific alert it means that the implementation SHOULD send the alert indicated by the descriptions below. The phrases "terminate the connection with a X alert" and "abort the handshake with a X alert" mean that the implementation MUST send alert X if it sends any alert. All alerts defined in this section below, as well as all unknown alerts, are universally considered fatal as of TLS 1.3 (see Section 6). The implementation SHOULD provide a way to facilitate logging the sending and receiving of alerts.

The following error alerts are defined:

unexpected_message An inappropriate message (e.g., the wrong handshake message, premature application data, etc.) was received. This alert should never be observed in communication between proper implementations.

bad_record_mac This alert is returned if a record is received which cannot be deprotected. Because AEAD algorithms combine decryption and verification, and also to avoid side channel attacks, this alert is used for all deprotection failures. This alert should never be observed in communication between proper implementations, except when messages were corrupted in the network.

record_overflow A TLSCiphertext record was received that had a length more than 2^14 + 256 bytes, or a record decrypted to a TLSPlaintext record with more than 2^14 bytes (or some other negotiated limit). This alert should never be observed in communication between proper implementations, except when messages were corrupted in the network.

handshake_failure Receipt of a "handshake_failure" alert message indicates that the sender was unable to negotiate an acceptable set of security parameters given the options available.

bad_certificate A certificate was corrupt, contained signatures that did not verify correctly, etc.

unsupported_certificate A certificate was of an unsupported type.

certificate_revoked A certificate was revoked by its signer.

certificate_expired A certificate has expired or is not currently valid.

certificate_unknown Some other (unspecified) issue arose in processing the certificate, rendering it unacceptable.

illegal_parameter A field in the handshake was incorrect or inconsistent with other fields. This alert is used for errors which conform to the formal protocol syntax but are otherwise incorrect.

unknown_ca A valid certificate chain or partial chain was received, but the certificate was not accepted because the CA certificate could not be located or could not be matched with a known trust anchor.

access_denied A valid certificate or PSK was received, but when access control was applied, the sender decided not to proceed with negotiation.

decode_error A message could not be decoded because some field was out of the specified range or the length of the message was incorrect. This alert is used for errors where the message does not conform to the formal protocol syntax. This alert should never be observed in communication between proper implementations, except when messages were corrupted in the network.

decrypt_error A handshake (not record-layer) cryptographic operation failed, including being unable to correctly verify a signature or validate a Finished message or a PSK binder.

protocol_version The protocol version the peer has attempted to negotiate is recognized but not supported. (see Appendix D)

insufficient_security Returned instead of "handshake_failure" when a negotiation has failed specifically because the server requires parameters more secure than those supported by the client.

internal_error An internal error unrelated to the peer or the correctness of the protocol (such as a memory allocation failure) makes it impossible to continue.

inappropriate_fallback Sent by a server in response to an invalid connection retry attempt from a client (see [RFC7507]).

missing_extension Sent by endpoints that receive a handshake message not containing an extension that is mandatory to send for the offered TLS version or other negotiated parameters.

unsupported_extension Sent by endpoints receiving any handshake message containing an extension known to be prohibited for inclusion in the given handshake message, or including any extensions in a ServerHello or Certificate not first offered in the corresponding ClientHello.

unrecognized_name Sent by servers when no server exists identified by the name provided by the client via the "server_name" extension (see [RFC6066]).

bad_certificate_status_response Sent by clients when an invalid or unacceptable OCSP response is provided by the server via the "status_request" extension (see [RFC6066]).

unknown_psk_identity Sent by servers when PSK key establishment is desired but no acceptable PSK identity is provided by the client. Sending this alert is OPTIONAL; servers MAY instead choose to send a "decrypt_error" alert to merely indicate an invalid PSK identity.

certificate_required Sent by servers when a client certificate is desired but none was provided by the client.

no_application_protocol Sent by servers when a client "application_layer_protocol_negotiation" extension advertises only protocols that the server does not support (see [RFC7301]).

New Alert values are assigned by IANA as described in Section 11.

7. Cryptographic Computations

The TLS handshake establishes one or more input secrets which are combined to create the actual working keying material, as detailed below. The key derivation process incorporates both the input secrets and the handshake transcript. Note that because the handshake transcript includes the random values from the Hello messages, any given handshake will have different traffic secrets, even if the same input secrets are used, as is the case when the same PSK is used for multiple connections.

7.1. Key Schedule

The key derivation process makes use of the HKDF-Extract and HKDF-Expand functions as defined for HKDF [RFC5869], as well as the functions defined below:

```
    HKDF-Expand-Label(Secret, Label, Context, Length) =
         HKDF-Expand(Secret, HkdfLabel, Length)

    Where HkdfLabel is specified as:

    struct {
        uint16 length = Length;
        opaque label<7..255> = "tls13 " + Label;
        opaque context<0..255> = Context;
    } HkdfLabel;

    Derive-Secret(Secret, Label, Messages) =
         HKDF-Expand-Label(Secret, Label,
                           Transcript-Hash(Messages), Hash.length)
```

The Hash function used by Transcript-Hash and HKDF is the cipher suite hash algorithm. Hash.length is its output length in bytes. Messages is the concatenation of the indicated handshake messages, including the handshake message type and length fields, but not including record layer headers. Note that in some cases a zero-length Context (indicated by "") is passed to HKDF-Expand-Label. The Labels specified in this document are all ASCII strings, and do not include a trailing NUL byte.

Note: with common hash functions, any label longer than 12 characters requires an additional iteration of the hash function to compute. The labels in this specification have all been chosen to fit within this limit.

Keys are derived from two input secrets using the HKDF-Extract and Derive-Secret functions. The general pattern for adding a new secret is to use HKDF-Extract with the salt being the current secret state and the IKM being the new secret to be added. In this version of TLS 1.3, the two input secrets are:

- PSK (a pre-shared key established externally or derived from the resumption_master_secret value from a previous connection)

- (EC)DHE shared secret (Section 7.4)

This produces a full key derivation schedule shown in the diagram below. In this diagram, the following formatting conventions apply:

- HKDF-Extract is drawn as taking the Salt argument from the top and the IKM argument from the left, with its output to the bottom and the name of the output on the right.

- Derive-Secret’s Secret argument is indicated by the incoming arrow. For instance, the Early Secret is the Secret for generating the client_early_traffic_secret.

- "0" indicates a string of Hash-lengths bytes set to 0.

```
                 0
                 |
                 v
   PSK ->  HKDF-Extract = Early Secret
                 |
                 +-----> Derive-Secret(.,
                 |                     "ext binder" |
                 |                     "res binder",
                 |                     "")
                 |                     = binder_key
                 |
                 +-----> Derive-Secret(., "c e traffic",
                 |                     ClientHello)
                 |                     = client_early_traffic_secret
                 |
                 +-----> Derive-Secret(., "e exp master",
                 |                     ClientHello)
                 |                     = early_exporter_master_secret
                 v
           Derive-Secret(., "derived", "")
                 |
                 v
(EC)DHE -> HKDF-Extract = Handshake Secret
                 |
                 +-----> Derive-Secret(., "c hs traffic",
                 |                     ClientHello...ServerHello)
                 |                     = client_handshake_traffic_secret
                 |
                 +-----> Derive-Secret(., "s hs traffic",
                 |                     ClientHello...ServerHello)
                 |                     = server_handshake_traffic_secret
                 v
           Derive-Secret(., "derived", "")
                 |
                 v
      0 -> HKDF-Extract = Master Secret
                 |
                 +-----> Derive-Secret(., "c ap traffic",
                 |                     ClientHello...server Finished)
                 |                     = client_application_traffic_secret_0
                 |
                 +-----> Derive-Secret(., "s ap traffic",
                 |                     ClientHello...server Finished)
                 |                     = server_application_traffic_secret_0
                 |
                 +-----> Derive-Secret(., "exp master",
                 |                     ClientHello...server Finished)
                 |                     = exporter_master_secret
                 |
                 +-----> Derive-Secret(., "res master",
                                       ClientHello...client Finished)
                                       = resumption_master_secret
```

The general pattern here is that the secrets shown down the left side of the diagram are just raw entropy without context, whereas the secrets down the right side include handshake context and therefore can be used to derive working keys without additional context. Note that the different calls to Derive-Secret may take different Messages arguments, even with the same secret. In a 0-RTT exchange, Derive-Secret is called with four distinct transcripts; in a 1-RTT-only exchange with three distinct transcripts.

If a given secret is not available, then the 0-value consisting of a string of Hash.length bytes set to zeros is used. Note that this does not mean skipping rounds, so if PSK is not in use Early Secret will still be HKDF-Extract(0, 0). For the computation of the binder_secret, the label is "ext binder" for external PSKs (those provisioned outside of TLS) and "res binder" for resumption PSKs (those provisioned as the resumption master secret of a previous handshake). The different labels prevent the substitution of one type of PSK for the other.

There are multiple potential Early Secret values depending on which PSK the server ultimately selects. The client will need to compute one for each potential PSK; if no PSK is selected, it will then need to compute the early secret corresponding to the zero PSK.

Once all the values which are to be derived from a given secret have been computed, that secret SHOULD be erased.

7.2. Updating Traffic Secrets

Once the handshake is complete, it is possible for either side to update its sending traffic keys using the KeyUpdate handshake message defined in Section 4.6.3. The next generation of traffic keys is computed by generating client_/server_application_traffic_secret_N+1 from client_/server_application_traffic_secret_N as described in this section then re-deriving the traffic keys as described in Section 7.3.

The next-generation application_traffic_secret is computed as:

```
    application_traffic_secret_N+1 =
        HKDF-Expand-Label(application_traffic_secret_N,
                          "traffic upd", "", Hash.length)
```

Once client/server_application_traffic_secret_N+1 and its associated traffic keys have been computed, implementations SHOULD delete client_/server_application_traffic_secret_N and its associated traffic keys.

7.3. Traffic Key Calculation

The traffic keying material is generated from the following input values:

- A secret value

- A purpose value indicating the specific value being generated

- The length of the key being generated

The traffic keying material is generated from an input traffic secret value using:

```
 [sender]_write_key = HKDF-Expand-Label(Secret, "key", "", key_length)
 [sender]_write_iv  = HKDF-Expand-Label(Secret, "iv" , "", iv_length)
```

[sender] denotes the sending side. The Secret value for each record type is shown in the table below.

Record Type	Secret
0-RTT Application	client_early_traffic_secret
Handshake	[sender]_handshake_traffic_secret
Application Data	[sender]_application_traffic_secret_N

All the traffic keying material is recomputed whenever the underlying Secret changes (e.g., when changing from the handshake to application data keys or upon a key update).

7.4. (EC)DHE Shared Secret Calculation

7.4.1. Finite Field Diffie-Hellman

For finite field groups, a conventional Diffie-Hellman [DH76] computation is performed. The negotiated key (Z) is converted to a byte string by encoding in big-endian and left padded with zeros up to the size of the prime. This byte string is used as the shared secret in the key schedule as specified above.

Note that this construction differs from previous versions of TLS which remove leading zeros.

7.4.2. Elliptic Curve Diffie-Hellman

For secp256r1, secp384r1 and secp521r1, ECDH calculations (including parameter and key generation as well as the shared secret calculation) are performed according to [IEEE1363] using the ECKAS-DH1 scheme with the identity map as key derivation function (KDF), so that the shared secret is the x-coordinate of the ECDH shared secret elliptic curve point represented as an octet string. Note that this octet string (Z in IEEE 1363 terminology) as output by FE2OSP, the Field Element to Octet String Conversion Primitive, has constant length for any given field; leading zeros found in this octet string MUST NOT be truncated.

(Note that this use of the identity KDF is a technicality. The complete picture is that ECDH is employed with a non-trivial KDF because TLS does not directly use this secret for anything other than for computing other secrets.)

ECDH functions are used as follows:

- The public key to put into the KeyShareEntry.key_exchange structure is the result of applying the ECDH scalar multiplication function to the secret key of appropriate length (into scalar input) and the standard public basepoint (into u-coordinate point input).

- The ECDH shared secret is the result of applying the ECDH scalar multiplication function to the secret key (into scalar input) and the peer’s public key (into u-coordinate point input). The output is used raw, with no processing.

For X25519 and X448, implementations SHOULD use the approach specified in [RFC7748] to calculate the Diffie-Hellman shared secret. Implementations MUST check whether the computed Diffie-Hellman shared secret is the all-zero value and abort if so, as described in

Section 6 of [RFC7748]. If implementors use an alternative implementation of these elliptic curves, they SHOULD perform the additional checks specified in Section 7 of [RFC7748].

7.5. Exporters

[RFC5705] defines keying material exporters for TLS in terms of the TLS pseudorandom function (PRF). This document replaces the PRF with HKDF, thus requiring a new construction. The exporter interface remains the same.

The exporter value is computed as:

```
TLS-Exporter(label, context_value, key_length) =
    HKDF-Expand-Label(Derive-Secret(Secret, label, ""),
                      "exporter", Hash(context_value), key_length)
```

Where Secret is either the early_exporter_master_secret or the exporter_master_secret. Implementations MUST use the exporter_master_secret unless explicitly specified by the application. The early_exporter_master_secret is defined for use in settings where an exporter is needed for 0-RTT data. A separate interface for the early exporter is RECOMMENDED; this avoids the exporter user accidentally using an early exporter when a regular one is desired or vice versa.

If no context is provided, the context_value is zero-length. Consequently, providing no context computes the same value as providing an empty context. This is a change from previous versions of TLS where an empty context produced a different output to an absent context. As of this document’s publication, no allocated exporter label is used both with and without a context. Future specifications MUST NOT define a use of exporters that permit both an empty context and no context with the same label. New uses of exporters SHOULD provide a context in all exporter computations, though the value could be empty.

Requirements for the format of exporter labels are defined in section 4 of [RFC5705].

8. 0-RTT and Anti-Replay

As noted in Section 2.3 and Appendix E.5, TLS does not provide inherent replay protections for 0-RTT data. There are two potential threats to be concerned with:

- Network attackers who mount a replay attack by simply duplicating a flight of 0-RTT data.

- Network attackers who take advantage of client retry behavior to arrange for the server to receive multiple copies of an application message. This threat already exists to some extent because clients that value robustness respond to network errors by attempting to retry requests. However, 0-RTT adds an additional dimension for any server system which does not maintain globally consistent server state. Specifically, if a server system has multiple zones where tickets from zone A will not be accepted in zone B, then an attacker can duplicate a ClientHello and early data intended for A to both A and B. At A, the data will be accepted in 0-RTT, but at B the server will reject 0-RTT data and instead force a full handshake. If the attacker blocks the ServerHello from A, then the client will complete the handshake with B and probably retry the request, leading to duplication on the server system as a whole.

The first class of attack can be prevented by sharing state to guarantee that the 0-RTT data is accepted at most once. Servers SHOULD provide that level of replay safety, by implementing one of the methods described in this section or by equivalent means. It is understood, however, that due to operational concerns not all deployments will maintain state at that level. Therefore, in normal operation, clients will not know which, if any, of these mechanisms servers actually implement and hence MUST only send early data which they deem safe to be replayed.

In addition to the direct effects of replays, there is a class of attacks where even operations normally considered idempotent could be exploited by a large number of replays (timing attacks, resource limit exhaustion and others described in Appendix E.5). Those can be mitigated by ensuring that every 0-RTT payload can be replayed only a limited number of times. The server MUST ensure that any instance of it (be it a machine, a thread or any other entity within the relevant serving infrastructure) would accept 0-RTT for the same 0-RTT handshake at most once; this limits the number of replays to the number of server instances in the deployment. Such a guarantee can be accomplished by locally recording data from recently-received ClientHellos and rejecting repeats, or by any other method that provides the same or a stronger guarantee. The "at most once per server instance" guarantee is a minimum requirement; servers SHOULD limit 0-RTT replays further when feasible.

The second class of attack cannot be prevented at the TLS layer and MUST be dealt with by any application. Note that any application whose clients implement any kind of retry behavior already needs to implement some sort of anti-replay defense.

8.1. Single-Use Tickets

The simplest form of anti-replay defense is for the server to only allow each session ticket to be used once. For instance, the server can maintain a database of all outstanding valid tickets; deleting each ticket from the database as it is used. If an unknown ticket is provided, the server would then fall back to a full handshake.

If the tickets are not self-contained but rather are database keys, and the corresponding PSKs are deleted upon use, then connections established using PSKs enjoy forward secrecy. This improves security for all 0-RTT data and PSK usage when PSK is used without (EC)DHE.

Because this mechanism requires sharing the session database between server nodes in environments with multiple distributed servers, it may be hard to achieve high rates of successful PSK 0-RTT connections when compared to self-encrypted tickets. Unlike session databases, session tickets can successfully do PSK-based session establishment even without consistent storage, though when 0-RTT is allowed they still require consistent storage for anti-replay of 0-RTT data, as detailed in the following section.

8.2. Client Hello Recording

An alternative form of anti-replay is to record a unique value derived from the ClientHello (generally either the random value or the PSK binder) and reject duplicates. Recording all ClientHellos causes state to grow without bound, but a server can instead record ClientHellos within a given time window and use the "obfuscated_ticket_age" to ensure that tickets aren't reused outside that window.

In order to implement this, when a ClientHello is received, the server first verifies the PSK binder as described Section 4.2.11. It then computes the expected_arrival_time as described in the next section and rejects 0-RTT if it is outside the recording window, falling back to the 1-RTT handshake.

If the expected arrival time is in the window, then the server checks to see if it has recorded a matching ClientHello. If one is found, it either aborts the handshake with an "illegal_parameter" alert or accepts the PSK but reject 0-RTT. If no matching ClientHello is found, then it accepts 0-RTT and then stores the ClientHello for as long as the expected_arrival_time is inside the window. Servers MAY also implement data stores with false positives, such as Bloom filters, in which case they MUST respond to apparent replay by rejecting 0-RTT but MUST NOT abort the handshake.

The server MUST derive the storage key only from validated sections of the ClientHello. If the ClientHello contains multiple PSK identities, then an attacker can create multiple ClientHellos with different binder values for the less-preferred identity on the assumption that the server will not verify it, as recommended by Section 4.2.11. I.e., if the client sends PSKs A and B but the server prefers A, then the attacker can change the binder for B without affecting the binder for A. If the binder for B is part of the storage key, then this ClientHello will not appear as a duplicate, which will cause the ClientHello to be accepted, and may cause side effects such as replay cache pollution, although any 0-RTT data will not be decryptable because it will use different keys. If the validated binder or the ClientHello.random are used as the storage key, then this attack is not possible.

Because this mechanism does not require storing all outstanding tickets, it may be easier to implement in distributed systems with high rates of resumption and 0-RTT, at the cost of potentially weaker anti-replay defense because of the difficulty of reliably storing and retrieving the received ClientHello messages. In many such systems, it is impractical to have globally consistent storage of all the received ClientHellos. In this case, the best anti-replay protection is provided by having a single storage zone be authoritative for a given ticket and refusing 0-RTT for that ticket in any other zone. This approach prevents simple replay by the attacker because only one zone will accept 0-RTT data. A weaker design is to implement separate storage for each zone but allow 0-RTT in any zone. This approach limits the number of replays to once per zone. Application message duplication of course remains possible with either design.

When implementations are freshly started, they SHOULD reject 0-RTT as long as any portion of their recording window overlaps the startup time. Otherwise, they run the risk of accepting replays which were originally sent during that period.

Note: If the client's clock is running much faster than the server's then a ClientHello may be received that is outside the window in the future, in which case it might be accepted for 1-RTT, causing a client retry, and then acceptable later for 0-RTT. This is another variant of the second form of attack described above.

8.3. Freshness Checks

Because the ClientHello indicates the time at which the client sent it, it is possible to efficiently determine whether a ClientHello was likely sent reasonably recently and only accept 0-RTT for such a ClientHello, otherwise falling back to a 1-RTT handshake. This is necessary for the ClientHello storage mechanism described in

Section 8.2 because otherwise the server needs to store an unlimited number of ClientHellos and is a useful optimization for self-contained single-use tickets because it allows efficient rejection of ClientHellos which cannot be used for 0-RTT.

In order to implement this mechanism, a server needs to store the time that the server generated the session ticket, offset by an estimate of the round trip time between client and server. I.e.,

```
    adjusted_creation_time = creation_time + estimated_RTT
```

This value can be encoded in the ticket, thus avoiding the need to keep state for each outstanding ticket. The server can determine the client's view of the age of the ticket by subtracting the ticket's "ticket_age_add value" from the "obfuscated_ticket_age" parameter in the client's "pre_shared_key" extension. The server can determine the "expected arrival time" of the ClientHello as:

```
  expected_arrival_time = adjusted_creation_time + clients_ticket_age
```

When a new ClientHello is received, the expected_arrival_time is then compared against the current server wall clock time and if they differ by more than a certain amount, 0-RTT is rejected, though the 1-RTT handshake can be allowed to complete.

There are several potential sources of error that might cause mismatches between the expected arrival time and the measured time. Variations in client and server clock rates are likely to be minimal, though potentially the absolute times may be off by large values. Network propagation delays are the most likely causes of a mismatch in legitimate values for elapsed time. Both the NewSessionTicket and ClientHello messages might be retransmitted and therefore delayed, which might be hidden by TCP. For clients on the Internet, this implies windows on the order of ten seconds to account for errors in clocks and variations in measurements; other deployment scenarios may have different needs. Clock skew distributions are not symmetric, so the optimal tradeoff may involve an asymmetric range of permissible mismatch values.

Note that freshness checking alone is not sufficient to prevent replays because it does not detect them during the error window, which, depending on bandwidth and system capacity could include billions of replays in real-world settings. In addition, this freshness checking is only done at the time the ClientHello is received, and not when later early application data records are received. After early data is accepted, records may continue to be streamed to the server over a longer time period.

9. Compliance Requirements

9.1. Mandatory-to-Implement Cipher Suites

In the absence of an application profile standard specifying otherwise, a TLS-compliant application MUST implement the TLS_AES_128_GCM_SHA256 [GCM] cipher suite and SHOULD implement the TLS_AES_256_GCM_SHA384 [GCM] and TLS_CHACHA20_POLY1305_SHA256 [RFC7539] cipher suites. (see Appendix B.4)

A TLS-compliant application MUST support digital signatures with rsa_pkcs1_sha256 (for certificates), rsa_pss_rsae_sha256 (for CertificateVerify and certificates), and ecdsa_secp256r1_sha256. A TLS-compliant application MUST support key exchange with secp256r1 (NIST P-256) and SHOULD support key exchange with X25519 [RFC7748].

9.2. Mandatory-to-Implement Extensions

In the absence of an application profile standard specifying otherwise, a TLS-compliant application MUST implement the following TLS extensions:

- Supported Versions ("supported_versions"; Section 4.2.1)

- Cookie ("cookie"; Section 4.2.2)

- Signature Algorithms ("signature_algorithms"; Section 4.2.3)

- Signature Algorithms Certificate ("signature_algorithms_cert"; Section 4.2.3)

- Negotiated Groups ("supported_groups"; Section 4.2.7)

- Key Share ("key_share"; Section 4.2.8)

- Server Name Indication ("server_name"; Section 3 of [RFC6066])

All implementations MUST send and use these extensions when offering applicable features:

- "supported_versions" is REQUIRED for all ClientHello, ServerHello and HelloRetryRequest messages.

- "signature_algorithms" is REQUIRED for certificate authentication.

- "supported_groups" is REQUIRED for ClientHello messages using DHE or ECDHE key exchange.

- "key_share" is REQUIRED for DHE or ECDHE key exchange.

- "pre_shared_key" is REQUIRED for PSK key agreement.

- "psk_key_exchange_modes" is REQUIRED for PSK key agreement.

A client is considered to be attempting to negotiate using this specification if the ClientHello contains a "supported_versions" extension with 0x0304 contained in its body. Such a ClientHello message MUST meet the following requirements:

- If not containing a "pre_shared_key" extension, it MUST contain both a "signature_algorithms" extension and a "supported_groups" extension.

- If containing a "supported_groups" extension, it MUST also contain a "key_share" extension, and vice versa. An empty KeyShare.client_shares vector is permitted.

Servers receiving a ClientHello which does not conform to these requirements MUST abort the handshake with a "missing_extension" alert.

Additionally, all implementations MUST support use of the "server_name" extension with applications capable of using it. Servers MAY require clients to send a valid "server_name" extension. Servers requiring this extension SHOULD respond to a ClientHello lacking a "server_name" extension by terminating the connection with a "missing_extension" alert.

9.3. Protocol Invariants

This section describes invariants that TLS endpoints and middleboxes MUST follow. It also applies to earlier versions of TLS.

TLS is designed to be securely and compatibly extensible. Newer clients or servers, when communicating with newer peers, should negotiate the most preferred common parameters. The TLS handshake provides downgrade protection: Middleboxes passing traffic between a newer client and newer server without terminating TLS should be unable to influence the handshake (see Appendix E.1). At the same time, deployments update at different rates, so a newer client or server MAY continue to support older parameters, which would allow it to interoperate with older endpoints.

For this to work, implementations MUST correctly handle extensible fields:

- A client sending a ClientHello MUST support all parameters advertised in it. Otherwise, the server may fail to interoperate by selecting one of those parameters.

- A server receiving a ClientHello MUST correctly ignore all unrecognized cipher suites, extensions, and other parameters. Otherwise, it may fail to interoperate with newer clients. In TLS 1.3, a client receiving a CertificateRequest or NewSessionTicket MUST also ignore all unrecognized extensions.

- A middlebox which terminates a TLS connection MUST behave as a compliant TLS server (to the original client), including having a certificate which the client is willing to accept, and as a compliant TLS client (to the original server), including verifying the original server's certificate. In particular, it MUST generate its own ClientHello containing only parameters it understands, and it MUST generate a fresh ServerHello random value, rather than forwarding the endpoint's value.

  Note that TLS's protocol requirements and security analysis only apply to the two connections separately. Safely deploying a TLS terminator requires additional security considerations which are beyond the scope of this document.

- An middlebox which forwards ClientHello parameters it does not understand MUST NOT process any messages beyond that ClientHello. It MUST forward all subsequent traffic unmodified. Otherwise, it may fail to interoperate with newer clients and servers.

  Forwarded ClientHellos may contain advertisements for features not supported by the middlebox, so the response may include future TLS additions the middlebox does not recognize. These additions MAY change any message beyond the ClientHello arbitrarily. In particular, the values sent in the ServerHello might change, the ServerHello format might change, and the TLSCiphertext format might change.

The design of TLS 1.3 was constrained by widely-deployed non-compliant TLS middleboxes (see Appendix D.4), however it does not relax the invariants. Those middleboxes continue to be non-compliant.

10. Security Considerations

Security issues are discussed throughout this memo, especially in Appendix C, Appendix D, and Appendix E.

11. IANA Considerations

This document uses several registries that were originally created in [RFC4346]. IANA [SHALL update/has updated] these to reference this document. The registries and their allocation policies are below:

- TLS Cipher Suite Registry: values with the first byte in the range 0-254 (decimal) are assigned via Specification Required [RFC8126]. Values with the first byte 255 (decimal) are reserved for Private Use [RFC8126].

  IANA [SHALL add/has added] the cipher suites listed in Appendix B.4 to the registry. The "Value" and "Description" columns are taken from the table. The "DTLS-OK" and "Recommended" columns are both marked as "Yes" for each new cipher suite. [[This assumes [I-D.ietf-tls-iana-registry-updates] has been applied.]]

- TLS ContentType Registry: Future values are allocated via Standards Action [RFC8126].

- TLS Alert Registry: Future values are allocated via Standards Action [RFC8126]. IANA [SHALL update/has updated] this registry to include values for "missing_extension" and "certificate_required". The "DTLS-OK" column is marked as "Yes" for each new alert.

- TLS HandshakeType Registry: Future values are allocated via Standards Action [RFC8126]. IANA [SHALL update/has updated] this registry to rename item 4 from "NewSessionTicket" to "new_session_ticket" and to add the "hello_retry_request_RESERVED", "encrypted_extensions", "end_of_early_data", "key_update", and "message_hash" values. The "DTLS-OK" are marked as "Yes" for each of these additions.

This document also uses the TLS ExtensionType Registry originally created in [RFC4366]. IANA has updated it to reference this document. Changes to the registry follow:

- IANA [SHALL update/has updated] the registration policy as follows:

  Values with the first byte in the range 0-254 (decimal) are assigned via Specification Required [RFC8126]. Values with the first byte 255 (decimal) are reserved for Private Use [RFC8126].

- IANA [SHALL update/has updated] this registry to include the "key_share", "pre_shared_key", "psk_key_exchange_modes", "early_data", "cookie", "supported_versions", "certificate_authorities", "oid_filters", "post_handshake_auth", and "signature_algorithms_cert", extensions with the values defined in this document and the Recommended value of "Yes".

- IANA [SHALL update/has updated] this registry to include a "TLS 1.3" column which lists the messages in which the extension may appear. This column [SHALL be/has been] initially populated from the table in Section 4.2 with any extension not listed there marked as "-" to indicate that it is not used by TLS 1.3.

In addition, this document defines two new registries to be maintained by IANA:

- TLS SignatureScheme Registry: Values with the first byte in the range 0-253 (decimal) are assigned via Specification Required [RFC8126]. Values with the first byte 254 or 255 (decimal) are reserved for Private Use [RFC8126]. Values with the first byte in the range 0-6 or with the second byte in the range 0-3 that are not currently allocated are reserved for backwards compatibility. This registry SHALL have a "Recommended" column. The registry [shall be/ has been] initially populated with the values described in Section 4.2.3. The following values SHALL be marked as "Recommended": ecdsa_secp256r1_sha256, ecdsa_secp384r1_sha384, rsa_pss_rsae_sha256, rsa_pss_rsae_sha384, rsa_pss_rsae_sha512, rsa_pss_pss_sha256, rsa_pss_pss_sha384, rsa_pss_pss_sha512, and ed25519.

- TLS PskKeyExchangeMode Registry: Values in the range 0-253 (decimal) are assigned via Specification Required [RFC8126]. Values with the first byte 254 or 255 (decimal) are reserved for Private Use [RFC8126]. This registry SHALL have a "Recommended" column. The registry [shall be/ has been] initially populated psk_ke (0) and psk_dhe_ke (1). Both SHALL be marked as "Recommended".

12. References

12.1. Normative References

[DH] Diffie, W. and M. Hellman, "New Directions in Cryptography", IEEE Transactions on Information Theory, V.IT-22 n.6 , June 1977.

[DH76] Diffie, W. and M. Hellman, "New directions in cryptography", IEEE Transactions on Information Theory Vol. 22, pp. 644-654, DOI 10.1109/tit.1976.1055638, November 1976.

[GCM] Dworkin, M., "Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC", NIST Special Publication 800-38D, November 2007.

[RFC2104] Krawczyk, H., Bellare, M., and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication", RFC 2104, DOI 10.17487/RFC2104, February 1997, <https://www.rfc-editor.org/info/rfc2104>.

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

[RFC5116] McGrew, D., "An Interface and Algorithms for Authenticated Encryption", RFC 5116, DOI 10.17487/RFC5116, January 2008, <https://www.rfc-editor.org/info/rfc5116>.

[RFC5280] Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R., and W. Polk, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", RFC 5280, DOI 10.17487/RFC5280, May 2008, <https://www.rfc-editor.org/info/rfc5280>.

[RFC5705] Rescorla, E., "Keying Material Exporters for Transport Layer Security (TLS)", RFC 5705, DOI 10.17487/RFC5705, March 2010, <https://www.rfc-editor.org/info/rfc5705>.

[RFC5756] Turner, S., Brown, D., Yiu, K., Housley, R., and T. Polk, "Updates for RSAES-OAEP and RSASSA-PSS Algorithm Parameters", RFC 5756, DOI 10.17487/RFC5756, January 2010, <https://www.rfc-editor.org/info/rfc5756>.

[RFC5869] Krawczyk, H. and P. Eronen, "HMAC-based Extract-and-Expand Key Derivation Function (HKDF)", RFC 5869, DOI 10.17487/RFC5869, May 2010, <https://www.rfc-editor.org/info/rfc5869>.

[RFC6066] Eastlake 3rd, D., "Transport Layer Security (TLS) Extensions: Extension Definitions", RFC 6066, DOI 10.17487/RFC6066, January 2011, <https://www.rfc-editor.org/info/rfc6066>.

[RFC6655] McGrew, D. and D. Bailey, "AES-CCM Cipher Suites for Transport Layer Security (TLS)", RFC 6655, DOI 10.17487/RFC6655, July 2012, <https://www.rfc-editor.org/info/rfc6655>.

[RFC6960] Santesson, S., Myers, M., Ankney, R., Malpani, A., Galperin, S., and C. Adams, "X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP", RFC 6960, DOI 10.17487/RFC6960, June 2013, <https://www.rfc-editor.org/info/rfc6960>.

[RFC6961] Pettersen, Y., "The Transport Layer Security (TLS) Multiple Certificate Status Request Extension", RFC 6961, DOI 10.17487/RFC6961, June 2013, <https://www.rfc-editor.org/info/rfc6961>.

[RFC6962] Laurie, B., Langley, A., and E. Kasper, "Certificate Transparency", RFC 6962, DOI 10.17487/RFC6962, June 2013, <https://www.rfc-editor.org/info/rfc6962>.

[RFC6979] Pornin, T., "Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA)", RFC 6979, DOI 10.17487/RFC6979, August 2013, <https://www.rfc-editor.org/info/rfc6979>.

[RFC7301] Friedl, S., Popov, A., Langley, A., and E. Stephan, "Transport Layer Security (TLS) Application-Layer Protocol Negotiation Extension", RFC 7301, DOI 10.17487/RFC7301, July 2014, <https://www.rfc-editor.org/info/rfc7301>.

[RFC7507] Moeller, B. and A. Langley, "TLS Fallback Signaling Cipher Suite Value (SCSV) for Preventing Protocol Downgrade Attacks", RFC 7507, DOI 10.17487/RFC7507, April 2015, <https://www.rfc-editor.org/info/rfc7507>.

[RFC7539] Nir, Y. and A. Langley, "ChaCha20 and Poly1305 for IETF Protocols", RFC 7539, DOI 10.17487/RFC7539, May 2015, <https://www.rfc-editor.org/info/rfc7539>.

[RFC7748] Langley, A., Hamburg, M., and S. Turner, "Elliptic Curves for Security", RFC 7748, DOI 10.17487/RFC7748, January 2016, <https://www.rfc-editor.org/info/rfc7748>.

[RFC7919] Gillmor, D., "Negotiated Finite Field Diffie-Hellman Ephemeral Parameters for Transport Layer Security (TLS)", RFC 7919, DOI 10.17487/RFC7919, August 2016, <https://www.rfc-editor.org/info/rfc7919>.

[RFC8017] Moriarty, K., Ed., Kaliski, B., Jonsson, J., and A. Rusch, "PKCS #1: RSA Cryptography Specifications Version 2.2", RFC 8017, DOI 10.17487/RFC8017, November 2016, <https://www.rfc-editor.org/info/rfc8017>.

[RFC8032] Josefsson, S. and I. Liusvaara, "Edwards-Curve Digital Signature Algorithm (EdDSA)", RFC 8032, DOI 10.17487/RFC8032, January 2017, <https://www.rfc-editor.org/info/rfc8032>.

[RFC8126] Cotton, M., Leiba, B., and T. Narten, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 8126, DOI 10.17487/RFC8126, June 2017, <https://www.rfc-editor.org/info/rfc8126>.

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

[SHS] Dang, Q., "Secure Hash Standard", National Institute of Standards and Technology report, DOI 10.6028/nist.fips.180-4, July 2015.

[X690] ITU-T, "Information technology - ASN.1 encoding Rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)", ISO/IEC 8825-1:2002, 2002.

[X962] ANSI, "Public Key Cryptography For The Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA)", ANSI X9.62, 1998.

12.2. Informative References

[AEAD-LIMITS]
Luykx, A. and K. Paterson, "Limits on Authenticated Encryption Use in TLS", 2016, <http://www.isg.rhul.ac.uk/~kp/TLS-AEbounds.pdf>.

[Anon18] Anonymous, A., "Secure Channels for Multiplexed Data Streams: Analyzing the TLS 1.3 Record Layer Without Elision", In submission to CRYPTO 2018. RFC EDITOR: PLEASE UPDATE THIS REFERENCE AFTER FINAL NOTIFICATION (2018-4-29). , 2018.

[BBFKZG16]
Bhargavan, K., Brzuska, C., Fournet, C., Kohlweiss, M., Zanella-Beguelin, S., and M. Green, "Downgrade Resilience in Key-Exchange Protocols", Proceedings of IEEE Symposium on Security and Privacy (Oakland) 2016 , 2016.

[BBK17] Bhargavan, K., Blanchet, B., and N. Kobeissi, "Verified Models and Reference Implementations for the TLS 1.3 Standard Candidate", Proceedings of IEEE Symposium on Security and Privacy (Oakland) 2017 , 2017.

[BDFKPPRSZZ16] Bhargavan, K., Delignat-Lavaud, A., Fournet, C., Kohlweiss, M., Pan, J., Protzenko, J., Rastogi, A., Swamy, N., Zanella-Beguelin, S., and J. Zinzindohoue, "Implementing and Proving the TLS 1.3 Record Layer", Proceedings of IEEE Symposium on Security and Privacy (Oakland) 2017 , December 2016, <https://eprint.iacr.org/2016/1178>.

[Ben17a] Benjamin, D., "Presentation before the TLS WG at IETF 100", 2017, <https://datatracker.ietf.org/meeting/100/materials/slides-100-tls-sessa-tls13/>.

[Ben17b] Benjamin, D., "Additional TLS 1.3 results from Chrome", 2017, <https://www.ietf.org/mail-archive/web/tls/current/msg25168.html>.

[BMMT15] Badertscher, C., Matt, C., Maurer, U., and B. Tackmann, "Augmented Secure Channels and the Goal of the TLS 1.3 Record Layer", ProvSec 2015 , September 2015, <https://eprint.iacr.org/2015/394>.

[BT16] Bellare, M. and B. Tackmann, "The Multi-User Security of Authenticated Encryption: AES-GCM in TLS 1.3", Proceedings of CRYPTO 2016 , 2016, <https://eprint.iacr.org/2016/564>.

[CCG16] Cohn-Gordon, K., Cremers, C., and L. Garratt, "On Post-Compromise Security", IEEE Computer Security Foundations Symposium , 2015.

[CHECKOWAY] Checkoway, S., Shacham, H., Maskiewicz, J., Garman, C., Fried, J., Cohney, S., Green, M., Heninger, N., Weinmann, R., and E. Rescorla, "A Systematic Analysis of the Juniper Dual EC Incident", Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security - CCS’16, DOI 10.1145/2976749.2978395, 2016.

[CHHSV17] Cremers, C., Horvat, M., Hoyland, J., van der Merwe, T., and S. Scott, "Awkward Handshake: Possible mismatch of client/server view on client authentication in post-handshake mode in Revision 18", 2017, <https://www.ietf.org/mail-archive/web/tls/current/msg22382.html>.

[CHSV16] Cremers, C., Horvat, M., Scott, S., and T. van der Merwe, "Automated Analysis and Verification of TLS 1.3: 0-RTT, Resumption and Delayed Authentication", Proceedings of IEEE Symposium on Security and Privacy (Oakland) 2016 , 2016, <http://ieeexplore.ieee.org/document/7546518/>.

[CK01] Canetti, R. and H. Krawczyk, "Analysis of Key-Exchange Protocols and Their Use for Building Secure Channels", Proceedings of Eurocrypt 2001 , 2001.

[CLINIC] Miller, B., Huang, L., Joseph, A., and J. Tygar, "I Know Why You Went to the Clinic: Risks and Realization of HTTPS Traffic Analysis", Privacy Enhancing Technologies pp. 143-163, DOI 10.1007/978-3-319-08506-7_8, 2014.

[DFGS15] Dowling, B., Fischlin, M., Guenther, F., and D. Stebila, "A Cryptographic Analysis of the TLS 1.3 draft-10 Full and Pre-shared Key Handshake Protocol", Proceedings of ACM CCS 2015 , 2015, <https://eprint.iacr.org/2015/914>.

[DFGS16] Dowling, B., Fischlin, M., Guenther, F., and D. Stebila, "A Cryptographic Analysis of the TLS 1.3 draft-10 Full and Pre-shared Key Handshake Protocol", TRON 2016 , 2016, <https://eprint.iacr.org/2016/081>.

[DOW92] Diffie, W., van Oorschot, P., and M. Wiener, "Authentication and authenticated key exchanges", Designs, Codes and Cryptography , 1992.

[DSS] National Institute of Standards and Technology, U.S. Department of Commerce, "Digital Signature Standard, version 4", NIST FIPS PUB 186-4, 2013.

[ECDSA] American National Standards Institute, "Public Key Cryptography for the Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA)", ANSI ANS X9.62-2005, November 2005.

[FG17] Fischlin, M. and F. Guenther, "Replay Attacks on Zero Round-Trip Time: The Case of the TLS 1.3 Handshake Candidates", Proceedings of Euro S"P 2017 , 2017, <https://eprint.iacr.org/2017/082>.

[FGSW16] Fischlin, M., Guenther, F., Schmidt, B., and B. Warinschi, "Key Confirmation in Key Exchange: A Formal Treatment and Implications for TLS 1.3", Proceedings of IEEE Symposium on Security and Privacy (Oakland) 2016 , 2016, <http://ieeexplore.ieee.org/document/7546517/>.

[FW15] Florian Weimer, ., "Factoring RSA Keys With TLS Perfect Forward Secrecy", September 2015.

[HCJ16] Husak, M., &#268;ermak, M., Jirsik, T., and P. &#268;eleda, "HTTPS traffic analysis and client identification using passive SSL/TLS fingerprinting", EURASIP Journal on Information Security Vol. 2016, DOI 10.1186/s13635-016-0030-7, February 2016.

[HGFS15] Hlauschek, C., Gruber, M., Fankhauser, F., and C. Schanes, "Prying Open Pandora’s Box: KCI Attacks against TLS", Proceedings of USENIX Workshop on Offensive Technologies , 2015.

[I-D.ietf-tls-iana-registry-updates]
Salowey, J. and S. Turner, "IANA Registry Updates for TLS and DTLS", draft-ietf-tls-iana-registry-updates-04 (work in progress), February 2018.

[I-D.ietf-tls-tls13-vectors]
Thomson, M., "Example Handshake Traces for TLS 1.3", draft-ietf-tls-tls13-vectors-03 (work in progress), December 2017.

[IEEE1363]
IEEE, "Standard Specifications for Public Key Cryptography", IEEE 1363 , 2000.

[JSS15] Jager, T., Schwenk, J., and J. Somorovsky, "On the Security of TLS 1.3 and QUIC Against Weaknesses in PKCS#1 v1.5 Encryption", Proceedings of ACM CCS 2015 , 2015, <https://www.nds.rub.de/media/nds/veroeffentlichungen/2015/08/21/Tls13QuicAttacks.pdf>.

[KEYAGREEMENT]
Barker, E., Chen, L., Roginsky, A., and M. Smid, "Recommendation for Pair-Wise Key Establishment Schemes Using Discrete Logarithm Cryptography", National Institute of Standards and Technology report, DOI 10.6028/nist.sp.800-56ar2, May 2013.

[Kraw10] Krawczyk, H., "Cryptographic Extraction and Key Derivation: The HKDF Scheme", Proceedings of CRYPTO 2010 , 2010, <https://eprint.iacr.org/2010/264>.

[Kraw16] Krawczyk, H., "A Unilateral-to-Mutual Authentication Compiler for Key Exchange (with Applications to Client Authentication in TLS 1.3", Proceedings of ACM CCS 2016 , 2016, <https://eprint.iacr.org/2016/711>.

[KW16] Krawczyk, H. and H. Wee, "The OPTLS Protocol and TLS 1.3", Proceedings of Euro S"P 2016 , 2016, <https://eprint.iacr.org/2015/978>.

[LXZFH16] Li, X., Xu, J., Feng, D., Zhang, Z., and H. Hu, "Multiple Handshakes Security of TLS 1.3 Candidates", Proceedings of IEEE Symposium on Security and Privacy (Oakland) 2016 , 2016, <http://ieeexplore.ieee.org/document/7546519/>.

[Mac17] MacCarthaigh, C., "Security Review of TLS1.3 0-RTT", 2017, <https://github.com/tlswg/tls13-spec/issues/1001>.

[PSK-FINISHED]
Cremers, C., Horvat, M., van der Merwe, T., and S. Scott, "Revision 10: possible attack if client authentication is allowed during PSK", 2015, <https://www.ietf.org/mail-archive/web/tls/current/msg18215.html>.

[REKEY] Abdalla, M. and M. Bellare, "Increasing the Lifetime of a Key: A Comparative Analysis of the Security of Re-keying Techniques", ASIACRYPT2000 , October 2000.

[Res17a] Rescorla, E., "Preliminary data on Firefox TLS 1.3 Middlebox experiment", 2017, <https://www.ietf.org/mail-archive/web/tls/current/msg25091.html>.

[Res17b] Rescorla, E., "More compatibility measurement results", 2017, <https://www.ietf.org/mail-archive/web/tls/current/msg25179.html>.

[RFC3552] Rescorla, E. and B. Korver, "Guidelines for Writing RFC Text on Security Considerations", BCP 72, RFC 3552, DOI 10.17487/RFC3552, July 2003, <https://www.rfc-editor.org/info/rfc3552>.

[RFC4086] Eastlake 3rd, D., Schiller, J., and S. Crocker, "Randomness Requirements for Security", BCP 106, RFC 4086, DOI 10.17487/RFC4086, June 2005, <https://www.rfc-editor.org/info/rfc4086>.

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

[RFC4366] Blake-Wilson, S., Nystrom, M., Hopwood, D., Mikkelsen, J., and T. Wright, "Transport Layer Security (TLS) Extensions", RFC 4366, DOI 10.17487/RFC4366, April 2006, <https://www.rfc-editor.org/info/rfc4366>.

[RFC4492] Blake-Wilson, S., Bolyard, N., Gupta, V., Hawk, C., and B. Moeller, "Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS)", RFC 4492, DOI 10.17487/RFC4492, May 2006, <https://www.rfc-editor.org/info/rfc4492>.

[RFC5077] Salowey, J., Zhou, H., Eronen, P., and H. Tschofenig, "Transport Layer Security (TLS) Session Resumption without Server-Side State", RFC 5077, DOI 10.17487/RFC5077, January 2008, <https://www.rfc-editor.org/info/rfc5077>.

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

[RFC5764] McGrew, D. and E. Rescorla, "Datagram Transport Layer Security (DTLS) Extension to Establish Keys for the Secure Real-time Transport Protocol (SRTP)", RFC 5764, DOI 10.17487/RFC5764, May 2010, <https://www.rfc-editor.org/info/rfc5764>.

[RFC5929] Altman, J., Williams, N., and L. Zhu, "Channel Bindings for TLS", RFC 5929, DOI 10.17487/RFC5929, July 2010, <https://www.rfc-editor.org/info/rfc5929>.

[RFC6091] Mavrogiannopoulos, N. and D. Gillmor, "Using OpenPGP Keys for Transport Layer Security (TLS) Authentication", RFC 6091, DOI 10.17487/RFC6091, February 2011, <https://www.rfc-editor.org/info/rfc6091>.

[RFC6176] Turner, S. and T. Polk, "Prohibiting Secure Sockets Layer (SSL) Version 2.0", RFC 6176, DOI 10.17487/RFC6176, March 2011, <https://www.rfc-editor.org/info/rfc6176>.

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

[RFC6520] Seggelmann, R., Tuexen, M., and M. Williams, "Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS) Heartbeat Extension", RFC 6520, DOI 10.17487/RFC6520, February 2012, <https://www.rfc-editor.org/info/rfc6520>.

[RFC7230] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing", RFC 7230, DOI 10.17487/RFC7230, June 2014, <https://www.rfc-editor.org/info/rfc7230>.

[RFC7250] Wouters, P., Ed., Tschofenig, H., Ed., Gilmore, J., Weiler, S., and T. Kivinen, "Using Raw Public Keys in Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)", RFC 7250, DOI 10.17487/RFC7250, June 2014, <https://www.rfc-editor.org/info/rfc7250>.

[RFC7465] Popov, A., "Prohibiting RC4 Cipher Suites", RFC 7465, DOI 10.17487/RFC7465, February 2015, <https://www.rfc-editor.org/info/rfc7465>.

[RFC7568] Barnes, R., Thomson, M., Pironti, A., and A. Langley, "Deprecating Secure Sockets Layer Version 3.0", RFC 7568, DOI 10.17487/RFC7568, June 2015, <https://www.rfc-editor.org/info/rfc7568>.

[RFC7627] Bhargavan, K., Ed., Delignat-Lavaud, A., Pironti, A., Langley, A., and M. Ray, "Transport Layer Security (TLS) Session Hash and Extended Master Secret Extension", RFC 7627, DOI 10.17487/RFC7627, September 2015, <https://www.rfc-editor.org/info/rfc7627>.

[RFC7685] Langley, A., "A Transport Layer Security (TLS) ClientHello Padding Extension", RFC 7685, DOI 10.17487/RFC7685, October 2015, <https://www.rfc-editor.org/info/rfc7685>.

[RFC7924] Santesson, S. and H. Tschofenig, "Transport Layer Security (TLS) Cached Information Extension", RFC 7924, DOI 10.17487/RFC7924, July 2016, <https://www.rfc-editor.org/info/rfc7924>.

[RFC8305] Schinazi, D. and T. Pauly, "Happy Eyeballs Version 2: Better Connectivity Using Concurrency", RFC 8305, DOI 10.17487/RFC8305, December 2017, <https://www.rfc-editor.org/info/rfc8305>.

[RSA] Rivest, R., Shamir, A., and L. Adleman, "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems", Communications of the ACM v. 21, n. 2, pp. 120-126., February 1978.

[SIGMA] Krawczyk, H., "SIGMA: the 'SIGn-and-MAc' approach to authenticated Diffie-Hellman and its use in the IKE protocols", Proceedings of CRYPTO 2003 , 2003.

[SLOTH] Bhargavan, K. and G. Leurent, "Transcript Collision Attacks: Breaking Authentication in TLS, IKE, and SSH", Network and Distributed System Security Symposium (NDSS 2016) , 2016.

[SSL2] Hickman, K., "The SSL Protocol", February 1995.

[SSL3] Freier, A., Karlton, P., and P. Kocher, "The SSL 3.0 Protocol", November 1996.

[TIMING] Boneh, D. and D. Brumley, "Remote timing attacks are practical", USENIX Security Symposium, 2003.

[X501] "Information Technology - Open Systems Interconnection - The Directory: Models", ITU-T X.501, 1993.

12.3. URIs

[1] mailto:tls@ietf.org

Appendix A. State Machine

This section provides a summary of the legal state transitions for the client and server handshakes. State names (in all capitals, e.g., START) have no formal meaning but are provided for ease of comprehension. Actions which are taken only in certain circumstances are indicated in []. The notation "K_{send,recv} = foo" means "set the send/recv key to the given key".

A.1. Client

```
                              START <----+
               Send ClientHello |        | Recv HelloRetryRequest
          [K_send = early data] |        |
                                v        |
           /                 WAIT_SH ----+
           |                    | Recv ServerHello
           |                    | K_recv = handshake
       Can |                    V
      send |                 WAIT_EE
     early |                    | Recv EncryptedExtensions
      data |           +--------+--------+
           |     Using |                 | Using certificate
           |       PSK |                 v
           |           |            WAIT_CERT_CR
           |           |        Recv |       | Recv CertificateRequest
           |           | Certificate |       v
           |           |             |    WAIT_CERT
           |           |             |       | Recv Certificate
           |           |             v       v
           |           |              WAIT_CV
           |           |                 | Recv CertificateVerify
           |           +> WAIT_FINISHED <+
           |                  | Recv Finished
           \                  | [Send EndOfEarlyData]
                              | K_send = handshake
                              | [Send Certificate [+ CertificateVerify]]
    Can send                  | Send Finished
    app data   -->            | K_send = K_recv = application
    after here                v
                          CONNECTED
```

Note that with the transitions as shown above, clients may send alerts that derive from post-ServerHello messages in the clear or with the early data keys. If clients need to send such alerts, they SHOULD first rekey to the handshake keys if possible.

A.2. Server

```
                              START <-----+
               Recv ClientHello |         | Send HelloRetryRequest
                                v         |
                             RECVD_CH ----+
                                | Select parameters
                                v
                             NEGOTIATED
                                | Send ServerHello
                                | K_send = handshake
                                | Send EncryptedExtensions
                                | [Send CertificateRequest]
 Can send                       | [Send Certificate + CertificateVerify]
 app data                       | Send Finished
 after   -->                    | K_send = application
 here                  +--------+--------+
              No 0-RTT |                 | 0-RTT
                       |                 |
   K_recv = handshake  |                 | K_recv = early data
 [Skip decrypt errors] |    +------> WAIT_EOED -+
                       |    |       Recv |      | Recv EndOfEarlyData
                       |    | early data |      | K_recv = handshake
                       |    +------------+      |
                       |                        |
                       +> WAIT_FLIGHT2 <--------+
                                |
                       +--------+--------+
               No auth |                 | Client auth
                       |                 |
                       |                 v
                       |             WAIT_CERT
                       |        Recv |       | Recv Certificate
                       |       empty |       v
                       | Certificate |    WAIT_CV
                       |             |       | Recv
                       |             v       | CertificateVerify
                       +-> WAIT_FINISHED <---+
                                | Recv Finished
                                | K_recv = application
                                v
                            CONNECTED
```

Appendix B. Protocol Data Structures and Constant Values

This section provides the normative protocol types and constants definitions. Values listed as _RESERVED were used in previous versions of TLS and are listed here for completeness. TLS 1.3

implementations MUST NOT send them but might receive them from older TLS implementations.

B.1. Record Layer

```
      enum {
          invalid(0),
          change_cipher_spec(20),
          alert(21),
          handshake(22),
          application_data(23),
          (255)
      } ContentType;

      struct {
          ContentType type;
          ProtocolVersion legacy_record_version;
          uint16 length;
          opaque fragment[TLSPlaintext.length];
      } TLSPlaintext;

      struct {
          opaque content[TLSPlaintext.length];
          ContentType type;
          uint8 zeros[length_of_padding];
      } TLSInnerPlaintext;

      struct {
          ContentType opaque_type = application_data; /* 23 */
          ProtocolVersion legacy_record_version = 0x0303; /* TLS v1.2 */
          uint16 length;
          opaque encrypted_record[TLSCiphertext.length];
      } TLSCiphertext;
```

B.2. Alert Messages

```
      enum { warning(1), fatal(2), (255) } AlertLevel;

      enum {
          close_notify(0),
          unexpected_message(10),
          bad_record_mac(20),
          decryption_failed_RESERVED(21),
          record_overflow(22),
          decompression_failure_RESERVED(30),
          handshake_failure(40),
          no_certificate_RESERVED(41),
          bad_certificate(42),
          unsupported_certificate(43),
          certificate_revoked(44),
          certificate_expired(45),
          certificate_unknown(46),
          illegal_parameter(47),
          unknown_ca(48),
          access_denied(49),
          decode_error(50),
          decrypt_error(51),
          export_restriction_RESERVED(60),
          protocol_version(70),
          insufficient_security(71),
          internal_error(80),
          inappropriate_fallback(86),
          user_canceled(90),
          no_renegotiation_RESERVED(100),
          missing_extension(109),
          unsupported_extension(110),
          certificate_unobtainable_RESERVED(111),
          unrecognized_name(112),
          bad_certificate_status_response(113),
          bad_certificate_hash_value_RESERVED(114),
          unknown_psk_identity(115),
          certificate_required(116),
          no_application_protocol(120),
          (255)
      } AlertDescription;

      struct {
          AlertLevel level;
          AlertDescription description;
      } Alert;
```

B.3. Handshake Protocol

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

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

B.3.1. Key Exchange Messages

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

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

   struct {
       ProtocolVersion legacy_version = 0x0303;    /* TLS v1.2 */
       Random random;
       opaque legacy_session_id<0..32>;
       CipherSuite cipher_suites<2..2^16-2>;
       opaque legacy_compression_methods<1..2^8-1>;
       Extension extensions<8..2^16-1>;
   } ClientHello;

   struct {
       ProtocolVersion legacy_version = 0x0303;    /* TLS v1.2 */
       Random random;
       opaque legacy_session_id_echo<0..32>;
       CipherSuite cipher_suite;
       uint8 legacy_compression_method = 0;
       Extension extensions<6..2^16-1>;
   } ServerHello;

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

   enum {
       server_name(0),                             /* RFC 6066 */
       max_fragment_length(1),                     /* RFC 6066 */
       status_request(5),                          /* RFC 6066 */
       supported_groups(10),                       /* RFC 4492, 7919 */
       signature_algorithms(13),                   /* [[this document]] */
       use_srtp(14),                               /* RFC 5764 */
       heartbeat(15),                              /* RFC 6520 */
       application_layer_protocol_negotiation(16), /* RFC 7301 */
       signed_certificate_timestamp(18),           /* RFC 6962 */
       client_certificate_type(19),                /* RFC 7250 */
       server_certificate_type(20),                /* RFC 7250 */
       padding(21),                                /* RFC 7685 */
       RESERVED(40),                               /* Used but never assigned */
       pre_shared_key(41),                         /* [[this document]] */
       early_data(42),                             /* [[this document]] */
       supported_versions(43),                     /* [[this document]] */
       cookie(44),                                 /* [[this document]] */
       psk_key_exchange_modes(45),                 /* [[this document]] */
       RESERVED(46),                               /* Used but never assigned */
       certificate_authorities(47),                /* [[this document]] */
       oid_filters(48),                            /* [[this document]] */
       post_handshake_auth(49),                    /* [[this document]] */
       signature_algorithms_cert(50),              /* [[this document]] */
       key_share(51),                              /* [[this document]] */
       (65535)
```

```
   } ExtensionType;

   struct {
       NamedGroup group;
       opaque key_exchange<1..2^16-1>;
   } KeyShareEntry;

   struct {
       KeyShareEntry client_shares<0..2^16-1>;
   } KeyShareClientHello;

   struct {
       NamedGroup selected_group;
   } KeyShareHelloRetryRequest;

   struct {
       KeyShareEntry server_share;
   } KeyShareServerHello;

   struct {
       uint8 legacy_form = 4;
       opaque X[coordinate_length];
       opaque Y[coordinate_length];
   } UncompressedPointRepresentation;

   enum { psk_ke(0), psk_dhe_ke(1), (255) } PskKeyExchangeMode;

   struct {
       PskKeyExchangeMode ke_modes<1..255>;
   } PskKeyExchangeModes;

   struct {} Empty;

   struct {
       select (Handshake.msg_type) {
           case new_session_ticket:   uint32 max_early_data_size;
           case client_hello:         Empty;
           case encrypted_extensions: Empty;
       };
   } EarlyDataIndication;

   struct {
       opaque identity<1..2^16-1>;
       uint32 obfuscated_ticket_age;
   } PskIdentity;

   opaque PskBinderEntry<32..255>;
   struct {
       PskIdentity identities<7..2^16-1>;
       PskBinderEntry binders<33..2^16-1>;
   } OfferedPsks;

   struct {
       select (Handshake.msg_type) {
           case client_hello: OfferedPsks;
           case server_hello: uint16 selected_identity;
       };
   } PreSharedKeyExtension;
```

B.3.1.1. Version Extension

```
      struct {
          select (Handshake.msg_type) {
              case client_hello:
                   ProtocolVersion versions<2..254>;

              case server_hello: /* and HelloRetryRequest */
                   ProtocolVersion selected_version;
          };
      } SupportedVersions;
```

B.3.1.2. Cookie Extension

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

B.3.1.3. Signature Algorithm Extension

```
      enum {
          /* RSASSA-PKCS1-v1_5 algorithms */
          rsa_pkcs1_sha256(0x0401),
          rsa_pkcs1_sha384(0x0501),
          rsa_pkcs1_sha512(0x0601),

          /* ECDSA algorithms */
          ecdsa_secp256r1_sha256(0x0403),
          ecdsa_secp384r1_sha384(0x0503),
          ecdsa_secp521r1_sha512(0x0603),

          /* RSASSA-PSS algorithms with public key OID rsaEncryption */
          rsa_pss_rsae_sha256(0x0804),
          rsa_pss_rsae_sha384(0x0805),
          rsa_pss_rsae_sha512(0x0806),

          /* EdDSA algorithms */
          ed25519(0x0807),
          ed448(0x0808),

          /* RSASSA-PSS algorithms with public key OID RSASSA-PSS */
          rsa_pss_pss_sha256(0x0809),
          rsa_pss_pss_sha384(0x080a),
          rsa_pss_pss_sha512(0x080b),

          /* Legacy algorithms */
          rsa_pkcs1_sha1(0x0201),
          ecdsa_sha1(0x0203),

          /* Reserved Code Points */
          obsolete_RESERVED(0x0000..0x0200),
          dsa_sha1_RESERVED(0x0202),
          obsolete_RESERVED(0x0204..0x0400),
          dsa_sha256_RESERVED(0x0402),
          obsolete_RESERVED(0x0404..0x0500),
          dsa_sha384_RESERVED(0x0502),
          obsolete_RESERVED(0x0504..0x0600),
          dsa_sha512_RESERVED(0x0602),
          obsolete_RESERVED(0x0604..0x06FF),
          private_use(0xFE00..0xFFFF),
          (0xFFFF)
      } SignatureScheme;

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

B.3.1.4. Supported Groups Extension

```
      enum {
          unallocated_RESERVED(0x0000),

          /* Elliptic Curve Groups (ECDHE) */
          obsolete_RESERVED(0x0001..0x0016),
          secp256r1(0x0017), secp384r1(0x0018), secp521r1(0x0019),
          obsolete_RESERVED(0x001A..0x001C),
          x25519(0x001D), x448(0x001E),

          /* Finite Field Groups (DHE) */
          ffdhe2048(0x0100), ffdhe3072(0x0101), ffdhe4096(0x0102),
          ffdhe6144(0x0103), ffdhe8192(0x0104),

          /* Reserved Code Points */
          ffdhe_private_use(0x01FC..0x01FF),
          ecdhe_private_use(0xFE00..0xFEFF),
          obsolete_RESERVED(0xFF01..0xFF02),
          (0xFFFF)
      } NamedGroup;

      struct {
          NamedGroup named_group_list<2..2^16-1>;
      } NamedGroupList;
```

Values within "obsolete_RESERVED" ranges are used in previous versions of TLS and MUST NOT be offered or negotiated by TLS 1.3 implementations. The obsolete curves have various known/theoretical weaknesses or have had very little usage, in some cases only due to unintentional server configuration issues. They are no longer considered appropriate for general use and should be assumed to be potentially unsafe. The set of curves specified here is sufficient for interoperability with all currently deployed and properly configured TLS implementations.

B.3.2. Server Parameters Messages

```
      opaque DistinguishedName<1..2^16-1>;

      struct {
          DistinguishedName authorities<3..2^16-1>;
      } CertificateAuthoritiesExtension;

      struct {
          opaque certificate_extension_oid<1..2^8-1>;
          opaque certificate_extension_values<0..2^16-1>;
      } OIDFilter;

      struct {
          OIDFilter filters<0..2^16-1>;
      } OIDFilterExtension;

      struct {} PostHandshakeAuth;

      struct {
          Extension extensions<0..2^16-1>;
      } EncryptedExtensions;

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

B.3.3. Authentication Messages

```
      /* Managed by IANA */
      enum {
          X509(0),
          OpenPGP_RESERVED(1),
          RawPublicKey(2),
          (255)
      } CertificateType;

      struct {
          select (certificate_type) {
              case RawPublicKey:
                /* From RFC 7250 ASN.1_subjectPublicKeyInfo */
                opaque ASN1_subjectPublicKeyInfo<1..2^24-1>;

              case X509:
                opaque cert_data<1..2^24-1>;
          };
          Extension extensions<0..2^16-1>;
      } CertificateEntry;

      struct {
          opaque certificate_request_context<0..2^8-1>;
          CertificateEntry certificate_list<0..2^24-1>;
      } Certificate;

      struct {
          SignatureScheme algorithm;
          opaque signature<0..2^16-1>;
      } CertificateVerify;

      struct {
          opaque verify_data[Hash.length];
      } Finished;
```

B.3.4. Ticket Establishment

```
      struct {
          uint32 ticket_lifetime;
          uint32 ticket_age_add;
          opaque ticket_nonce<0..255>;
          opaque ticket<1..2^16-1>;
          Extension extensions<0..2^16-2>;
      } NewSessionTicket;
```

B.3.5. Updating Keys

```
   struct {} EndOfEarlyData;

   enum {
       update_not_requested(0), update_requested(1), (255)
   } KeyUpdateRequest;

   struct {
       KeyUpdateRequest request_update;
   } KeyUpdate;
```

B.4. Cipher Suites

A symmetric cipher suite defines the pair of the AEAD algorithm and hash algorithm to be used with HKDF. Cipher suite names follow the naming convention:

```
CipherSuite TLS_AEAD_HASH = VALUE;
```

Component	Contents
TLS	The string "TLS"
AEAD	The AEAD algorithm used for record protection
HASH	The hash algorithm used with HKDF
VALUE	The two byte ID assigned for this cipher suite

This specification defines the following cipher suites for use with TLS 1.3.

Description	Value
TLS_AES_128_GCM_SHA256	{0x13,0x01}
TLS_AES_256_GCM_SHA384	{0x13,0x02}
TLS_CHACHA20_POLY1305_SHA256	{0x13,0x03}
TLS_AES_128_CCM_SHA256	{0x13,0x04}
TLS_AES_128_CCM_8_SHA256	{0x13,0x05}

The corresponding AEAD algorithms AEAD_AES_128_GCM, AEAD_AES_256_GCM, and AEAD_AES_128_CCM are defined in [RFC5116]. AEAD_CHACHA20_POLY1305 is defined in [RFC7539]. AEAD_AES_128_CCM_8 is defined in [RFC6655]. The corresponding hash algorithms are defined in [SHS].

Although TLS 1.3 uses the same cipher suite space as previous versions of TLS, TLS 1.3 cipher suites are defined differently, only specifying the symmetric ciphers, and cannot be used for TLS 1.2. Similarly, TLS 1.2 and lower cipher suites cannot be used with TLS 1.3.

New cipher suite values are assigned by IANA as described in Section 11.

Appendix C. Implementation Notes

The TLS protocol cannot prevent many common security mistakes. This section provides several recommendations to assist implementors. [I-D.ietf-tls-tls13-vectors] provides test vectors for TLS 1.3 handshakes.

C.1. Random Number Generation and Seeding

TLS requires a cryptographically secure pseudorandom number generator (CSPRNG). In most cases, the operating system provides an appropriate facility such as /dev/urandom, which should be used absent other (performance) concerns. It is RECOMMENDED to use an existing CSPRNG implementation in preference to crafting a new one. Many adequate cryptographic libraries are already available under favorable license terms. Should those prove unsatisfactory, [RFC4086] provides guidance on the generation of random values.

TLS uses random values both in public protocol fields such as the public Random values in the ClientHello and ServerHello and to generate keying material. With a properly functioning CSPRNG, this does not present a security problem as it is not feasible to determine the CSPRNG state from its output. However, with a broken CSPRNG, it may be possible for an attacker to use the public output to determine the CSPRNG internal state and thereby predict the keying material, as documented in [CHECKOWAY]. Implementations can provide extra security against this form of attack by using separate CSPRNGs to generate public and private values.

C.2. Certificates and Authentication

Implementations are responsible for verifying the integrity of certificates and should generally support certificate revocation messages. Absent a specific indication from an application profile, Certificates should always be verified to ensure proper signing by a trusted Certificate Authority (CA). The selection and addition of trust anchors should be done very carefully. Users should be able to view information about the certificate and trust anchor. Applications SHOULD also enforce minimum and maximum key sizes. For example, certification paths containing keys or signatures weaker than 2048-bit RSA or 224-bit ECDSA are not appropriate for secure applications.

C.3. Implementation Pitfalls

Implementation experience has shown that certain parts of earlier TLS specifications are not easy to understand and have been a source of interoperability and security problems. Many of these areas have been clarified in this document but this appendix contains a short list of the most important things that require special attention from implementors.

TLS protocol issues:

- Do you correctly handle handshake messages that are fragmented to multiple TLS records (see Section 5.1)? Including corner cases like a ClientHello that is split to several small fragments? Do you fragment handshake messages that exceed the maximum fragment size? In particular, the Certificate and CertificateRequest handshake messages can be large enough to require fragmentation.

- Do you ignore the TLS record layer version number in all unencrypted TLS records? (see Appendix D)

- Have you ensured that all support for SSL, RC4, EXPORT ciphers, and MD5 (via the "signature_algorithms" extension) is completely removed from all possible configurations that support TLS 1.3 or later, and that attempts to use these obsolete capabilities fail correctly? (see Appendix D)

- Do you handle TLS extensions in ClientHello correctly, including unknown extensions?

- When the server has requested a client certificate, but no suitable certificate is available, do you correctly send an empty Certificate message, instead of omitting the whole message (see Section 4.4.2.3)?

- When processing the plaintext fragment produced by AEAD-Decrypt and scanning from the end for the ContentType, do you avoid scanning past the start of the cleartext in the event that the peer has sent a malformed plaintext of all-zeros?

- Do you properly ignore unrecognized cipher suites (Section 4.1.2), hello extensions (Section 4.2), named groups (Section 4.2.7), key shares (Section 4.2.8), supported versions (Section 4.2.1), and signature algorithms (Section 4.2.3) in the ClientHello?

- As a server, do you send a HelloRetryRequest to clients which support a compatible (EC)DHE group but do not predict it in the "key_share" extension? As a client, do you correctly handle a HelloRetryRequest from the server?

Cryptographic details:

- What countermeasures do you use to prevent timing attacks [TIMING]?

- When using Diffie-Hellman key exchange, do you correctly preserve leading zero bytes in the negotiated key (see Section 7.4.1)?

- Does your TLS client check that the Diffie-Hellman parameters sent by the server are acceptable, (see Section 4.2.8.1)?

- Do you use a strong and, most importantly, properly seeded random number generator (see Appendix C.1) when generating Diffie-Hellman private values, the ECDSA "k" parameter, and other security-critical values? It is RECOMMENDED that implementations implement "deterministic ECDSA" as specified in [RFC6979].

- Do you zero-pad Diffie-Hellman public key values to the group size (see Section 4.2.8.1)?

- Do you verify signatures after making them to protect against RSA-CRT key leaks? [FW15]

C.4. Client Tracking Prevention

Clients SHOULD NOT reuse a ticket for multiple connections. Reuse of a ticket allows passive observers to correlate different connections. Servers that issue tickets SHOULD offer at least as many tickets as the number of connections that a client might use; for example, a web browser using HTTP/1.1 [RFC7230] might open six connections to a server. Servers SHOULD issue new tickets with every connection. This ensures that clients are always able to use a new ticket when creating a new connection.

C.5. Unauthenticated Operation

Previous versions of TLS offered explicitly unauthenticated cipher suites based on anonymous Diffie-Hellman. These modes have been deprecated in TLS 1.3. However, it is still possible to negotiate parameters that do not provide verifiable server authentication by several methods, including:

- Raw public keys [RFC7250].

- Using a public key contained in a certificate but without validation of the certificate chain or any of its contents.

Either technique used alone is vulnerable to man-in-the-middle attacks and therefore unsafe for general use. However, it is also possible to bind such connections to an external authentication mechanism via out-of-band validation of the server's public key, trust on first use, or a mechanism such as channel bindings (though the channel bindings described in [RFC5929] are not defined for TLS 1.3). If no such mechanism is used, then the connection has no protection against active man-in-the-middle attack; applications MUST NOT use TLS in such a way absent explicit configuration or a specific application profile.

Appendix D. Backward Compatibility

The TLS protocol provides a built-in mechanism for version negotiation between endpoints potentially supporting different versions of TLS.

TLS 1.x and SSL 3.0 use compatible ClientHello messages. Servers can also handle clients trying to use future versions of TLS as long as the ClientHello format remains compatible and there is at least one protocol version supported by both the client and the server.

Prior versions of TLS used the record layer version number (TLSPlaintext.legacy_record_version and TLSCiphertext.legacy_record_version) for various purposes. As of TLS 1.3, this field is deprecated. The value of TLSPlaintext.legacy_record_version MUST be ignored by all implementations. The value of TLSCiphertext.legacy_record_version is included in the additional data for deprotection but MAY otherwise be ignored or MAY be validated to match the fixed constant value. Version negotiation is performed using only the handshake versions (ClientHello.legacy_version, ServerHello.legacy_version, as well as the ClientHello, HelloRetryRequest and ServerHello "supported_versions" extensions). In order to maximize interoperability with older endpoints, implementations that negotiate the use of TLS 1.0-1.2 SHOULD set the record layer version number to the negotiated version for the ServerHello and all records thereafter.

For maximum compatibility with previously non-standard behavior and misconfigured deployments, all implementations SHOULD support validation of certification paths based on the expectations in this document, even when handling prior TLS versions' handshakes. (see Section 4.4.2.2)

TLS 1.2 and prior supported an "Extended Master Secret" [RFC7627] extension which digested large parts of the handshake transcript into the master secret. Because TLS 1.3 always hashes in the transcript up to the server CertificateVerify, implementations which support both TLS 1.3 and earlier versions SHOULD indicate the use of the Extended Master Secret extension in their APIs whenever TLS 1.3 is used.

D.1. Negotiating with an older server

A TLS 1.3 client who wishes to negotiate with servers that do not support TLS 1.3 will send a normal TLS 1.3 ClientHello containing 0x0303 (TLS 1.2) in ClientHello.legacy_version but with the correct version(s) in the "supported_versions" extension. If the server does not support TLS 1.3 it will respond with a ServerHello containing an older version number. If the client agrees to use this version, the negotiation will proceed as appropriate for the negotiated protocol. A client using a ticket for resumption SHOULD initiate the connection using the version that was previously negotiated.

Note that 0-RTT data is not compatible with older servers and SHOULD NOT be sent absent knowledge that the server supports TLS 1.3. See Appendix D.3.

If the version chosen by the server is not supported by the client (or not acceptable), the client MUST abort the handshake with a "protocol_version" alert.

Some legacy server implementations are known to not implement the TLS specification properly and might abort connections upon encountering TLS extensions or versions which they are not aware of. Interoperability with buggy servers is a complex topic beyond the scope of this document. Multiple connection attempts may be required in order to negotiate a backwards compatible connection; however, this practice is vulnerable to downgrade attacks and is NOT RECOMMENDED.

D.2. Negotiating with an older client

A TLS server can also receive a ClientHello indicating a version number smaller than its highest supported version. If the "supported_versions" extension is present, the server MUST negotiate using that extension as described in Section 4.2.1. If the "supported_versions" extension is not present, the server MUST negotiate the minimum of ClientHello.legacy_version and TLS 1.2. For example, if the server supports TLS 1.0, 1.1, and 1.2, and legacy_version is TLS 1.0, the server will proceed with a TLS 1.0 ServerHello. If the "supported_versions" extension is absent and the server only supports versions greater than ClientHello.legacy_version, the server MUST abort the handshake with a "protocol_version" alert.

Note that earlier versions of TLS did not clearly specify the record layer version number value in all cases (TLSPlaintext.legacy_record_version). Servers will receive various TLS 1.x versions in this field, but its value MUST always be ignored.

D.3. 0-RTT backwards compatibility

0-RTT data is not compatible with older servers. An older server will respond to the ClientHello with an older ServerHello, but it will not correctly skip the 0-RTT data and will fail to complete the handshake. This can cause issues when a client attempts to use 0-RTT, particularly against multi-server deployments. For example, a deployment could deploy TLS 1.3 gradually with some servers implementing TLS 1.3 and some implementing TLS 1.2, or a TLS 1.3 deployment could be downgraded to TLS 1.2.

A client that attempts to send 0-RTT data MUST fail a connection if it receives a ServerHello with TLS 1.2 or older. A client that attempts to repair this error SHOULD NOT send a TLS 1.2 ClientHello, but instead send a TLS 1.3 ClientHello without 0-RTT data.

To avoid this error condition, multi-server deployments SHOULD ensure a uniform and stable deployment of TLS 1.3 without 0-RTT prior to enabling 0-RTT.

D.4. Middlebox Compatibility Mode

Field measurements [Ben17a], [Ben17b], [Res17a], [Res17b] have found that a significant number of middleboxes misbehave when a TLS client/server pair negotiates TLS 1.3. Implementations can increase the chance of making connections through those middleboxes by making the TLS 1.3 handshake look more like a TLS 1.2 handshake:

- The client always provides a non-empty session ID in the ClientHello, as described in the legacy_session_id section of Section 4.1.2.

- If not offering early data, the client sends a dummy change_cipher_spec record (see the third paragraph of Section 5.1) immediately before its second flight. This may either be before its second ClientHello or before its encrypted handshake flight. If offering early data, the record is placed immediately after the first ClientHello.

- The server sends a dummy change_cipher_spec record immediately after its first handshake message. This may either be after a ServerHello or a HelloRetryRequest.

When put together, these changes make the TLS 1.3 handshake resemble TLS 1.2 session resumption, which improves the chance of successfully connecting through middleboxes. This "compatibility mode" is partially negotiated: The client can opt to provide a session ID or not and the server has to echo it. Either side can send change_cipher_spec at any time during the handshake, as they must be ignored by the peer, but if the client sends a non-empty session ID, the server MUST send the change_cipher_spec as described in this section.

D.5. Backwards Compatibility Security Restrictions

Implementations negotiating use of older versions of TLS SHOULD prefer forward secret and AEAD cipher suites, when available.

The security of RC4 cipher suites is considered insufficient for the reasons cited in [RFC7465]. Implementations MUST NOT offer or negotiate RC4 cipher suites for any version of TLS for any reason.

Old versions of TLS permitted the use of very low strength ciphers. Ciphers with a strength less than 112 bits MUST NOT be offered or negotiated for any version of TLS for any reason.

The security of SSL 3.0 [SSL3] is considered insufficient for the reasons enumerated in [RFC7568], and it MUST NOT be negotiated for any reason.

The security of SSL 2.0 [SSL2] is considered insufficient for the reasons enumerated in [RFC6176], and it MUST NOT be negotiated for any reason.

Implementations MUST NOT send an SSL version 2.0 compatible CLIENT-HELLO. Implementations MUST NOT negotiate TLS 1.3 or later using an

SSL version 2.0 compatible CLIENT-HELLO. Implementations are NOT RECOMMENDED to accept an SSL version 2.0 compatible CLIENT-HELLO in order to negotiate older versions of TLS.

Implementations MUST NOT send a ClientHello.legacy_version or ServerHello.legacy_version set to 0x0300 or less. Any endpoint receiving a Hello message with ClientHello.legacy_version or ServerHello.legacy_version set to 0x0300 MUST abort the handshake with a "protocol_version" alert.

Implementations MUST NOT send any records with a version less than 0x0300. Implementations SHOULD NOT accept any records with a version less than 0x0300 (but may inadvertently do so if the record version number is ignored completely).

Implementations MUST NOT use the Truncated HMAC extension, defined in Section 7 of [RFC6066], as it is not applicable to AEAD algorithms and has been shown to be insecure in some scenarios.

Appendix E. Overview of Security Properties

A complete security analysis of TLS is outside the scope of this document. In this section, we provide an informal description the desired properties as well as references to more detailed work in the research literature which provides more formal definitions.

We cover properties of the handshake separately from those of the record layer.

E.1. Handshake

The TLS handshake is an Authenticated Key Exchange (AKE) protocol which is intended to provide both one-way authenticated (server-only) and mutually authenticated (client and server) functionality. At the completion of the handshake, each side outputs its view of the following values:

- A set of "session keys" (the various secrets derived from the master secret) from which can be derived a set of working keys.

- A set of cryptographic parameters (algorithms, etc.)

- The identities of the communicating parties.

We assume the attacker to be an active network attacker, which means it has complete control over the network used to communicate between the parties [RFC3552]. Even under these conditions, the handshake should provide the properties listed below. Note that these properties are not necessarily independent, but reflect the protocol consumers' needs.

Establishing the same session keys. The handshake needs to output the same set of session keys on both sides of the handshake, provided that it completes successfully on each endpoint (See [CK01]; defn 1, part 1).

Secrecy of the session keys. The shared session keys should be known only to the communicating parties and not to the attacker (See [CK01]; defn 1, part 2). Note that in a unilaterally authenticated connection, the attacker can establish its own session keys with the server, but those session keys are distinct from those established by the client.

Peer Authentication. The client's view of the peer identity should reflect the server's identity. If the client is authenticated, the server's view of the peer identity should match the client's identity.

Uniqueness of the session keys: Any two distinct handshakes should produce distinct, unrelated session keys. Individual session keys produced by a handshake should also be distinct and independent.

Downgrade protection. The cryptographic parameters should be the same on both sides and should be the same as if the peers had been communicating in the absence of an attack (See [BBFKZG16]; defns 8 and 9}).

Forward secret with respect to long-term keys If the long-term keying material (in this case the signature keys in certificate-based authentication modes or the external/resumption PSK in PSK with (EC)DHE modes) is compromised after the handshake is complete, this does not compromise the security of the session key (See [DOW92]), as long as the session key itself has been erased. The forward secrecy property is not satisfied when PSK is used in the "psk_ke" PskKeyExchangeMode.

Key Compromise Impersonation (KCI) resistance In a mutually-authenticated connection with certificates, compromising the long-term secret of one actor should not break that actor's authentication of their peer in the given connection (see [HGFS15]). For example, if a client's signature key is compromised, it should not be possible to impersonate arbitrary servers to that client in subsequent handshakes.

Protection of endpoint identities. The server's identity (certificate) should be protected against passive attackers. The client's identity should be protected against both passive and active attackers.

Informally, the signature-based modes of TLS 1.3 provide for the establishment of a unique, secret, shared key established by an (EC)DHE key exchange and authenticated by the server's signature over the handshake transcript, as well as tied to the server's identity by a MAC. If the client is authenticated by a certificate, it also signs over the handshake transcript and provides a MAC tied to both identities. [SIGMA] describes the design and analysis of this type of key exchange protocol. If fresh (EC)DHE keys are used for each connection, then the output keys are forward secret.

The external PSK and resumption PSK bootstrap from a long-term shared secret into a unique per-connection set of short-term session keys. This secret may have been established in a previous handshake. If PSK with (EC)DHE key establishment is used, these session keys will also be forward secret. The resumption PSK has been designed so that the resumption master secret computed by connection N and needed to form connection N+1 is separate from the traffic keys used by connection N, thus providing forward secrecy between the connections. In addition, if multiple tickets are established on the same connection, they are associated with different keys, so compromise of the PSK associated with one ticket does not lead to the compromise of connections established with PSKs associated with other tickets. This property is most interesting if tickets are stored in a database (and so can be deleted) rather than if they are self-encrypted.

The PSK binder value forms a binding between a PSK and the current handshake, as well as between the session where the PSK was established and the current session. This binding transitively includes the original handshake transcript, because that transcript is digested into the values which produce the Resumption Master Secret. This requires that both the KDF used to produce the resumption master secret and the MAC used to compute the binder be collision resistant. See Appendix E.1.1 for more on this. Note: The binder does not cover the binder values from other PSKs, though they are included in the Finished MAC.

Note: TLS does not currently permit the server to send a certificate_request message in non-certificate-based handshakes (e.g., PSK). If this restriction were to be relaxed in future, the client's signature would not cover the server's certificate directly. However, if the PSK was established through a NewSessionTicket, the client's signature would transitively cover the server's certificate through the PSK binder. [PSK-FINISHED] describes a concrete attack on constructions that do not bind to the server's certificate (see also [Kraw16]). It is unsafe to use certificate-based client authentication when the client might potentially share the same PSK/key-id pair with two different endpoints. Implementations MUST NOT combine external PSKs with certificate-based authentication of either the client or the server unless negotiated by some extension.

If an exporter is used, then it produces values which are unique and secret (because they are generated from a unique session key). Exporters computed with different labels and contexts are computationally independent, so it is not feasible to compute one from another or the session secret from the exported value. Note: exporters can produce arbitrary-length values. If exporters are to be used as channel bindings, the exported value MUST be large enough to provide collision resistance. The exporters provided in TLS 1.3 are derived from the same handshake contexts as the early traffic keys and the application traffic keys respectively, and thus have similar security properties. Note that they do not include the client's certificate; future applications which wish to bind to the client's certificate may need to define a new exporter that includes the full handshake transcript.

For all handshake modes, the Finished MAC (and where present, the signature), prevents downgrade attacks. In addition, the use of certain bytes in the random nonces as described in Section 4.1.3 allows the detection of downgrade to previous TLS versions. See [BBFKZG16] for more detail on TLS 1.3 and downgrade.

As soon as the client and the server have exchanged enough information to establish shared keys, the remainder of the handshake is encrypted, thus providing protection against passive attackers, even if the computed shared key is not authenticated. Because the server authenticates before the client, the client can ensure that if it authenticates to the server, it only reveals its identity to an authenticated server. Note that implementations must use the provided record padding mechanism during the handshake to avoid leaking information about the identities due to length. The client's proposed PSK identities are not encrypted, nor is the one that the server selects.

E.1.1. Key Derivation and HKDF

Key derivation in TLS 1.3 uses the HKDF function defined in [RFC5869] and its two components, HKDF-Extract and HKDF-Expand. The full rationale for the HKDF construction can be found in [Kraw10] and the rationale for the way it is used in TLS 1.3 in [KW16]. Throughout this document, each application of HKDF-Extract is followed by one or more invocations of HKDF-Expand. This ordering should always be followed (including in future revisions of this document), in particular, one SHOULD NOT use an output of HKDF-Extract as an input to another application of HKDF-Extract without an HKDF-Expand in between. Consecutive applications of HKDF-Expand are allowed as long as these are differentiated via the key and/or the labels.

Note that HKDF-Expand implements a pseudorandom function (PRF) with both inputs and outputs of variable length. In some of the uses of HKDF in this document (e.g., for generating exporters and the resumption_master_secret), it is necessary that the application of HKDF-Expand be collision-resistant, namely, it should be infeasible to find two different inputs to HKDF-Expand that output the same value. This requires the underlying hash function to be collision resistant and the output length from HKDF-Expand to be of size at least 256 bits (or as much as needed for the hash function to prevent finding collisions).

E.1.2. Client Authentication

A client that has sent authentication data to a server, either during the handshake or in post-handshake authentication, cannot be sure if the server afterwards considers the client to be authenticated or not. If the client needs to determine if the server considers the connection to be unilaterally or mutually authenticated, this has to be provisioned by the application layer. See [CHHSV17] for details. In addition, the analysis of post-handshake authentication from [Kraw16] shows that the client identified by the certificate sent in the post-handshake phase possesses the traffic key. This party is therefore the client that participated in the original handshake or one to whom the original client delegated the traffic key (assuming that the traffic key has not been compromised).

E.1.3. 0-RTT

The 0-RTT mode of operation generally provides similar security properties as 1-RTT data, with the two exceptions that the 0-RTT encryption keys do not provide full forward secrecy and that the server is not able to guarantee uniqueness of the handshake (non-replayability) without keeping potentially undue amounts of state. See Section 8 for mechanisms to limit the exposure to replay.

E.1.4. Exporter Independence

The exporter_master_secret and early_exporter_master_secret are derived to be independent of the traffic keys and therefore do not represent a threat to the security of traffic encrypted with those keys. However, because these secrets can be used to compute any exporter value, they SHOULD be erased as soon as possible. If the total set of exporter labels is known, then implementations SHOULD pre-compute the inner Derive-Secret stage of the exporter computation for all those labels, then erase the [early_]exporter_master_secret, followed by each inner values as soon as it is known that it will not be needed again.

E.1.5. Post-Compromise Security

TLS does not provide security for handshakes which take place after the peer's long-term secret (signature key or external PSK) is compromised. It therefore does not provide post-compromise security [CCG16], sometimes also referred to as backwards or future secrecy. This is in contrast to KCI resistance, which describes the security guarantees that a party has after its own long-term secret has been compromised.

E.1.6. External References

The reader should refer to the following references for analysis of the TLS handshake: [DFGS15] [CHSV16] [DFGS16] [KW16] [Kraw16] [FGSW16] [LXZFH16] [FG17] [BBK17].

E.2. Record Layer

The record layer depends on the handshake producing strong traffic secrets which can be used to derive bidirectional encryption keys and nonces. Assuming that is true, and the keys are used for no more data than indicated in Section 5.5 then the record layer should provide the following guarantees:

Confidentiality. An attacker should not be able to determine the plaintext contents of a given record.

Integrity. An attacker should not be able to craft a new record which is different from an existing record which will be accepted by the receiver.

Order protection/non-replayability An attacker should not be able to cause the receiver to accept a record which it has already accepted or cause the receiver to accept record N+1 without having first processed record N.

Length concealment. Given a record with a given external length, the attacker should not be able to determine the amount of the record that is content versus padding.

Forward secrecy after key change. If the traffic key update mechanism described in Section 4.6.3 has been used and the previous generation key is deleted, an attacker who compromises the endpoint should not be able to decrypt traffic encrypted with the old key.

Informally, TLS 1.3 provides these properties by AEAD-protecting the plaintext with a strong key. AEAD encryption [RFC5116] provides confidentiality and integrity for the data. Non-replayability is provided by using a separate nonce for each record, with the nonce being derived from the record sequence number (Section 5.3), with the sequence number being maintained independently at both sides thus records which are delivered out of order result in AEAD deprotection failures. In order to prevent mass cryptanalysis when the same plaintext is repeatedly encrypted by different users under the same key (as is commonly the case for HTTP), the nonce is formed by mixing the sequence number with a secret per-connection initialization vector derived along with the traffic keys. See [BT16] for analysis of this construction.

The re-keying technique in TLS 1.3 (see Section 7.2) follows the construction of the serial generator in [REKEY], which shows that re-keying can allow keys to be used for a larger number of encryptions than without re-keying. This relies on the security of the HKDF-Expand-Label function as a pseudorandom function (PRF). In addition, as long as this function is truly one way, it is not possible to compute traffic keys from prior to a key change (forward secrecy).

TLS does not provide security for data which is communicated on a connection after a traffic secret of that connection is compromised. That is, TLS does not provide post-compromise security/future secrecy/backward secrecy with respect to the traffic secret. Indeed, an attacker who learns a traffic secret can compute all future traffic secrets on that connection. Systems which want such guarantees need to do a fresh handshake and establish a new connection with an (EC)DHE exchange.

E.2.1. External References

The reader should refer to the following references for analysis of the TLS record layer: [BMMT15] [BT16] [BDFKPPRSZZ16] [BBK17] [Anon18].

E.3. Traffic Analysis

TLS is susceptible to a variety of traffic analysis attacks based on observing the length and timing of encrypted packets [CLINIC] [HCJ16]. This is particularly easy when there is a small set of possible messages to be distinguished, such as for a video server hosting a fixed corpus of content, but still provides usable information even in more complicated scenarios.

TLS does not provide any specific defenses against this form of attack but does include a padding mechanism for use by applications: The plaintext protected by the AEAD function consists of content plus variable-length padding, which allows the application to produce arbitrary length encrypted records as well as padding-only cover traffic to conceal the difference between periods of transmission and periods of silence. Because the padding is encrypted alongside the actual content, an attacker cannot directly determine the length of the padding, but may be able to measure it indirectly by the use of timing channels exposed during record processing (i.e., seeing how long it takes to process a record or trickling in records to see which ones elicit a response from the server). In general, it is not known how to remove all of these channels because even a constant time padding removal function will likely feed the content into data-dependent functions. At minimum, a fully constant time server or client would require close cooperation with the application layer protocol implementation, including making that higher level protocol constant time.

Note: Robust traffic analysis defences will likely lead to inferior performance due to delay in transmitting packets and increased traffic volume.

E.4. Side Channel Attacks

In general, TLS does not have specific defenses against side-channel attacks (i.e., those which attack the communications via secondary channels such as timing) leaving those to the implementation of the relevant cryptographic primitives. However, certain features of TLS are designed to make it easier to write side-channel resistant code:

- Unlike previous versions of TLS which used a composite MAC-then-encrypt structure, TLS 1.3 only uses AEAD algorithms, allowing implementations to use self-contained constant-time implementations of those primitives.

- TLS uses a uniform "bad_record_mac" alert for all decryption errors, which is intended to prevent an attacker from gaining piecewise insight into portions of the message. Additional resistance is provided by terminating the connection on such errors; a new connection will have different cryptographic material, preventing attacks against the cryptographic primitives that require multiple trials.

Information leakage through side channels can occur at layers above TLS, in application protocols and the applications that use them. Resistance to side-channel attacks depends on applications and application protocols separately ensuring that confidential information is not inadvertently leaked.

E.5. Replay Attacks on 0-RTT

Replayable 0-RTT data presents a number of security threats to TLS-using applications, unless those applications are specifically engineered to be safe under replay (minimally, this means idempotent, but in many cases may also require other stronger conditions, such as constant-time response). Potential attacks include:

- Duplication of actions which cause side effects (e.g., purchasing an item or transferring money) to be duplicated, thus harming the site or the user.

- Attackers can store and replay 0-RTT messages in order to re-order them with respect to other messages (e.g., moving a delete to after a create).

- Exploiting cache timing behavior to discover the content of 0-RTT messages by replaying a 0-RTT message to a different cache node and then using a separate connection to measure request latency, to see if the two requests address the same resource.

If data can be replayed a large number of times, additional attacks become possible, such as making repeated measurements of the speed of cryptographic operations. In addition, they may be able to overload rate-limiting systems. For further description of these attacks, see [Mac17].

Ultimately, servers have the responsibility to protect themselves against attacks employing 0-RTT data replication. The mechanisms described in Section 8 are intended to prevent replay at the TLS layer but do not provide complete protection against receiving multiple copies of client data. TLS 1.3 falls back to the 1-RTT handshake when the server does not have any information about the client, e.g., because it is in a different cluster which does not share state or because the ticket has been deleted as described in Section 8.1. If the application layer protocol retransmits data in this setting, then it is possible for an attacker to induce message duplication by sending the ClientHello to both the original cluster (which processes the data immediately) and another cluster which will fall back to 1-RTT and process the data upon application layer replay. The scale of this attack is limited by the client’s willingness to retry transactions and therefore only allows a limited amount of duplication, with each copy appearing as a new connection at the server.

If implemented correctly, the mechanisms described in Section 8.1 and Section 8.2 prevent a replayed ClientHello and its associated 0-RTT data from being accepted multiple times by any cluster with consistent state; for servers which limit the use of 0-RTT to one cluster for a single ticket, then a given ClientHello and its associated 0-RTT data will only be accepted once. However, if state is not completely consistent, then an attacker might be able to have multiple copies of the data be accepted during the replication window. Because clients do not know the exact details of server behavior, they MUST NOT send messages in early data which are not safe to have replayed and which they would not be willing to retry across multiple 1-RTT connections.

Application protocols MUST NOT use 0-RTT data without a profile that defines its use. That profile needs to identify which messages or interactions are safe to use with 0-RTT and how to handle the situation when the server rejects 0-RTT and falls back to 1-RTT.

In addition, to avoid accidental misuse, TLS implementations MUST NOT enable 0-RTT (either sending or accepting) unless specifically requested by the application and MUST NOT automatically resend 0-RTT data if it is rejected by the server unless instructed by the application. Server-side applications may wish to implement special processing for 0-RTT data for some kinds of application traffic (e.g., abort the connection, request that data be resent at the application layer, or delay processing until the handshake completes). In order to allow applications to implement this kind of processing, TLS implementations MUST provide a way for the application to determine if the handshake has completed.

E.5.1. Replay and Exporters

Replays of the ClientHello produce the same early exporter, thus requiring additional care by applications which use these exporters. In particular, if these exporters are used as an authentication channel binding (e.g., by signing the output of the exporter) an attacker who compromises the PSK can transplant authenticators between connections without compromising the authentication key.

In addition, the early exporter SHOULD NOT be used to generate server-to-client encryption keys because that would entail the reuse of those keys. This parallels the use of the early application traffic keys only in the client-to-server direction.

E.6. PSK Identity Exposure

Because implementations respond to an invalid PSK binder by aborting the handshake, it may be possible for an attacker to verify whether a given PSK identity is valid. Specifically, if a server accepts both external PSK and certificate-based handshakes, a valid PSK identity will result in a failed handshake, whereas an invalid identity will just be skipped and result in a successful certificate handshake. Servers which solely support PSK handshakes may be able to resist this form of attack by treating the cases where there is no valid PSK identity and where there is an identity but it has an invalid binder identically.

E.7. Attacks on Static RSA

Although TLS 1.3 does not use RSA key transport and so is not directly susceptible to Bleichenbacher-type attacks, if TLS 1.3 servers also support static RSA in the context of previous versions of TLS, then it may be possible to impersonate the server for TLS 1.3 connections [JSS15]. TLS 1.3 implementations can prevent this attack by disabling support for static RSA across all versions of TLS. In principle, implementations might also be able to separate certificates with different keyUsage bits for static RSA decryption and RSA signature, but this technique relies on clients refusing to accept signatures using keys in certificates that do not have the digitalSignature bit set, and many clients do not enforce this restriction.

Appendix F. 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://www.ietf.org/mailman/listinfo/tls

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

Appendix G. Contributors

- Martin Abadi
  University of California, Santa Cruz
  abadi@cs.ucsc.edu

- Christopher Allen (co-editor of TLS 1.0)
  Alacrity Ventures
  ChristopherA@AlacrityManagement.com

- Richard Barnes
  Cisco
  rlb@ipv.sx

- Steven M. Bellovin
  Columbia University
  smb@cs.columbia.edu

- David Benjamin
  Google
  davidben@google.com

- Benjamin Beurdouche
  INRIA & Microsoft Research
  benjamin.beurdouche@ens.fr

- Karthikeyan Bhargavan (co-author of [RFC7627])
  INRIA
  karthikeyan.bhargavan@inria.fr

- Simon Blake-Wilson (co-author of [RFC4492])
  BCI
  sblakewilson@bcisse.com

- Nelson Bolyard (co-author of [RFC4492])
  Sun Microsystems, Inc.
  nelson@bolyard.com

- Ran Canetti
  IBM
  canetti@watson.ibm.com

- Matt Caswell
  OpenSSL
  matt@openssl.org

- Stephen Checkoway
  University of Illinois at Chicago
  sfc@uic.edu

- Pete Chown
  Skygate Technology Ltd
  pc@skygate.co.uk

- Katriel Cohn-Gordon
  University of Oxford
  me@katriel.co.uk

- Cas Cremers
  University of Oxford
  cas.cremers@cs.ox.ac.uk

- Antoine Delignat-Lavaud (co-author of [RFC7627])
  INRIA
  antdl@microsoft.com

- Tim Dierks (co-editor of TLS 1.0, 1.1, and 1.2)
  Independent
  tim@dierks.org

- Roelof DuToit
  Symantec Corporation
  roelof_dutoit@symantec.com

- Taher Elgamal
  Securify
  taher@securify.com

- Pasi Eronen
  Nokia
  pasi.eronen@nokia.com

- Cedric Fournet
  Microsoft
  fournet@microsoft.com

- Anil Gangolli
  anil@busybuddha.org

- David M. Garrett
  dave@nulldereference.com

- Illya Gerasymchuk
  Independent
  illya@iluxonchik.me

- Alessandro Ghedini
  Cloudflare Inc.
  alessandro@cloudflare.com

- Daniel Kahn Gillmor
  ACLU
  dkg@fifthhorseman.net

- Matthew Green
  Johns Hopkins University

mgreen@cs.jhu.edu

- Jens Guballa
  ETAS
  jens.guballa@etas.com

- Felix Guenther
  TU Darmstadt
  mail@felixguenther.info

- Vipul Gupta (co-author of [RFC4492])
  Sun Microsystems Laboratories
  vipul.gupta@sun.com

- Chris Hawk (co-author of [RFC4492])
  Corriente Networks LLC
  chris@corriente.net

- Kipp Hickman

- Alfred Hoenes

- David Hopwood
  Independent Consultant
  david.hopwood@blueyonder.co.uk

- Marko Horvat
  MPI-SWS
  mhorvat@mpi-sws.org

- Jonathan Hoyland
  Royal Holloway, University of London jonathan.hoyland@gmail.com

- Subodh Iyengar
  Facebook
  subodh@fb.com

- Benjamin Kaduk
  Akamai
  kaduk@mit.edu

- Hubert Kario
  Red Hat Inc.
  hkario@redhat.com

- Phil Karlton (co-author of SSL 3.0)

- Leon Klingele

Independent
mail@leonklingele.de

- Paul Kocher (co-author of SSL 3.0)
  Cryptography Research
  paul@cryptography.com

- Hugo Krawczyk
  IBM
  hugokraw@us.ibm.com

- Adam Langley (co-author of [RFC7627])
  Google
  agl@google.com

- Olivier Levillain
  ANSSI
  olivier.levillain@ssi.gouv.fr

- Xiaoyin Liu
  University of North Carolina at Chapel Hill
  xiaoyin.l@outlook.com

- Ilari Liusvaara
  Independent
  ilariliusvaara@welho.com

- Atul Luykx
  K.U. Leuven
  atul.luykx@kuleuven.be

- Colm MacCarthaigh
  Amazon Web Services
  colm@allcosts.net

- Carl Mehner
  USAA
  carl.mehner@usaa.com

- Jan Mikkelsen
  Transactionware
  janm@transactionware.com

- Bodo Moeller (co-author of [RFC4492])
  Google
  bodo@acm.org

- Kyle Nekritz

Facebook
knekritz@fb.com

- Erik Nygren
  Akamai Technologies
  erik+ietf@nygren.org

- Magnus Nystrom
  Microsoft
  mnystrom@microsoft.com

- Kazuho Oku
  DeNA Co., Ltd.
  kazuhooku@gmail.com

- Kenny Paterson
  Royal Holloway, University of London
  kenny.paterson@rhul.ac.uk

- Alfredo Pironti (co-author of [RFC7627])
  INRIA
  alfredo.pironti@inria.fr

- Andrei Popov
  Microsoft
  andrei.popov@microsoft.com

- Marsh Ray (co-author of [RFC7627])
  Microsoft
  maray@microsoft.com

- Robert Relyea
  Netscape Communications
  relyea@netscape.com

- Kyle Rose
  Akamai Technologies
  krose@krose.org

- Jim Roskind
  Amazon
  jroskind@amazon.com

- Michael Sabin

- Joe Salowey
  Tableau Software
  joe@salowey.net

- Rich Salz
  Akamai
  rsalz@akamai.com

- David Schinazi
  Apple Inc.
  dschinazi@apple.com

- Sam Scott
  Royal Holloway, University of London
  me@samjs.co.uk

- Dan Simon
  Microsoft, Inc.
  dansimon@microsoft.com

- Brian Smith
  Independent
  brian@briansmith.org

- Brian Sniffen
  Akamai Technologies
  ietf@bts.evenmere.org

- Nick Sullivan
  Cloudflare Inc.
  nick@cloudflare.com

- Bjoern Tackmann
  University of California, San Diego
  btackmann@eng.ucsd.edu

- Tim Taubert
  Mozilla
  ttaubert@mozilla.com

- Martin Thomson
  Mozilla
  mt@mozilla.com

- Sean Turner
  sn3rd
  sean@sn3rd.com

- Steven Valdez
  Google
  svaldez@google.com

- Filippo Valsorda
  Cloudflare Inc.
  filippo@cloudflare.com

- Thyla van der Merwe
  Royal Holloway, University of London
  tjvdmerwe@gmail.com

- Victor Vasiliev
  Google
  vasilvv@google.com

- Tom Weinstein

- Hoeteck Wee
  Ecole Normale Superieure, Paris
  hoeteck@alum.mit.edu

- David Wong
  NCC Group
  david.wong@nccgroup.trust

- Christopher A. Wood
  Apple Inc.
  cawood@apple.com

- Tim Wright
  Vodafone
  timothy.wright@vodafone.com

- Peter Wu
  Independent
  peter@lekensteyn.nl

- Kazu Yamamoto
  Internet Initiative Japan Inc.
  kazu@iij.ad.jp

Author's Address

Eric Rescorla
RTFM, Inc.

EMail: ekr@rtfm.com

TLS Working Group
Internet-Draft
Intended status: Standards Track
Expires: September 6, 2018

E. Rescorla
Mozilla
N. Sullivan
Cloudflare

March 05, 2018

Semi-Static Diffie-Hellman Key Establishment for TLS 1.3
draft-rescorla-tls13-semistatic-dh-00

Abstract

TLS 1.3 [I-D.ietf-tls-tls13] specifies a signed Diffie-Hellman exchange modelled after SIGMA [SIGMA]. This design is suitable for endpoints whose certified credential is a signing key, which is the common situation for current TLS servers. This document describes a mode of TLS 1.3 in which one or both endpoints have a certified DH key which is used to authenticate the exchange.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

DISCLAIMER: This is a work-in-progress draft and is curenty totally handwavy, so it has not yet seen significant security analysis. It should not be used as a basis for building production systems.

TLS 1.3 [I-D.ietf-tls-tls13] specifies a signed Diffie-Hellman exchange modelled after SIGMA [SIGMA]. This design is suitable for endpoints whose certified credential is a signing key, which is the common situation for current TLS servers, which is why it was selected for TLS 1.3.

However, it is also possible - although currently rare - for endpoints to have a credential which is an (EC)DH key. This can happen in one of two ways:

- They may be issued a certificate with an (EC)DH key, as specified for instance in [I-D.ietf-curdle-pkix]

- They may have a signing key which they use to generate a delegated credential [I-D.ietf-tls-subcerts] containing an (EC)DH key.

In these situations, a signed DH exchange is not appropriate, and instead a design in which the server authenticates via its long-term (EC)DH key is suitable. This document describes such a design modelled on that described in OPTLS [KW16].

This design has a number of potential advantages over the signed exchange in TLS 1.3, specifically:

- If the end-entity certificate contains an (EC)DH key, TLS can operate with a single asymmetric primitive (Diffie-Hellman). The PKI component will still need signatures, but the TLS stack need not have one. Note that this advantage is somewhat limited if the (EC)DH key is in a delegated credential, but that allows for a clean transition to (EC)DH certificates.

- It is more resistant to random number generation failures on the server because the attacker needs to have both the server's long-term (EC)DH key and the ephemeral (EC)DH key in order to compute the traffic secrets. [Note: [I-D.cremers-cfrg-randomness-improvements] describes a technique for accomplishing this with a signed exchange.]

- If the server has a comparatively slow signing cert (e.g., P-256) it can amortize that signature over a large number of connections by creating a delegated credential with an (EC)DH key from a faster group (e.g., X25519).

- Because there is no signature, the server has deniability for the existence of the communication. Note that it could always have denied the contents of the communication.

This exchange is not generally faster than a signed exchange if comparable groups are used. In fact, if delegated credentials are used, it may be slower on the client as it has to validate the delegated credential, though this operation is cacheable.

2. Protocol Overview

The overall protocol flow remains the same as that in ordinary TLS 1.3, as shown below:

```
       Client                                               Server

Key  ^ ClientHello
Exch | + key_share*
     | + signature_algorithms*
     | + psk_key_exchange_modes*
     v + pre_shared_key*         -------->
                                                       ServerHello  ^ Key
                                                      + key_share*  | Exch
                                                 + pre_shared_key*  v
                                             {EncryptedExtensions}  ^  Server
                                             {CertificateRequest*}  v  Params
                                                    {Certificate*}  ^
                                              {CertificateVerify*}  | Auth
                                                        {Finished}  v
                                 <--------     [Application Data*]
     ^ {Certificate*}
Auth | {CertificateVerify*}
     v {Finished}                -------->
       [Application Data]        <------->      [Application Data]
```

As usual, the client and server each supply an (EC)DH share in their "key_share" extensions. However, in addition, the server supplies a static (EC)DH share in its Certificate message, either directly in its end-entity certificate or in a delegated credential. The client and server then perform two (EC)DH exchanges:

- Between the client and server "key_share" values to form an ephemeral secret (ES). This is the same value as is computed in TLS 1.3 currently.

- Between the client's "key_share" and the server's static share, to form a static secret (SS).

Note that this means that the server's static secret MUST be in the same group as selected group for the ephemeral (EC)DH exchange.

The handshake then proceeds as usual, except that:

- Instead of containing a signature, the CertificateVerify contains a MAC of the handshake transcript, computed based on SS.

- SS is mixed into the key schedule at the last HKDF-Extract stage (where currently a 0 is used as the IKM input).

3. Negotiation

In order to negotiate this mode, we treat the (EC)DH MAC as if it were a signature and negotiate it with a set of new signature scheme values:

```
enum {
  sig_p256(0x0901),
  sig_p384(0x0902),
  sig_p521(0x0903),
  sig_x52219(0x0904),
  sig_x448(0x0905),
} SignatureScheme;
```

When present in the "signature_algorithms" extension or CertificateVerify.signature_scheme, these values indicate DH MAC with the specified key exchange mode. These values MUST NOT appear in "signature_algorithms_cert".

Before sending and upon receipt, endpoints MUST ensure that the signature scheme is consistent with the ephemeral (EC)DH group in use.

4. Certificate Format

Like signing keys, static DH keys are carried in the Certificate message, either directly in the EE certificate, or in a delegated credential. In either case, the OID for the SubjectPublicKeyInfo MUST be appropriate for use with (EC)DH key establishment. If in a certificate, the key usage and EKU MUST also be set appropriately See [I-D.ietf-curdle-pkix] and [[TBD: P-256, etc.]] for specific details about these formats.

5. Cryptographic Details

5.1. Certificate Verify Computation

Instead of a signature, the server proves knowledge of the private key associated with its static share by computing a MAC over the handshake transcript using SS. The transcript thus far includes all messages up to and including Certificate, i.e.:

Transcript-Hash(Handshake Context, Certificate)

The MAC key - SS-Base-Key - is derived from SS as follows:

```
SS-Base-Key = HKDF-Extract(0, SS)
```

The MAC is then computed using the Finished computation described in [I-D.ietf-tls-tls13] Section 4.4, with SS-Base-Key as the Base Key value. Receivers MUST validate the MAC and terminate the handshake with a "decrypt_error" alert upon failure.

Note that this means that the server sends two MAC computations in the handshake, one in CertificateVerify using SS and the other in Finished using the Master Secret. These MACs serve different purposes: the first authenticates the handshake and the second proves possession of the ephemeral secret. [[OPEN ISSUE: Verify that this is OK because neither MAC is computed with the mixed key. At least one version of OPTLS was somewhat like that, however.]]

5.2. Key Schedule

The final HKDF-Extract stage of the TLS 1.3 key schedule has an HKDF-Extract with the IKM of 0. When static key exchange is negotiated, that 0 is replaced with SS, as shown below.

```
...
           Derive-Secret(., "derived", "")
                 |
                 v
     SS -> HKDF-Extract = Master Secret
                 |
                 +-----> Derive-Secret(., "c ap traffic",
                 |                     ClientHello...server Finished)
                 |                     = client_application_traffic_secret_0
                 |
...
```

6. Client Authentication

[[OPEN ISSUE]] In principle, we can do client authentication the same way, with the client's DH key in Certificate and a MAC in CertificateVerity. However, it's less good because the client's static key doesn't get mixed in at all. Also, client DH keys seem even further off.

7. 0-RTT

[[OPEN ISSUE]] It seems like one ought to be able to publish the server's static key and use it for 0-RTT, but actually we don't know how to do the publication piece, so I think we should leave this out for now.

8. Security Considerations

[[OPEN ISSUE: This is a -00, so the security considerations are kind of sketchy.]]

- This is intended to have roughly equivalent security properties to current TLS 1.3, except for the points raised in the introduction.

- There are open questions about how much key mixing we want to do, especially with respect to client authentication.

- I'm not sure I like the double extract of SS. I've looked it over and the SS-Base-Key and the HKDF-Extract to make the MS should be independent, but I'd like to give it another look-over to see if there is a cleaner way to do it.

9. IANA Considerations

IANA [SHOULD add/has added] the new code points specified in Section 3 to the TLS 1.3 signature scheme registry, with a "recommended" value of TBD.

10. References

10.1. Normative References

[I-D.ietf-curdle-pkix]
Josefsson, S. and J. Schaad, "Algorithm Identifiers for Ed25519, Ed448, X25519 and X448 for use in the Internet X.509 Public Key Infrastructure", draft-ietf-curdle-pkix-07 (work in progress), November 2017.

[I-D.ietf-tls-subcerts]
Barnes, R., Iyengar, S., Sullivan, N., and E. Rescorla, "Delegated Credentials for TLS", draft-ietf-tls-subcerts-00 (work in progress), October 2017.

[I-D.ietf-tls-tls13]
Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", draft-ietf-tls-tls13-26 (work in progress), March 2018.

[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>.

10.2. Informative References

[I-D.cremers-cfrg-randomness-improvements]
Cremers, C., Garratt, L., Smyshlyaev, S., Sullivan, N., and C. Wood, "Randomness Improvements for Security Protocols", draft-cremers-cfrg-randomness-improvements-00 (work in progress), March 2018.

[KW16] Krawczyk, H. and H. Wee, "The OPTLS Protocol and TLS 1.3", Proceedings of Euro S"P 2016 , 2016, <https://eprint.iacr.org/2015/978>.

[SIGMA] Krawczyk, H., "SIGMA: the 'SIGn-and-MAc' approach to authenticated Diffie-Hellman and its use in the IKE protocols", Proceedings of CRYPTO 2003 , 2003.

Authors' Addresses

Eric Rescorla
Mozilla

Email: ekr@rtfm.com

Nick Sullivan
Cloudflare

Email: nick@cloudflare.com

```
Network Working Group                                         R. Housley
Internet-Draft                                            Vigil Security
Intended status: Standards Track                                R. Droms
Expires: September 3, 2018                                        Google
                                                           March 2, 2018
```

TLS 1.3 Option for Negotiation of Visibility in the Datacenter
draft-rhrd-tls-tls13-visibility-01

Abstract

Current drafts of TLS 1.3 do not include the use of the RSA handshake. While (EC) Diffie-Hellman is in nearly all ways an improvement over the TLS RSA handshake, the use of (EC)DH has impacts certain enterprise network operational requirements. The TLS Visibility Extension addresses one of the impacts of (EC)DH through an opt-in mechanism that allows a TLS client and server to explicitly grant access to the TLS session plaintext.

Status of This Memo

Copyright Notice

1. Introduction

Unlike earlier versions of TLS, current drafts of TLS 1.3 [I-D.ietf-tls-tls13] do not provide support for the RSA handshake -- and have instead adopted ephemeral-mode Diffie-Hellman (DHE) and elliptic-curve Diffie-Hellman (ECDHE) as the primary cryptographic key exchange mechanism used in TLS.

While ephemeral (EC) Diffie-Hellman is in nearly all ways an improvement over the TLS RSA handshake, the use of these mechanisms has impacts on certain enterprise operational requirements. Specifically, the use of ephemeral ciphersuites prevents the use of current enterprise network monitoring tools such as Intrusion Detection Systems (IDS) and application monitoring systems, which leverage the current TLS RSA handshake to passively decrypt and monitor intranet TLS connections made between endpoints under the enterprise's control. This traffic includes TLS connections made from enterprise network security devices (firewalls) and load balancers at the edge of the enterprise network to internal enterprise TLS servers. It does not include TLS connections traveling over the external Internet.

Such monitoring of the enterprise network is ubiquitous and indispensable in some industries, and is required for effective and safe operation of their enterprise networks. Loss of this capability may slow adoption of TLS 1.3 or force enterprises to continue to use outdated and potentially vulnerable technology.

The TLS Visibility Extension provides an option to enable visibility into a TLS 1.3 session by an authorized third party. Use of the extension requires opt-in by the TLS client when it initiates a TLS 1.3 session. The TLS server then opts-in by including keying material that will enable decryption in the TLS Visibility Extension. The presence of the TLS Visibility Extension provides a clear indication that other parties have been granted access to the TLS session plaintext. The keying material in the TLS Visibility Extension is encrypted and can only be decrypted by authorized parties that have been given the private key from a managed Diffie-Hellman key pair.

2. Terminology

Two key pairs are used with the TLS Visibility Extension for encryption of the session secrets:

SSWrapDH1: generated externally and the public key is provided to the TLS 1.3 server prior to use of the TLS Visibility Extension; the corresponding private key is provided to the parties that are authorized to access the TLS session plaintext.

SSWrapDH2: an ephemeral key pair that is generated by the TLS 1.3 server for each TLS 1.3 session that uses the TLS Visibility Extension; the server keeps the private key confidential, and passes the public key to the other parties in the TLS Visibility session.

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

3. Extension Overview

Prior to the use of the TLS Visibility Extension, the SSWrapDH1 key pair is generated, possibly by an enterprise key manager. The private key is passed to the parties that are authorized to access the TLS session plaintext. The server is provisioned with the public key. When a new TLS 1.3 session is initiated, the client includes an empty TLS Visibility Extension in the ClientHello. The server then generates a SSWrapDH2 ephemeral key pair. The server will then:

- Generate a key, Ke, from the SSWrapDH1 public key and the SSWrapDH2 private key.

- Encrypt the TLS 1.3 session Early Secret (if one exists) and Handshake Secret (session secret) using Ke.

- Send an identifier for the SSWrapDH1 public key (called the fingerprint), the SSWrapDH2 public key, and the encrypted session secrets in the TLS Visibility Extension in the ServerHello message.

To decrypt the TLS 1.3 session, a party that is authorized to access the TLS session plaintext must be given the SSWrapDH1 private key. The party then:

- Obtains the SSWrapDH1 public key from the TLS Visibility extension

- Uses the SSHWrapDH1 private key and the SSWrapDH2 public key to generate Ke

- Uses Ke to decrypt the session secrets carried in the TLS Visibility extension

- Uses the session secrets to derive the keying material needed decrypt the TLS 1.3 session

4. TLS Visibility Extension

This section specifies the "tls_visibility" extension, which is carried in the ClientHello message and the ServerHello message.

The general extension mechanisms enable clients and servers to negotiate the use of specific extensions. As specified in [I-D.ietf-tls-tls13], clients request extended functionality from servers with the extensions field in the ClientHello message. If the server responds HelloRetryRequest, then the client sends another ClientHello message that includes the same extensions field as the original ClientHello message.

Most server extensions are carried in the EncryptedExtensions message; however, the "tls_visibility" extension is carried in the ServerHello message in a manner similar to the "key_share" and "pre_shared_key" extensions. It is only present in the ServerHello message if the server wants to enable TLS Visibility for some other parties and the client has offered the "tls_visibility" extension in the ClientHello message.

The "tls_visibility" extension MAY appear in the CH (ClientHello message) and SH (ServerHello message). It MUST NOT appear in any other messages. The "tls_visibility" extension MUST NOT appear in the ServerHello message unless "tls_visibility" extension appeared in the preceding ClientHello message. If an implementation recognizes the "tls_visibility" extension and receives it in any other message, then the implementation MUST abort the handshake with an "illegal_parameter" alert.

The Extension structure is defined in [I-D.ietf-tls-tls13]; 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_visibility" extension type, adding one new type to ExtensionType:

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

The "tls_visibility" extension is relevant when the client and server choose to enable one or more other parties to decrypt the TLS session.

Clients MUST include the "tls_visibility" extension in the ClientHello message to indicate their willingness for other parties to decrypt the TLS session. The server responds with data that enables the other parties to derive the keying material needed to decrypt the session if they are in possession of the indicated ECDH private key.

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

  struct {
      opaque early_secret<1..255>;
      opaque hs_secret<1..255>;
  } SessionSecrets;

  struct {
      opaque fingerprint<20>;
      opaque key_exchange<1..2^16-1>;
      opaque wrapped_secrets<1..2^16-1>;
  } WrappedSessionSecrets;
```

The fields in WrappedSessionSecrets are used as follows:

- "fingerprint" contains the leftmost 20 octets of the SHA-256 hash of SSWrapDH1 public key that was used by the server to compute the session secret wrapping key. The public key is DER-encoded in the SubjectPublicKeyInfo [RFC5280] for the SHA-256 hash computation. The key manager tells the server which AEAD algorithm to use with this SSWrapDH1 public key at the time it is distributed.

- "key_exchange" contains the SSWrapDH2 ephemeral public key generated by the server on the same elliptic curve as the SSWrapDH1 public key identified by the "fingerprint". The server uses the SSWrapDH2 ephemeral private key and the SSWrapDH1 public key identified by the "fingerprint" to compute a shared secret value, called Z, and then uses HKDF [RFC5869] to produce the session secret wrapping key, called Ke, and an AEAD nonce, if one is needed by the AEAD algorithm [RFC5116]. The details of the key agreement process are described in Section 5.

- "wrapped_secrets" contains the SessionSecrets structure encrypted with the AEAD algorithm under Ke. The details of the encryption process are described in Section 5.

The fields in SessionSecrets are used as follows:

- "early_secret" contains the Early Secret that was derived from the pre-shared key. If this session did not use a pre-shared key, then the Early Secret is HKDF-Extract(0, 0).

- "hs_secret" contains the handshake key that was computed using (EC)DHE.

5. Session Secret Wrapping

The input to the encryption process is the encoded SessionSecrets structure, and the ciphertext is carried in the "wrapped_secrets" field in the WrappedSessionSecrets structure. The session secret wrapping key, called Ke, and an AEAD nonce, if one is needed by the AEAD algorithm [RFC5116] are used to perform the encryption. For example, AES-KEY-WRAP-256 [RFC5649] does not require a nonce, but AES-GCM-128 [GCM] does require a nonce.

The "key_exchange" field of the WrappedSessionSecrets structure contains the SSWrapDH2 ephemeral public key generated by the server on the same elliptic curve as the SSWrapDH1 public key identified by the "fingerprint" field of the WrappedSessionSecrets structure. The server uses the SSWrapDH2 ephemeral private key and the SSWrapDH1 public key to compute a shared secret value, called Z, and then uses HKDF [RFC5869] to produce the Ke and the nonce:

```
      PRK = HKDF-Extract(0x00, Z)
      Ke = HKDF-Expand(PRK, "tls_vis_key", AEAD_key_size)
      nonce = HKDF-Expand(PRK, "tls_vis_nonce", AEAD_nonce_size)
```

The length of the ciphertext can be longer than the input plaintext, depending on the AEAD algorithm that is used. The AEAD algorithm is distributed to the server along with the SSWrapDH1 public key, so there is no need to carry an explicit algorithm identifier.

Encryption is performed as follows:

```
      wrapped_secrets = AEAD-Encrypt(Ke, nonce, SessionSecrets)
```

Other parties use the SSWrapDH2 ephemeral public key from the "key_exchange" field of the WrappedSessionSecrets structure and the SSWrapDH1 private key that is associated with the "fingerprint" field of the WrappedSessionSecrets structure to compute a shared secret value, called Z. The SSWrapDH1 private key and the AEAD algorithm are obtained in advance. Then, Z is used to produce the Ke and the nonce as specified above. To unwrap the session secrets, decryption is performed as follows:

```
      SessionSecrets = AEAD-Encrypt(Ke, nonce, wrapped_secrets)
```

The result is either the plaintext of the SessionSecrets structure or an error indicating that the decryption failed. An integrity check is performed as part of the decrypt operation.

6. Alternative Approaches

This section captures the rationale for pursuing this approach to TLS visibility instead of the various alternative approaches.

Server uses a static Diffie-Hellman key pair: Instead of generating ephemeral Diffie-Hellman key pairs, the server reuses a static Diffie-Hellman key pair. The static private Diffie-Hellman key gets shared with the points that need visibility. While this approach scales, the TLS client is unaware of the sharing. In addition, this enables visibility of data of all clients communicating with the server, versus only those that opt-in to visibility.

Export of ephemeral keys: In large enterprises there will be billions of ephemeral keys to export and distribute. Transporting these keys to tools for decryption of packets in real time will be difficult, adding greatly to the complexity of the solution.

Export of decrypted traffic from TLS proxy devices: Decrypting traffic only at the edge of the enterprise datacenter does not meet all of the enterprise requirements, which include troubleshooting, fraud detection, and network security monitoring. Further, the number of TLS proxies needed are quite costly, add latency, and increase production risk.

Continue to use TLS 1.2 within the enterprise network: TLS 1.2 could be used within the enterprise network (with TLS 1.3 outside) to enable TLS visibility via RSA key transport. However, TLS 1.3 has security improvements over TLS 1.2. At some point in the future, TLS 1.2 will not longer be supported and available in enterprise applications and protocol implementations. In addition, based on experience, standards bodies will deprecate the use of TLS 1.2 and require enterprise networks to move to TLS 1.3.

Reliance on TCP/IP headers: TCP and IP headers are not adequate for enterprise requirements. Troubleshooting, fraud detection, and network security monitoring need access to the plaintext payload. For example, troubleshooters must be able to find specific transactions, user identifiers, session identifiers, URLs, and time stamps.

Reliance on application and server logs: Logging is not adequate for enterprise requirements. Code developers cannot anticipate every possible problem for logging, and system administrators turn much of the logging off to conserve system resources.

Troubleshooting and malware analysis at the endpoint: Endpoints are focused on providing a service, and they cannot handle the additional burden of the various enterprise monitoring requirements.

Adding TCP/UDP extensions: An important part of troubleshooting, network security monitoring, etc. is analysis of the application-specific payload of the packet. It is not possible to anticipate ahead of time, among thousands of unique applications, which fields in the application payload will be important.

7. IANA Considerations

IANA is requested to update the TLS ExtensionType Registry to include "tls_visibility" with a value of [TBD] and the list of messages "CH, SH" in which the "tls_visibility" extension may appear.

8. Security Considerations

The use of a TLS protocol extension ensures that both the TLS client and the TLS server are aware that other parties have visibility into the TLS session plaintext. However, the approach used here does not allow those parties to masquerade since they do not have the ability to sign the Finished message in the TLS handshake.

Use of the TLS Visibility extension represents a deliberate introduction by the client and server of other parties that can access the TLS session plaintext. Deployments that choose to make use of this extension should carefully consider the risks associated with the change to the Forward Secrecy. In particular, Forward Secrecy will not begin for sessions where the TLS Visibility Extension is used until all of these events take place:

(1) The server has securely discarded the session secrets.

(2) The server has securely discarded the session secret wrapping key.

(3) The client has securely discarded the session secrets.

(4) The other parties have securely discarded the session secrets.

(5) The other parties have securely discarded the session secret wrapping key.

(6) The other parties have securely discarded the ECDHE private key that was used to derive the session secret wrapping key.

By agreeing to the use of the TLS Visibility extension, the client is aware that the TLS session plaintext will be accessible to any other party that has access to the ECDHE private key that was used to derive the session secret wrapping key. It is envisioned that the server and other parties will all be under a single administrative control; however, the TLS Visibility extension does not guarantee any particular scope for the distribution of the ECDHE private keys.

The SSWrapDH1 and SSWrapDH2 key size and parameters MUST be selected to provide the same level (or more) of security as the (EC)DHE key used in the TLS Handshake. Similarly, the Sessions Secret Wrapping key size and algorithm MUST be selected to provide the same level (or more) of security as the AEAD cipher used with the TLS Record protocol. If weaker key sizes, parameters or algorithms are used, the attacker will find it easier to obtain the session secrets from the TLS Visibility extension.

9. Acknowledgments

Matthew Green was the primary author of [I-D.green-tls-static-dh-in-tls13], which describes an earlier solution to the TLS 1.3 session visibility problem. Nick Sullivan and Richard Barnes suggested the use of client and server opt-in. Peter Wu suggested the use of HKDF-Expand to get a nonce. Nalini Elkins, Steven Fenter, Sinok Lao, Andrew Kennedy, Darin Pettis, Tim Polk, Andrew Regenscheid, Murugiah Souppaya, and Paul Turner contributed through discussion to the development of this document.

10. References

10.1. Normative References

[I-D.ietf-tls-tls13]
Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", draft-ietf-tls-tls13-24 (work in progress), February 2018.

[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>.

[RFC5280] Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R., and W. Polk, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", RFC 5280, DOI 10.17487/RFC5280, May 2008, <https://www.rfc-editor.org/info/rfc5280>.

[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>.

10.2. Informative References

[GCM] Dworkin, M., "Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC", NIST Special Publication 800-38D, November 2007.

<http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38d.pdf>

[I-D.green-tls-static-dh-in-tls13]
Green, M., Droms, R., Housley, R., Turner, P., and S. Fenter, "Data Center use of Static Diffie-Hellman in TLS 1.3", draft-green-tls-static-dh-in-tls13-01 (work in progress), July 2017.

[RFC5116] McGrew, D., "An Interface and Algorithms for Authenticated Encryption", RFC 5116, DOI 10.17487/RFC5116, January 2008, <https://www.rfc-editor.org/info/rfc5116>.

[RFC5649] Housley, R. and M. Dworkin, "Advanced Encryption Standard (AES) Key Wrap with Padding Algorithm", RFC 5649, DOI 10.17487/RFC5649, September 2009, <https://www.rfc-editor.org/info/rfc5649>.

Authors' Addresses

Russ Housley
Vigil Security, LLC
918 Spring Knoll Drive
Herndon, VA 20170
USA

Email: housley@vigilsec.com

Ralph Droms
Google
355 Main Street
Cambridge, MA 02142
USA

Email: rdroms.ietf@gmail.com