Internet Engineering Task Force
Internet-Draft
Intended status: Informational
Expires: January 9, 2017

T. Fossati
Nokia
H. Tschofenig
ARM
N. Mavrogiannopoulos
Red Hat

July 8, 2016

TLS/DTLS Optimizations for Internet of Things Deployments

draft-fossati-tls-iot-optimizations-00

Abstract

Internet protocols work well in a variety of environments, including Internet of Things (IoT) deployments. While there are some optimization possibilities to reduce code size, bandwidth utilization, and to improve battery lifetime, in general most Internet protocols are also applicable to constrained environments. TLS and DTLS are two such security protocols that can be used by many IoT devices since DTLS/TLS provide lot of flexiblity in terms credential choice, ciphersuite usage, etc. The DICE working group has developed a specification that profiles the use of TLS and DTLS for IoT environments, without changing the TLS/DTLS specifications.

This memo goes a step further and proposes changes to the DTLS/TLS protocol to introduce further optimizations. Since the ongoing work on TLS/DTLS 1.3 already offers several improvements (compared to previous versions) this document focuses on the use of version 1.3 and suggests further optimizations.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

Internet protocols work well in a variety of environments, including Internet of Things (IoT) deployments. While there are some optimization possibilities to reduce code size, bandwidth utilization, and to improve battery lifetime, in general most Internet protocols are also applicable to constrained environments. TLS and DTLS are two such security protocols that can be used by many IoT devices since DTLS/TLS provide lot of flexiblity in terms credential choice, ciphersuite usage, etc. The DICE working group has developed a specification that profiles the use of TLS and DTLS for IoT environments, without changing the TLS/DTLS specifications.

This memo goes a step further and proposes changes to the DTLS/TLS protocol to introduce further optimizations. Since the ongoing work on TLS/DTLS 1.3 [I-D.ietf-tls-tls13] already offers several improvements (compared to previous versions) this document focuses on the use of version 1.3 and suggests further optimizations.

This document discusses four extensions, namely:

Selective Fragment Retransmission: This extension improves retransmissions of lost handshake packets.

Transport Agnostic Security Associations: Changes to a connection (such as an IP address change) requires a new handshake. This extension introduces a transport independent identifier.

Reducing the DTLS Record Layer Header Overhead: This extension changes the record layer format to reduce the overhead.

Reducing Buffers: This extension allows a DTLS/TLS server running on a constrained node to indicate its buffer size.

2. Requirements Language

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

3. Selective Fragment Retransmission

3.1. Problem Statement

The unit of retransmission used by the DTLS handshake is a whole flight (see Section 4.2.4 of [RFC6347]. If the underlying media is inherently lossy, or shows high latency variance that might fire spurious retransmission, a single fragment that gets lost or is excessively delayed will force the whole flight to be retransmitted.

This is further exacerbated when the effective MTU is very low and therefore handshake messages have higher probability to be fragmented. For example, IEEE 802.15.4 networks have a 128-byte MTU size. In such an environment a very "ordinary" TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA negotiation can take up to 30 individual fragments, 2/3 of which are sent in flight 4. The loss of a single fragment in flight 4 implies a retransmission that is 20x the magnitude of the original loss.

The retransmission timer settings suggested in Section 11 of [I-D.ietf-dice-profile] offer mitigation for the spurious retransmit issue and, in general, help with congestion. However, they do not solve the retransmission of the entire flight.

3.2. Proposal

A potential solution is to add a NACK-based retransmission scheme to the DTLS handshake and the granularity of retransmission would be a message fragment. We note that each fragment in a DTLS handshake is effectively associated to a unique identifier defined by the tuple Handshake.{message_seq,fragment_offset,fragment_length} that can be used in the NACK report to identify the exact geometry of the missing data in the current flight, together with the right-most received byte.

4. Transport Agnostic Security Associations

4.1. Problem Statement

In DTLS, the security context demultiplexing is done via the 5-tuple. This implies that the associated DTLS context needs to be re-negotiated from scratch whenever the IP address changes. For example, when moving the network attachment from WLAN to a cellular connection, or when the IP address of the IoT devices changes during a sleep cyle. A NAT device may also modify the source UDP port after an idle period. In such situations, there is not enough information in the DTLS record header for a DTLS server, which handles multiple clients, to associate the changed address to an existing client.

4.2. Proposal

A potential solution is to add the ability to negotiate, at handshake time, a transport independent identifier that is unique per security association. We call this identifier the ’Connection ID (CID)’ in Figure 1. It decouples the DTLS session from the underlying transport protocol allowing the same DTLS security association to be migrated across different transport sessions.

```
                                        00
                                        /\
                                        :
    IP                     UDP          : DTLS Record Header
    +-----+-----+-------+ +-----+-----+ : +---------+-------+------
    | src | dst | proto | | src | dst | : | Seq#i   |  CID  | ...
    +-----+-----+-------+ +-----+-----+ : +---------+-------+------
    `----------------+----------------' :              ^
                      `................ : .............'
     <Handover event>                   :
                      GSM-SMS           : DTLS Record Header
                      +-------+-------+ : +---------+-------+-----
                      | tp-oa | tp-da | : | Seq#i+1 |  CID  | ...
                      +-------+-------+ : +---------+-------+-----
                                        :
                                        \/
                                        00
```

Figure 1: Transparent Handover of DTLS Session

That approach modifies the DTLS record layer header to the format described in Figure 2.

```
     struct {
          ContentType type;
          ProtocolVersion version;
          uint16 epoch;
          uint48 sequence_number;
          uint32 connection_id;                          // New field
          uint16 length;
          opaque fragment[DTLSPlaintext.length];
     } DTLSPlaintext;
```

Figure 2: Modified DTLS Record Format

A similar approach to support transparent handover of a DTLS session has been described in [I-D.barrett-mobile-dtls] and [Seggelmann].

The privacy issue associated with the use of a long-term identifier must be taken into consideration. For example, client and server could use a hash chain [Lamport] derived from the shared secret and pick the next unused id on handover.

5. Reducing the DTLS Record Layer Header Overhead

5.1. Problem Statement

The DTLS record layer header adds 13 bytes of overhead, as described in Appendix B of [I-D.ietf-dice-profile]. While some of the information carried in the header is unavoidable, other parameters are redundant and included for backwards compatibility reasons. This burden becomes quite substantial in networks with small frame sizes (e.g., low power wide area networks).

Overhead that is not strictly needed could be removed to allow applications to transmit more data in a single packet or to make space for other DTLS features, such as the proposal described in Section 4.

5.2. Proposal

It is possible to at least remove the following parameters in the header:

- o Protocol Version (2 bytes)

- o The sequence number component of the nonce_explicit field at the AES-CCM layer is an exact copy of the sequence number in the record layer header field. This leads to a duplication of 8-bytes per record.

6. Reducing Buffers

6.1. Problem Statement

The Maximum Fragment Length extension [RFC6066] allows a client with limited buffer space to specify a different (smaller) maximum size for fragments that the server is allowed to send. The mechanism is not symmetrical: a server cannot state their buffer size. The assumption made in [RFC6066] is that the server is never going to be a constrained device, and therefore does not need such a capability. This may be true for many IoT deployments where the TLS client is implemented in an IoT device that connects to a server on the Internet that does not have memory limitations, such as a server in the cloud. However, with the desire to also deploy CoAPS and HTTPS-based servers in IoT devices, a constrained node may also need to run a DTLS/TLS server. In such a scenario, allowing a constrained server to advertise its Maximum Fragment Length helps to lower memory requirements.

6.2. Proposal

The semantics of the max_fragment_length extension could be modified to allow the server as well as the client to express their buffer sizes.

7. Acknowledgements

We would like to thank Stephen Farrell for suggesting the use of hash chains to implement a privacy-friendly connection id.

8. Security Considerations

This document suggests various extensions to DTLS/TLS and each of them comes with their own security and privacy considerations. Since this version of the document only suggests strawman proposals further discussions are needed to specify the details.

9. References

9.1. Normative References

[I-D.ietf-tls-tls13]
Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", draft-ietf-tls-tls13-13 (work in progress), May 2016.

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

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

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

9.2. Informative References

[I-D.barrett-mobile-dtls]
Williams, M. and J. Barrett, "Mobile DTLS", draft-barrett-mobile-dtls-00 (work in progress), March 2009.

[I-D.ietf-dice-profile]
Tschofenig, H. and T. Fossati, "TLS/DTLS Profiles for the Internet of Things", draft-ietf-dice-profile-17 (work in progress), October 2015.

[Lamport] Lamport, L., "Password Authentication with Insecure Communication", Commun. ACM, 1981.

[Seggelmann]
Seggelmann, R., Tuexen, M., and E. Rathgeb, "DTLS Mobility", ICDCN 12, 2012.

Authors' Addresses

Thomas Fossati
Nokia
Cambridge
UK

Email: thomas.fossati@nokia.com

Hannes Tschofenig
ARM
Cambridge
UK

Email: hannes.tschofenig@arm.com

Nikos Mavrogiannopoulos
Red Hat
Brno
Czech Republic

Email: nmav@redhat.com

Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: January 4, 2018

M. Green
Cryptography Engineering LLC
R. Droms
Interisle Consulting
R. Housley
Vigil Security, LLC
P. Turner
Venafi
S. Fenter
July 3, 2017

Data Center use of Static Diffie-Hellman in TLS 1.3
draft-green-tls-static-dh-in-tls13-01

Abstract

Unlike earlier versions of TLS, current drafts of TLS 1.3 have instead adopted ephemeral-mode Diffie-Hellman and elliptic-curve Diffie-Hellman as the primary cryptographic key exchange mechanism used in TLS. This document describes an optional configuration for TLS servers that allows for the use of a static Diffie-Hellman private key for all TLS connections made to the server. Passive monitoring of TLS connections can be enabled by installing a corresponding copy of this key in each monitoring device.

Status of This Memo

Copyright Notice

Table of Contents

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 complicates certain enterprise settings. Specifically, the use of ephemeral ciphersuites is not compatible with current enterprise network monitoring tools such as Intrusion Detection Systems (IDS) and application monitoring systems, which leverage the current TLS RSA handshake passively 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. This monitoring is required for effective and safe operation of enterprise networks. Loss of this capability may slow adoption of TLS 1.3.

This document describes an optional configuration for TLS servers that is compatible with the TLS 1.3 ephemeral ciphersuites without precluding enterprise network monitoring. This configuration allows for the use of a static (EC) Diffie-Hellman private key for all TLS connections made to the server. Passive monitoring of TLS connections can be enabled by installing a corresponding copy of this key in each authorized monitoring device.

An advantage of this proposal is that it can be implemented using software modifications to the TLS server and enterprise network monitoring tools, without the need to make changes to TLS client implementations.

1.1. Terminology

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].

This document introduces the term "static (elliptic curve) Diffie-Hellman ephemeral", generally written as "static (EC)DHE", to refer to long-lived finite field or elliptic curve Diffie-Hellman keys or key pairs that will be used with the TLS 1.3 ephemeral ciphersuites to negotiate traffic keys for multiple TLS sessions.

For clarity, this document also introduces the term "ephemeral (elliptic curve) Diffie-Hellman ephemeral", generally written as "ephemeral (EC)DHE", to denote finite field or elliptic curve Diffie-Hellman keys or key pairs that will be used with the TLS 1.3 ephemeral ciphersuites to negotiate traffic keys for a single TLS sessions.

1.2. ASN.1

The Cryptographic Message Syntax (CMS) [RFC5652] and asymmetric key packages [RFC5958] are generated using ASN.1 [X680], which uses the

Basic Encoding Rules (BER) and the Distinguished Encoding Rules (DER) [X690].

2. Enterprise Out-of-band TLS Decryption Architecture

This document describes the use of a static (elliptic-curve) Diffie-Hellman (static (EC)DHE) private key by servers for use in TLS 1.3 sessions internal to an enterprise network where network monitoring is required. In Figure 1, the Web Servers use a static (EC)DHE key pair with the standard TLS 1.3 handshake for connections from the Load Balancer, and the Back-End Services use static (EC)DHE for connections from the Web Servers. The Load Balancer uses ephemeral (EC)DHE key pairs with the standard TLS 1.3 handshake for connections from external Browsers over the Internet, to provide Forward Secrecy on those connections that are exposed to third-party monitoring. Internally, the static (EC)DHE keys are provided to authorized TLS Decrypter devices, such as intrusion detection systems, application monitoring systems or network packet capture devices.

```
                                   ********************************
                                  *                                *
                                 *            +--------+            *
                                *   TLS       |  Web   |             *
                               * Termination  + Server +              *
                              *      |       /|        |\              *
    +---------+  +----------+ * +----|-----+/ +--------+ \+----------+ *
    |         |  |          | * |   Load    +              + Back-end | *
    | Browser +--+ Internet |-*-+ Balancer |              |  Server  | *
    |         |  |          | * |    |     +              +          | *
    +---------+  +----------+ * +----------+\ +--------+ /+----------+ *
                              *      |      .\|  Web   |/.             *
                              *            .  + Server +  .            *
                              *            .  |        |  .            *
                              *            .  +--------+  .            *
                              *            .              .            *
                              *            .   --------   .            *
                              *             . /  TLS   \ .             *
                              *              | Decrypter|              *
                               *              \        /              *
                                *              --------              *
                                 *                                  *
                                  *** Enterprise Network Boundary **

                                     |
    <------ Ephemeral (EC)DHE ------>|<-------- Static (EC)DHE -------->
                                     |
```

Figure 1: Enterprise TLS Decryption Architecture

3. Enterprise Requirements for Passive (out-of-band) TLS Decryption

Enterprise networks based on this architecture have operational requirements for traffic monitoring and ex post facto analysis for purposes of:

- o Application troubleshooting and performance analysis
- o Fraud monitoring
- o Security, including intrusion detection, malware detection, confidential data exfiltration and layer 7 DDoS protection
- o Audit compliance
- o Customer Experience Monitoring

Specific requirements to meet the listed operational requirements include:

- TLS decryption for network security monitoring tools must be done in real time with no gaps in decryption.

- The solution must be able to decrypt passively captured pcap traces.

- The solution must scale to handle thousands of TLS sessions/sec.

- Key material must be preserved for back-in-time analysis. The period for key retention depends upon local policy, reflecting operational, security and compliance requirements.

- Key material must be encrypted during network transit

- The solution must not negatively impact the enterprise infrastructure (servers, network, etc.)

- The solution must be able to decrypt the session when a TLS session is reused. This may involve the use of a TLS decryption appliance.

- The solution must be able to decrypt in a physical data center, in a virtual environment, and in a cloud.

4. Summary of the Existing Diffie-Hellman Handshake

In TLS 1.3, servers exchange keys using two primary modes, DHE and ECDHE. In a simplified view of the full handshake, the following steps occur:

1. The client generates an ephemeral public and private key, and transmits the public key within a "key_share" message, along with a random nonce (ClientHello.random).
2. The server generates an ephemeral public and private key, and transmits the public key within a "key_share" message, along with a random nonce (ServerHello.random).
3. The two parties now calculate a shared (EC)DHE secret by combining the other party’s ephemeral public key with their own ephemeral private key.
4. A series of traffic and handshake keys is derived by combining this shared secret with various inputs from the handshake, including the ClientHello.random and ServerHello.random.
5. Data encryption is performed using the shared secret.

5. Using static (EC)DHE on the server

The proposal embodied in this draft modifies the standard TLS handshake summarized above in the following ways:

For each elliptic curve (and FF-DH parameter length) supported by the server, the server is provisioned with a static (EC)DHE private/public key pair. This key pair may be either:

* generated at server installation, and rotated at periodic intervals appropriate for any long-term server key,
* generated at a central key management server and distributed (in a secure encrypted form) to the appropriate endpoint servers.

All steps of the original handshake proceed as above, with the following modification to server behavior. Step (2) proceeds as follows:

2. The server transmits the static public key within a "key_share" message, along with a random nonce (ServerHello.random).

6. Key Representation

The Asymmetric Key Package [RFC5958] MUST be used to transfer the centrally managed Diffie-Hellman key pair. The key package contains at least one Diffie-Hellman key pair. Each Diffie-Hellman key pair is associated with a set of attributes, including the key validity period for that Diffie-Hellman key pair.

OneAsymmetricKey is defined in Section 2 of [RFC5958]. The fields are used as follows:

- version MUST be set to v2, which has an integer value of 1.
- privateKeyAlgorithm MUST be set to the algorithm identifier of the Diffie-Hellman key pair. For convenience, some popular algorithm identifiers are listed in Figure 2.
- privateKey MUST be set to the Diffie-Hellman private key encoded as an OCTET STRING.
- attributes MUST be included even though the field is optional. The set of attributes MUST include the key validity period attribute defined in Section 15 of [RFC7906]. Other attributes MAY be included as well.

- publicKey MUST be included even though the field is optional. It MUST be set to the Diffie-Hellman public key, encoded as a BIT STRING. This is the same BIT STRING that would be included in an X.509 certificate [RFC5280] for this public key.

```
+--------------------------------------------------------------------+
|                                                                    |
| Finite Field Diffie-Hellman                                        |
|  object identifier: { 1 2 840 10046 2 1 }                          |
|  parameter encoding: DomainParameters, Section 2.3.3 of [RFC3279] |
|  private key encoding: INTEGER                                     |
|  public key encoding: INTEGER                                      |
|                                                                    |
| Elliptic Curve Diffie-Hellman                                      |
|  object identifier: { 1 3 132 1 12 }                               |
|  parameter encoding: ECParameters, Section 2.1.2 of [RFC5480]      |
|                        (MUST use the namedCurve CHOICE)            |
|  private key encoding: ECPrivateKey, Section 3 of [RFC5915]        |
|  public key encoding:  ECPoint, Section 2.2 of [RFC5480]           |
|                                                                    |
+--------------------------------------------------------------------+
```

Figure 2: Popular Diffie-Hellman Algorithm Identifiers

The CMS protecting content types [RFC5652][RFC5083] can be used to provide authentication and confidentiality protection for the Asymmetric Key Package:

- SignedData can be used to apply a digital signature to the Asymmetric Key Package.

- EncryptedData can be used to encrypt the Asymmetric Key Package with previously distributed symmetric encryption key.

- EnvelopedData can be used to encrypt the Asymmetric Key Package, where the sender and the receiver establish a symmetric encryption key using Diffie-Hellman key agreement.

- AuthEnvelopedData can be used to protect the Asymmetric Key Package where the sender and the receiver establish a symmetric authenticated encryption key using Diffie-Hellman key agreement.

7. TLS Static DH Key (TSK) Protocol

The TLS Static DH Key (TSK) Protocol is used in cases where the Diffie-Hellman keys are centrally managed. The two main roles in the TSK protocol are "key manager" and "key consumer". Key consumers can be TLS servers or TLS decrypters. The key manager generates, distributes, and tracks static (EC)DHE keys used by key consumers. TSK messaging is based on HTTPS [RFC2818]. Keys are transferred as Asymmetric Key Packages [RFC5958], using the profile in Section 6 of this document.

```
                    --------------       -----------------
                    | TLS server |-------|  key manager  |
                    --------------       -----------------
                           |                     |
                           |                     |
                           |                     |
                           |             -----------------
                           |------------>| TLS decrypter |
                           |             -----------------
                           |
                           |
                    --------------
                    | TLS client |
                    --------------
```

Figure 3: TSK protocol components

The key manager can push keys to key consumers:

```
        TLS server               key manager             TLS decrypter
            |                         |                         |
            |                         |--                       |
            |                         |  \ Generate             |
            |                         |  / key pair             |
            |                         |<-                       |
            |                         |                         |
            |                         |------------------------>|
            |                         |      Push key pair      |
            |<------------------------|                         |
            |       Push key pair     |                         |
```

Figure 4: TSK protocol push model

Alternatively, key consumers can request (or pull) keys from the key manager.

```
        TLS server               key manager             TLS decrypter
            |                         |                        |
            |                         |--                      |
            |                         |  \ Generate            |
            |                         |  / key pair            |
            |                         |<-                      |
            |                         |                        |
            |                         |<-----------------------|
            |                         |    Request key pair    |
            |------------------------>|                        |
            |     Request key pair    |                        |
```

Figure 5: TSK protocol pull model

7.1. Key Push

An HTTPS-based TSK push is composed of the appropriate HTTP headers, followed by the binary value of the BER (Basic Encoding Rules) encoding of the Asymmetric Key Package.

The Content-Type header MUST be application/cms [RFC7193] if the Asymmetric Key Package is encrypted with CMS [RFC6032]. The Content-Type header MUST be application/pkcs8 if the Asymmetric Key Package is transferred in plain text (within the encrypted HTTPS stream).

7.2. Key Request

A key consumer may request a key by providing a fingerprint [RFC6234] of the public key. The key manager is responsible for determining if the key consumer is authorized to receive a copy of the key being requested.

Example with plain text Asymmetric Key Package:

```
GET /tsk/key/PublicKeyFingerprint
Accept: application/pkcs8
```

Example with CMS encrypted and/or signed Asymmetric Key Package:

```
GET /tsk/key/PublicKeyFingerprint
Accept: application/cms
```

The response to the TSK push is composed of the appropriate HTTP headers, followed by the binary value of the BER (Basic Encoding Rules) encoding of the Asymmetric Key Package.

The Content-Type header MUST be application/cms [RFC7193] if the Asymmetric Key Package is encrypted with CMS [RFC6032]. The Content-Type header MUST be application/pkcs8 if the Asymmetric Key Package is transferred in plain text (within the encrypted HTTPS stream).

8. Alternative Solutions for Enterprise Monitoring and Troubleshooting

- Export of ephemeral keys
- Export of decrypted traffic from TLS proxy devices at the edge of the enterprise network
- Placement of TLS proxies in the enterprise network
- Reliance on TCP/IP headers not encrypted by TLS
- Reliance on application/server logs
- Doing troubleshooting and malware analysis at the endpoint.
- Adding a TCP or UDP extension to provide the information needed to do packet analysis.

9. Weaknesses of Alternative Solutions

Export of ephemeral keys: Scale - In a large enterprise there will be billions of ephemeral keys to export and manage. There will also be difficulty in transporting these keys to real time tools that need decrypted packets. The complexity of the solution is a problem that adds risk.

Export of decrypted traffic from TLS proxy devices: Decrypted traffic at only the edge of the network is not adequate for the enterprise requirements listed above (troubleshooting, network security monitoring, etc...)

TLS proxies in the network: Inline TLS proxies will not scale to the number of decryption points needed within an enterprise. Each inline proxy adds cost, latency, and production risk.

Reliance on TCP/IP headers: IP and/or TCP headers are not adequate for the enterprise requirements listed above. Troubleshooters must be able to find transactions in a pcap trace, identified by markers like userids, session ids, URLs, and time stamps. Threat Detection teams must be able to look for Indicators of Compromise in the payload of packets, etc.

Reliance on Application/server logs: Logging is not adequate for the enterprise requirements listed above. Code developers cannot anticipate every possible problem and put a log message in just the right place. There are billions of lines of code in a data center, and it’s not scalable to try and improve logging.

Troubleshooting and malware analysis at the endpoint: Endpoints don’t have the robustness to do their own workload and handle the burden of the various enterprise requirements listed above. These requirements would include always-on full packet capture at the endpoint with no packet drops.

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.

10. Security considerations

We now consider the security implications of the change described above:

1. The shift from fully-ephemeral (EC)HDE to static (EC)DHE affects the security properties offered by the TLS 1.3 handshake by eliminating the Forward Secrecy property provided by the server. If a server is compromised and the private key is stolen, then an attacker who observes any TLS handshake (even one that occurred prior to the compromise) performed with this static (EC)DHE key pair will be able to recover session traffic encryption keys and will be able to decrypt traffic.

2. As long as the server static secret key is not compromised, the resulting protocol will provide strong cryptographic security, as long as the Diffie-Hellman parameters (e.g., finite-field group or elliptic curve) are correctly generated and provide security at a sufficient cryptographic security level.

3. A flaw in the generation of finite-field Diffie-Hellman parameters or the use of an insecure implementation could leak some bits of the static secret key over time. This risk is not present in ephemeral DH implementations. Implementers should use care to avoid such pitfalls.

Thus the modification described in Section 10 represents a deliberate weakening of some security properties. Implementers who choose to include this capability should carefully consider the risks to their infrastructure of using a handshake without Forward Secrecy. Static (EC)DHE key pairs should be rotated regularly.

11. IANA Considerations

This document contains no actions for IANA.

12. Acknowledgements

This modification to TLS was initially suggested by Hugo Krawczyk.

13. Normative References

[I-D.ietf-tls-tls13]
Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", draft-ietf-tls-tls13-20 (work in progress), April 2017.

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

[RFC2818] Rescorla, E., "HTTP Over TLS", RFC 2818, DOI 10.17487/RFC2818, May 2000, <http://www.rfc-editor.org/info/rfc2818>.

[RFC3279] Bassham, L., Polk, W., and R. Housley, "Algorithms and Identifiers for the Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", RFC 3279, DOI 10.17487/RFC3279, April 2002, <http://www.rfc-editor.org/info/rfc3279>.

[RFC5083] Housley, R., "Cryptographic Message Syntax (CMS) Authenticated-Enveloped-Data Content Type", RFC 5083, DOI 10.17487/RFC5083, November 2007, <http://www.rfc-editor.org/info/rfc5083>.

[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, <http://www.rfc-editor.org/info/rfc5280>.

[RFC5480] Turner, S., Brown, D., Yiu, K., Housley, R., and T. Polk, "Elliptic Curve Cryptography Subject Public Key Information", RFC 5480, DOI 10.17487/RFC5480, March 2009, <http://www.rfc-editor.org/info/rfc5480>.

[RFC5652] Housley, R., "Cryptographic Message Syntax (CMS)", STD 70, RFC 5652, DOI 10.17487/RFC5652, September 2009, <http://www.rfc-editor.org/info/rfc5652>.

[RFC5915] Turner, S. and D. Brown, "Elliptic Curve Private Key Structure", RFC 5915, DOI 10.17487/RFC5915, June 2010, <http://www.rfc-editor.org/info/rfc5915>.

[RFC5958] Turner, S., "Asymmetric Key Packages", RFC 5958, DOI 10.17487/RFC5958, August 2010, <http://www.rfc-editor.org/info/rfc5958>.

[RFC6032] Turner, S. and R. Housley, "Cryptographic Message Syntax (CMS) Encrypted Key Package Content Type", RFC 6032, DOI 10.17487/RFC6032, December 2010, <http://www.rfc-editor.org/info/rfc6032>.

[RFC6234] Eastlake 3rd, D. and T. Hansen, "US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF)", RFC 6234, DOI 10.17487/RFC6234, May 2011, <http://www.rfc-editor.org/info/rfc6234>.

[RFC7193] Turner, S., Housley, R., and J. Schaad, "The application/cms Media Type", RFC 7193, DOI 10.17487/RFC7193, April 2014, <http://www.rfc-editor.org/info/rfc7193>.

[RFC7906] Timmel, P., Housley, R., and S. Turner, "NSA's Cryptographic Message Syntax (CMS) Key Management Attributes", RFC 7906, DOI 10.17487/RFC7906, June 2016, <http://www.rfc-editor.org/info/rfc7906>.

[X680] ITU-T, "Information technology -- Abstract Syntax Notation One (ASN.1): Specification of basic notation", ITU-T Recommendation X.680, 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)", ITU-T Recommendation X.690, 2015.

Authors' Addresses

Matthew Green
Cryptography Engineering LLC
4506 Roland Ave
Baltimore, MD 21210
USA

Email: mgreen@cryptographyengineering.com

Ralph Droms
Interisle Consulting

Email: rdroms.ietf@gmail.com

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

Email: housley@vigilsec.com

Paul Turner
Venafi
175 East 400 South, Suite 300
Salt Lake City, UT 84111
USA

Email: paul.turner@venafi.com

Steve Fenter

Email: steven.fenter58@gmail.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

Network Working Group   E. Rescorla
Internet-Draft   RTFM, Inc.
Obsoletes: 5077, 5246, 6961 (if   March 20, 2018
approved)
Updates: 4492, 5705, 6066 (if approved)
Intended status: Standards Track
Expires: September 21, 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:

```
20202020202020202020202020202020202020202020202020202020202020202020
20202020202020202020202020202020202020202020202020202020202020202020
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., Čermak, M., Jirsik, T., and P. Čeleda, "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
Internet-Draft
Obsoletes: 6347 (if approved)
Intended status: Standards Track
Expires: September 14, 2017

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

March 13, 2017

The Datagram Transport Layer Security (DTLS) Protocol Version 1.3

draft-rescorla-tls-dtls13-01

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. Datagram semantics of the underlying transport are preserved by the DTLS protocol.

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/dtls13-spec. Instructions are on that page as well. Editorial changes can be managed in GitHub, but any substantive change should be discussed on the TLS mailing list.

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

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

DTLS 1.0 was originally defined as a delta from TLS 1.1 and DTLS 1.2 was defined as a series of deltas to TLS 1.2. 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 aligning with the efforts around TLS 1.3.

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 [I-D.ietf-tls-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 RFC 2119 [RFC2119].

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 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 TLS connection.

The reader is assumed to be familiar with the TLS 1.3 specification since this document defined as a delta from TLS 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 keysderived from traffic_secret_N.

3. DTLS Design Rational and Overview

The basic design philosophy of DTLS is to construct "TLS over datagram transport". Datagram transport does not require or 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 re-ordered data traffic.

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

1. TLS does not allow independent decryption of individual records. Because the integrity check indirectly depends on a sequence number, if record N is not received, then the integrity check on record N+1 will be based on the wrong sequence number and thus will fail. DTLS solves this problem by adding explicit 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. Clearly, this is incompatible with reordering and message loss.

3. Not all TLS 1.3 handshake messages (such as the NewSessionTicket message) are acknowledged. Hence, a new acknowledgement message has to be added to detect message loss.

4. Handshake messages are potentially larger than any given datagram, thus creating the problem of IP fragmentation.

5. Datagram transport protocols, like UDP, are more vulnerable to denial of service attacks and require a return-routability check with the help of cookies to be integrated into the handshake. A detailed discussion of countermeasures can be found in Section 5.1.

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 from the server. However, if the server's message is lost, the client knows that either the ClientHello or the HelloRetryRequest 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.

3.1.1. Reordering

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

3.1.2. Message Size

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

3.2. Replay Detection

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

4. The DTLS Record Layer

The DTLS record layer is similar to that of TLS 1.3 unless noted otherwise. The only change is the inclusion of an explicit epoch and sequence number in the record. This sequence number allows the recipient to correctly verify the TLS MAC. The DTLS record format is shown below:

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

   struct {
       ContentType opaque_type = 23; /* application_data */
       ProtocolVersion legacy_record_version = {254,253); // DTLSv1.2
       uint16 epoch;                         // DTLS-related field
       uint48 sequence_number;               // DTLS-related field
       uint16 length;
       opaque encrypted_record[length];
} DTLSCiphertext;
```

type: The content type of the record.

legacy_record_version: This field is redundant and it is treated in the same way as specified in the TLS 1.3 specification. The DTLS version 1.2 version number is reused, namely { 254, 253 }. This field is deprecated and MUST be ignored.

epoch: A counter value that is incremented on every cipher state change.

sequence_number: The sequence number for this record.

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

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

4.1. Sequence Number Handling

DTLS uses an 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. For instance, if a handshake message from epoch 0 is retransmitted, it might have a sequence number after a message from epoch 1, even if the message from epoch 1 was transmitted first. Note that some care needs to be taken during the handshake to ensure that retransmitted messages use the right epoch and keying material.

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 5.9. In order to ensure that any given sequence/epoch pair is unique, implementations MUST NOT allow the same epoch value to be reused within two times the TCP maximum segment lifetime.

Note that because DTLS records may be reordered, a record from epoch 1 may be received after epoch 2 has begun. In general, implementations SHOULD discard packets from earlier epochs, but if packet loss causes noticeable problems they 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, not that they attempt to interrogate the MSL that the system TCP stack is using.) Until the handshake has completed, implementations MUST accept packets from the old epoch.

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

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. Transport Layer Mapping

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.

Note that unlike IPsec, DTLS records do not contain any association identifiers. 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.

Multiple DTLS records may be placed in a single datagram. They are simply encoded consecutively. The DTLS record framing is sufficient to determine the boundaries. Note, however, that the first byte of the datagram payload must be the beginning of a record. Records may not span datagrams.

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.

4.3. PMTU Issues

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

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

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

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

- The DTLS handshake messages can exceed the PMTU.

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

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

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

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

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

The DTLS record layer SHOULD 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.

4.4. 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.

4.4.1. Anti-Replay

DTLS records contain 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 receiver packet counter for this session MUST be initialized to zero when the 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 life of this session. This SHOULD be the first check applied to a packet after it has been matched to a session, to speed rejection of duplicate records.

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.) A minimum window size of 32 MUST be supported, but a window size of 64 is preferred and SHOULD be employed as the default. Another window size (larger than the minimum) MAY be chosen by the receiver. (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 this session. Records that contain sequence numbers lower than the "left" edge of the window are rejected. Packets falling within the window are checked against a list of received packets within the window. An efficient means for performing this check, based on the use of a bit mask, is described in Section 3.4.3 of [RFC4303].

If the received record falls within the window and is new, or if the packet is to the right of the window, then the receiver proceeds to MAC verification. If the MAC validation fails, the receiver MUST discard the received record as invalid. The receive window is updated only if the MAC verification succeeds.

4.4.2. 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.

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

5. The DTLS Handshake Protocol

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

1. To handle message loss, reordering, and fragmentation modifications to the handshake header are necessary.

2. Retransmission timers are introduced to handle message loss.

3. The TLS 1.3 KeyUpdate message is not used in DTLS 1.3 for re-keying.

4. A new ACK message has been added for reliable message delivery of certain handshake messages.

Note that TLS 1.3 already supports a cookie extension, which 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.

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 ciphersuite 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 [RFC5996]. 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 defence against DoS attacks mounted from valid IP addresses.

The DTLS 1.3 specification changes the way how cookies are exchanged compared to DTLS 1.2. DTLS 1.3 re-uses the HelloRetryRequest message and conveys the cookie to the client via an extension. The client receiving the cookie uses the same extension to place the cookie subsequently into a ClientHello message.
DTLS 1.2 on the other hand used a separate message, namely the HelloVerifyRequest, to pass a cookie to the client and did not utilize the extension mechanism. For backwards compatibility reason 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 2. 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 2: DTLS Exchange with HelloRetryRequest contain the Cookie Extension

The cookie extension is defined in Section 4.2.2 of [I-D.ietf-tls-tls13]. When sending the initial ClientHello, the client does not have a cookie yet. In this case, the cookie extension is omitted and the legacy_cookie field in the ClientHello message SHOULD be set to a zero length vector (i.e., a single zero byte length field) and MUST be ignored by a server negotiating DTLS 1.3.

When responding to a HelloRetryRequest, the client MUST create a new ClientHello message following the description in Section 4.1.2 of [I-D.ietf-tls-tls13].

The server SHOULD use information received in the ClientHello to generate its cookie, such as version, random, ciphersuites. The server MUST use the same version number in the HelloRetryRequest that it would use when sending a ServerHello. Upon receipt of the ServerHello, the client MUST verify that the server version values match and MUST terminate the connection with an "illegal_parameter" alert otherwise.

If the HelloRetryRequest message is used, the initial ClientHello and the HelloRetryRequest are included in the calculation of the handshake_messages (for the CertificateVerify message) and verify_data (for the Finished message). However, the computation of the message hash for the HelloRetryRequest is done according to the description in Section 4.4.1 of [I-D.ietf-tls-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.

One potential attack on this scheme is for the attacker to collect a number of cookies from different addresses 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 that legitimate clients be able to handshake through the transition (e.g., they 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. [RFC5996] 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 those cookies that were not generated within a certain amount of time.

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

If a server receives a ClientHello with an invalid cookie, it MUST NOT respond with a HelloRetryRequest. Restarting the handshake from scratch, without a cookie, allows the client to recover from a situation where it obtained a cookie that cannot be verified by the server. As described in Section 4.1.4 of [I-D.ietf-tls-tls13],clients SHOULD also 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(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_RESERVED(24),
    ack([[TBD RFC Editor -- Proposal: 25]]),
    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 hello_retry_request:   HelloRetryRequest;
        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; /* reserved */
        case ack:                   ACK; /* DTLS-required field */
    } body;
} Handshake;
```

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.

A DTLS 1.3 MUST NOT use the KeyUpdate message to change keying material used for the protection of traffic data. Instead the epoch field is used, which is explained in Section 5.9.

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<0..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 [I-D.ietf-tls-tls13]) and the legacy_version field MUST be set to {254, 253}, which was the version number for DTLS 1.2.

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.

cipher_suites: Same as for TLS 1.3

legacy_compression_methods: Same as for TLS 1.3

extensions: Same as for TLS 1.3

The first message each side transmits in each handshake 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.

Here is an example:

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

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

                             X<-----      HelloRetryRequest
                             (lost)         (message_seq=0)


[Timer Expires]

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


                         <--------        HelloRetryRequest
                                            (message_seq=0)

ClientHello              -------->
(message_seq=1)
  +cookie

                         <--------              ServerHello
                                            (message_seq=1)
                                        EncryptedExtensions
                                            (message_seq=2)
                                                Certificate
                                            (message_seq=3)
                                          CertificateVerify
                                            (message_seq=4)
                                                   Finished
                                            (message_seq=5)

Certificate             -------->
(message_seq=2)
CertificateVerify
(message_seq=3)
Finished
(message_seq=4)

                    <--------                          Ack
                                            (message_seq=6)
```

Figure 3: Example DTLS Exchange illustrating Message Loss

From the perspective of the DTLS record layer, the retransmission is a new record. This record will have a new DTLSPlaintext.sequence_number value.

DTLS implementations maintain (at least notionally) a next_receive_seq counter. This counter is initially set to zero. When a message is received, if its sequence number 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).

5.3. ACK Message

```
struct {} ACK;
```

The ACK handshake message is used by an endpoint to respond to a message where the TLS 1.3 handshake does not foresee such return message. With the use of the ACK message the sender is able to determine whether a transmitted request has been lost and needs to be retransmitted. Since the ACK message does not contain any correlation information the sender MUST only have one such message outstanding at a time.

The ACK message uses a handshake content type and is encrypted under the appropriate application traffic key. [[OPEN ISSUE: It seems odd to have the ACK that responds to CFIN encrypted under the application key. Also, what do you do about ACKs that have to deal with key changes.]]

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. These ranges MUST NOT be larger than the maximum handshake fragment size and MUST jointly contain the entire handshake message. 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.

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. Timeout and Retransmission

DTLS messages are grouped into a series of message flights, according to the diagrams below. Although each flight of messages may consist of a number of messages, they should be viewed as monolithic for the purpose of timeout and retransmission.

```
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 4: Message Flights for 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 5: 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*]


                                                               +----------+
 (EndOfEarlyData)                                              | Flight 3 |
 {Finished}            -------->                               +----------+
 [Application Data*]
                                                               +----------+
                       <--------                        [Ack]  | Flight 4 |
                                          [Application Data*]  +----------+

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

Figure 6: Message Flights for the Zero-RTT Handshake

```
Client                                                Server

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

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

Figure 7: Message Flights for New Session Ticket Message

```
Client                                              Server

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

[Certificate]                                               +----------+
[CertificateVerify]                                         | Flight 2 |
[Finished]             -------->                            +----------+
```

Figure 8: Message Flights for Post-Handshake Authentication (Success)

```
Client                                              Server

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

[Certificate]                                               +----------+
[Finished]             -------->                            | Flight 2 |
                                                            +----------+
```

Figure 9: Message Flights for Post-Handshake Authentication (Decline)

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

5.5.1. 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             |
            flight |  +--------+                         |
                   |  |        | Set retransmit timer    |
                   |  |       \|/                        |
                   |  |  +-----------+                   |
                   |  |  |           |                   |
                   +--)--|  WAITING  |-------------------+
                   |  |  |           |   Timer expires   |
                   |  |  +-----------+                   |
                   |  |         |                        |
                   |  |         |                        |
                   |  |         +------------------------+
                   |  |                Read retransmit
           Receive |  |
              last |  |
            flight |  |
                   |  |
                  \|/\|/

               +-----------+
               |           |
               | FINISHED  |
               |           |
               +-----------+
                   |  /|\
                   |   |
                   |   |
                   +---+

             Server read retransmit
                 Retransmit ACK
```

Figure 10: DTLS Timeout and Retransmission State Machine

The state machine has three basic states.

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. Once the messages have been sent, the implementation then enters the FINISHED state if this is the last flight in the handshake. Or, if the implementation expects to receive more messages, it sets a retransmit timer and then enters the WAITING state.

There are three 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 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.

3. The implementation receives the next flight of messages: if this is the final flight of messages, the implementation transitions to FINISHED. If the implementation needs to send a new flight, it transitions to the PREPARING state. Partial reads (whether partial messages or only some of the messages in the flight) do not cause state transitions or timer resets.

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

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

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

5.5.2. 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 we recommend a 100 msec timer 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. One situation where this might occur is when a rehandshake is used after substantial data transfer.

5.6. 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 [I-D.ietf-tls-tls13], respectively.

5.7. 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.

5.8. 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.

5.9. Epoch Values and Rekeying

A recipient of a DTLS message needs to select the correct keying material in order to process an incoming message. With the possibility of message loss and re-order an identifier is needed to determine which cipher state has been used to protect the record payload. The epoch value fulfills this role in DTLS. In addition to the key derivation steps described in Section 7 of [I-D.ietf-tls-tls13] triggered by the states during the handshake a sender may want to rekey at any time during the lifetime of the connection and has to have a way to indicate that it is updating its sending cryptographic keys.

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

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

- epoch value (1) is used for messages protected using keys derived from early_traffic_secret. This includes early data sent by the client and the EndOfEarlyData message.

- epoch value (2) is used for messages protected using keys derived from the 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 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 traffic_secret_N (N>0).

Using these reserved epoch values a receiver knows what cipher state has been used to encrypt and integrity protect a message. Implementations that receive a payload with an epoch value for which no corresponding cipher state can be determined MUST generate a "unexpected_message" alert. For example, client incorrectly uses epoch value 5 when sending early application data in a 0-RTT exchange. A server will not be able to compute the appropriate keys and will therefore have to respond with an alert.

Increasing the epoch value by a sender (starting with value 4 upwards) corresponds semantically to rekeying using the KeyUpdate message in TLS 1.3. Instead of utilizing an dedicated message in DTLS 1.3 the sender uses an increase in the epoch value to signal rekeying. Hence, a sender that decides to increment the epoch value MUST send all its traffic using the next generation of keys, computed as described in Section 7.2 of [I-D.ietf-tls-tls13]. Upon receiving a payload with such a new epoch value, the receiver MUST update their receiving keys and if they have not already updated their sending state up to or past the then current receiving generation MUST send messages with the new epoch value prior to sending any other messages. For epoch values lower than 4 the key schedule described in Section 7.1 of [I-D.ietf-tls-tls13] is applicable. As a difference to the functionality of the KeyUpdate in TLS 1.3 the sender forces the receiver to increase the epoch value for outgoing data as well.

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 [I-D.ietf-tls-tls13].

Figure 11 illustrates the epoch values in an example DTLS handshake.

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

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

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

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

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

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

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

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

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

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

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

```
   [Ack]                     -------->
   (epoch=3)

                            Some time later ...
                              (Rekeying)

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

Figure 11: Example DTLS Exchange with Epoch Information

6. 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.

7. Security Considerations

Security issues are discussed primarily in [I-D.ietf-tls-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.

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

8. Changes to DTLS 1.2

Since TLS 1.3 introduce 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

- Support for AEAD-only ciphers

- 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
- Removed support for weaker and older cryptographic algorithms
- Improved version negotation

9. IANA Considerations

IANA is requested to allocate a new value in the TLS HandshakeType Registry for the ACK message defined in Section 5.3.

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-tls13-19 (work in progress), March 2017.

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

[RFC1191] Mogul, J. and S. Deering, "Path MTU discovery", RFC 1191, DOI 10.17487/RFC1191, November 1990, <http://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, <http://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", RFC 4443, DOI 10.17487/RFC4443, March 2006, <http://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, <http://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, <http://www.rfc-editor.org/info/rfc6298>.

10.2. Informative References

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

[RFC4303] Kent, S., "IP Encapsulating Security Payload (ESP)", RFC 4303, DOI 10.17487/RFC4303, December 2005, <http://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, <http://www.rfc-editor.org/info/rfc4340>.

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

[RFC5996] Kaufman, C., Hoffman, P., Nir, Y., and P. Eronen, "Internet Key Exchange Protocol Version 2 (IKEv2)", RFC 5996, DOI 10.17487/RFC5996, September 2010, <http://www.rfc-editor.org/info/rfc5996>.

[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, <http://www.rfc-editor.org/info/rfc7525>.

10.3. URIs

[1] mailto:tls@ietf.org

Appendix A. History

RFC EDITOR: PLEASE REMOVE THE THIS SECTION

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 B. Working Group Information

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

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

Appendix C. Contributors

Many people have contributed to previous DTLS versions and they are acknowledged in prior versions of DTLS specifications.

For this version of the document we would like to thank:

* Ilari Liusvaara
  Independent
  ilariliusvaara@welho.com

* Martin Thomson
  Mozilla
  martin.thomson@gmail.com

Authors' Addresses

Eric Rescorla
RTFM, Inc.

EMail: ekr@rtfm.com

Hannes Tschofenig
ARM Limited

EMail: hannes.tschofenig@arm.com

Nagendra Modadugu
Google, Inc.

EMail: nagendra@cs.stanford.edu

Network Working Group Y. Sheffer
Internet-Draft Intuit
Intended status: Experimental D. Migault
Expires: December 28, 2019 Ericsson
June 26, 2019

TLS Server Identity Pinning with Tickets
draft-sheffer-tls-pinning-ticket-12

Abstract

Misissued public-key certificates can prevent TLS clients from appropriately authenticating the TLS server. Several alternatives have been proposed to detect this situation and prevent a client from establishing a TLS session with a TLS end point authenticated with an illegitimate public-key certificate. These mechanisms are either not widely deployed or limited to public web browsing.

This document proposes experimental extensions to TLS with opaque pinning tickets as a way to pin the server's identity. During an initial TLS session, the server provides an original encrypted pinning ticket. In subsequent TLS session establishment, upon receipt of the pinning ticket, the server proves its ability to decrypt the pinning ticket and thus the ownership of the pinning protection key. The client can now safely conclude that the TLS session is established with the same TLS server as the original TLS session. One of the important properties of this proposal is that no manual management actions are required.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

Misissued public-key certificates can prevent TLS [RFC8446] clients from appropriately authenticating the TLS server. This is a significant risk in the context of the global public key infrastructure (PKI), and similarly for large scale deployments of certificates within enterprises.

This document proposes experimental extensions to TLS with opaque pinning tickets as a way to pin the server's identity. The approach is intended to be easy to implement and deploy, and reuses some of the ideas behind TLS session resumption [RFC5077].

Ticket pinning is a second factor server authentication method and is not proposed as a substitute for the authentication method provided in the TLS key exchange. More specifically, the client only uses the pinning identity method after the TLS key exchange is successfully completed. In other words, the pinning identity method is only performed over an authenticated TLS session. Note that Ticket Pinning does not pin certificate information and therefore is truly an independent second factor authentication.

Ticket pinning is a Trust On First Use (TOFU) mechanism, in that the first server authentication is only based on PKI certificate validation, but for any follow-on sessions, the client is further ensuring the server's identity based on the server's ability to decrypt the ticket, in addition to normal PKI certificate authentication.

During initial TLS session establishment, the client requests a pinning ticket from the server. Upon receiving the request the server generates a pinning secret which is expected to be unpredictable for peers other than the client or the server. In our case, the pinning secret is generated from parameters exchanged during the TLS key exchange, so client and server can generate it locally and independently. The server constructs the pinning ticket with the necessary information to retrieve the pinning secret. The server then encrypts the ticket and returns the pinning ticket to the client with an associated pinning lifetime.

The pinning lifetime value indicates for how long the server promises to retain the server-side ticket-encryption key, which allows it to complete the protocol exchange correctly and prove its identity. The server commitment (and ticket lifetime) is typically on the order of weeks.

Once the key exchange is completed and the server is deemed authenticated, the client generates locally the pinning secret and caches the server's identifiers to index the pinning secret as well as the pinning ticket and its associated lifetime.

When the client re-establishes a new TLS session with the server, it sends the pinning ticket to the server. Upon receiving it, the server returns a proof of knowledge of the pinning secret. Once the key exchange is completed and the server has been authenticated, the client checks the pinning proof returned by the server using the client's stored pinning secret. If the proof matches, the client can conclude that the server it is currently connecting to is in fact the correct server.

This document only applies to TLS 1.3. We believe that the idea can also be back-fitted into earlier versions of the protocol, but this would require significant changes. One example is that TLS 1.2 [RFC5246] and earlier versions do not provide a generic facility of encrypted handshake extensions, such as is used here to transport the ticket.

The main advantages of this protocol over earlier pinning solutions are:

- The protocol is at the TLS level, and as a result is not restricted to HTTP at the application level.

- The protocol is robust to server IP, Certificate Authority (CA), and public key changes. The server is characterized by the ownership of the pinning protection key, which is never provided to the client. Server configuration parameters such as the CA and the public key may change without affecting the pinning ticket protocol.

- Once a single parameter is configured (the ticket’s lifetime), operation is fully automated. The server administrator need not bother with the management of backup certificates or explicit pins.

- For server clusters, we reuse the existing [RFC5077] infrastructure where it exists.

- Pinning errors, presumably resulting from man-in-the-middle (MITM) attacks, can be detected both by the client and the server. This allows for server-side detection of MITM attacks using large-scale analytics, and with no need to rely on clients to explicitly report the error.

A note on terminology: unlike other solutions in this space, we do not do "certificate pinning" (or "public key pinning"), since the protocol is oblivious to the server’s certificate. We prefer the term "server identity pinning" for this new solution. In our solution, the server proves its identity by generating a proof that it can read and decrypt an encrypted ticket. As a result, the identity proof relies on proof of ownership of the pinning protection key. However, this key is never exchanged with the client or known by it, and so cannot itself be pinned.

1.1. Conventions used in this document

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.

1.2. Scope of Experimentation

This document describes an experimental extension to the TLS protocol. This section defines constraints on this experiment and how it can yield useful information, potentially resulting in a standard.

The protocol is designed so that if the server does not support it, the client and server fall back to a normal TLS exchange, with the exception of a single PinningTicket extension being initially sent by the client. In addition, the protocol is designed to only strengthen the validation of the server's identity ("second factor"). As a result, implementation or even protocol errors should not result in weakened security compared to the normal TLS exchange. Given these two points, experimentation can be run on the open Internet between consenting client and server implementations.

The goal of the experiment is to prove that:

- Non-supporting clients and servers are unaffected.

- Connectivity between supporting clients and servers is retained under normal circumstances, whether the client connects to the server frequently (relative to the ticket's lifetime) or very rarely.

- Enterprise middleboxes do not interrupt such connectivity.

- Misissued certificates and rogue TLS-aware middleboxes do result in broken connectivity, and these cases are detected on the client and/or server side. Clients and servers can be recovered even after such events and the normal connectivity restored.

Following two years of successful deployment, the authors will publish a document that summarizes the experiment's findings and will resubmit the protocol for consideration as a Proposed Standard.

2. Protocol Overview

The protocol consists of two phases: the first time a particular client connects to a server, and subsequent connections.

This protocol supports full TLS handshakes, as well as 0-RTT handshakes. Below we present it in the context of a full handshake, but behavior in 0-RTT handshakes should be identical.

The document presents some similarities with the ticket resumption mechanism described in [RFC5077]. However the scope of this document differs from session resumption mechanisms implemented with [RFC5077] or with other mechanisms. Specifically, the pinning ticket does not carry any state associated with a TLS session and thus cannot be used for session resumption, or to authenticate the client. Instead, the pinning ticket only contains the encrypted pinning secret. The pinning ticket is used by the server to prove its ability to decrypt it, which implies ownership of the pinning protection key.

[RFC5077] has been obsoleted by [RFC8446] and ticket resumption is now defined by Sec. 2.2 of [RFC8446]. This document references [RFC5077] as an informational document since it contains a more thorough discussion of stateless ticket resumption and because ticket resumption benefits from significant operational experience with TLS 1.2 that is still widely deployed at the time of writing this document. This experience as well as deployment can easily be re-used for identity pinning.

With TLS 1.3, session resumption is based on a preshared key (PSK). This is orthogonal to this protocol. With TLS 1.3, a TLS session can be established using PKI and a pinning ticket, and later resumed with PSK.

However, the protocol described in this document addresses the problem of misissued certificates. Thus, it is not expected to be used outside a certificate-based TLS key exchange, such as in PSK. As a result, PSK handshakes MUST NOT include the extension defined here.

2.1. Initial Connection

When a client first connects to a server, it requests a pinning ticket by sending an empty PinningTicket extension, and receives it as part of the server's first response, in the returned PinningTicket extension.

```
    Client                                               Server

    ClientHello
      + key_share
      + signature_algorithms
      + PinningTicket         -------->
                                                    ServerHello
                                                    + key_share
                                         {EncryptedExtensions
                                              + PinningTicket}
                                         {CertificateRequest*}
                                                {Certificate*}
                                          {CertificateVerify*}
                              <--------              {Finished}
    {Certificate*}
    {CertificateVerify*}
    {Finished}                -------->
    [Application Data]        <------->      [Application Data]

           *  Indicates optional or situation-dependent
              messages that are not always sent.

           {} Indicates messages protected using keys
              derived from the ephemeral secret.

           [] Indicates messages protected using keys
              derived from the master secret.
```

If a client supports the PinningTicket extension and does not have any pinning ticket associated with the server, the exchange is considered as an initial connection. Other reasons the client may not have a pinning ticket include the client having flushed its pinning ticket store, or the committed lifetime of the pinning ticket having expired.

Upon receipt of the PinningTicket extension, the server computes a pinning secret (Section 4.1), and sends the pinning ticket (Section 4.2) encrypted with the pinning protection key (Section 4.3). The pinning ticket is associated with a lifetime value by which the server assumes the responsibility of retaining the pinning protection key and being able to decrypt incoming pinning tickets during the period indicated by the committed lifetime.

Once the pinning ticket has been generated, the server returns the pinning ticket and the committed lifetime in a PinningTicket extension embedded in the EncryptedExtensions message. We note that a PinningTicket extension MUST NOT be sent as part of a HelloRetryRequest.

Upon receiving the pinning ticket, the client MUST NOT accept it until the key exchange is completed and the server authenticated. If the key exchange is not completed successfully, the client MUST ignore the received pinning ticket. Otherwise, the client computes the pinning secret and SHOULD cache the pinning secret and the pinning ticket for the duration indicated by the pinning ticket lifetime. The client SHOULD clean up the cached values at the end of the indicated lifetime.

2.2. Subsequent Connections

When the client initiates a connection to a server it has previously seen (see Section 2.3 on identifying servers), it SHOULD send the pinning ticket for that server. The pinning ticket, pinning secret and pinning ticket lifetime computed during the establishment of the previous TLS session are designated in this document as the "original" ones, to distinguish them from a new ticket that may be generated during the current session.

The server MUST extract the original pinning_secret value from the ticket and MUST respond with a PinningTicket extension, which includes:

- A proof that the server can understand the ticket that was sent by the client; this proof also binds the pinning ticket to the server's (current) public key, as well as the ongoing TLS session. The proof is mandatory and MUST be included if a pinning ticket was sent by the client.

- A fresh pinning ticket. The main reason for refreshing the ticket on each connection is privacy: to avoid the ticket serving as a fixed client identifier. While a fresh pinning ticket might be of zero length, it is RECOMMENDED to include a fresh ticket with a non zero length with each response.

If the server cannot validate the received ticket, that might indicate an earlier MITM attack on this client. The server MUST then abort the connection with a handshake_failure alert, and SHOULD log this failure.

The client MUST verify the proof, and if it fails to do so, MUST issue a handshake_failure alert and abort the connection (see also Section 7.5). It is important that the client does not attempt to "fall back" by omitting the PinningTicket extension.

When the connection is successfully set up, i.e. after the Finished message is verified, the client SHOULD store the new ticket along with the corresponding pinning_secret, replacing the original ticket.

Although this is an extension, if the client already has a ticket for a server, the client MUST interpret a missing PinningTicket extension in the server's response as an attack, because of the server's prior commitment to respect the ticket. The client MUST abort the connection in this case. See also Section 5.5 on ramping down support for this extension.

2.3. Indexing the Pins

Each pin is associated with a set of identifiers which include among others host name, protocol (TLS or DTLS) and port number. In other words, the pin for port TCP/443 may be different from that for DTLS or from the pin for port TCP/8443. These identifiers are expected to be relevant to characterize the identity of the server as well as the establishing TLS session. When a host name is used, it MUST be the value sent inside the Server Name Indication (SNI) extension. This definition is similar to a Web Origin [RFC6454], but does not assume the existence of a URL.

The purpose of ticket pinning is to pin the server identity. As a result, any information orthogonal to the server's identity MUST NOT be considered in indexing. More particularly, IP addresses are ephemeral and forbidden in SNI and therefore pins MUST NOT be associated with IP addresses. Similarly, CA names or public keys associated with server MUST NOT be used for indexing as they may change over time.

3. Message Definitions

This section defines the format of the PinningTicket extension. We follow the message notation of [RFC8446].

```
 opaque pinning_ticket<0..2^16-1>;

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

 struct {
   select (Role) {
     case client:
       pinning_ticket ticket<0..2^16-1>; //omitted on 1st connection

     case server:
       pinning_proof proof<0..2^8-1>; //no proof on 1st connection
       pinning_ticket ticket<0..2^16-1>; //omitted on ramp down
       uint32 lifetime;
   }
} PinningTicketExtension;
```

ticket a pinning ticket sent by the client or returned by the server. The ticket is opaque to the client. The extension MUST contain exactly 0 or 1 tickets.

proof a demonstration by the server that it understands the received ticket and therefore that it is in possession of the secret that was used to generate it originally. The extension MUST contain exactly 0 or 1 proofs.

lifetime the duration (in seconds) that the server commits to accept offered tickets in the future.

4. Cryptographic Operations

This section provides details on the cryptographic operations performed by the protocol peers.

4.1. Pinning Secret

The pinning secret is generated locally by the client and the server which means they must use the same inputs to generate it. This value must be generated before the ServerHello message is sent, as the server includes the corresponding pinning ticket in the same flight as the ServerHello message. In addition, the pinning secret must be unpredictable to any party other than the client and the server.

The pinning secret is derived using the Derive-Secret function provided by TLS 1.3, described in Section "Key Schedule" of [RFC8446].

```
pinning secret = Derive-Secret(Handshake Secret, "pinning secret",
         ClientHello...ServerHello)
```

4.2. Pinning Ticket

The pinning ticket contains the pinning secret. The pinning ticket is provided by the client to the server which decrypts it in order to extract the pinning secret and responds with a pinning proof. As a result, the characteristics of the pinning ticket are:

- Pinning tickets MUST be encrypted and integrity-protected using strong cryptographic algorithms.

- Pinning tickets MUST be protected with a long-term pinning protection key.

- Pinning tickets MUST include a pinning protection key ID or serial number as to enable the pinning protection key to be refreshed.

- The pinning ticket MAY include other information, in addition to the pinning secret. When additional information is included, a careful review needs to be performed to evaluate its impact on privacy.

The pinning ticket's format is not specified by this document, but we RECOMMEND a format similar to the one proposed by [RFC5077].

4.3. Pinning Protection Key

The pinning protection key is only used by the server and so remains server implementation specific. [RFC5077] recommends the use of two keys, but when using AEAD algorithms only a single key is required.

When a single server terminates TLS for multiple virtual servers using the Server Name Indication (SNI) mechanism, we strongly RECOMMEND to use a separate protection key for each one of them, in order to allow migrating virtual servers between different servers while keeping pinning active.

As noted in Section 5.1, if the server is actually a cluster of machines, the protection key MUST be synchronized between all the nodes that accept TLS connections to the same server name. When [RFC5077] is deployed, an easy way to do it is to derive the protection key from the session-ticket protection key, which is already synchronized. For example:

```
pinning_protection_key = HKDF-Expand(resumption_protection_key,
                                     "pinning protection", L)
```

Where resumption_protection_key is the ticket protection key defined in [RFC5077]. Both resumption_protection_key and pinning_protection_key are only used by the server.

The above solution attempts to minimize code changes related to management of the resumption_protection_key. The drawback is that this key would be used both to directly encrypt session tickets and to derive the pinning_protection_key, and such mixed usage of a single key is not in line with cryptographic best practices. Where possible, we RECOMMEND to have the resumption_protection_key and pinning_protection_key as two, unrelated keys that are separately shared among the relevant servers.

4.4. Pinning Proof

The pinning proof is sent by the server to demonstrate that it has been able to decrypt the pinning ticket and retrieve the pinning secret. The proof must be unpredictable and must not be replayed.

Similarly to the pinning ticket, the pinning proof is sent by the server in the ServerHello message. In addition, it must not be possible for a MITM server with a fake certificate to obtain a pinning proof from the original server.

In order to address these requirements, the pinning proof is bound to the TLS session as well as the public key of the server:

```
pinning_proof_secret=Derive-Secret(Handshake Secret, "pinning proof 1",
            ClientHello...ServerHello)

proof = HMAC(original_pinning_secret, "pinning proof 2" +
            pinning_proof_secret + Hash(server_public_key))
```

where HMAC [RFC2104] uses the Hash algorithm that was negotiated in the handshake, and the same hash is also used over the server's public key. The original_pinning_secret value refers to the secret value extracted from the ticket sent by the client, to distinguish it from a new pinning secret value that is possibly computed in the current exchange. The server_public_key value is the DER representation of the public key, specifically the SubjectPublicKeyInfo structure as-is.

5. Operational Considerations

The main motivation behind the current protocol is to enable identity pinning without the need for manual operations. Manual operations are susceptible to human error and in the case of public key pinning, can easily result in "server bricking": the server becoming inaccessible to some or all of its users. To achieve this goal operations described in identity pinning are only performed within the current TLS session, and there is no dependence on any TLS configuration parameters such as CA identity or public keys. As a result, configuration changes are unlikely to lead to desynchronized state between the client and the server.

5.1. Protection Key Synchronization

The only operational requirement when deploying this protocol is that if the server is part of a cluster, protection keys (the keys used to encrypt tickets) MUST be synchronized between all cluster members. The protocol is designed so that if resumption ticket protection keys [RFC5077] are already synchronized between cluster members, nothing more needs to be done.

Moreover, synchronization does not need to be instantaneous, e.g. protection keys can be distributed a few minutes or hours in advance of their rollover. In such scenarios, each cluster member MUST be able to accept tickets protected with a new version of the protection key, even while it is still using an old version to generate keys. This ensures that a client that receives a "new" ticket does not next hit a cluster member that still rejects this ticket.

Misconfiguration can lead to the server’s clock being off by a large amount of time. Consider a case where a server’s clock is misconfigured, for example, to be 1 year in the future, and the system is allowed to delete expired keys automatically. The server will then delete many outstanding keys because they are now long expired and will end up rejecting valid tickets that are stored by clients. Such a scenario could make the server inaccessible to a large number of clients.

The decision to delete a key should at least consider the largest value of the ticket lifetime as well as the expected time desynchronisation between the servers of the cluster and the time difference for distributing the new key among the different servers in the cluster.

5.2. Ticket Lifetime

The lifetime of the ticket is a commitment by the server to retain the ticket’s corresponding protection key for this duration, so that the server can prove to the client that it knows the secret embedded in the ticket. For production systems, the lifetime SHOULD be between 7 and 31 days.

5.3. Certificate Renewal

The protocol ensures that the client will continue speaking to the correct server even when the server’s certificate is renewed. In this sense, pinning is not associated with certificates which is the reason we designate the protocol described in this document as "server identity pinning".

Note that this property is not impacted by the use of the server’s public key in the pinning proof, because the scope of the public key used is only the current TLS session.

5.4. Certificate Revocation

The protocol is orthogonal to certificate validation in the sense that, if the server’s certificate has been revoked or is invalid for some other reason, the client MUST refuse to connect to it regardless of any ticket-related behavior.

5.5. Disabling Pinning

A server implementing this protocol MUST have a "ramp down" mode of operation where:

- The server continues to accept valid pinning tickets and responds correctly with a proof.

- The server does not send back a new pinning ticket.

After a while no clients will hold valid tickets any more and the feature may be disabled. Note that clients that do not receive a new pinning ticket do not necessarily need to remove the original ticket. Instead, the client may keep on using the ticket until its lifetime expires. However, as detailed in section Section 7.7, re-use of a ticket by the client may result in privacy concerns as the ticket value may be used to correlate TLS sessions.

Issuing a new pinning ticket with a shorter lifetime would only delay the ramp down process, as the shorter lifetime can only affect clients that actually initiated a new connection. Other clients would still see the original lifetime for their pinning tickets.

5.6. Server Compromise

If a server compromise is detected, the pinning protection key MUST be rotated immediately, but the server MUST still accept valid tickets that use the old, compromised key. Clients that still hold old pinning tickets will remain vulnerable to MITM attacks, but those that connect to the correct server will immediately receive new tickets protected with the newly generated pinning protection key.

The same procedure applies if the pinning protection key is compromised directly, e.g. if a backup copy is inadvertently made public.

5.7. Disaster Recovery

All web servers in production need to be backed up, so that they can be recovered if a disaster (including a malicious activity) ever wipes them out. Backup often includes the certificate and its private key, which must be backed up securely. The pinning secret, including earlier versions that are still being accepted, must be backed up regularly. However since it is only used as an authentication second factor, it does not require the same level of confidentiality as the server’s private key.

Readers should note that [RFC5077] session resumption keys are more security sensitive, and should normally not be backed up but rather treated as ephemeral keys. Even when servers derive pinning secrets from resumption keys (Section 4.1), they MUST NOT back up resumption keys.

6. Implementation Status

Note to RFC Editor: please remove this section before publication, including the reference to [RFC7942].

This section records the status of known implementations of the protocol defined by this specification at the time of posting of this Internet-Draft, and is based on a proposal described in [RFC7942]. The description of implementations in this section is intended to assist the IETF in its decision processes in progressing drafts to RFCs. Please note that the listing of any individual implementation here does not imply endorsement by the IETF. Furthermore, no effort has been spent to verify the information presented here that was supplied by IETF contributors. This is not intended as, and must not be construed to be, a catalog of available implementations or their features. Readers are advised to note that other implementations may exist.

According to RFC 7942, "this will allow reviewers and working groups to assign due consideration to documents that have the benefit of running code, which may serve as evidence of valuable experimentation and feedback that have made the implemented protocols more mature. It is up to the individual working groups to use this information as they see fit".

6.1. Mint Fork

6.1.1. Overview

A fork of the Mint TLS 1.3 implementation, developed by Yaron Sheffer and available at https://github.com/yaronf/mint.

6.1.2. Description

This is a fork of the TLS 1.3 implementation, and includes client and server code. In addition to the actual protocol, several utilities are provided allowing to manage pinning protection keys on the server side, and pinning tickets on the client side.

6.1.3. Level of Maturity

This is a prototype.

6.1.4. Coverage

The entire protocol is implemented.

6.1.5. Version Compatibility

The implementation is compatible with draft-sheffer-tls-pinning-ticket-02.

6.1.6. Licensing

Mint itself and this fork are available under an MIT license.

6.1.7. Contact Information

See author details below.

7. Security Considerations

This section reviews several security aspects related to the proposed extension.

7.1. Trust on First Use (TOFU) and MITM Attacks

This protocol is a "trust on first use" protocol. If a client initially connects to the "right" server, it will be protected against MITM attackers for the lifetime of each received ticket. If it connects regularly (depending of course on the server-selected lifetime), it will stay constantly protected against fake certificates.

However if it initially connects to an attacker, subsequent connections to the "right" server will fail. Server operators might want to advise clients on how to remove corrupted pins, once such large scale attacks are detected and remediated.

The protocol is designed so that it is not vulnerable to an active MITM attacker who has real-time access to the original server. The pinning proof includes a hash of the server's public key, to ensure the client that the proof was in fact generated by the server with which it is initiating the connection.

7.2. Pervasive Monitoring

Some organizations, and even some countries perform pervasive monitoring on their constituents [RFC7258]. This often takes the form of always-active SSL proxies. Because of the TOFU property, this protocol does not provide any security in such cases.

Pervasive monitoring may also result in privacy concerns detailed in section Section 7.7.

7.3. Server-Side Error Detection

Uniquely, this protocol allows the server to detect clients that present incorrect tickets and therefore can be assumed to be victims of a MITM attack. Server operators can use such cases as indications of ongoing attacks, similarly to fake certificate attacks that took place in a few countries in the past.

7.4. Client Policy and SSL Proxies

Like it or not, some clients are normally deployed behind an SSL proxy. Similarly to [RFC7469], it is acceptable to allow pinning to be disabled for some hosts according to local policy. For example, a User Agent (UA) MAY disable pinning for hosts whose validated certificate chain terminates at a user-defined trust anchor, rather than a trust anchor built-in to the UA (or underlying platform). Moreover, a client MAY accept an empty PinningTicket extension from such hosts as a valid response.

7.5. Client-Side Error Behavior

When a client receives a malformed or empty PinningTicket extension from a pinned server, it MUST abort the handshake and MUST NOT retry with no PinningTicket in the request. Doing otherwise would expose the client to trivial fallback attacks, similar to those described in [RFC7507].

This rule can however have negative affects on clients that move from behind SSL proxies into the open Internet and vice versa, if the advice in Section 7.4 is not followed. Therefore, we RECOMMEND that browser and library vendors provide a documented way to remove stored pins.

7.6. Stolen and Forged Tickets

Stealing pinning tickets even in conjunction with other pinning parameters, such as the associated pinning secret, provides no benefit to the attacker since pinning tickets are used to secure the client rather than the server. Similarly, it is useless to forge a ticket for a particular server.

7.7. Client Privacy

This protocol is designed so that an external attacker cannot correlate between different requests of a single client, provided the client requests and receives a fresh ticket upon each connection. This may be of concern particularly during ramp-down, if the server does not provide any new ticket and the client re-uses the same ticket. To reduce or avoid such privacy concerns, it is RECOMMENDED for the server to issue a fresh ticket with a reduced life time. This would at least reduce the time period under which TLS session of the client are correlated. The server MAY also issue tickets with a zero second lifetime until it is confident all tickets are expired.

On the other hand, the server to which the client is connecting can easily track the client. This may be an issue when the client expects to connect to the server (e.g., a mail server) with multiple identities. Implementations SHOULD allow the user to opt out of pinning, either in general or for particular servers.

This document does not define the exact content of tickets. Including client-specific information in tickets would raise privacy concerns and is NOT RECOMMENDED.

7.8. Ticket Protection Key Management

While the ticket format is not mandated by this document, we RECOMMEND using authenticated encryption to protect it. Some of the algorithms commonly used for authenticated encryption, e.g. GCM, are highly vulnerable to nonce reuse, and this problem is magnified in a cluster setting. Therefore implementations that choose AES-GCM or any AEAD equivalent MUST adopt one of these three alternatives:

- Partition the nonce namespace between cluster members and use monotonic counters on each member, e.g. by setting the nonce to the concatenation of the cluster member ID and an incremental counter.

- Generate random nonces but avoid the so-called birthday bound, i.e. never generate more than the maximum allowed number of encrypted tickets (2**64 for AES-128-GCM) for the same ticket pinning protection Key.

- An alternative design which has been attributed to Karthik Bhargavan is as follows. Start with a 128-bit master key "K_master" and then for each encryption, generate a 256-bit random nonce and compute: K = HKDF(K_master, Nonce || "key"), then N = HKDF(K_master, Nonce || "nonce"). Use these values to encrypt the ticket, AES-GCM(K, N, data). This nonce should then be stored and transmitted with the ticket.

8. IANA Considerations

IANA is requested to allocate a TicketPinning extension value in the TLS ExtensionType Registry.

[RFC8447] defines the procedure and requirements and the necessary information for the IANA to update the "TLS ExtensionType Values" registry [TLS-EXT].

According to [RFC8447] the update of the "TLS ExtensionType Values" registry is "Specification Required" [RFC8126] which is fulfilled by the current document, when it is published as an RFC.

The TicketPinning Extension is not limited to Private use and as such the TicketPinning Extension Value is expected to have its first byte in the range 0-254.

The TicketPinning Extension Name is expected to be ticket_pinning.

The TicketPinning Extension Recommended value should be set to "No" with the publication of the current document as "Experimental".

The TicketPinning Extension TLS.13 column should be set to CH, EE to indicate that the TicketPinning Extension is present in ClientHello and EncryptedExtensions messages.

9. Acknowledgements

The original idea behind this proposal was published in [Oreo] by Moti Yung, Benny Pinkas and Omer Berkman. The current protocol is but a distant relative of the original Oreo protocol, and any errors are the responsibility of the authors of this document alone.

We would like to thank Adrian Farrel, Dave Garrett, Daniel Kahn Gillmor, Alexey Melnikov, Yoav Nir, Eric Rescorla, Benjamin Kaduk and Rich Salz for their comments on this document. Special thanks to Craig Francis for contributing the HPKP deployment script, and to Ralph Holz for several fruitful discussions.

10. References

10.1. Normative References

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

[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>.

10.2. Informative References

[I-D.perrin-tls-tack] Marlinspike, M., "Trust Assertions for Certificate Keys", draft-perrin-tls-tack-02 (work in progress), January 2013.

[Netcraft] Mutton, P., "HTTP Public Key Pinning: You’re doing it wrong!", March 2016, <http://news.netcraft.com/archives/2016/03/30/http-public-key-pinning-youre-doing-it-wrong.html>.

[Oreo] Berkman, O., Pinkas, B., and M. Yung, "Firm Grip Handshakes: A Tool for Bidirectional Vouching", Cryptology and Network Security, pp. 142-157 , 2012.

[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>.

[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>.

[RFC6454] Barth, A., "The Web Origin Concept", RFC 6454, DOI 10.17487/RFC6454, December 2011, <https://www.rfc-editor.org/info/rfc6454>.

[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>.

[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>.

[RFC7469] Evans, C., Palmer, C., and R. Sleevi, "Public Key Pinning Extension for HTTP", RFC 7469, DOI 10.17487/RFC7469, April 2015, <https://www.rfc-editor.org/info/rfc7469>.

[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>.

[RFC7942] Sheffer, Y. and A. Farrel, "Improving Awareness of Running Code: The Implementation Status Section", BCP 205, RFC 7942, DOI 10.17487/RFC7942, July 2016, <https://www.rfc-editor.org/info/rfc7942>.

[TLS-EXT] IANA, ., "TLS Extension Type Value", 2018, <https://www.iana.org/assignments/tls-extensiontype-values/tls-extensiontype-values.xhtml#tls-extensiontype-values-1>.

Appendix A. Previous Work

The global PKI system relies on the trust of a CA issuing certificates. As a result, a corrupted trusted CA may issue a certificate for any organization without the organization's approval (a misissued or "fake" certificate), and use the certificate to impersonate the organization. There are many attempts to resolve these weaknesses, including Certificate Transparency (CT) [RFC6962], HTTP Public Key Pinning (HPKP) [RFC7469], and TACK [I-D.perrin-tls-tack].

CT requires cooperation of a large portion of the hundreds of extant certificate authorities (CAs) before it can be used "for real", in enforcing mode. It is noted that the relevant industry forum (CA/Browser Forum) is indeed pushing for such extensive adoption. However the public nature of CT often makes it inappropriate for enterprise use, because many organizations are not willing to expose their internal infrastructure publicly.

TACK has some similarities to the current proposal, but work on it seems to have stalled. Appendix A.2 compares our proposal to TACK.

HPKP is an IETF standard, but so far has proven hard to deploy. HPKP pins (fixes) a public key, one of the public keys listed in the certificate chain. As a result, HPKP needs to be coordinated with the certificate management process. Certificate management impacts HPKP and thus increases the probability of HPKP failures. This risk is made even higher given the fact that, even though work has been done at the ACME WG to automate certificate management, in many or even most cases, certificates are still managed manually. As a result, HPKP cannot be completely automated resulting in error-prone manual configuration. Such errors could prevent the web server from being accessed by some clients. In addition, HPKP uses a HTTP header which makes this solution HTTPS specific and not generic to TLS. On the other hand, the current document provides a solution that is independent of the server's certificate management and that can be entirely and easily automated. Appendix A.1 compares HPKP to the current document in more detail.

The ticket pinning proposal augments these mechanisms with a much easier to implement and deploy solution for server identity pinning, by reusing some of the ideas behind TLS session resumption.

This section compares ticket pinning to two earlier proposals, HPKP and TACK.

A.1. Comparison: HPKP

The current IETF standard for pinning the identity of web servers is the Public Key Pinning Extension for HTTP, or HPKP [RFC7469].

The main differences between HPKP and the current document are the following:

- HPKP limits its scope to HTTPS, while the current document considers all application above TLS.

- HPKP pins the public key of the server (or another public key along the certificate chain) and as such is highly dependent on the management of certificates. Such dependency increases the potential error surface, especially as certificate management is not yet largely automated. The current proposal, on the other hand, is independent of certificate management.

- HPKP pins public keys which are public and used for the standard TLS authentication. Identity pinning relies on the ownership of the pinning key which is not disclosed to the public and not involved in the standard TLS authentication. As a result, identity pinning is a completely independent second factor authentication mechanism.

- HPKP relies on a backup key to recover the misissuance of a key. We believe such backup mechanisms add excessive complexity and cost. Reliability of the current mechanism is primarily based on its being highly automated.

- HPKP relies on the client to report errors to the report-uri. The current document does not need any out-of band mechanism, and the server is informed automatically. This provides an easier and more reliable health monitoring.

On the other hand, HPKP shares the following aspects with identity pinning:

- Both mechanisms provide hard failure. With HPKP only the client is aware of the failure, while with the current proposal both client and server are informed of the failure. This provides room for further mechanisms to automatically recover such failures.

- Both mechanisms are subject to a server compromise in which users are provided with an invalid ticket (e.g. a random one) or HTTP Header, with a very long lifetime. For identity pinning, this lifetime SHOULD NOT be longer than 31 days. In both cases, clients will not be able to reconnect the server during this lifetime. With the current proposal, an attacker needs to compromise the TLS layer, while with HPKP, the attacker needs to compromise the HTTP server. Arguably, the TLS-level compromise is typically more difficult for the attacker.

Unfortunately HPKP has not seen wide deployment yet. As of March 2016, the number of servers using HPKP was less than 3000 [Netcraft]. This may simply be due to inertia, but we believe the main reason is the interactions between HPKP and manual certificate management which is needed to implement HPKP for enterprise servers. The penalty for making mistakes (e.g. being too early or too late to deploy new pins) is having the server become unusable for some of the clients.

To demonstrate this point, we present a list of the steps involved in deploying HPKP on a security-sensitive Web server.

1. Generate two public/private key-pairs on a computer that is not the Live server. The second one is the "backup1" key-pair.

   "openssl genrsa -out "example.com.key" 2048;"

   "openssl genrsa -out "example.com.backup1.key" 2048;"

2. Generate hashes for both of the public keys. These will be used in the HPKP header:

   "openssl rsa -in "example.com.key" -outform der -pubout | openssl dgst -sha256 -binary | openssl enc -base64"

   "openssl rsa -in "example.com.backup1.key" -outform der -pubout | openssl dgst -sha256 -binary | openssl enc -base64"

3. Generate a single CSR (Certificate Signing Request) for the first key-pair, where you include the domain name in the CN (Common Name) field:

   "openssl req -new -subj "/C=GB/ST=Area/L=Town/O=Company/CN=example.com" -key "example.com.key" -out "example.com.csr";"

4. Send this CSR to the CA (Certificate Authority), and go though the dance to prove you own the domain. The CA will give you back a single certificate that will typically expire within a year or two.

5. On the Live server, upload and setup the first key-pair (and its certificate). At this point you can add the "Public-Key-Pins" header, using the two hashes you created in step 2.

Note that only the first key-pair has been uploaded to the server so far.

6. Store the second (backup1) key-pair somewhere safe, probably somewhere encrypted like a password manager. It won't expire, as it's just a key-pair, it just needs to be ready for when you need to get your next certificate.

7. Time passes... probably just under a year (if waiting for a certificate to expire), or maybe sooner if you find that your server has been compromised and you need to replace the key-pair and certificate.

8. Create a new CSR (Certificate Signing Request) using the "backup1" key-pair, and get a new certificate from your CA.

9. Generate a new backup key-pair (backup2), get its hash, and store it in a safe place (again, not on the Live server).

10. Replace your old certificate and old key-pair, and update the "Public-Key-Pins" header to remove the old hash, and add the new "backup2" key-pair.

Note that in the above steps, both the certificate issuance as well as the storage of the backup key pair involve manual steps. Even with an automated CA that runs the ACME protocol, key backup would be a challenge to automate.

A.2. Comparison: TACK

Compared with HPKP, TACK [I-D.perrin-tls-tack] is a lot more similar to the current document. It can even be argued that this document is a symmetric-cryptography variant of TACK. That said, there are still a few significant differences:

- Probably the most important difference is that with TACK, validation of the server certificate is no longer required, and in fact TACK specifies it as a "MAY" requirement (Sec. 5.3). With ticket pinning, certificate validation by the client remains a MUST requirement, and the ticket acts only as a second factor. If the pinning secret is compromised, the server's security is not immediately at risk.

- Both TACK and the current document are mostly orthogonal to the server certificate as far as their life cycle, and so both can be deployed with no manual steps.

- TACK uses ECDSA to sign the server's public key. This allows cooperating clients to share server assertions between themselves. This is an optional TACK feature, and one that cannot be done with pinning tickets.

- TACK allows multiple servers to share its public keys. Such sharing is disallowed by the current document.

- TACK does not allow the server to track a particular client, and so has better privacy properties than the current document.

- TACK has an interesting way to determine the pin's lifetime, setting it to the time period since the pin was first observed, with a hard upper bound of 30 days. The current document makes the lifetime explicit, which may be more flexible to deploy. For example, Web sites which are only visited rarely by users may opt for a longer period than other sites that expect users to visit on a daily basis.

Appendix B. Document History

B.1. draft-sheffer-tls-pinning-ticket-12

- IETF-Conflict Review comments.

- IANA: removed request for a specific extension value.

B.2. draft-sheffer-tls-pinning-ticket-11

- Comments by Ben Kaduk. Specifically, changed the derivation of the pinning proof to make it more in line with the TLS 1.3 key schedule.

B.3. draft-sheffer-tls-pinning-ticket-10

- ISE comments by Adrian Farrel, the ISE.

B.4. draft-sheffer-tls-pinning-ticket-09

- ISE comments by Yoav Nir.

B.5. draft-sheffer-tls-pinning-ticket-08

- ISE comments by Rich Salz.

B.6. draft-sheffer-tls-pinning-ticket-07

- Refer to published RFCs.

B.7. draft-sheffer-tls-pinning-ticket-06

- IANA Considerations in preparation for Experimental publication.

B.8. draft-sheffer-tls-pinning-ticket-05

- Multiple comments from Eric Rescorla.

B.9. draft-sheffer-tls-pinning-ticket-04

- Editorial changes.

- Two-phase rotation of protection key.

B.10. draft-sheffer-tls-pinning-ticket-03

- Deleted redundant length fields in the extension's formal definition.

- Modified cryptographic operations to align with the current state of TLS 1.3.

- Numerous textual improvements.

B.11. draft-sheffer-tls-pinning-ticket-02

- Added an Implementation Status section.

- Added lengths into the extension structure.

- Changed the computation of the pinning proof to be more robust.

- Clarified requirements on the length of the pinning_secret.

- Revamped the HPKP section to be more in line with current practices, and added recent statistics on HPKP deployment.

B.12. draft-sheffer-tls-pinning-ticket-01

- Corrected the notation for variable-sized vectors.

- Added a section on disaster recovery and backup.

- Added a section on privacy.

- Clarified the assumptions behind the HPKP procedure in the comparison section.

- Added a definition of pin indexing (origin).

- Adjusted to the latest TLS 1.3 notation.

B.13. draft-sheffer-tls-pinning-ticket-00

Initial version.

Authors' Addresses

Yaron Sheffer
Intuit

EMail: yaronf.ietf@gmail.com

Daniel Migault
Ericsson

EMail: daniel.migault@ericsson.com

TLS N. Sullivan
Internet-Draft Cloudflare Inc.
Intended status: Standards Track March 13, 2017
Expires: September 14, 2017

Exported Authenticators in TLS
draft-sullivan-tls-exported-authenticator-01

Abstract

This document describes a mechanism in Transport Layer Security (TLS) to provide an exportable proof of ownership of a certificate that can be transmitted out of band and verified by the other party.

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) communication to another using a certificate 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 the other party.

This mechanism is useful in the following situations:

- servers that are authoritative for multiple domains the same connection but do not have a certificate that is simultaneously authoritative for all of them

- servers that have resources that require client authentication to access and need to request client authentication after the connection has started

- clients that want to assert ownership over an identity to a server after a connection has been established

This document intends to replace much of the functionality of renegotiation in previous versions of TLS. It has the advantages over renegotiation of not requiring additional on-the-wire changes during a connection. For simplicity, only TLS 1.2 and later are supported.

2. Authenticator

The authenticator is a structured message that can be exported from either party of a TLS connection. It can be sent out-of-band to the other party of a TLS connection to be validated.

An authenticator message can be constructed by either the client or the server given an established TLS connection, a certificate, and a corresponding private key. This authenticator uses the message structures from section 4.4. of [I-D.ietf-tls-tls13], but different parameters. Also, unlike the Certificate and CertificateRequest messages in TLS 1.3, the messages described in this draft are not encryped with a handshake key.

Each Authenticator is computed using a Handshake Context and Finished MAC Key derived from the TLS session. The Handshake Context is identical for both parties of the TLS connection, the Finished MAC Key is dependent on whether the Authenticator is created by the client or the server.

- o The Handshake Context is an [RFC5705] (for TLS 1.2) or [I-D.ietf-tls-tls13] exporter value derived using the label "authenticator handshake context" and length 64 bytes.

- o The Finished MAC Key is an exporter value derived using the label "server authenticator finished key" or "client authenticator finished key", depending on the sender. The length of this key is equal to the length of the output of the hash function negotiated in TLS. For TLS 1.3, it’s the hash algorithm of the cipher suite. For TLS 1.2, it’s the hash algorithm selected for the PRF for AEAD ciphers, or the hash algorithm used as the HMAC in non-AEAD ciphers.

If the connection is TLS 1.2, the master secret MUST have been computed with the extended master secret [RFC7627] to avoid key synchronization attacks.

Certificate The certificate to be used for authentication and any supporting certificates in the chain.

The certificate message contains an opaque string called certificate_request_context which MUST be unique for a given connection. Its format should be defined by the application layer protocol and MUST be non-zero length. For example, it may be a randomly chosen identifier used by the higher-level protocol during the transport of the Authenticator to the other party.

CertificateVerify A signature over the value Hash(Handshake Context || Certificate)

Finished A HMAC over the value Hash(Handshake Context || Certificate || CertificateVerify) using the hash function from the handshake and the Finished MAC Key as a key.

The certificates used in the Certificate message MUST conform to the requirements of a Certificate message in the version of TLS negotiated. This is described in section 4.2.3. of [I-D.ietf-tls-tls13] and sections 7.4.2. and 7.4.6. of [RFC5246].

The exported authenticator message is the concatenation of messages: Certificate || CertificateVerify || Finished

3. API considerations

TLS implementations supporting the use of exported authenticators MUST provide application programming interfaces by which clients and servers may request and verify exported authenticator messages.

Given an established connection, the application should be able to obtain an authenticator by providing the following:

- o certificate_request_context (from 1 to 255 bytes)
- o valid certificate chain for the connection and associated extensions (OCSP, SCT, etc.)
- o signer (either the private key associated with the certificate, or interface to perform private key operation)

Given an established connection and an exported authenticator message, the application should be able to provide the authenticator to the connection. If the Finished and CertificateVerify messages verify, the TLS library should return the following:

- o certificate chain and extensions
- o certificate_request_context

In order for the application layer to communicate which certificates it will accept, an API should be exposed that returns an array of TLS 1.3 SignatureScheme objects that corresponds to the signature algorithms that the library is willing to validate in an exported authenticator message.

4. Security Considerations

The Certificate/Verify/Finished pattern intentionally looks like the TLS 1.3 pattern which now has been analyzed several times. In the case where the client presents an authenticator to a server, [SIGMAC] presents a relevant framework for analysis.

From a formal security perspective, one drawback of this mechanism is that there is no explicit signaling mechanism for one party to acknowledge an Authenticator to the party who computed it. Nothing about the state of the connection is changed when a new Authenticator is exported, and the Handshake Context of the TLS connection is unchanged after creating or validating an authenticator. This property makes it difficult to formally prove that a server is jointly authoritative over multiple certificates, rather than individually authoritative on each certificate.

Another result of the unidirectional nature of Authenticator messages is that the view of which certificates the other party is authoritative over does not reside in the TLS state machine. Not knowing when the exported authenticator was created or validated at the TLS layer also means that assumptions about when the other party is considered authoritative can not be determined purely from where in the in the TLS record layer it was sent. A valid authenticator can be created at any time during the connection. If it matters to the application whether or not an authenticator was acknowledged before or after a particular piece of data, it should be tracked as part of the application layer semantics.

5. Acknowledgements

Comments on this proposal were provided by Martin Thomson. Suggestions for the security considerations section were provided by Karthikeyan Bhargavan.

6. Normative References

[I-D.ietf-tls-tls13]
Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", draft-ietf-tls-tls13-19 (work in progress), March 2017.

[RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, DOI 10.17487/RFC5246, August 2008, <http://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, <http://www.rfc-editor.org/info/rfc5705>.

[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, <http://www.rfc-editor.org/info/rfc7627>.

[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

HTTP M. Thomson
Internet-Draft Mozilla
Intended status: Standards Track November 13, 2016
Expires: May 17, 2017

Example Handshake Traces for TLS 1.3
draft-thomson-tls-tls13-vectors-01

Abstract

Examples of TLS 1.3 handshakes are shown. Private keys and inputs are provided so that these handshakes might be reproduced. Intermediate values, including secrets, traffic keys and ivs are shown so that implementations might be checked incrementally against these values.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

TLS 1.3 [I-D.ietf-tls-tls13] defines a new key schedule and a number new cryptographic operations. This document includes sample handshakes that show all intermediate values. This allows an implementation to be verified incrementally, examining inputs and outputs of each cryptographic computation independently.

Private keys are included with the traces so that implementations can be checked by importing these values and verifying that the same outputs are produced.

2. Private Keys

Ephemeral private keys are shown as they are generated in the traces.

The server in most examples uses an RSA certificate with a private key of:

```
modulus (public):  b4bb498f8279303d 980836399b36c698 8c0c68de55e1bdb8
   26d3901a2461eafd 2de49a91d015abbc 9a95137ace6c1af1
   9eaa6af98c7ced43 120998e187a80ee0 ccb0524b1b018c3e
   0b63264d449a6d38 e22a5fda43084674 8030530ef0461c8c
   a9d9efbfae8ea6d1 d03e2bd193eff0ab 9a8002c47428a6d3
   5a8d88d79f7f1e3f

public exponent:  010001

private exponent:  04dea705d43a6ea7 209dd8072111a83c 81e322a59278b334
   80641eaf7c0a6985 b8e31c44f6de62e1 b4c2309f6126e77b
   7c41e923314bbfa3 881305dc1217f16c 819ce538e922f369
   828d0e57195d8c84 88460207b2faa726 bcf708bbd7db7f67
   9f893492fc2a622e 08970aac441ce4e0 c3088df25ae67923
   3df8a3bda2ff9941
```

```
   prime1:  e435fb7cc8373775 6dacea96ab7f59a2 cc1069db7deb190e
      17e33a532b273f30 a327aa0aaabc58cd 67466af9845fadc6
      75fe094af92c4bd1 f2c1bc33dd2e0515

   prime2:  cabd3bc0e0438664 c8d4cc9f99977a94 d9bbfead8e43870a
      bae3f7eb8b4e0eee 8af1d9b4719ba619 6cf2cbbaeeebf8b3
      490afe9e9ffa74a8 8aa51fc645629303

   exponent1:  3f57345c27fe1b68 7e6e761627b78b1b 826433dd760fa0be
      a6a6acf39490aa1b 47cda4869d68f584 dd5b5029bd32093b
      8258661fe715025e 5d70a45a08d3d319

   exponent2:  183da01363bd2f28 85cacbdc9964bf47 64f1517636f86401
      286f71893c52ccfe 40a6c23d0d086b47 c6fb10d8fd1041e0
      4def7e9a40ce957c 417794e10412d139

   coefficient:  839ca9a085e4286b 2c90e466997a2c68 1f21339aa3477814
      e4dec11833050ed5 0dd13cc038048a43 c59b2acc416889c0
      37665fe5afa60596 9f8c01dfa5ca969d
```

3. Simple 1-RTT Handshake

In this example, the simplest possible handshake is completed. The server is authenticated, but the client remains anonymous. After connecting, a few application data octets are exchanged. The server sends a session ticket that permits the use of 0-RTT in any resumed session.

Note: This example doesn’t include the calculation of the exporter secret. Support for that will be added to NSS soon.

{client} create an ephemeral x25519 key pair:

```
      private key (32 octets):  03bd8bca70c19f65 7e897e366dbe21a4
         66e4924af6082dbd f573827bcdde5def

      public key (32 octets):  2a981db6cdd02a06 c1763102c9e74136
         5ac4e6f72b3176a6 bd6a3523d3ec0f4c
```

{client} send a ClientHello handshake message

{client} send record:

```
      cleartext (512 octets):  010001fc0303ce05 cfa3d92170cbc246
         5cdc3e3a2f577f6e ac809361708ab244 b07d8fad86160000
         3e130113031302c0 2bc02fcca9cca8c0 0ac009c013c023c0
         27c014009eccaa00 3300320067003900 38006b0016001300
         9c002f003c003500 3d000a0005000401 000195001500fc00
```

```
         0000000000000000 0000000000000000 0000000000000000
         0000000000000000 0000000000000000 0000000000000000
         0000000000000000 0000000000000000 0000000000000000
         0000000000000000 0000000000000000 0000000000000000
         0000000000000000 0000000000000000 0000000000000000
         0000000000000000 0000000000000000 0000000000000000
         0000000000000000 0000000000000000 0000000000000000
         0000000000000000 0000000000000000 0000000000000000
         0000000000000000 0000000000000000 0000000000000000
         0000000000000000 0000000000000000 0000000000000000
         0000000000000000 000000000000b00 0900000673657276
         6572ff0100010000 0a00140012001d00 1700180019010001
         0101020103010400 0b00020100002300 0000280026002400
         1d00202a981db6cd d02a06c1763102c9 e741365ac4e6f72b
         3176a6bd6a3523d3 ec0f4c002b000706 7f1203030302000d
         0020001e04030503 0603020308040805 0806040105010601
         0201040205020602 0202002d00020101

      ciphertext (517 octets):  1603010200010001 fc0303ce05cfa3d9
         2170cbc2465cdc3e 3a2f577f6eac8093 61708ab244b07d8f
         ad861600003e1301 13031302c02bc02f cca9cca8c00ac009
         c013c023c027c014 009eccaa00330032 006700390038006b
         00160013009c002f 003c0035003d000a 0005000401000195
         001500fc00000000 0000000000000000 0000000000000000
         0000000000000000 0000000000000000 0000000000000000
         0000000000000000 0000000000000000 0000000000000000
         0000000000000000 0000000000000000 0000000000000000
         0000000000000000 0000000000000000 0000000000000000
         0000000000000000 0000000000000000 0000000000000000
         0000000000000000 0000000000000000 0000000000000000
         0000000000000000 0000000000000000 0000000000000000
         0000000000000000 0000000000000000 0000000000000000
         0000000000000000 0000000000000000 0000000000000000
         0000000000000000 0000000000000000 0000000b00090000
         06736572766572ff 01000100000a0014 0012001d00170018
         0019010001010102 01030104000b0002 0100002300000028
         00260024001d0020 2a981db6cdd02a06 c1763102c9e74136
         5ac4e6f72b3176a6 bd6a3523d3ec0f4c 002b0007067f1203
         030302000d002000 1e04030503060302 0308040805080604
         0105010601020104 0205020602020200 2d00020101

   {server}  extract secret "early":

      salt (0 octets):  (empty)

      ikm (32 octets):  0000000000000000 0000000000000000
         0000000000000000 0000000000000000
```

```
      secret (32 octets):  33ad0a1c607ec03b 09e6cd9893680ce2
         10adf300aa1f2660 e1b22e10f170f92a

   {server}  create an ephemeral x25519 key pair:

      private key (32 octets):  0cc3d0a7806ef6bc df69be30c6855597
         7b51e0f5edbf1d1c c7b28eead93b34b4

      public key (32 octets):  9c1b0a7421919a73 cb57b3a0ad9d6805
         861a9c47e11df863 9d25323b79ce201c

   {server}  send a ServerHello handshake message

   {server}  extract secret "handshake":

      salt (32 octets):  33ad0a1c607ec03b 09e6cd9893680ce2
         10adf300aa1f2660 e1b22e10f170f92a

      ikm (32 octets):  0dfa4c5e11a6f606 d4b75f138412d85a
         4b2da0d5f981ffc1 d2e8ceff2e00a12c

      secret (32 octets):  1b3f45dcdc375a9a e91bf34d669f24c7
         53132f1d394553af bfffe6568a27e22c

   {server}  derive secret "client handshake traffic secret":

      PRK (32 octets):  1b3f45dcdc375a9a e91bf34d669f24c7
         53132f1d394553af bfffe6568a27e22c

      handshake hash (32 octets):  79027f438271dba2 d8e207b6e36a5180
         bdd916869ab43f24 f2e2fa98b2db135c

      info (76 octets):  002028544c532031 2e332c20636c6965
         6e742068616e6473 68616b6520747261 6666696320736563
         7265742079027f43 8271dba2d8e207b6 e36a5180bdd91686
         9ab43f24f2e2fa98 b2db135c

      output (32 octets):  f737c2b29be2ef48 9d145dd3df485103
         86e812edcf799925 27e9ad5479967193

   {server}  derive secret "server handshake traffic secret":

      PRK (32 octets):  1b3f45dcdc375a9a e91bf34d669f24c7
         53132f1d394553af bfffe6568a27e22c

      handshake hash (32 octets):  79027f438271dba2 d8e207b6e36a5180
         bdd916869ab43f24 f2e2fa98b2db135c
```

```
      info (76 octets):  002028544c532031 2e332c2073657276
         65722068616e6473 68616b6520747261 6666696320736563
         7265742079027f43 8271dba2d8e207b6 e36a5180bdd91686
         9ab43f24f2e2fa98 b2db135c

      output (32 octets):  3550ca3a8c219272 9cc385313e3bc832
         92a14f4ecb3d2b92 18ea7907c67ab3a7

   {server}  extract secret "master":

      salt (32 octets):  1b3f45dcdc375a9a e91bf34d669f24c7
         53132f1d394553af bfffe6568a27e22c

      ikm (32 octets):  0000000000000000 0000000000000000
         0000000000000000 0000000000000000

      secret (32 octets):  cab4645a3995d0d8 5bea9942596284e7
         2058a3d4d8f3e0d9 885aa92c517ad9e4

   {server}  derive write traffic keys using label "handshake data":

      PRK (32 octets):  3550ca3a8c219272 9cc385313e3bc832
         92a14f4ecb3d2b92 18ea7907c67ab3a7

      key info (16 octets):  00100c544c532031 2e332c206b657900

      key output (16 octets):  d2dd45f87ad87801 a85ac38187f9023b

      iv info (15 octets):  000c0b544c532031 2e332c20697600

      iv output (12 octets):  f0a14f808692cef8 7a3daf70

   {server}  send record:

      cleartext (82 octets):  0200004e7f1220b9 c9201cd171a15abb
         a4e7eddcf3e8488e 7192ffe01ea5c19f 3d4b52ffeebe1301
         002800280024001d 00209c1b0a742191 9a73cb57b3a0ad9d
         6805861a9c47e11d f8639d25323b79ce 201c

      ciphertext (87 octets):  1603010052020000 4e7f1220b9c9201c
         d171a15abba4e7ed dcf3e8488e7192ff e01ea5c19f3d4b52
         ffeebe1301002800 280024001d00209c 1b0a7421919a73cb
         57b3a0ad9d680586 1a9c47e11df8639d 25323b79ce201c

   {server}  send a EncryptedExtensions handshake message

   {server}  send a Certificate handshake message
```

{server} send a CertificateVerify handshake message

{server} calculate finished:

PRK (32 octets): 3550ca3a8c219272 9cc385313e3bc832
92a14f4ecb3d2b92 18ea7907c67ab3a7

handshake hash (0 octets): (empty)

info (21 octets): 002011544c532031 2e332c2066696e69 7368656400

output (32 octets): 1ba8c586468bb93d cd9264e62929e77d
eba36e5bfc5e06ad 029f667448e5e6c8

{server} send a Finished handshake message

{server} send record:

cleartext (651 octets): 0800001e001c000a 00140012001d0017
0018001901000101 0102010301040000 00000b0001b90000
01b50001b0308201 ac30820115a00302 0102020102300d06
092a864886f70d01 010b0500300e310c 300a060355040313
03727361301e170d 3136303733303031 323335395a170d32
3630373330303132 3335395a300e310c 300a060355040313
0372736130819f30 0d06092a864886f7 0d01010105000381
8d00308189028181 00b4bb498f827930 3d980836399b36c6
988c0c68de55e1bd b826d3901a2461ea fd2de49a91d015ab
bc9a95137ace6c1a f19eaa6af98c7ced 43120998e187a80e
e0ccb0524b1b018c 3e0b63264d449a6d 38e22a5fda430846
748030530ef0461c 8ca9d9efbfae8ea6 d1d03e2bd193eff0
ab9a8002c47428a6 d35a8d88d79f7f1e 3f0203010001a31a
301830090603551d 1304023000300b06 03551d0f04040302
05a0300d06092a86 4886f70d01010b05 00038181008 5aad2
a0e5b9276b908c65 f73a7267170618a5 4c5f8a7b337d2df7
a594365417f2eae8 f8a58c8f8172f931 9cf36b7fd6c55b80
f21a030151567260 96fd335e5e67f2db f102702e608ccae6
bec1fc63a42a99be 5c3eb7107c3c54e9 b9eb2bd5203b1c3b
84e0a8b2f759409b a3eac9d91d402dcc 0cc8f8961229ac91
87b42b4de100000f 0000840804008013 4e22eac57321ab47
db6b38b2992cec2d d79bd065a034a9af 6b9e3d03475e4309
e6523ccdf055453f b480804a3a7e9962 29eb28e734f6702b
ea2b32149899ac04 3a4b44468197868d a77147ce9f73c054
3c4e3fc33e306cac 8506faa80a959c5f 1edccbee76eda1ad
7a4fa440de35dcb8 7e82ec94e8725355 ce7507713a609e14
0000207304bb7332 1f01b71dd94622fa e98daf634490d220
e4c8f3ffa2559911 a56e51

```
      ciphertext (673 octets):  170301029c40ae92 071a3a548b26af31
         e116dfc0ba454921 0b17e70da16cfbda 9ccdad844d94264a
         9ae65b786b3eaf0d e20aa89c6babb448 b6f32d07f2335842
         96eefe19316bd979 659472ee8567cb01 d70b0366cddb3c60
         eb9e1d789a3691dc 254c14de73f4f201 00504544ce184d44
         547e124b1f18303b 4859f8f2e2b04423 d23a866b43866374
         d54af41649d25f4a 3ec2cecd5d4e6de1 b24953440b46fbb7
         4c1dbec6fbb1f16b c21d4aa0e1e936a4 9c07127e19719bc6
         52a2f0b7f8df4a15 0b2b3c9e9e353d6e d101970ddc611aba
         d0632c6793f9379c 9d06846c311fcbd6 f85edd569b8782c4
         c5f62294c4611ae6 0f83230a53aa95e3 bcbed204f19a7a1d
         b83c0fbfec1edd2c 17498fa7b5aa2321 248a92592d891e49
         47df6bcef52f4481 797d032ad332046a 384abece6454b3e3
         56d7249bfa569679 3c7f7d3048dc87fa 7409a4691887caaf
         0982c402b902d699 f62dc4d5e153f13e 8589e4a6206c7f74
         eb26ddefbb92309f b753decfea972dec 7de02eda9c6d26ac
         d7be53a8aa20f1a9 3f082ae6eb927a6a 1b7bd9153551aedf
         af94f61dd4cb9355 ad7ab09f615d9f92 c21712c732c0e7e1
         17797f38cbdc184e 3a65e15a89f46cb3 624f5fdb8dbbd275
         f2c8492f8d95bdbd 8d1dc1b9f21107bd 433acbbac247239c
         073a2f24a4a9f807 4f325f277d579b6b ff0269ff19aed380
         9a9ddd21dd29c136 3c9dc44812dd41d2 111f9c2e8342046c
         14133b853262676f 15e94de18660e04a e5c0c661ea43559a
         f5842e161c83dd29 f64508b2ec3e635a 2134fc0e1a39d3ec
         b51dcddfcf8382c8 8ffe2a737842ad1d e7fe505b6c4d1673
         870f6fc2a0f2f797 2acaee368a1599d6 4ba18798f10333f9
         779bd5b05f9b084d 03dab2f3d80c2eb7 4ec70c9866ea31c1
         8b491cd597aae3e9 41205fcc38a3a10c e8c0269f02ccc9c5
         1278e25f1a0f0731 a9

   {server}  derive secret "client application traffic secret":

      PRK (32 octets):  cab4645a3995d0d8 5bea9942596284e7
         2058a3d4d8f3e0d9 885aa92c517ad9e4

      handshake hash (32 octets):  16756399da565370 337a4ede5774b9e6
         0bf328086272dc39 3b8b1d8ba6e6ebbb

      info (78 octets):  00202a544c532031 2e332c20636c6965
         6e74206170706c69 636174696f6e2074 7261666669632073
         6563726574201675 6399da565370337a 4ede5774b9e60bf3
         28086272dc393b8b 1d8ba6e6ebbb

      output (32 octets):  2a1d25e6f9f13f92 e4b482fa06bc4447
         1218368d2d4e03e0 504d4e342b16ff8f

   {server}  derive secret "server application traffic secret":
```

```
      PRK (32 octets):  cab4645a3995d0d8 5bea9942596284e7
         2058a3d4d8f3e0d9 885aa92c517ad9e4

      handshake hash (32 octets):  16756399da565370 337a4ede5774b9e6
         0bf328086272dc39 3b8b1d8ba6e6ebbb

      info (78 octets):  00202a544c532031 2e332c2073657276
         6572206170706c69 636174696f6e2074 7261666669632073
         6563726574201675 6399da565370337a 4ede5774b9e60bf3
         28086272dc393b8b 1d8ba6e6ebbb

      output (32 octets):  56231ff04300e7f7 4964da88c8bbdf12
         42a31ade351ce974 46598d28632e79ca

   {server}  derive secret "exporter master secret":

      PRK (32 octets):  cab4645a3995d0d8 5bea9942596284e7
         2058a3d4d8f3e0d9 885aa92c517ad9e4

      handshake hash (32 octets):  16756399da565370 337a4ede5774b9e6
         0bf328086272dc39 3b8b1d8ba6e6ebbb

      info (67 octets):  00201f544c532031 2e332c206578706f
         72746572206d6173 7465722073656372 65742016756399da
         565370337a4ede57 74b9e60bf3280862 72dc393b8b1d8ba6 e6ebbb

      output (32 octets):  407265d811f66c24 30de0832fbc4bd25
         719a4736301f1312 98fd9107653a78f2

   {server}  derive write traffic keys using label "application data":

      PRK (32 octets):  56231ff04300e7f7 4964da88c8bbdf12
         42a31ade351ce974 46598d28632e79ca

      key info (16 octets):  00100c544c532031 2e332c206b657900

      key output (16 octets):  3381f6b3f94500f1 6226de440193e858

      iv info (15 octets):  000c0b544c532031 2e332c20697600

      iv output (12 octets):  4f1d73cc1d465eb3 0021c41f

   {server}  derive read traffic keys using label "handshake data":

      PRK (32 octets):  f737c2b29be2ef48 9d145dd3df485103
         86e812edcf799925 27e9ad5479967193

      key info (16 octets):  00100c544c532031 2e332c206b657900
```

```
      key output (16 octets):  40e1201d75d41962 7f04c88530a15c9d

      iv info (15 octets):  000c0b544c532031 2e332c20697600

      iv output (12 octets):  a0f073f3b35e18f9 6969696b

   {client}  extract secret "early":

      salt (0 octets):  (empty)

      ikm (32 octets):  0000000000000000 0000000000000000
         0000000000000000 0000000000000000

      secret (32 octets):  33ad0a1c607ec03b 09e6cd9893680ce2
         10adf300aa1f2660 e1b22e10f170f92a

   {client}  extract secret "handshake":

      salt (32 octets):  33ad0a1c607ec03b 09e6cd9893680ce2
         10adf300aa1f2660 e1b22e10f170f92a

      ikm (32 octets):  0dfa4c5e11a6f606 d4b75f138412d85a
         4b2da0d5f981ffc1 d2e8ceff2e00a12c

      secret (32 octets):  1b3f45dcdc375a9a e91bf34d669f24c7
         53132f1d394553af bfffe6568a27e22c

   {client}  derive secret "client handshake traffic secret":

      PRK (32 octets):  1b3f45dcdc375a9a e91bf34d669f24c7
         53132f1d394553af bfffe6568a27e22c

      handshake hash (32 octets):  79027f438271dba2 d8e207b6e36a5180
         bdd916869ab43f24 f2e2fa98b2db135c

      info (76 octets):  002028544c532031 2e332c20636c6965
         6e742068616e6473 68616b6520747261 6666696320736563
         7265742079027f43 8271dba2d8e207b6 e36a5180bdd91686
         9ab43f24f2e2fa98 b2db135c

      output (32 octets):  f737c2b29be2ef48 9d145dd3df485103
         86e812edcf799925 27e9ad5479967193

   {client}  derive secret "server handshake traffic secret":

      PRK (32 octets):  1b3f45dcdc375a9a e91bf34d669f24c7
         53132f1d394553af bfffe6568a27e22c
```

```
      handshake hash (32 octets):  79027f438271dba2 d8e207b6e36a5180
         bdd916869ab43f24 f2e2fa98b2db135c

      info (76 octets):  002028544c532031 2e332c2073657276
         65722068616e6473 68616b6520747261 6666696320736563
         7265742079027f43 8271dba2d8e207b6 e36a5180bdd91686
         9ab43f24f2e2fa98 b2db135c

      output (32 octets):  3550ca3a8c219272 9cc385313e3bc832
         92a14f4ecb3d2b92 18ea7907c67ab3a7

   {client}  extract secret "master" (same as server)

   {client}  derive read traffic keys using label "handshake data":

      PRK (32 octets):  3550ca3a8c219272 9cc385313e3bc832
         92a14f4ecb3d2b92 18ea7907c67ab3a7

      key info (16 octets):  00100c544c532031 2e332c206b657900

      key output (16 octets):  d2dd45f87ad87801 a85ac38187f9023b

      iv info (15 octets):  000c0b544c532031 2e332c20697600

      iv output (12 octets):  f0a14f808692cef8 7a3daf70

   {client}  calculate finished:

      PRK (32 octets):  3550ca3a8c219272 9cc385313e3bc832
         92a14f4ecb3d2b92 18ea7907c67ab3a7

      handshake hash (0 octets):  (empty)

      info (21 octets):  002011544c532031 2e332c2066696e69 7368656400

      output (32 octets):  1ba8c586468bb93d cd9264e62929e77d
         eba36e5bfc5e06ad 029f667448e5e6c8

   {client}  derive write traffic keys using label "handshake data"
      (same as server read traffic keys)

   {client}  derive secret "client application traffic secret":

      PRK (32 octets):  cab4645a3995d0d8 5bea9942596284e7
         2058a3d4d8f3e0d9 885aa92c517ad9e4

      handshake hash (32 octets):  16756399da565370 337a4ede5774b9e6
         0bf328086272dc39 3b8b1d8ba6e6ebbb
```

```
      info (78 octets):  00202a544c532031 2e332c20636c6965
         6e74206170706c69 636174696f6e2074 7261666669632073
         6563726574201675 6399da565370337a 4ede5774b9e60bf3
         28086272dc393b8b 1d8ba6e6ebbb

      output (32 octets):  2a1d25e6f9f13f92 e4b482fa06bc4447
         1218368d2d4e03e0 504d4e342b16ff8f

   {client}  derive secret "server application traffic secret":

      PRK (32 octets):  cab4645a3995d0d8 5bea9942596284e7
         2058a3d4d8f3e0d9 885aa92c517ad9e4

      handshake hash (32 octets):  16756399da565370 337a4ede5774b9e6
         0bf328086272dc39 3b8b1d8ba6e6ebbb

      info (78 octets):  00202a544c532031 2e332c2073657276
         6572206170706c69 636174696f6e2074 7261666669632073
         6563726574201675 6399da565370337a 4ede5774b9e60bf3
         28086272dc393b8b 1d8ba6e6ebbb

      output (32 octets):  56231ff04300e7f7 4964da88c8bbdf12
         42a31ade351ce974 46598d28632e79ca

   {client}  derive secret "exporter master secret" (same as server)

   {client}  derive read traffic keys using label "application data"
      (same as server write traffic keys)

   {client}  calculate finished:

      PRK (32 octets):  f737c2b29be2ef48 9d145dd3df485103
         86e812edcf799925 27e9ad5479967193

      handshake hash (0 octets):  (empty)

      info (21 octets):  002011544c532031 2e332c2066696e69 7368656400

      output (32 octets):  ea2fe9596714c959 d1cdd8f8cd893b96
         6429ee678bc7105e a10e6b4c03e2425a

   {client}  send a Finished handshake message

   {client}  send record:

      cleartext (36 octets):  1400002078367856 d3c8cc4e0a95eb98
         906ca7a48bd3cc70 29f48bd4ae0dc91a b903ca89
```

```
      ciphertext (58 octets):  1703010035fa15e9 2daa21cd05d8f9c3
         152a61748d9aaf04 9da559718e583f95 aacecad657b52a65
         62da09a5819e864d 86ac2989360a1eb2 2795

   {client}  derive write traffic keys using label "application data":

      PRK (32 octets):  2a1d25e6f9f13f92 e4b482fa06bc4447
         1218368d2d4e03e0 504d4e342b16ff8f

      key info (16 octets):  00100c544c532031 2e332c206b657900

      key output (16 octets):  eb23a804904b80ba 4fe8399e09b1ce42

      iv info (15 octets):  000c0b544c532031 2e332c20697600

      iv output (12 octets):  efa8c50c06b9c9b8 c483e174

   {client}  derive secret "resumption master secret":

      PRK (32 octets):  cab4645a3995d0d8 5bea9942596284e7
         2058a3d4d8f3e0d9 885aa92c517ad9e4

      handshake hash (32 octets):  e74cc34c780d9562 b1b3e7321f2ebcb0
         e6646246dbae060d 5d1335ac5f8db917

      info (69 octets):  002021544c532031 2e332c2072657375
         6d7074696f6e206d 6173746572207365 6372657420e74cc3
         4c780d9562b1b3e7 321f2ebcb0e66462 46dbae060d5d1335 ac5f8db917

      output (32 octets):  05438edfa0f6e663 0d7a9ffe81dc6773
         6d753a4ee351a79d 296975918b16039e

   {server}  calculate finished:

      PRK (32 octets):  f737c2b29be2ef48 9d145dd3df485103
         86e812edcf799925 27e9ad5479967193

      handshake hash (0 octets):  (empty)

      info (21 octets):  002011544c532031 2e332c2066696e69 7368656400

      output (32 octets):  ea2fe9596714c959 d1cdd8f8cd893b96
         6429ee678bc7105e a10e6b4c03e2425a

   {server}  derive read traffic keys using label "application data"
      (same as client write traffic keys)

   {server}  derive secret "resumption master secret" (same as client)
```

```
{server}  send a SessionTicket handshake message

{server}  send record:

   cleartext (170 octets):  040000a60002a300 4abe594b00924e53
      5321cadc96238da0 9caf9b02fecafdd6 5e3e418f03e43772
      cf512ed806610050 3b1c08abbbf298a9 d138ce821dd12fe1
      710e2137cd12e6a8 5cd3fd7f73706e7f 5dddefb87c1ef838
      24638464099c9d13 63e3c64ed2075c16 b8ccd8e524a6bbd7
      a6a6e34ea1579782 b15bbe7dfed5c0c0 d980fb330f9d8ab2
      52ffe7be1277d418 b6828ead4dae3b30 d448442417ef76af
      0008002e00040002 0000

   ciphertext (192 octets):  17030100bb45a662 6fa13b66ce2c5b3e
      f807e299a118296f 26a2dd9ec7487a06 73e2460d4c79f400
      87dcd014c59c5137 9c90d26b4e4f9bb2 b78f5b6761594f01
      3ff3e4c78d836905 229eac811c4ef8b2 faa89867e9ffc586
      f7f03c216591aa5e 620eac3c62dfe60f 846036bd7ecc4464
      b584af184e9644e9 4ee1d7834dba408a 51cbe4248004796e
      d9c558e0f5f96115 a6f6ba487e17d16a 2e20a3d3a650a9a0
      70fb53d9da82864b 5621d77650bd0c79 47e9889917b53d05
      15627c72b0ded521

{client}  send record:

   cleartext (50 octets):  0001020304050607 08090a0b0c0d0e0f
      1011121314151617 18191a1b1c1d1e1f 2021222324252627
      28292a2b2c2d2e2f 3031

   ciphertext (72 octets):  1703010043e30617 8ad97f74bb64f35e
      af3c39846b83aef8 472cbc9046749b81 a949dfb12cfbc65c
      babd20ade92c1f94 4605892ceeb12fde e8a927bce77c8303
      6ac5a794a8f54a69

{server}  send record:

   cleartext (50 octets):  0001020304050607 08090a0b0c0d0e0f
      1011121314151617 18191a1b1c1d1e1f 2021222324252627
      28292a2b2c2d2e2f 3031

   ciphertext (72 octets):  170301004346 7d99 a807dbf778e6ffd8
      be52456c70665f89 0811ef2f3c495d5b be983feedab0c251
      dde596bc7e2b1359 09ec9f9166fb0152 e8c16a84e4b10392
      56467f9538be4463

{client}  send record:

   cleartext (2 octets):  0100
```

```
      ciphertext (24 octets):  17030100136bdf60 847ba6fb650da36e
         872adc684a4af2e8

   {server}  send record:

      cleartext (2 octets):  0100

      ciphertext (24 octets):  1703010013621b7c c1962cd8a70109fe
         e68a52efedf87d2e
```

4. Resumed 0-RTT Handshake

This handshake resumes from the handshake in Section 3. Since the server provided a session ticket that permitted 0-RTT, and the client is configured for 0-RTT, the client is able to send 0-RTT data.

```
   {client}  create an ephemeral x25519 key pair:

      private key (32 octets):  0944d93ff58c924f a9d8915d05ab99cb
         48eb9d3c932710a6 e44feb46b1ded481

      public key (32 octets):  2c1a71f7cedf5fad 8e8433be7c85533a
         615a8d1140c8984d bfdf5391e18b4e74

   {client}  extract secret "early":

      salt (0 octets):  (empty)

      ikm (32 octets):  05438edfa0f6e663 0d7a9ffe81dc6773
         6d753a4ee351a79d 296975918b16039e

      secret (32 octets):  99853a47f018f8b2 123e742a14b06549
         87fd96262ec8b893 e3dc5c087dc10f4f

   {client}  derive secret "resumption psk binder key":

      PRK (32 octets):  99853a47f018f8b2 123e742a14b06549
         87fd96262ec8b893 e3dc5c087dc10f4f

      handshake hash (32 octets):  e3b0c44298fc1c14 9afbf4c8996fb924
         27ae41e4649b934c a495991b7852b855

      info (70 octets):  002022544c532031 2e332c2072657375
         6d7074696f6e2070 736b2062696e6465 72206b657920e3b0
         c44298fc1c149afb f4c8996fb92427ae 41e4649b934ca495 991b7852b855

      output (32 octets):  1590d475bebda581 fd7d7008a92140d9
         baf1b75bfcb7e033 a736591ecba7bb42
```

```
{client}  derive secret "early exporter master secret":

   PRK (32 octets):  99853a47f018f8b2 123e742a14b06549
      87fd96262ec8b893 e3dc5c087dc10f4f

   handshake hash (32 octets):  e3b0c44298fc1c14 9afbf4c8996fb924
      27ae41e4649b934c a495991b7852b855

   info (73 octets):  002025544c532031 2e332c206561726c
      79206578706f7274 6572206d61737465 7220736563726574
      20e3b0c44298fc1c 149afbf4c8996fb9 2427ae41e4649b93
      4ca495991b7852b8 55

   output (32 octets):  399ca522c8bdbd22 9a1db3f4f97632d4
      250ed6ecd5568419 6ba9953033956f94

{client}  send a ClientHello handshake message

{client}  calculate finished:

   PRK (32 octets):  1590d475bebda581 fd7d7008a92140d9
      baf1b75bfcb7e033 a736591ecba7bb42

   handshake hash (0 octets):  (empty)

   info (21 octets):  002011544c532031 2e332c2066696e69 7368656400

   output (32 octets):  fe36c444491b0082 e4683625da4dcadf
      99aebd2dab5a1621 ae25542ec266d6a7

{client}  send record:

   cleartext (512 octets):  010001fc030302d2 254d2bde0890e202
      8ebb36a14a128bce bc498d9ebcc5eaf0 c1d258cc0a290000
      3e130113031302c0 2bc02fcca9cca8c0 0ac009c013c023c0
      27c014009eccaa00 3300320067003900 38006b0016001300
      9c002f003c003500 3d000a0005000401 0001950015003b00
      0000000000000000 0000000000000000 0000000000000000
      0000000000000000 0000000000000000 0000000000000000
      0000000000000000 00000000000b0009 0000067365727665
      72ff01000100000a 00140012001d0017 0018001901000101
      010201030104000b 0002010000280026 0024001d00202c1a
      71f7cedf5fad8e84 33be7c85533a615a 8d1140c8984dbfdf
      5391e18b4e74002a 0000002b0007067f 1203030302000d00
      20001e0403050306 0302030804080508 0604010501060102
      0104020502060202 02002d0002010100 2900bd009800924e
      535321cadc96238d a09caf9b02fecafd d65e3e418f03e437
      72cf512ed8066100 503b1c08abbbf298 a9d138ce821dd12f
```

```
         e1710e2137cd12e6 a85cd3fd7f73706e 7f5dddefb87c1ef8
         3824638464099c9d 1363e3c64ed2075c 16b8ccd8e524a6bb
         d7a6a6e34ea15797 82b15bbe7dfed5c0 c0d980fb330f9d8a
         b252ffe7be1277d4 18b6828ead4dae3b 30d448442417ef76
         af4abe594b002120 56d264e68d59a053 7d872a47a2f0a72d
         5051f1aa5dcbbc5d a1e43ec781580e0a

      ciphertext (517 octets):  1603010200010001 fc030302d2254d2b
         de0890e2028ebb36 a14a128bcebc498d 9ebcc5eaf0c1d258
         cc0a2900003e1301 13031302c02bc02f cca9cca8c00ac009
         c013c023c027c014 009eccaa00330032 006700390038006b
         00160013009c002f 003c0035003d000a 0005000401000195
         0015003b00000000 0000000000000000 0000000000000000
         0000000000000000 0000000000000000 0000000000000000
         0000000000000000 0000000000000000 00000b0009000006
         736572766572ff01 000100000a001400 12001d0017001800
         1901000101010201 030104000b000201 0000280026002400
         1d00202c1a71f7ce df5fad8e8433be7c 85533a615a8d1140
         c8984dbfdf5391e1 8b4e74002a000000 2b0007067f120303
         0302000d0020001e 0403050306030203 0804080508060401
         0501060102010402 0502060202002d 00020101002900bd
         009800924e535321 cadc96238da09caf 9b02fecafdd65e3e
         418f03e43772cf51 2ed8066100503b1c 08abbbf298a9d138
         ce821dd12fe1710e 2137cd12e6a85cd3 fd7f73706e7f5ddd
         efb87c1ef8382463 8464099c9d1363e3 c64ed2075c16b8cc
         d8e524a6bbd7a6a6 e34ea1579782b15b be7dfed5c0c0d980
         fb330f9d8ab252ff e7be1277d418b682 8ead4dae3b30d448
         442417ef76af4abe 594b00212056d264 e68d59a0537d872a
         47a2f0a72d5051f1 aa5dcbbc5da1e43e c781580e0a

   {client}  derive secret "client early traffic secret":

      PRK (32 octets):  99853a47f018f8b2 123e742a14b06549
         87fd96262ec8b893 e3dc5c087dc10f4f

      handshake hash (32 octets):  5abe42e4bb8e0bcf f118e9e02e78c793
         c0f8bf0461a62ce4 5a7c541edf06c204

      info (72 octets):  002024544c532031 2e332c20636c6965
         6e74206561726c79 2074726166666963 2073656372657420
         5abe42e4bb8e0bcf f118e9e02e78c793 c0f8bf0461a62ce4
         5a7c541edf06c204

      output (32 octets):  560df53cb4604f16 954e5f63869fcf11
         d656be054f92c803 f93017a506032016

   {client}  derive write traffic keys using label "early application
      data":
```

```
      PRK (32 octets):  560df53cb4604f16 954e5f63869fcf11
         d656be054f92c803 f93017a506032016

      key info (16 octets):  00100c544c532031 2e332c206b657900

      key output (16 octets):  ee1188babbf83c53 5f8fa55f8f8a20a7

      iv info (15 octets):  000c0b544c532031 2e332c20697600

      iv output (12 octets):  22a3d48298c8b820 bef80201

   {client}  send record:

      cleartext (6 octets):  414243444546

      ciphertext (28 octets):  1703010017c07b71 c7200dab007e9ebc
         45c182721f06cd88 6bf785ab

   {server}  extract secret "early" (same as client)

   {server}  derive secret "resumption psk binder key":

      PRK (32 octets):  99853a47f018f8b2 123e742a14b06549
         87fd96262ec8b893 e3dc5c087dc10f4f

      handshake hash (32 octets):  e3b0c44298fc1c14 9afbf4c8996fb924
         27ae41e4649b934c a495991b7852b855

      info (70 octets):  002022544c532031 2e332c2072657375
         6d7074696f6e2070 736b2062696e6465 72206b657920e3b0
         c44298fc1c149afb f4c8996fb92427ae 41e4649b934ca495 991b7852b855

      output (32 octets):  1590d475bebda581 fd7d7008a92140d9
         baf1b75bfcb7e033 a736591ecba7bb42

   {server}  derive secret "early exporter master secret":

      PRK (32 octets):  99853a47f018f8b2 123e742a14b06549
         87fd96262ec8b893 e3dc5c087dc10f4f

      handshake hash (32 octets):  e3b0c44298fc1c14 9afbf4c8996fb924
         27ae41e4649b934c a495991b7852b855

      info (73 octets):  002025544c532031 2e332c206561726c
         79206578706f7274 6572206d61737465 7220736563726574
         20e3b0c44298fc1c 149afbf4c8996fb9 2427ae41e4649b93
         4ca495991b7852b8 55
```

```
      output (32 octets):  399ca522c8bdbd22 9a1db3f4f97632d4
         250ed6ecd5568419 6ba9953033956f94

   {server}  calculate finished:

      PRK (32 octets):  1590d475bebda581 fd7d7008a92140d9
         baf1b75bfcb7e033 a736591ecba7bb42

      handshake hash (0 octets):  (empty)

      info (21 octets):  002011544c532031 2e332c2066696e69 7368656400

      output (32 octets):  fe36c444491b0082 e4683625da4dcadf
         99aebd2dab5a1621 ae25542ec266d6a7

   {server}  create an ephemeral x25519 key pair:

      private key (32 octets):  084cf2ecb7e94256 f575cd6e3dde2f21
         9c4f9029143e4f6a 85e86700b7d5eb77

      public key (32 octets):  7897ec11458a449d 3c73f5e3846c5062
         8c35faa8876e602e 996c2620deafbe0d

   {server}  derive secret "client early traffic secret" (same as
      client)

   {server}  send a ServerHello handshake message

   {server}  extract secret "handshake":

      salt (32 octets):  99853a47f018f8b2 123e742a14b06549
         87fd96262ec8b893 e3dc5c087dc10f4f

      ikm (32 octets):  7edd226788b92bf9 3b2b33396e06ef84
         059693fa9c199da2 3f41224c2b84e97d

      secret (32 octets):  6423cd6207ff4ea4 7b73af91b6f8db82
         706e5ee4691b27ca 3c743445186ed12c

   {server}  derive secret "client handshake traffic secret":

      PRK (32 octets):  6423cd6207ff4ea4 7b73af91b6f8db82
         706e5ee4691b27ca 3c743445186ed12c

      handshake hash (32 octets):  a80310ec3b531838 1c8db6495965f2fa
         cf9ca85a391fcf37 d85cadd1bc7443d4
```

```
      info (76 octets):  002028544c532031 2e332c20636c6965
         6e742068616e6473 68616b6520747261 6666696320736563
         72657420a80310ec 3b5318381c8db649 5965f2facf9ca85a
         391fcf37d85cadd1 bc7443d4

      output (32 octets):  d60ef6f4d7eda53d cc21d02d26ebd575
         f9663f84ef4af32e 5bed4fbb6af833e0

   {server}  derive secret "server handshake traffic secret":

      PRK (32 octets):  6423cd6207ff4ea4 7b73af91b6f8db82
         706e5ee4691b27ca 3c743445186ed12c

      handshake hash (32 octets):  a80310ec3b531838 1c8db6495965f2fa
         cf9ca85a391fcf37 d85cadd1bc7443d4

      info (76 octets):  002028544c532031 2e332c2073657276
         65722068616e6473 68616b6520747261 6666696320736563
         72657420a80310ec 3b5318381c8db649 5965f2facf9ca85a
         391fcf37d85cadd1 bc7443d4

      output (32 octets):  c41576b7adda04fb eb128b8cb48e4b46
         e9954abc6dd2dfc3 0856d028dedcfdd7

   {server}  extract secret "master":

      salt (32 octets):  6423cd6207ff4ea4 7b73af91b6f8db82
         706e5ee4691b27ca 3c743445186ed12c

      ikm (32 octets):  0000000000000000 0000000000000000
         0000000000000000 0000000000000000

      secret (32 octets):  838095f760b7ff7a 207ff3c3c818e6f9
         86c87db36fcf063f 09e8451dc55b97e2

   {server}  derive write traffic keys using label "handshake data":

      PRK (32 octets):  c41576b7adda04fb eb128b8cb48e4b46
         e9954abc6dd2dfc3 0856d028dedcfdd7

      key info (16 octets):  00100c544c532031 2e332c206b657900

      key output (16 octets):  3b6b7a6360a82cf2 5bf22e59e3d170c3

      iv info (15 octets):  000c0b544c532031 2e332c20697600

      iv output (12 octets):  3e94717fb3af82cd e82642b9
```

```
   {server}  send record:

      cleartext (88 octets):  020000547f124f9b fff8d7d6e5e445e8
         67330150aa680274 59e8d59262ac183e a8d7e5b9c4981301
         002e002900020000 00280024001d0020 7897ec11458a449d
         3c73f5e3846c5062 8c35faa8876e602e 996c2620deafbe0d

      ciphertext (93 octets):  1603010058020000 547f124f9bfff8d7
         d6e5e445e8673301 50aa68027459e8d5 9262ac183ea8d7e5
         b9c4981301002e00 2900020000002800 24001d00207897ec
         11458a449d3c73f5 e3846c50628c35fa a8876e602e996c26 20deafbe0d

   {server}  send a EncryptedExtensions handshake message

   {server}  calculate finished:

      PRK (32 octets):  c41576b7adda04fb eb128b8cb48e4b46
         e9954abc6dd2dfc3 0856d028dedcfdd7

      handshake hash (0 octets):  (empty)

      info (21 octets):  002011544c532031 2e332c2066696e69 7368656400

      output (32 octets):  675bd9b07376e6a6 56ef9fbe9297ce8a
         cabbc804e1001d0d c4a810b918aad2d3

   {server}  send a Finished handshake message

   {server}  send record:

      cleartext (74 octets):  080000220020000a 00140012001d0017
         0018001901000101 0102010301040000 0000002a00001400
         00206b2d3c33b880 827d22789897cf52 ced3a06fd4a1b927
         106cad93e8145ecf e9ee

      ciphertext (96 octets):  170301005b29076d 479ff50c63291217
         5bc8d31b77425359 8be825a729656425 3acf12baa202f07a
         29c686489aa76bb5 d8b1bb64d6502ee9 7954302c4a8a528f
         f27506e35fabb67b 7bf7623cfb23ac56 24942c10ffbae8a7
         79ffcec31860a481

   {server}  derive secret "client application traffic secret":

      PRK (32 octets):  838095f760b7ff7a 207ff3c3c818e6f9
         86c87db36fcf063f 09e8451dc55b97e2

      handshake hash (32 octets):  25678f29cd74c323 e2c410f6163f1560
         8bbe70f367f330f9 f316a3b91a98a5cb
```

```
      info (78 octets):  00202a544c532031 2e332c20636c6965
         6e74206170706c69 636174696f6e2074 7261666669632073
         6563726574202567 8f29cd74c323e2c4 10f6163f15608bbe
         70f367f330f9f316 a3b91a98a5cb

      output (32 octets):  642d05445f11316d d9f94a0b64af1f07
         37ca6429219cd7fb 1f33c4b2fe3ab632

   {server}  derive secret "server application traffic secret":

      PRK (32 octets):  838095f760b7ff7a 207ff3c3c818e6f9
         86c87db36fcf063f 09e8451dc55b97e2

      handshake hash (32 octets):  25678f29cd74c323 e2c410f6163f1560
         8bbe70f367f330f9 f316a3b91a98a5cb

      info (78 octets):  00202a544c532031 2e332c2073657276
         6572206170706c69 636174696f6e2074 7261666669632073
         6563726574202567 8f29cd74c323e2c4 10f6163f15608bbe
         70f367f330f9f316 a3b91a98a5cb

      output (32 octets):  125f0e573a686d07 92ed788646fedd3e
         4407728929607077 745cd1a98f240daa

   {server}  derive secret "exporter master secret":

      PRK (32 octets):  838095f760b7ff7a 207ff3c3c818e6f9
         86c87db36fcf063f 09e8451dc55b97e2

      handshake hash (32 octets):  25678f29cd74c323 e2c410f6163f1560
         8bbe70f367f330f9 f316a3b91a98a5cb

      info (67 octets):  00201f544c532031 2e332c206578706f
         72746572206d6173 7465722073656372 65742025678f29cd
         74c323e2c410f616 3f15608bbe70f367 f330f9f316a3b91a 98a5cb

      output (32 octets):  94afc03877de24ce 1a14ecd098ad891c
         5d54b37369bc98f8 3c136fb7f56e1490

   {server}  derive write traffic keys using label "application data":

      PRK (32 octets):  125f0e573a686d07 92ed788646fedd3e
         4407728929607077 745cd1a98f240daa

      key info (16 octets):  00100c544c532031 2e332c206b657900

      key output (16 octets):  dab117e37b791fec 925a71f88c376fa6
```

```
      iv info (15 octets):  000c0b544c532031 2e332c20697600

      iv output (12 octets):  bbe980ebee1ba6c0 38a2e244

   {server}  derive read traffic keys using label "early application
      data" (same as client write traffic keys)

   {client}  extract secret "handshake":

      salt (32 octets):  99853a47f018f8b2 123e742a14b06549
         87fd96262ec8b893 e3dc5c087dc10f4f

      ikm (32 octets):  7edd226788b92bf9 3b2b33396e06ef84
         059693fa9c199da2 3f41224c2b84e97d

      secret (32 octets):  6423cd6207ff4ea4 7b73af91b6f8db82
         706e5ee4691b27ca 3c743445186ed12c

   {client}  derive secret "client handshake traffic secret":

      PRK (32 octets):  6423cd6207ff4ea4 7b73af91b6f8db82
         706e5ee4691b27ca 3c743445186ed12c

      handshake hash (32 octets):  a80310ec3b531838 1c8db6495965f2fa
         cf9ca85a391fcf37 d85cadd1bc7443d4

      info (76 octets):  002028544c532031 2e332c20636c6965
         6e742068616e6473 68616b6520747261 6666696320736563
         72657420a80310ec 3b5318381c8db649 5965f2facf9ca85a
         391fcf37d85cadd1 bc7443d4

      output (32 octets):  d60ef6f4d7eda53d cc21d02d26ebd575
         f9663f84ef4af32e 5bed4fbb6af833e0

   {client}  derive secret "server handshake traffic secret":

      PRK (32 octets):  6423cd6207ff4ea4 7b73af91b6f8db82
         706e5ee4691b27ca 3c743445186ed12c

      handshake hash (32 octets):  a80310ec3b531838 1c8db6495965f2fa
         cf9ca85a391fcf37 d85cadd1bc7443d4

      info (76 octets):  002028544c532031 2e332c2073657276
         65722068616e6473 68616b6520747261 6666696320736563
         72657420a80310ec 3b5318381c8db649 5965f2facf9ca85a
         391fcf37d85cadd1 bc7443d4
```

```
      output (32 octets):  c41576b7adda04fb eb128b8cb48e4b46
         e9954abc6dd2dfc3 0856d028dedcfdd7

   {client}  extract secret "master" (same as server)

   {client}  derive read traffic keys using label "handshake data":

      PRK (32 octets):  c41576b7adda04fb eb128b8cb48e4b46
         e9954abc6dd2dfc3 0856d028dedcfdd7

      key info (16 octets):  00100c544c532031 2e332c206b657900

      key output (16 octets):  3b6b7a6360a82cf2 5bf22e59e3d170c3

      iv info (15 octets):  000c0b544c532031 2e332c20697600

      iv output (12 octets):  3e94717fb3af82cd e82642b9

   {client}  calculate finished:

      PRK (32 octets):  c41576b7adda04fb eb128b8cb48e4b46
         e9954abc6dd2dfc3 0856d028dedcfdd7

      handshake hash (0 octets):  (empty)

      info (21 octets):  002011544c532031 2e332c2066696e69 7368656400

      output (32 octets):  675bd9b07376e6a6 56ef9fbe9297ce8a
         cabbc804e1001d0d c4a810b918aad2d3

   {client}  send record:

      cleartext (2 octets):  0101

      ciphertext (24 octets):  17030100130aba56 52f18ac0971329d7
         5fa54b8d4477f693

   {client}  derive write traffic keys using label "handshake data":

      PRK (32 octets):  d60ef6f4d7eda53d cc21d02d26ebd575
         f9663f84ef4af32e 5bed4fbb6af833e0

      key info (16 octets):  00100c544c532031 2e332c206b657900

      key output (16 octets):  bd8d8cc78152c42f 15b5d2ae85d85391

      iv info (15 octets):  000c0b544c532031 2e332c20697600
```

```
      iv output (12 octets):  9e379b5677dda474 9dd45fd5

   {client}  derive secret "client application traffic secret":

      PRK (32 octets):  838095f760b7ff7a 207ff3c3c818e6f9
         86c87db36fcf063f 09e8451dc55b97e2

      handshake hash (32 octets):  25678f29cd74c323 e2c410f6163f1560
         8bbe70f367f330f9 f316a3b91a98a5cb

      info (78 octets):  00202a544c532031 2e332c20636c6965
         6e74206170706c69 636174696f6e2074 7261666669632073
         6563726574202567 8f29cd74c323e2c4 10f6163f15608bbe
         70f367f330f9f316 a3b91a98a5cb

      output (32 octets):  642d05445f11316d d9f94a0b64af1f07
         37ca6429219cd7fb 1f33c4b2fe3ab632

   {client}  derive secret "server application traffic secret":

      PRK (32 octets):  838095f760b7ff7a 207ff3c3c818e6f9
         86c87db36fcf063f 09e8451dc55b97e2

      handshake hash (32 octets):  25678f29cd74c323 e2c410f6163f1560
         8bbe70f367f330f9 f316a3b91a98a5cb

      info (78 octets):  00202a544c532031 2e332c2073657276
         6572206170706c69 636174696f6e2074 7261666669632073
         6563726574202567 8f29cd74c323e2c4 10f6163f15608bbe
         70f367f330f9f316 a3b91a98a5cb

      output (32 octets):  125f0e573a686d07 92ed788646fedd3e
         4407728929607077 745cd1a98f240daa

   {client}  derive secret "exporter master secret" (same as server)

   {client}  derive read traffic keys using label "application data"
      (same as server write traffic keys)

   {client}  calculate finished:

      PRK (32 octets):  d60ef6f4d7eda53d cc21d02d26ebd575
         f9663f84ef4af32e 5bed4fbb6af833e0

      handshake hash (0 octets):  (empty)

      info (21 octets):  002011544c532031 2e332c2066696e69 7368656400
```

```
      output (32 octets):  9d18ee7c846ea450 0c9884d3b3741107
         1cb93b42db69a46c 101e65e976a20417

   {client}  send a Finished handshake message

   {client}  send record:

      cleartext (36 octets):  1400002055f849f1 a03006f7ec3d5384
         aba84782b4c37df3 d3c7b92543d5e8b0 24b38aea

      ciphertext (58 octets):  170301003561ad40 384d8ffd77d6ea42
         28ca06247041fccf edc89e8f4f575a3b 79a01e61f6d3961a
         5a6251e79594620a 62067c3a245dff64 b2fe

   {client}  derive write traffic keys using label "application data":

      PRK (32 octets):  642d05445f11316d d9f94a0b64af1f07
         37ca6429219cd7fb 1f33c4b2fe3ab632

      key info (16 octets):  00100c544c532031 2e332c206b657900

      key output (16 octets):  ed504bf560f8c1e6 867659dd6527cdfa

      iv info (15 octets):  000c0b544c532031 2e332c20697600

      iv output (12 octets):  005434eeaac2d2b6 b3dc186d

   {client}  derive secret "resumption master secret":

      PRK (32 octets):  838095f760b7ff7a 207ff3c3c818e6f9
         86c87db36fcf063f 09e8451dc55b97e2

      handshake hash (32 octets):  10e631557cc36de9 c9e1698cd932420d
         8388263513d401f0 a8a2d5bbf8ab8500

      info (69 octets):  002021544c532031 2e332c2072657375
         6d7074696f6e206d 6173746572207365 637265742010e631
         557cc36de9c9e169 8cd932420d838826 3513d401f0a8a2d5 bbf8ab8500

      output (32 octets):  ddb7ba1feb09673a ebc36db7e08c410b
         de864b2eb7be9bda ded9be89bac6649c

   {server}  derive read traffic keys using label "handshake data":

      PRK (32 octets):  d60ef6f4d7eda53d cc21d02d26ebd575
         f9663f84ef4af32e 5bed4fbb6af833e0

      key info (16 octets):  00100c544c532031 2e332c206b657900
```

```
      key output (16 octets):  bd8d8cc78152c42f 15b5d2ae85d85391

      iv info (15 octets):  000c0b544c532031 2e332c20697600

      iv output (12 octets):  9e379b5677dda474 9dd45fd5

   {server}  calculate finished:

      PRK (32 octets):  d60ef6f4d7eda53d cc21d02d26ebd575
         f9663f84ef4af32e 5bed4fbb6af833e0

      handshake hash (0 octets):  (empty)

      info (21 octets):  002011544c532031 2e332c2066696e69 7368656400

      output (32 octets):  9d18ee7c846ea450 0c9884d3b3741107
         1cb93b42db69a46c 101e65e976a20417

   {server}  derive read traffic keys using label "application data"
      (same as client write traffic keys)

   {server}  derive secret "resumption master secret" (same as client)

   {client}  send record:

      cleartext (50 octets):  0001020304050607 08090a0b0c0d0e0f
         1011121314151617 18191a1b1c1d1e1f 2021222324252627
         28292a2b2c2d2e2f 3031

      ciphertext (72 octets):  1703010043215c81 57730ca2101ad6ee
         50335a7216d5565e 3391c1d920b4c126 4285994032dbe9bc
         f077bfdd6f0fa1c9 e0c610c0b74605b2 a24448e4a7cb45ef
         8b0193ea95b4d860

   {server}  send record:

      cleartext (50 octets):  0001020304050607 08090a0b0c0d0e0f
         1011121314151617 18191a1b1c1d1e1f 2021222324252627
         28292a2b2c2d2e2f 3031

      ciphertext (72 octets):  17030100434255b4 8f15b947f760ed76
         29e130e5d4aaabea 7d06fa74fd3c9901 0997853776caf2c6
         5c8ccc6e33567dc7 f4ac50467eddf42c c76241aeda237a07
         422ac51a643773e9

   {client}  send record:

      cleartext (2 octets):  0100
```

```
      ciphertext (24 octets):  1703010013422dd5 2ef4a92aaac69e06
         6846b7e507d4a2ca

   {server}  send record:

      cleartext (2 octets):  0100

      ciphertext (24 octets):  1703010013c6f797 8bf3ce7e86f54ffe
         a9edc9e61dfdd967
```

5. Security Considerations

It probably isn't a good idea to use the private key here. If it weren't for the fact that it is too small to provide any meaningful security, it is now very well known.

6. Normative References

[I-D.ietf-tls-tls13]
Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", draft-ietf-tls-tls13-18 (work in progress), October 2016.

Appendix A. Acknowledgements

None of this would have been possible without Franziskus Kiefer, Eric Rescorla and Tim Taubert, who did a lot of the work in NSS.

Author's Address

Martin Thomson
Mozilla

Email: martin.thomson@gmail.com