Network Working Group C. Bartle
Internet-Draft Apple, Inc.
Intended status: Standards Track N. Aviram
Expires: 28 August 2021

F. Valsorda
24 February 2021

Deprecating FFDH(E) Ciphersuites in TLS
draft-bartle-tls-deprecate-ffdhe-00

Abstract

This document deprecates and discourages use of finite field and elliptic curve Diffie Hellman cipher suites that have known vulnerabilities or improper security properties when implemented incorrectly.

Discussion Venues

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

Discussion of this document takes place on the Transport Layer Security Working Group mailing list (tls@ietf.org), which is archived at https://mailarchive.ietf.org/arch/browse/tls/.

Source for this draft and an issue tracker can be found at https://github.com/cbartle891/draft-deprecate-ffdhe.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

TLS supports a variety of key exchange algorithms, including those based on finite field and elliptic curve Diffie Hellman (DH) groups. Each of these also come in ephemeral and non-ephemeral varieties. Non-ephemeral DH algorithms use static DH public keys included in the authenticating peer's certificate; see [RFC4492] for discussion. In contrast, ephemeral DH algorithms use ephemeral DH public keys sent in the handshake and authenticated by the peer's certificate. Ephemeral and non-ephemeral finite field DH algorithms are called DHE and DH, respectively, and ephemeral and non-ephemeral elliptic curve DH algorithms are called ECDHE and ECDH, respectively [RFC4492].

In general, non-ephemeral cipher suites are not recommended due to their lack of forward secrecy. However, as demonstrated by the [Raccoon] attack, public key reuse, either via non-ephemeral cipher suites or reused keys with ephemeral cipher suites, can lead to timing side channels that may leak connection secrets. (Note that Raccoon only applies to finite field DH cipher suites, and not those based on elliptic curves.) While these side channels can be avoided in implementations, doing is demonstrably difficult given the prevalence of related side channels in TLS implementations.

Given these problems, this document updates [RFC4346], [RFC5246], [RFC4162], [RFC6347], [RFC5932], [RFC5288], [RFC6209], [RFC6367], [RFC8422], [RFC5289], and [RFC5469] to deprecate, prohibiting and discouraging, cipher suites with key reuse.

1.1. Requirements

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. Non-Ephemeral Diffie Hellman

Clients MUST NOT offer non-ephemeral DH cipher suites in TLS 1.0, 1.1, and 1.2 connections. This includes all cipher suites listed in the following table.

Ciphersuite	Reference
TLS_DH_DSS_EXPORT_WITH_DES40_CBC_SHA	[RFC4346]

TLS_DH_DSS_WITH_DES_CBC_SHA	[RFC5469]
TLS_DH_DSS_WITH_3DES_EDE_CBC_SHA	[RFC5246]
TLS_DH_RSA_EXPORT_WITH_DES40_CBC_SHA	[RFC4346]
TLS_DH_RSA_WITH_DES_CBC_SHA	[RFC5469]
TLS_DH_RSA_WITH_3DES_EDE_CBC_SHA	[RFC5246]
TLS_DH_anon_EXPORT_WITH_RC4_40_MD5	[RFC4346][RFC6347]
TLS_DH_anon_WITH_RC4_128_MD5	[RFC5246][RFC6347]
TLS_DH_anon_EXPORT_WITH_DES40_CBC_SHA	[RFC4346]
TLS_DH_anon_WITH_DES_CBC_SHA	[RFC5469]
TLS_DH_anon_WITH_3DES_EDE_CBC_SHA	[RFC5246]
TLS_DH_DSS_WITH_AES_128_CBC_SHA	[RFC5246]
TLS_DH_RSA_WITH_AES_128_CBC_SHA	[RFC5246]
TLS_DH_anon_WITH_AES_128_CBC_SHA	[RFC5246]
TLS_DH_DSS_WITH_AES_256_CBC_SHA	[RFC5246]
TLS_DH_RSA_WITH_AES_256_CBC_SHA	[RFC5246]
TLS_DH_anon_WITH_AES_256_CBC_SHA	[RFC5246]
TLS_DH_DSS_WITH_AES_128_CBC_SHA256	[RFC5246]
TLS_DH_RSA_WITH_AES_128_CBC_SHA256	[RFC5246]
TLS_DH_DSS_WITH_CAMELLIA_128_CBC_SHA	[RFC5932]
TLS_DH_RSA_WITH_CAMELLIA_128_CBC_SHA	[RFC5932]
TLS_DH_anon_WITH_CAMELLIA_128_CBC_SHA	[RFC5932]
TLS_DH_DSS_WITH_AES_256_CBC_SHA256	[RFC5246]
TLS_DH_RSA_WITH_AES_256_CBC_SHA256	[RFC5246]
TLS_DH_anon_WITH_AES_128_CBC_SHA256	[RFC5246]
TLS_DH_anon_WITH_AES_256_CBC_SHA256	[RFC5246]
TLS_DH_DSS_WITH_CAMELLIA_256_CBC_SHA	[RFC5932]
TLS_DH_RSA_WITH_CAMELLIA_256_CBC_SHA	[RFC5932]
TLS_DH_anon_WITH_CAMELLIA_256_CBC_SHA	[RFC5932]
TLS_DH_DSS_WITH_SEED_CBC_SHA	[RFC4162]
TLS_DH_RSA_WITH_SEED_CBC_SHA	[RFC4162]
TLS_DH_anon_WITH_SEED_CBC_SHA	[RFC4162]

TLS_DH_RSA_WITH_AES_128_GCM_SHA256	[RFC5288]
TLS_DH_RSA_WITH_AES_256_GCM_SHA384	[RFC5288]
TLS_DH_DSS_WITH_AES_128_GCM_SHA256	[RFC5288]
TLS_DH_DSS_WITH_AES_256_GCM_SHA384	[RFC5288]
TLS_DH_anon_WITH_AES_128_GCM_SHA256	[RFC5288]
TLS_DH_anon_WITH_AES_256_GCM_SHA384	[RFC5288]
TLS_DH_DSS_WITH_CAMELLIA_128_CBC_SHA256	[RFC5932]
TLS_DH_RSA_WITH_CAMELLIA_128_CBC_SHA256	[RFC5932]
TLS_DH_anon_WITH_CAMELLIA_128_CBC_SHA256	[RFC5932]
TLS_DH_DSS_WITH_CAMELLIA_256_CBC_SHA256	[RFC5932]
TLS_DH_RSA_WITH_CAMELLIA_256_CBC_SHA256	[RFC5932]
TLS_DH_anon_WITH_CAMELLIA_256_CBC_SHA256	[RFC5932]
TLS_DH_DSS_WITH_ARIA_128_CBC_SHA256	[RFC6209]
TLS_DH_DSS_WITH_ARIA_256_CBC_SHA384	[RFC6209]
TLS_DH_RSA_WITH_ARIA_128_CBC_SHA256	[RFC6209]
TLS_DH_RSA_WITH_ARIA_256_CBC_SHA384	[RFC6209]
TLS_DH_anon_WITH_ARIA_128_CBC_SHA256	[RFC6209]
TLS_DH_anon_WITH_ARIA_256_CBC_SHA384	[RFC6209]
TLS_DH_RSA_WITH_ARIA_128_GCM_SHA256	[RFC6209]
TLS_DH_RSA_WITH_ARIA_256_GCM_SHA384	[RFC6209]
TLS_DH_DSS_WITH_ARIA_128_GCM_SHA256	[RFC6209]
TLS_DH_DSS_WITH_ARIA_256_GCM_SHA384	[RFC6209]
TLS_DH_anon_WITH_ARIA_128_GCM_SHA256	[RFC6209]
TLS_DH_anon_WITH_ARIA_256_GCM_SHA384	[RFC6209]
TLS_DH_RSA_WITH_CAMELLIA_128_GCM_SHA256	[RFC6367]
TLS_DH_RSA_WITH_CAMELLIA_256_GCM_SHA384	[RFC6367]
TLS_DH_DSS_WITH_CAMELLIA_128_GCM_SHA256	[RFC6367]
TLS_DH_DSS_WITH_CAMELLIA_256_GCM_SHA384	[RFC6367]
TLS_DH_anon_WITH_CAMELLIA_128_GCM_SHA256	[RFC6367]
TLS_DH_anon_WITH_CAMELLIA_256_GCM_SHA384	[RFC6367]

Table 1

Clients SHOULD NOT offer non-ephemeral ECDH cipher suites in TLS 1.0, 1.1, and 1.2 connections. This includes all cipher suites listed in the following table.

Ciphersuite	Reference
TLS_ECDH_ECDSA_WITH_NULL_SHA	[RFC8422]
TLS_ECDH_ECDSA_WITH_RC4_128_SHA	[RFC8422][RFC6347]
TLS_ECDH_ECDSA_WITH_3DES_EDE_CBC_SHA	[RFC8422]
TLS_ECDH_ECDSA_WITH_AES_128_CBC_SHA	[RFC8422]
TLS_ECDH_ECDSA_WITH_AES_256_CBC_SHA	[RFC8422]
TLS_ECDH_RSA_WITH_NULL_SHA	[RFC8422]
TLS_ECDH_RSA_WITH_RC4_128_SHA	[RFC8422][RFC6347]
TLS_ECDH_RSA_WITH_3DES_EDE_CBC_SHA	[RFC8422]
TLS_ECDH_RSA_WITH_AES_128_CBC_SHA	[RFC8422]
TLS_ECDH_RSA_WITH_AES_256_CBC_SHA	[RFC8422]
TLS_ECDH_anon_WITH_NULL_SHA	[RFC8422]
TLS_ECDH_anon_WITH_RC4_128_SHA	[RFC8422][RFC6347]
TLS_ECDH_anon_WITH_3DES_EDE_CBC_SHA	[RFC8422]
TLS_ECDH_anon_WITH_AES_128_CBC_SHA	[RFC8422]
TLS_ECDH_anon_WITH_AES_256_CBC_SHA	[RFC8422]
TLS_ECDH_ECDSA_WITH_AES_128_CBC_SHA256	[RFC5289]
TLS_ECDH_ECDSA_WITH_AES_256_CBC_SHA384	[RFC5289]
TLS_ECDH_RSA_WITH_AES_128_CBC_SHA256	[RFC5289]
TLS_ECDH_RSA_WITH_AES_256_CBC_SHA384	[RFC5289]
TLS_ECDH_ECDSA_WITH_AES_128_GCM_SHA256	[RFC5289]
TLS_ECDH_ECDSA_WITH_AES_256_GCM_SHA384	[RFC5289]
TLS_ECDH_RSA_WITH_AES_128_GCM_SHA256	[RFC5289]
TLS_ECDH_RSA_WITH_AES_256_GCM_SHA384	[RFC5289]
TLS_ECDH_ECDSA_WITH_ARIA_128_CBC_SHA256	[RFC6209]
TLS_ECDH_ECDSA_WITH_ARIA_256_CBC_SHA384	[RFC6209]
TLS_ECDH_RSA_WITH_ARIA_128_CBC_SHA256	[RFC6209]
TLS_ECDH_RSA_WITH_ARIA_256_CBC_SHA384	[RFC6209]

TLS_ECDH_ECDSA_WITH_ARIA_128_GCM_SHA256	[RFC6209]
TLS_ECDH_ECDSA_WITH_ARIA_256_GCM_SHA384	[RFC6209]
TLS_ECDH_RSA_WITH_ARIA_128_GCM_SHA256	[RFC6209]
TLS_ECDH_RSA_WITH_ARIA_256_GCM_SHA384	[RFC6209]
TLS_ECDH_ECDSA_WITH_CAMELLIA_128_CBC_SHA256	[RFC6367]
TLS_ECDH_ECDSA_WITH_CAMELLIA_256_CBC_SHA384	[RFC6367]
TLS_ECDH_RSA_WITH_CAMELLIA_128_CBC_SHA256	[RFC6367]
TLS_ECDH_RSA_WITH_CAMELLIA_256_CBC_SHA384	[RFC6367]
TLS_ECDH_ECDSA_WITH_CAMELLIA_128_GCM_SHA256	[RFC6367]
TLS_ECDH_ECDSA_WITH_CAMELLIA_256_GCM_SHA384	[RFC6367]
TLS_ECDH_RSA_WITH_CAMELLIA_128_GCM_SHA256	[RFC6367]
TLS_ECDH_RSA_WITH_CAMELLIA_256_GCM_SHA384	[RFC6367]

Table 2

3. Ephemeral Diffie Hellman

Clients and servers MUST NOT reuse ephemeral DHE or ECDHE public keys across TLS connections for all existing (and future) TLS versions. Doing so invalidates forward secret properties of these connections. In the case of DHE (finite field DH) cipher suites, such reuse may also lead to vulnerabilities such as those use in the [Raccoon] attack. See Section 5 for related discussion.

4. IANA Considerations

This document makes no requests to IANA. All cipher suites listed in Section 2 are already marked as not recommended in the "TLS Cipher Suites" registry.

5. Security Considerations

Non-ephemeral finite field DH cipher suites (TLS_DH_*), as well as ephemeral key reuse for finite field DH cipher suites, are prohibited due to the [Raccoon] attack. Both are already considered bad practice since they do not provide forward secrecy. However, Raccoon revealed that timing side channels in processing TLS premaster secrets may be exploited to reveal the encrypted premaster secret.

Raccoon does not apply to non-ephemeral elliptic curve DH suites, since the same timing side channel does not exist. However, such re-use is still discouraged, and thus deprecated in this document.

6. Acknowledgments

This document was inspired by discussion on the TLS WG mailing list and a suggestion by Filippo Valsorda following release of the [Raccoon] attack.

7. References

7.1. Normative References

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

[RFC4162] Lee, H.J., Yoon, J.H., and J.I. Lee, "Addition of SEED Cipher Suites to Transport Layer Security (TLS)", RFC 4162, DOI 10.17487/RFC4162, August 2005, <https://www.rfc-editor.org/info/rfc4162>.

[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>.

[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>.

[RFC5288] Salowey, J., Choudhury, A., and D. McGrew, "AES Galois Counter Mode (GCM) Cipher Suites for TLS", RFC 5288, DOI 10.17487/RFC5288, August 2008, <https://www.rfc-editor.org/info/rfc5288>.

[RFC5289] Rescorla, E., "TLS Elliptic Curve Cipher Suites with SHA-256/384 and AES Galois Counter Mode (GCM)", RFC 5289, DOI 10.17487/RFC5289, August 2008, <https://www.rfc-editor.org/info/rfc5289>.

[RFC5469] Eronen, P., Ed., "DES and IDEA Cipher Suites for Transport Layer Security (TLS)", RFC 5469, DOI 10.17487/RFC5469, February 2009, <https://www.rfc-editor.org/info/rfc5469>.

[RFC5932] Kato, A., Kanda, M., and S. Kanno, "Camellia Cipher Suites for TLS", RFC 5932, DOI 10.17487/RFC5932, June 2010, <https://www.rfc-editor.org/info/rfc5932>.

[RFC6209] Kim, W., Lee, J., Park, J., and D. Kwon, "Addition of the ARIA Cipher Suites to Transport Layer Security (TLS)", RFC 6209, DOI 10.17487/RFC6209, April 2011, <https://www.rfc-editor.org/info/rfc6209>.

[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>.

[RFC6367] Kanno, S. and M. Kanda, "Addition of the Camellia Cipher Suites to Transport Layer Security (TLS)", RFC 6367, DOI 10.17487/RFC6367, September 2011, <https://www.rfc-editor.org/info/rfc6367>.

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

[RFC8422] Nir, Y., Josefsson, S., and M. Pegourie-Gonnard, "Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer

Security (TLS) Versions 1.2 and Earlier", RFC 8422, DOI 10.17487/RFC8422, August 2018, <https://www.rfc-editor.org/info/rfc8422>.

7.2. Informative References

[Raccoon] Merget, R., Brinkmann, M., Aviram, N., Somorovsky, J., Mittmann, J., and J. Schwenk, "Raccoon Attack: Finding and Exploiting Most-Significant-Bit-Oracles in TLS-DH(E)", 1 August 2012, <https://raccoon-attack.com/RacoonAttack.pdf>.

[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>.

Authors' Addresses

Carrick Bartle
Apple, Inc.

Email: cbartle@apple.com

Nimrod Aviram

Email: nimrod.aviram@gmail.com

Filippo Valsorda

Email: ietf@filippo.io

Network Working Group   O. Friel
Internet-Draft   Cisco
Intended status: Standards Track   D. Harkins
Expires: January 14, 2021   Hewlett-Packard Enterprise
                                                          July 13, 2020

Bootstrapped TLS Authentication
draft-friel-tls-eap-dpp-01

Abstract

This document defines a TLS extension that enables a server to prove to a client that it has knowledge of the public key of a key pair where the client has knowledge of the private key of the key pair. Unlike standard TLS key exchanges, the public key is never exchanged in TLS protocol messages. Proof of knowledge of the public key is used by the client to bootstrap trust in the server. The use case outlined in this document is to establish trust in an EAP server.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

On-boarding of devices with no, or limited, user interface can be difficult. Typically, a credential is needed to access the network and network connectivity is needed to obtain a credential. This poses a catch-22.

If trust in the integrity of a device's public key can be obtained in an out-of-band fashion, a device can be authenticated and provisioned with a usable credential for network access. While this authentication can be strong, the device's authentication of the network is somewhat weaker. [Stajano]] presents a functional security model to address this asymmetry.

There are on-boarding protocols, such as [DPP], to address this use case but they have drawbacks. [DPP] for instance does not support wired network access. This document describes an on-boarding protocol, which we refer to as TLS Proof of Knowledge or TLS-POK.

1.1. Bootstrap Key Pair

The mechanism for on-boarding of devices defined in this document relies on bootstrap key pairs. A client device has an associated elliptic curve (EC) key pair. The key pair may be static and baked into device firmware at manufacturing time, or may be dynamic and generated at on-boarding time by the device. If this public key, specifically the ASN.1 SEQUENCE SubjectPublicKeyInfo from [RFC5280], can be shared in a trustworthy manner with a TLS server, a form of "origin entity authentication" (the step from which all subsequent authentication proceeds) can be obtained.

The exact mechanism by which the server gains knowledge of the public key is out of scope of this specification, but possible mechanisms include scanning a QR code to obtain a base64 encoding of the ASN.1-formatted public key or upload of a Bill of Materials (BOM). If the QR code is physically attached to the client device, or the BOM is associated with the device, the assumption is that the public key obtained in this bootstrapping method belongs to the client. In this model, physical possession of the device implies legitimate ownership.

The server may have knowledge of multiple bootstrap public keys corresponding to multiple devices, and TLS extensions are defined in this document that enable the server to identity a specific bootstrap public key correspinding to a specific device.

Using the extensions defined herein, the client proves to the server that it has possession of the private analog to its public bootstrapping key. Provided that the mechanism in which the server obtained the bootstrapping key is trustworthy, a commensurate amount of authenticity of the resulting connection can be obtained. The server also proves that it knows the client's public key which, if the client does not gratuitously expose its public key, can be used to obtain a modicum of correctness, that the client is connecting to the correct network (see [Stajano]).

1.2. Alignment with Wi-Fi Alliance Device Provisioning Profile

The definition of the boostrap public key aligns with that given in [DPP]. This, for example, enables the QR code format as defined in [DPP] to be reused for TLS-POK. Therefore, a device that supports both wired LAN and Wi-Fi LAN connections can have a single QR code printed on its label, and the bootstrap key can be used for DPP if the device bootstraps against a Wi-Fi network, or TLS-POK if the device bootstraps against a wired network. Similarly, a common bootstrap public key format could be imported in a BOM into a server that handles devices connecting over both wired and Wi-Fi networks.

Any bootstrapping method defined for, or used by, [DPP] is compatible with TLS-POK.

2. Bootstrapping in TLS 1.3

The bootstrapping modifications introduce an extension to identify a "bootstrapping" key into the TLS 1.3 handshake. This key MUST be from a cryptosystem suitable for doing (EC)DH. When using the bootstrapping extension, two (EC)DH operations are performed, a static-ephemeral (using the client's bootstrapping key and an ephemeral key generated by the server) and an ephemeral-ephemeral (using the client's ephemeral key and an ephemeral key generated by the server).

2.1. Server Ephemeral Key Options

[[TODO: Need to decide which mechanism to use here]]

There are two options for generation of the server's ephemeral key.

- o server generates one ephemeral key pair: the key pair is reused by the server for both static-ephemeral (bootstrap) and ephemeral-ephemeral (TLS key_share) ECDH exchanges. This means that the bootstrap and key_share keys MUST be on the same curve.

- o server generates two ephemeral key pairs: one that is used for the static-ephemeral (bootstrap) ECDH exchange, and one that is used for the ephemeral-ephemeral (TLS key_share) DH exchange. The bootstrap and key_share keys are independent. The key_share keys may be Finite Field DH keys.

This document assumes the latter and that the server generates unique ephemeral key pairs for bootstrap and key_share.

2.2. Bootstrap Key Extension

```
struct {
    select (Handshake.msg_type) {
        case client_hello:
            opaque bskey[32];

        case server_hello:
            opaque key_exchange<1..2^16-1>;
    };
} BootstrapKey;
```

The BootstrapKey extension is used by the client in its ClientHello message to specify its bootstrapping key identifier. The 'bskey' field of this extension SHALL consist of the base64 encoded SHA256 digest of the DER-encoded ASN.1 subjectPublicKeyInfo representation of the bootstrapping public key.

The BoostrapKey extension is used by the server in its ServerHello message to specify its ephemeral ECDH keying information. The 'key_exchange' field contains the key exchange information on the curve that the bootstrapping key is on.

2.3. Changes to TLS 1.3 Handshake

The client identifies the bootstrapping key in the ClientHello using the BootstrapKey extension 'bskey' field. The server looks up the client's bootstrapping key in its database by checking the SHA256 hash of each entry with the value received in the ClientHello. If no match is found, the server MAY proceed with the TLS handshake without using the bootstrapping key as input to the key schedule, or MAY terminate the TLS handshake with an alert.

[[TODO: should we define an explicit unknown_bsk_identity alert, similar to unknown_psk_identity]]

If the server found the matching bootstrap key, the server generates an ephemeral ECDH keypair on the curve indicated in the bootstrap public key information, and performs an ECDH operation using the client bootstrap key and the server's ephemeral keypair. The server includes a BootstrapKey extension in its ServerHello that includes its ephemeral ECDH public key in the 'key_exchange' field. This explicitly confirms to the client that the server has performed an ECDH operation using the bootstrap key, and has injected the output into the key schedule.

This is in addition to, and independent from, the (EC)DH that the server carries out when handling the key_share extension.

The handshake is shown in Figure 1.

```
Client                                                Server
--------                                            --------
ClientHello
+ bskey
+ key_share                  -------->
                                                 ServerHello
                                                     + bskey
                                                 + key_share
                                       {EncryptedExtensions}
                                                  {Finished}
                             <--------   [Application Data*]
{Finished}                   -------->
[Application Data]           <------->    [Application Data]
```

Figure 1: TLS 1.3 TLS-POK Handshake

2.4. Changes to TLS 1.3 Key Schedule

[[TODO: The key schedule mechanism needs to closed.]]

Multiple options for modifying the TLS 1.3 key schedule have been proporsed recently including [I-D.stebila-tls-hybrid-design] and [I-D.jhoyla-tls-extended-key-schedule]. The key schedule used for TLS-POK will align with the final direction chosen by the TLS WG.

This document proposes aligning with the model outlined in [I-D.jhoyla-tls-extended-key-schedule] where the shared secrets derived from the bskey and key_share key exchanges are injected in sequence into the key schedule.

The key schedule for TLS-POK is as follows:

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

3. Using TLS Bootstrapping in EAP

Enterprise deployments typically require an 802.1X/EAP-based authentication to obtain network access. Protocols like [RFC7030] can be used to enroll devices into a Certification Authority to allow them to authenticate using 802.1X/EAP. But this creates a Catch-22 where a certificate is needed for network access and network access is needed to obtain certificate.

Devices whose bootstrapping key can been obtained in an out-of-band fashion can perform an EAP-TLS-based exchange, for instance [RFC7170], and authenticate the TLS exchange using the bootstrapping extensions defined in Section 2. This network connectivity can then be used to perform an enrollment protocol (such as provided by [RFC7170]) to obtain a credential for subsequent network connectivity and certificate lifecycle maintenance.

Upon "link up", an Authenticator on an 802.1X-protected port will issue an EAP Identify request to the newly connected peer. For unprovisioned devices that desire to take advantage of TLS-POK, there is no initial realm in which to construct an NAI (see [RFC4282]) so the initial EAP Identity response SHOULD contain simply the name "TLS-POK" in order to indicate to the Authenticator that an EAP method that supports TLS-POK SHOULD be started.

```
Authenticating Peer     Authenticator
-------------------     -------------
                         <- EAP-Request/
                         Identity

 EAP-Response/
 Identity (TLS-POK) ->

                         <- EAP-Request/
                         EAP-Type=TEAP
                         (TLS Start)
                    .
                    .
                    .
```

4. Summary of Work

[TODO: agree with WG chairs where this work lives and where it should be documented.]

The protocol outlined here can be broadly broken up into 3 distinct areas:

- o TLS extensions to transport the bootstrap public key identifier
- o TLS key schedule enhancements to inject bootstrap public key keying material
- o TEAP extensions to leverage the new TLS-POK handshake for trust establishment

This document captures all 3 areas, but it may be more appropriate to split the work into multiple documents e.g.:

- o piggy back on top of [I-D.jhoyla-tls-extended-key-schedule] for TLS key schedule enhancements o include the TEAP extensions in [I-D.lear-eap-teap-brski] draft

5. IANA Considerations

IANA will allocated an ExtensionType for the bskey extension from the appropriate TLS 1.3 repository and replace TBD in this document with that number.

6. Security Considerations

Bootstrap and trust establishment by the TLS server is based on proof of knowledge of the client's bootstrap public key. An attack on the bootstrapping method which substitutes the public key of a corrupted device for the public key of an honest device can result in the TLS sever on-boarding and trusting the corrupted device.

Trust on the part of the client is not strong and is based on an assumption that the public bootstrapping key is not widely disseminated. If an adversary has knowledge of the bootstrap public key, the adversary may be able to make the client bootstrap against the adversary's network. For example, if an adversary intercepts and scans QR labels on clients, and the adversary can force the client to connect to its server, then the adversary can complete the TLS-POK handshake with the client and the client will connect to the adversary's server. Since physical possession implies ownership, there is nothing to prevent a stolen device from being on-boarded.

7. Informative References

[DPP] Wi-Fi Alliance, "Device Provisioning Profile", 2020.

[I-D.jhoyla-tls-extended-key-schedule]
Hoyland, J. and C. Wood, "TLS 1.3 Extended Key Schedule", draft-jhoyla-tls-extended-key-schedule-01 (work in progress), March 2020.

[I-D.lear-eap-teap-brski]
Lear, E., Friel, O., Cam-Winget, N., and D. Harkins, "TEAP Update and Extensions for Bootstrapping", draft-lear-eap-teap-brski-05 (work in progress), November 2019.

[I-D.stebila-tls-hybrid-design]
Steblia, D., Fluhrer, S., and S. Gueron, "Hybrid key exchange in TLS 1.3", draft-stebila-tls-hybrid-design-03 (work in progress), February 2020.

[RFC4282] Aboba, B., Beadles, M., Arkko, J., and P. Eronen, "The Network Access Identifier", RFC 4282, DOI 10.17487/RFC4282, December 2005, <https://www.rfc-editor.org/info/rfc4282>.

[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>.

[RFC7030] Pritikin, M., Ed., Yee, P., Ed., and D. Harkins, Ed., "Enrollment over Secure Transport", RFC 7030, DOI 10.17487/RFC7030, October 2013, <https://www.rfc-editor.org/info/rfc7030>.

[RFC7170] Zhou, H., Cam-Winget, N., Salowey, J., and S. Hanna, "Tunnel Extensible Authentication Protocol (TEAP) Version 1", RFC 7170, DOI 10.17487/RFC7170, May 2014, <https://www.rfc-editor.org/info/rfc7170>.

[Stajano] Anderson, S., "The Ressurecting Ducking: Security Issues for Ad-Hoc Wireless Networks", 1999.

Authors' Addresses

Owen Friel
Cisco

Email: ofriel@cisco.com

Dan Harkins
Hewlett-Packard Enterprise

Email: daniel.harkins@hpe.com

Network Working Group O. Friel
Internet-Draft Cisco
Intended status: Standards Track D. Harkins
Expires: August 26, 2021 Hewlett-Packard Enterprise
February 22, 2021

Bootstrapped TLS Authentication
draft-friel-tls-eap-dpp-02

Abstract

This document defines a TLS extension that enables a server to prove to a client that it has knowledge of the public key of a key pair where the client has knowledge of the private key of the key pair. Unlike standard TLS key exchanges, the public key is never exchanged in TLS protocol messages. Proof of knowledge of the public key is used by the client to bootstrap trust in the server. The use case outlined in this document is to establish trust in an EAP server.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

On-boarding of devices with no, or limited, user interface can be difficult. Typically, a credential is needed to access the network and network connectivity is needed to obtain a credential. This poses a catch-22.

If trust in the integrity of a device's public key can be obtained in an out-of-band fashion, a device can be authenticated and provisioned with a usable credential for network access. While this authentication can be strong, the device's authentication of the network is somewhat weaker. [Stajano]] presents a functional security model to address this asymmetry.

There are on-boarding protocols, such as [DPP], to address this use case but they have drawbacks. [DPP] for instance does not support wired network access. This document describes an on-boarding protocol, which we refer to as TLS Proof of Knowledge or TLS-POK.

1.1. Bootstrap Key Pair

The mechanism for on-boarding of devices defined in this document relies on bootstrap key pairs. A client device has an associated elliptic curve (EC) key pair. The key pair may be static and baked into device firmware at manufacturing time, or may be dynamic and generated at on-boarding time by the device. If this public key, specifically the ASN.1 SEQUENCE SubjectPublicKeyInfo from [RFC5280], can be shared in a trustworthy manner with a TLS server, a form of "origin entity authentication" (the step from which all subsequent authentication proceeds) can be obtained.

The exact mechanism by which the server gains knowledge of the public key is out of scope of this specification, but possible mechanisms include scanning a QR code to obtain a base64 encoding of the ASN.1-formatted public key or upload of a Bill of Materials (BOM). If the QR code is physically attached to the client device, or the BOM is associated with the device, the assumption is that the public key obtained in this bootstrapping method belongs to the client. In this model, physical possession of the device implies legitimate ownership.

The server may have knowledge of multiple bootstrap public keys corresponding to multiple devices, and TLS extensions are defined in this document that enable the server to identity a specific bootstrap public key correspinding to a specific device.

Using the extensions defined herein, the client proves to the server that it has possession of the private analog to its public bootstrapping key. Provided that the mechanism in which the server obtained the bootstrapping key is trustworthy, a commensurate amount of authenticity of the resulting connection can be obtained. The server also proves that it knows the client's public key which, if the client does not gratuitously expose its public key, can be used to obtain a modicum of correctness, that the client is connecting to the correct network (see [Stajano]).

1.2. Alignment with Wi-Fi Alliance Device Provisioning Profile

The definition of the boostrap public key aligns with that given in [DPP]. This, for example, enables the QR code format as defined in [DPP] to be reused for TLS-POK. Therefore, a device that supports both wired LAN and Wi-Fi LAN connections can have a single QR code printed on its label, and the bootstrap key can be used for DPP if the device bootstraps against a Wi-Fi network, or TLS-POK if the device bootstraps against a wired network. Similarly, a common bootstrap public key format could be imported in a BOM into a server that handles devices connecting over both wired and Wi-Fi networks.

Any bootstrapping method defined for, or used by, [DPP] is compatible with TLS-POK.

2. Bootstrapping in TLS 1.3

The bootstrapping modifications introduce an extension to identify a "bootstrapping" key into the TLS 1.3 handshake. This key MUST be from a cryptosystem suitable for doing (EC)DH. When using the bootstrapping extension, two (EC)DH operations are performed, a static-ephemeral (using the client's bootstrapping key and an ephemeral key generated by the server) and an ephemeral-ephemeral (using the client's ephemeral key and an ephemeral key generated by the server).

2.1. Bootstrap Key Extension

This document defines the "bskey_share" extension.

```
struct {
    select (Handshake.msg_type) {
        case client_hello:
            opaque bskey[32];

        case server_hello:
            opaque bskey_exchange<1..2^16-1>;
    };
} BootstrapKey;
```

The BootstrapKey extension is used by the client in its ClientHello message to specify its bootstrapping key identifier. The 'bskey' field of this extension SHALL consist of the base64 encoded SHA256 digest of the DER-encoded ASN.1 subjectPublicKeyInfo representation of the bootstrapping public key.

The BoostrapKey extension is used by the server in its ServerHello message to specify its ephemeral ECDH keying information. The 'key_exchange' field contains the key exchange information on the curve that the bootstrapping key is on.

2.2. Server Ephemeral Keys

The server generates two ephemeral key pairs: one that is used for the static-ephemeral (bootstrap) ECDH exchange and is exchanged in the "bskey_share" extension, and one that is used for the ephemeral-ephemeral (TLS key_share) DH exchange and is exchanged in the "key_share" extension. The "bskey_share" and "key_share" keys are independent. The key_share keys may be Finite Field DH keys.

2.3. Changes to TLS 1.3 Handshake

The client identifies the bootstrapping key in the ClientHello using the BootstrapKey extension 'bskey' field. The server looks up the client's bootstrapping key in its database by checking the SHA256 hash of each entry with the value received in the ClientHello. If no match is found, the server MAY proceed with the TLS handshake without using the bootstrapping key as input to the key schedule, or MAY terminate the TLS handshake with an alert.

[[TODO: should we define an explicit unknown_bsk_identity alert, similar to unknown_psk_identity]]

If the server found the matching bootstrap key, the server generates an ephemeral ECDH keypair on the curve indicated in the bootstrap public key information, and performs an ECDH operation using the client bootstrap key and the server's ephemeral keypair. The server includes a BootstrapKey extension in its ServerHello that includes its ephemeral ECDH public key in the 'key_exchange' field. This explicitly confirms to the client that the server has performed an ECDH operation using the bootstrap key, and has injected the output into the key schedule.

This is in addition to, and independent from, the (EC)DH that the server carries out when handling the key_share extension.

The handshake is shown in Figure 1.

```
Client                                              Server
--------                                          --------
ClientHello
+ bskey_share
+ key_share               -------->
                                               ServerHello
                                             + bskey_share
                                               + key_share
                                     {EncryptedExtensions}
                                                {Finished}
                          <--------    [Application Data*]
{Finished}                -------->
[Application Data]        <------->     [Application Data]
```

Figure 1: TLS 1.3 TLS-POK Handshake

2.4. Changes to TLS 1.3 Key Schedule

[[TODO: The key schedule mechanism needs to closed.]]

Multiple options for modifying the TLS 1.3 key schedule have been proporsed recently including [I-D.stebila-tls-hybrid-design] and [I-D.jhoyla-tls-extended-key-schedule]. The key schedule used for TLS-POK will align with the final direction chosen by the TLS WG.

This document proposes aligning with the model outlined in [I-D.jhoyla-tls-extended-key-schedule] where the shared secrets derived from the bskey and key_share key exchanges are injected together into the key schedule to create the Handshake Secret.

The key schedule for TLS-POK is as follows:

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

3. Using TLS Bootstrapping in EAP

Enterprise deployments typically require an 802.1X/EAP-based authentication to obtain network access. Protocols like [RFC7030] can be used to enroll devices into a Certification Authority to allow them to authenticate using 802.1X/EAP. But this creates a Catch-22 where a certificate is needed for network access and network access is needed to obtain certificate.

Devices whose bootstrapping key can been obtained in an out-of-band fashion can perform an EAP-TLS-based exchange, for instance [RFC7170], and authenticate the TLS exchange using the bootstrapping extensions defined in Section 2. This network connectivity can then be used to perform an enrollment protocol (such as provided by [RFC7170]) to obtain a credential for subsequent network connectivity and certificate lifecycle maintenance.

Upon "link up", an Authenticator on an 802.1X-protected port will issue an EAP Identify request to the newly connected peer. For unprovisioned devices that desire to take advantage of TLS-POK, there is no initial realm in which to construct an NAI (see [RFC4282]) so the initial EAP Identity response SHOULD contain simply the name "TLS-POK" in order to indicate to the Authenticator that an EAP method that supports TLS-POK SHOULD be started.

```
           Authenticating Peer     Authenticator
           -------------------     -------------
                                    <- EAP-Request/
                                    Identity

            EAP-Response/
            Identity (TLS-POK) ->

                                    <- EAP-Request/
                                    EAP-Type=TEAP
                                    (TLS Start)
                               .
                               .
                               .
```

4. Summary of Work

[TODO: agree with WG chairs where this work lives and where it should be documented.]

The protocol outlined here can be broadly broken up into 3 distinct areas:

- o TLS extensions to transport the bootstrap public key identifier

- o TLS key schedule enhancements to inject bootstrap public key keying material

- o TEAP extensions to leverage the new TLS-POK handshake for trust establishment

This document captures all 3 areas, but it may be more appropriate to split the work into multiple documents e.g.:

- o piggy back on top of [I-D.jhoyla-tls-extended-key-schedule] for TLS key schedule enhancements

- o include the TEAP extensions in [I-D.lear-eap-teap-brski] draft

5. IANA Considerations

IANA will allocated an ExtensionType for the bskey extension from the appropriate TLS 1.3 repository and replace TBD in this document with that number.

6. Security Considerations

Bootstrap and trust establishment by the TLS server is based on proof of knowledge of the client's bootstrap public key. An attack on the bootstrapping method which substitutes the public key of a corrupted device for the public key of an honest device can result in the TLS sever on-boarding and trusting the corrupted device.

Trust on the part of the client is not strong and is based on an assumption that the public bootstrapping key is not widely disseminated. If an adversary has knowledge of the bootstrap public key, the adversary may be able to make the client bootstrap against the adversary's network. For example, if an adversary intercepts and scans QR labels on clients, and the adversary can force the client to connect to its server, then the adversary can complete the TLS-POK handshake with the client and the client will connect to the adversary's server. Since physical possession implies ownership, there is nothing to prevent a stolen device from being on-boarded.

7. Informative References

[DPP] Wi-Fi Alliance, "Device Provisioning Profile", 2020.

[I-D.jhoyla-tls-extended-key-schedule]
Hoyland, J. and C. Wood, "TLS 1.3 Extended Key Schedule", draft-jhoyla-tls-extended-key-schedule-03 (work in progress), December 2020.

[I-D.lear-eap-teap-brski]
Lear, E., Friel, O., Cam-Winget, N., and D. Harkins, "TEAP Update and Extensions for Bootstrapping", draft-lear-eap-teap-brski-05 (work in progress), November 2019.

[I-D.stebila-tls-hybrid-design]
Steblia, D., Fluhrer, S., and S. Gueron, "Hybrid key exchange in TLS 1.3", draft-stebila-tls-hybrid-design-03 (work in progress), February 2020.

[RFC4282] Aboba, B., Beadles, M., Arkko, J., and P. Eronen, "The Network Access Identifier", RFC 4282, DOI 10.17487/RFC4282, December 2005, <https://www.rfc-editor.org/info/rfc4282>.

[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>.

[RFC7030] Pritikin, M., Ed., Yee, P., Ed., and D. Harkins, Ed., "Enrollment over Secure Transport", RFC 7030, DOI 10.17487/RFC7030, October 2013, <https://www.rfc-editor.org/info/rfc7030>.

[RFC7170] Zhou, H., Cam-Winget, N., Salowey, J., and S. Hanna, "Tunnel Extensible Authentication Protocol (TEAP) Version 1", RFC 7170, DOI 10.17487/RFC7170, May 2014, <https://www.rfc-editor.org/info/rfc7170>.

[Stajano] Anderson, S., "The Ressurecting Ducking: Security Issues for Ad-Hoc Wireless Networks", 1999.

Authors' Addresses

Owen Friel
Cisco

Email: ofriel@cisco.com

Dan Harkins
Hewlett-Packard Enterprise

Email: daniel.harkins@hpe.com

TLS Working Group E. Rescorla
Internet-Draft Mozilla
Intended status: Informational R. Barnes
Expires: 6 November 2021 Cisco
H. Tschofenig
Arm Limited
5 May 2021

Compact TLS 1.3
draft-ietf-tls-ctls-02

Abstract

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

Table of Contents

1. Introduction

DISCLAIMER: This is a work-in-progress draft of cTLS and has not yet seen significant security analysis, so could contain major errors. It should not be used as a basis for building production systems.

This document specifies a "compact" version of TLS 1.3 [RFC8446]. It is isomorphic to TLS 1.3 but designed to take up minimal bandwidth. The space reduction is achieved by four basic techniques:

* Omitting unnecessary values that are a holdover from previous versions of TLS.

* Omitting the fields and handshake messages required for preserving backwards-compatibility with earlier TLS versions.

* More compact encodings, omitting unnecessary values.

* A template-based specialization mechanism that allows for the creation of application specific versions of TLS that omit unnecessary values.

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

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

2. Conventions and Definitions

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

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

3. Common Primitives

3.1. Varints

cTLS makes use of variable-length integers in order to allow a wide integer range while still providing for a minimal encoding. The width of the integer is encoded in the first two bits of the field as follows, with xs indicating bits that form part of the integer.

Bit pattern	Length (bytes)
0xxxxxxx	1
10xxxxxx xxxxxxxx	2
11xxxxxx xxxxxxxx xxxxxxxx	3

Table 1

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

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

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

cTLS uses the varint encoding for all multi-byte integers in TLS, including:

* Values of type uint16, uint24, uint32, uint64

* Array and vector entries of these types

* Encoded lengths for vectors that allow more than 255 entries

* Enums that allow more than 255 entries

Values of type uint8, opaque values, and one-byte enums are not affected. We do not show the structures which only change in this way.

3.2. Record Layer

The only cTLS records that are sent in plaintext are handshake records (ClientHello and ServerHello/HRR). The content type is therefore constant (it is always "handshake"), so we instead set the "content_type" field to a fixed cTLS-specific value to distinguish cTLS plaintext records from encrypted records, TLS/DTLS records, and other protocols using the same 5-tuple.

The "profile_id" field allows the client and server to agree on which compression profile should be used for this session (see Section 5). This field MUST be set to zero if and only if no compression profile is used. Non-zero values are negotiated out of band between the client and server, as part of the specification of the compression profile.

```
       struct {
           ContentType content_type = ctls_handshake;
           varint profile_id;
           opaque fragment<0..V>;
       } CTLSPlaintext;
```

[[OPEN ISSUE: The profile_id is needed in the ClientHello to inform the server what compression profile to use. For a ServerHello this field is not required. Should we make this field optional?]]

Encrypted records use DTLS 1.3 record framing, comprising a configuration octet followed by optional connection ID, sequence number, and length fields.

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

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

The presence and size of the connection ID field is negotiated as in DTLS.

As with DTLS, the length field MAY be omitted by clearing the L bit, which means that the record consumes the entire rest of the data in the lower level transport. In this case it is not possible to have multiple DTLSCiphertext format records without length fields in the same datagram. In stream-oriented transports (e.g., TCP), the length field MUST be present. For use over other transports length information may be inferred from the underlying layer.

Normal DTLS does not provide a mechanism for suppressing the sequence number field entirely. In cases where a sequence number is not required (e.g., when a reliable transport is in use), a cTLS implementation may suppress it by setting the "suppressSequenceNumber" flag in the compression profile being used (see Section 5.1). When this flag is enabled, the S bit in the configuration octet MUST be cleared.

3.3. Handshake Layer

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

1. The length field is omitted

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

```
struct {
    HandshakeType msg_type;    /* handshake type */
    select (Handshake.msg_type) {
        case client_hello:          ClientHello;
        case server_hello:          ServerHello;
        case hello_retry_request:   HelloRetryRequest;
        case end_of_early_data:     EndOfEarlyData;
        case encrypted_extensions:  EncryptedExtensions;
        case certificate_request:   CertificateRequest;
        case certificate:           Certificate;
        case certificate_verify:    CertificateVerify;
        case finished:              Finished;
        case new_session_ticket:    NewSessionTicket;
        case key_update:            KeyUpdate;
    };
} Handshake;
```

4. Handshake Messages

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

4.1. ClientHello

The cTLS ClientHello is as follows.

```
opaque Random[RandomLength];      // variable length

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

4.2. ServerHello

We redefine ServerHello in a similar way:

```
struct {
    Random random;
    CipherSuite cipher_suite;
    Extension extensions<1..V>;
} ServerHello;
```

4.3. HelloRetryRequest

The HelloRetryRequest has the following format:

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

It is the same as the ServerHello above but without the unnecessary sentinel Random value.

5. Template-Based Specialization

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

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

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

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

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

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

5.1. Specifying a Specialization

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

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

```
{
   "profileID": 33,
   "version" : 772,
   "cipherSuite" : "TLS_AES_128_GCM_SHA256"
}
```

cTLS allows specialization along the following axes:

profileID (integer): The identifier for this profile, to be sent in the "profile_id" field of CTLSPlaintext records. This value MUST NOT be zero. If this value is not present, the default "profile_id" is 1.

suppressSequenceNumber (boolean): If present and set to true, the sequence number field is omitted from encrypted record headers.

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

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

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

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

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

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

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

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

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

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

finishedSize (integer): indicates that the Finished value is to be truncated to the given length. When the transcript is reconstructed, the remainder of the Finished value is filled in by the receiving side. [[OPEN ISSUE: How short should we allow this to be? TLS 1.3 uses the native hash and TLS 1.2 used 12 bytes. More analysis is needed to know the minimum safe Finished size. See [RFC8446]; Section E.1 for more on this, as well as https://mailarchive.ietf.org/arch/msg/tls/TugB5ddJu3nYg7chcyeIyUqWSbA.]]

5.1.1. Requirements on the TLS Implementation

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

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

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

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

5.1.2. Predefined Extensions

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

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

* If the extensions list in the message is not sorted in ascending order by extension type, it is an error, because the decompressed message will not match.

* If there is no entry in the predefined extensions object for the type of the extension, then the extension is included in the compressed message

* If there is an entry:

  - If the ExtensionData of the extension does not match the value in the dictionary, it is an error, because decompression will not produce the correct result.

  - If the ExtensionData matches, then the extension is removed, and not included in the compressed message.

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

* If there is an extension in the compressed message with a type that exists in the predefined extensions object, it is an error, because such an extension would not have been sent by a sender with a compatible compression profile.

* For each entry in the predefined extensions dictionary, an extension is added to the decompressed message with the specified type and value.

* The resulting vector of extensions MUST be sorted in ascending order by extension type.

Note that the "version", "dhGroup", and "signatureAlgorithm" fields in the compression profile are specific instances of this algorithm for the corresponding extensions.

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

5.1.3. Known Certificates

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

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

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

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

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

6. Examples

The following section provides some example specializations.

TLS 1.3 only:

```
{
   "version" : 0x0304
}
```

TLS 1.3 with AES_GCM and X25519 and ALPN h2, short random values, and everything else is ordinary TLS 1.3.

```
{
   "version" : 772,
   "randomSize": 16,
   "cipherSuite" : "TLS_AES_128_GCM_SHA256",
   "dhGroup": "X25519",
   "clientHelloExtensions": {
      "named_groups": 29,
      "application_layer_protocol_negotiation" : "030016832",
      "..." : null
    }
}
```

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

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

7. Security Considerations

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

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

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

8. IANA Considerations

This document requests that a code point be allocated from the "TLS ContentType registry. This value must be in the range 0-31 (inclusive). The row to be added in the registry has the following form:

Value	Description	DTLS-OK	Reference
TBD	ctls	N	RFCXXXX

Table 2

[[RFC EDITOR: Please replace the value TBD with the value assigned by IANA, and the value XXXX to the RFC number assigned for this document.]]

[[OPEN ISSUE: Should we require standards action for all profile IDs that would fit in 2 octets.]]

9. Normative References

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

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

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

Appendix A. Sample Transcripts

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

	ECDHE			PSK		
	TLS	CTLS	Overhead	TLS	CTLS	Overhead
ClientHello	132	50	10	147	67	15
ServerHello	90	48	8	56	18	2
ServerFlight	478	104	16	42	12	3
ClientFlight	458	100	11	36	10	1
Total	1158	302	45	280	107	21

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

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

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

A.1. ECDHE and Mutual Certificate-based Authentication

Compression Profile:

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


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

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

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

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

Server Flight: 96 = SIG(71) + MAC(8) + CERTID(1) + Overhead(16)

```
   08                  // EncryptedExtensions
     00                //   Extensions.length
   0d                  // CertificateRequest
     00                //   CertificateRequestContext.length
     00                //   Extensions.length
   0b                  // Certificate
     00                //   CertificateRequestContext
     03                //   CertificateList
       01              //     CertData.length
         61            //       CertData = 'a'
       00              //   Extensions.length
   0f                  // CertificateVerify
     0403              //   SignatureAlgorithm
     4047              //   Signature.length
        3045...f60e    //      Signature
   14                  // Finished
     bfc9d66715bb2b04 //   VerifyData
```

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

```
   0b                  // Certificate
     00                //   CertificateRequestContext
     03                //   CertificateList
       01              //     CertData.length
         62            //       CertData = 'b'
       00              //     Extensions.length
   0f                  // CertificateVerify
     0403              //   SignatureAlgorithm
     4047              //   Signature.length
        3045...f60e    //      Signature
   14                  // Finished
     35e9c34eec2c5dc1 //   VerifyData
```

A.2. PSK

Compression Profile:

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


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

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

ServerHello: 18 bytes = RANDOM(16) + 2

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

Server Flight: 3 bytes = Overhead(3)

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

Client Flight: 1 byte = Overhead(3)

```
14    // Finished
```

Acknowledgments

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

Authors' Addresses

Eric Rescorla
Mozilla

Email: ekr@rtfm.com

Richard Barnes
Cisco

Email: rlb@ipv.sx

Hannes Tschofenig
Arm Limited

Email: hannes.tschofenig@arm.com

tls E. Rescorla
Internet-Draft RTFM, Inc.
Intended status: Standards Track K. Oku
Expires: 9 September 2021 Fastly
N. Sullivan
C.A. Wood
Cloudflare
8 March 2021

TLS Encrypted Client Hello
draft-ietf-tls-esni-10

Abstract

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

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

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

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

The target domain may also be visible through other channels, such as plaintext client DNS queries, visible server IP addresses (assuming the server does not use domain-based virtual hosting), or other indirect mechanisms such as traffic analysis. DoH [RFC8484] and DPRIVE [RFC7858] [RFC8094] provide mechanisms for clients to conceal DNS lookups from network inspection, and many TLS servers host multiple domains on the same IP address. In such environments, the SNI remains the primary explicit signal used to determine the server's identity.

The TLS Working Group has studied the problem of protecting the SNI, but has been unable to develop a completely generic solution. [RFC8744] provides a description of the problem space and some of the proposed techniques. One of the more difficult problems is "Do not stick out" ([RFC8744], Section 3.4): if only sensitive or private services use SNI encryption, then SNI encryption is a signal that a client is going to such a service. For this reason, much recent work has focused on concealing the fact that the SNI is being protected. Unfortunately, the result often has undesirable performance consequences, incomplete coverage, or both.

The protocol specified by this document takes a different approach. It assumes that private origins will co-locate with or hide behind a provider (reverse proxy, application server, etc.) that protects sensitive ClientHello parameters, including the SNI, for all of the domains it hosts. These co-located servers form an anonymity set wherein all elements have a consistent configuration, e.g., the set of supported application protocols, ciphersuites, TLS versions, and so on. Usage of this mechanism reveals that a client is connecting to a particular service provider, but does not reveal which server from the anonymity set terminates the connection. Thus, it leaks no more than what is already visible from the server IP address.

This document specifies a new TLS extension, called Encrypted Client Hello (ECH), that allows clients to encrypt their ClientHello to a supporting server. This protects the SNI and other potentially sensitive fields, such as the ALPN list [RFC7301]. This extension is only supported with (D)TLS 1.3 [RFC8446] and newer versions of the protocol.

2. Conventions and Definitions

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

3. Overview

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

3.1. Topologies

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

Figure 1: Shared Mode Topology

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

```
           +--------------------+     +---------------------+
           |                    |     |                     |
           |   2001:DB8::1111   |     |   2001:DB8::EEEE    |
Client <----------------------------->|                     |
           | public.example.com |     | private.example.com |
           |                    |     |                     |
           +--------------------+     +---------------------+
            Client-Facing Server            Backend Server
```

Figure 2: Split Mode Topology

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

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

3.2. Encrypted ClientHello (ECH)

ECH allows the client to encrypt sensitive ClientHello extensions, e.g., SNI, ALPN, etc., under the public key of the client-facing server. This requires the client-facing server to publish the public key and metadata it uses for ECH for all the domains for which it serves directly or indirectly (via Split Mode). This document defines the format of the ECH encryption public key and metadata, referred to as an ECH configuration, and delegates DNS publication details to [HTTPS-RR], though other delivery mechanisms are possible. In particular, if some of the clients of a private server are applications rather than Web browsers, those applications might have the public key and metadata preconfigured.

When a client wants to establish a TLS session with the backend server, it constructs its ClientHello as indicated in Section 6.1. We will refer to this as the ClientHelloInner message. The client encrypts this message using the public key of the ECH configuration. It then constructs a new ClientHello, the ClientHelloOuter, with innocuous values for sensitive extensions, e.g., SNI, ALPN, etc., and with an "encrypted_client_hello" extension, which this document defines (Section 5). The extension's payload carries the encrypted ClientHelloInner and specifies the ECH configuration used for encryption. Finally, it sends ClientHelloOuter to the server.

Upon receiving the ClientHelloOuter, a TLS server takes one of the following actions:

1. If it does not support ECH, it ignores the "encrypted_client_hello" extension and proceeds with the handshake as usual, per [RFC8446], Section 4.1.2.

2. If it is a client-facing server for the ECH protocol, but cannot decrypt the extension, then it terminates the handshake using the ClientHelloOuter. This is referred to as "ECH rejection". When ECH is rejected, the client-facing server sends an acceptable ECH configuration in its EncryptedExtensions message.

3. If it supports ECH and decrypts the extension, it forwards the ClientHelloInner to the backend server, who terminates the connection. This is referred to as "ECH acceptance".

Upon receiving the server's response, the client determines whether or not ECH was accepted and proceeds with the handshake accordingly. (See Section 6 for details.)

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

4. Encrypted ClientHello Configuration

ECH uses draft-08 of HPKE for public key encryption [I-D.irtf-cfrg-hpke]. The ECH configuration is defined by the following "ECHConfig" structure.

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

struct {
    HpkeKdfId kdf_id;
    HpkeAeadId aead_id;
} HpkeSymmetricCipherSuite;

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

struct {
    HpkeKeyConfig key_config;
    uint16 maximum_name_length;
    opaque public_name<1..2^16-1>;
    Extension extensions<0..2^16-1>;
} ECHConfigContents;

struct {
    uint16 version;
    uint16 length;
    select (ECHConfig.version) {
      case 0xfe0a: ECHConfigContents contents;
    }
} ECHConfig;
```

The structure contains the following fields:

version The version of ECH for which this configuration is used.

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

length The length, in bytes, of the next field.

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

The "ECHConfigContents" structure contains the following fields:

key_config A "HpkeKeyConfig" structure carrying the configuration information associated with the HPKE public key. Note that this structure contains the "config_id" field, which applies to the entire ECHConfigContents. Sites MUST NOT publish two different "ECHConfigContents" values with the same "HpkeKeyConfig" value. The RECOMMENDED technique for choosing "config_id" is to choose a random byte. This process is repeated if this config_id matches that of any valid ECHConfig, which could include any ECHConfig that has been recently removed from active use.

maximum_name_length The longest name of a backend server, if known. If this value is not known it can be set to zero, in which case clients SHOULD use the inner ClientHello padding scheme described below. That could happen if wildcard names are in use, or if names can be added or removed from the anonymity set during the lifetime of a particular ECH configuration.

public_name The non-empty name of the client-facing server, i.e., the entity trusted to update the ECH configuration. This is used to correct misconfigured clients, as described in Section 6.1.3.

extensions A list of extensions that the client must take into consideration when generating a ClientHello message. These are described below (Section 4.1).

The "HpkeKeyConfig" structure contains the following fields:

config_id A one-byte identifier for the given HPKE key configuration. This is used by clients to indicate the key used for ClientHello encryption.

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

public_key The HPKE public key used by the client to encrypt

ClientHelloInner.

cipher_suites The list of HPKE KDF and AEAD identifier pairs clients can use for encrypting ClientHelloInner.

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

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

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

4.1. Configuration Extensions

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

The format is as defined in [RFC8446], Section 4.2. The same interpretation rules apply: extensions MAY appear in any order, but there MUST NOT be more than one extension of the same type in the extensions block. An extension can be tagged as mandatory by using an extension type codepoint with the high order bit set to 1. A client that receives a mandatory extension they do not understand MUST reject the "ECHConfig" content.

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

5. The "encrypted_client_hello" Extension

The encrypted ClientHelloInner is carried in an "encrypted_client_hello" extension, defined as follows:

```
enum {
   encrypted_client_hello(0xfe0a), (65535)
} ExtensionType;
```

When offered by the client, the extension appears only in the ClientHelloOuter. The payload MUST have the following structure:

```
       struct {
          HpkeSymmetricCipherSuite cipher_suite;
          uint8 config_id;
          opaque enc<1..2^16-1>;
          opaque payload<1..2^16-1>;
       } ClientECH;
```

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

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

enc The HPKE encapsulated key, used by servers to decrypt the corresponding "payload" field.

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

When the client offers the "encrypted_client_hello" extension, the server MAY include an "encrypted_client_hello" extension in its EncryptedExtensions message with the following payload:

```
       struct {
          ECHConfigList retry_configs;
       } ServerECH;
```

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

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

5.1. Encoding the ClientHelloInner

Some TLS 1.3 extensions can be quite large, thus repeating them in the ClientHelloInner and ClientHelloOuter can lead to an excessive overall size. One pathological example is "key_share" with post-quantum algorithms. To reduce the impact of duplicated extensions, the client may use the "ech_outer_extensions" extension.

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

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

OuterExtensions consists of one or more ExtensionType values, each of which reference an extension in ClientHelloOuter.

When sending ClientHello, the client first computes ClientHelloInner, including any PSK binders. It then computes a new value, the EncodedClientHelloInner, by first making a copy of ClientHelloInner. It then replaces the legacy_session_id field with an empty string.

The client then MAY substitute extensions which it knows will be duplicated in ClientHelloOuter. To do so, the client removes and replaces extensions from EncodedClientHelloInner with a single "ech_outer_extensions" extension. Removed extensions MUST be ordered consecutively in ClientHelloInner. The list of outer extensions, OuterExtensions, includes those which were removed from EncodedClientHelloInner, in the order in which they were removed.

Finally, EncodedClientHelloInner is serialized as a ClientHello structure, defined in Section 4.1.2 of [RFC8446]. Note this does not include the four-byte header included in the Handshake structure.

The client-facing server computes ClientHelloInner by reversing this process. First it makes a copy of EncodedClientHelloInner and copies the legacy_session_id field from ClientHelloOuter. It then looks for an "ech_outer_extensions" extension. If found, it replaces the extension with the corresponding sequence of extensions in the ClientHelloOuter. If any referenced extensions are missing or if "encrypted_client_hello" appears in the list, the server MUST abort the connection with an "illegal_parameter" alert.

The "ech_outer_extensions" extension is only used for compressing the ClientHelloInner. It MUST NOT be sent in either ClientHelloOuter or ClientHelloInner.

Note that it is possible to implement decoding of the EncodedClientHelloInner in a way that creates a denial-of-service vulnerability. Specifically, the server needs to check that each extension in the OuterExtensions list appears in the ClientHelloOuter. The naive strategy would require O(N*M) time, where N is the number of extensions in the ClientHelloOuter and M is the number of extensions in the OuterExtensions list. Malicious clients could exploit this behavior in order to cause excessive work for the server, possibly making it unavailable. This problem can be mitigated by representing OuterExtensions in a way that allows it to be searched more quickly. For example, the runtime can be improved to O(N*log(M)) by sorting the OuterExtensions and using binary search to access it.

5.2. Authenticating the ClientHelloOuter

To prevent a network attacker from modifying the reconstructed ClientHelloInner (see Section 10.10.3), ECH authenticates ClientHelloOuter by computing ClientHelloOuterAAD as described below and passing it in as the associated data for HPKE sealing and opening operations. ClientHelloOuterAAD has the following structure:

```
struct {
   HpkeSymmetricCipherSuite cipher_suite;
   uint8 config_id;
   opaque enc<1..2^16-1>;
   opaque outer_hello<1..2^24-1>;
} ClientHelloOuterAAD;
```

The first three parameters are equal to, respectively, the "ClientECH.cipher_suite", "ClientECH.config_id", and "ClientECH.enc" fields of the payload of the "encrypted_client_hello" extension. The last parameter, "outer_hello", is computed by serializing ClientHelloOuter with the "encrypted_client_hello" extension removed. Note this does not include the four-byte header included in the Handshake structure.

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

6. Client Behavior

Clients that implement the ECH extension behave in one of two ways: either they offer a real ECH extension, as described in Section 6.1; or they send a GREASE ECH extension, as described in Section 6.2. Clients of the latter type do not negotiate ECH. Instead, they generate a dummy ECH extension that is ignored by the server. (See Section 10.8.4 for an explanation.) The client offers ECH if it is in possession of a compatible ECH configuration and sends GREASE ECH otherwise.

6.1. Offering ECH

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

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

1. It MUST NOT offer to negotiate TLS 1.2 or below. This is necessary to ensure the backend server does not negotiate a TLS version that is incompatible with ECH.

2. It MUST NOT offer to resume any session for TLS 1.2 and below.

3. It SHOULD contain TLS padding [RFC7685] as described in Section 6.1.2.

4. If it intends to compress any extensions (see Section 5.1), it MUST order those extensions consecutively.

5. It MUST include the "ech_is_inner" extension as defined in Section 6.1.1. (This requirement is not applicable when the "encrypted_client_hello" extension is generated as described in Section 6.2.)

The client then constructs EncodedClientHelloInner as described in Section 5.1. Finally, it constructs the ClientHelloOuter message just as it does a standard ClientHello, with the exception of the following rules:

1. It MUST offer to negotiate TLS 1.3 or above.

2. If it compressed any extensions in EncodedClientHelloInner, it MUST copy the corresponding extensions from ClientHelloInner.

3. It MUST ensure that all extensions or parameters in ClientHelloInner that might change in response to receiving HelloRetryRequest match that in ClientHelloOuter. See Section 6.1.4 for more information.

4. It MUST copy the legacy_session_id field from ClientHelloInner. This allows the server to echo the correct session ID for TLS 1.3's compatibility mode (see Appendix D.4 of [RFC8446]) when ECH is negotiated.

5. It MAY copy any other field from the ClientHelloInner except ClientHelloInner.random. Instead, It MUST generate a fresh ClientHelloOuter.random using a secure random number generator. (See Section 10.10.1.)

6. It MUST include an "encrypted_client_hello" extension with a payload constructed as described below.

7. The value of "ECHConfig.contents.public_name" MUST be placed in the "server_name" extension.

8. It MUST NOT include the "pre_shared_key" extension. (See Section 10.10.3.)

[[OPEN ISSUE: We currently require HRR-sensitive parameters to match in ClientHelloInner and ClientHelloOuter in order to simplify client-side logic in the event of HRR. See https://github.com/tlswg/draft-ietf-tls-esni/pull/316 for more information. We might also solve this by including an explicit signal in HRR noting ECH acceptance. We need to decide if inner/outer variance is important for HRR-sensitive parameters, and if so, how to best deal with it without complicated client logic.]]

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

To encrypt EncodedClientHelloInner, the client first computes ClientHelloOuterAAD as described in Section 5.2. Note this requires the "encrypted_client_hello" be computed after all other extensions. In particular, this is possible because the "pre_shared_key" extension is forbidden in ClientHelloOuter.

The client then generates the HPKE encryption context and computes the encapsulated key, context, and payload as:

```
pkR = Deserialize(ECHConfig.contents.public_key)
enc, context = SetupBaseS(pkR,
                          "tls ech" || 0x00 || ECHConfig)
payload = context.Seal(ClientHelloOuterAAD,
                       EncodedClientHelloInner)
```

Note that the HPKE functions Deserialize and SetupBaseS are those which match "ECHConfig.contents.kem_id" and the AEAD/KDF used with "context" are those which match the client’s chosen preference from "ECHConfig.contents.cipher_suites". The "info" parameter to SetupBaseS is the concatenation of "tls ech", a zero byte, and the serialized ECHConfig.

The value of the "encrypted_client_hello" extension in the ClientHelloOuter is a "ClientECH" with the following values:

* "config_id", the identifier corresponding to the chosen ECHConfig structure;

* "cipher_suite", the client’s chosen cipher suite;

* "enc", as computed above; and

* "payload", as computed above.

If optional configuration identifiers (see Section 10.4)) are used, "config_id" SHOULD be set to a randomly generated byte. Unless specified by the application using (D)TLS or externally configured on both sides, implementations MUST set the field as specified in Section 5.

6.1.1. ClientHelloInner Indication Extension

If, in a ClientHello, the "encrypted_client_hello" extension is not present and an "ech_is_inner" extension is present, the ClientHello is a ClientHelloInner. This extension MUST only be sent in the ClientHello message.

```
   enum {
      ech_is_inner(0xda09), (65535)
   } ExtensionType;
```

The "extension_data" field of the "ech_is_inner" extension is zero length.

Backend servers (as described in Section 7) MUST support the "ech_is_inner" extension.

6.1.2. Recommended Padding Scheme

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

By way of example, clients typically support a small number of application profiles. For instance, a browser might support HTTP with ALPN values ["http/1.1, "h2"] and WebRTC media with ALPNs ["webrtc", "c-webrtc"]. Clients SHOULD pad this extension by rounding up to the total size of the longest ALPN extension across all application profiles. The target padding length of most ClientHello extensions can be computed in this way.

In contrast, clients do not know the longest SNI value in the client-facing server's anonymity set without server input. For the "server_name" extension with length D, clients SHOULD use the server's length hint L (ECHConfig.contents.maximum_name_length) when computing the padding as follows:

1. If L >= D, add L - D bytes of padding. This rounds to the server's advertised hint, i.e., ECHConfig.contents.maximum_name_length.

2. Otherwise, let P = 31 - ((D - 1) % 32), and add P bytes of padding, plus an additional 32 bytes if D + P < L + 32. This rounds D up to the nearest multiple of 32 bytes that permits at least 32 bytes of length ambiguity.

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

6.1.3. Handling the Server Response

As described in Section 7, the server MAY either accept ECH and use ClientHelloInner or reject it and use ClientHelloOuter. In handling the server's response, the client's first step is to determine which value was used. The client presumes acceptance if the last 8 bytes of ServerHello.random are equal to the first 8 bytes of "accept_confirmation" as defined in Section 7.2. Otherwise, it presumes rejection.

6.1.3.1. Accepted ECH

If the server used ClientHelloInner, the client proceeds with the connection as usual, authenticating the connection for the true server name.

6.1.3.2. Rejected ECH

If the server used ClientHelloOuter, the client proceeds with the handshake, authenticating for ECHConfig.contents.public_name as described in Section 6.1.3.3. If authentication or the handshake fails, the client MUST return a failure to the calling application. It MUST NOT use the retry configurations.

Otherwise, if both authentication and the handshake complete successfully, the client MUST abort the connection with an "ech_required" alert. It then processes the "retry_configs" field from the server's "encrypted_client_hello" extension.

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

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

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

If the server negotiates an earlier version of TLS, or if it does not provide an "encrypted_client_hello" extension in EncryptedExtensions, the client proceeds with the handshake, authenticating for ECHConfig.contents.public_name as described in Section 6.1.3.3. If an earlier version was negotiated, the client MUST NOT enable the False Start optimization [RFC7918] for this handshake. If authentication or the handshake fails, the client MUST return a failure to the calling application. It MUST NOT treat this as a secure signal to disable ECH.

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

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

6.1.3.3. Authenticating for the Public Name

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

* The client MUST verify that the certificate is valid for ECHConfig.contents.public_name. If invalid, it MUST abort the connection with the appropriate alert.

* If the server requests a client certificate, the client MUST respond with an empty Certificate message, denoting no client certificate.

Note that authenticating a connection for the public name does not authenticate it for the origin. The TLS implementation MUST NOT report such connections as successful to the application. It additionally MUST ignore all session tickets and session IDs presented by the server. These connections are only used to trigger retries, as described in Section 6.1.3. This may be implemented, for instance, by reporting a failed connection with a dedicated error code.

6.1.4. Handling HelloRetryRequest

As required in Section 6.1, clients offering ECH MUST ensure that all extensions or parameters that might change in response to receiving a HelloRetryRequest have the same values in ClientHelloInner and ClientHelloOuter. That is, if a HelloRetryRequest causes a parameter to be changed, the same change is applied to both ClientHelloInner and ClientHelloOuter. Applicable parameters include:

1. TLS 1.3 [RFC8446] ciphersuites in the ClientHello.cipher_suites list.

2. The "key_share" and "supported_groups" extensions [RFC8446]. (These extensions may be copied from ClientHelloOuter into ClientHelloInner as described in Section 6.1.)

3. Versions in the "supported_versions" extension, excluding TLS 1.2 and earlier. Note the ClientHelloOuter MAY include these older versions, while the ClientHelloInner MUST omit them.

Future extensions that might change across first and second ClientHello messages in response to a HelloRetryRequest MUST have the same value.

If the server sends a HelloRetryRequest in response to the ClientHello, the client sends a second updated ClientHello per the rules in [RFC8446]. However, at this point, the client does not know whether the server processed ClientHelloOuter or ClientHelloInner, and MUST regenerate both values to be acceptable. Note: if ClientHelloOuter and ClientHelloInner use different groups for their key shares or differ in some other way, then the HelloRetryRequest may actually be invalid for one or the other ClientHello, in which case a fresh ClientHello MUST be generated, ignoring the instructions in HelloRetryRequest. Otherwise, the usual rules for HelloRetryRequest processing apply.

The client encodes the second ClientHelloInner as in Section 5.1, using the second ClientHelloOuter for any referenced extensions. It then encrypts the new EncodedClientHelloInner value as a second message with the previous HPKE context:

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

ClientHelloOuterAAD is computed as described in Section 5.2, but again using the second ClientHelloOuter. Note that the HPKE context maintains a sequence number, so this operation internally uses a fresh nonce for each AEAD operation. Reusing the HPKE context avoids an attack described in Section 10.10.2.

The client then modifies the "encrypted_client_hello" extension in ClientHelloOuter as follows:

* "config_id" is unchanged and contains the "config_id" corresponding to the client’s chosen ECHConfig.

* "cipher_suite" is unchanged and contains the client’s chosen HPKE cipher suite.

* "enc" is replaced with the empty string.

* "payload" is replaced with the value computed above.

If the client offered ECH in the first ClientHello, then it MUST offer ECH in the second. Likewise, if the client did not offer ECH in the first ClientHello, then it MUST NOT not offer ECH in the second.

6.2. GREASE ECH

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

* Set the "config_id" field to a random byte.

* Set the "cipher_suite" field to a supported HpkeSymmetricCipherSuite. The selection SHOULD vary to exercise all supported configurations, but MAY be held constant for successive connections to the same server in the same session.

* Set the "enc" field to a randomly-generated valid encapsulated public key output by the HPKE KEM.

* Set the "payload" field to a randomly-generated string of L+C bytes, where C is the ciphertext expansion of the selected AEAD scheme and L is the size of the EncodedClientHelloInner the client would compute when offering ECH, padded according to Section 6.1.2.

When sending a second ClientHello in response to a HelloRetryRequest, the client copies the entire "encrypted_client_hello" extension from the first ClientHello.

[[OPEN ISSUE: The above doesn't match HRR handling for either ECH acceptance or rejection. See issue https://github.com/tlswg/draft-ietf-tls-esni/issues/358.]]

If the server sends an "encrypted_client_hello" extension, the client MUST check the extension syntactically and abort the connection with a "decode_error" alert if it is invalid. It otherwise ignores the extension and MUST NOT use the retry keys.

[[OPEN ISSUE: if the client sends a GREASE "encrypted_client_hello" extension, should it also send a GREASE "pre_shared_key" extension? If not, GREASE+ticket is a trivial distinguisher.]]

Offering a GREASE extension is not considered offering an encrypted ClientHello for purposes of requirements in Section 6. In particular, the client MAY offer to resume sessions established without ECH.

7. Server Behavior

Servers that support ECH play one of two roles, depending on which of the "ech_is_inner" (Section 6.1.1) and "encrypted_client_hello" (Section 5) extensions are present in the ClientHello:

* If both the "ech_is_inner" and "encrypted_client_hello" extensions are present in the ClientHello, the backend server MUST abort with an "illegal_parameter" alert.

* If only the "encrypted_client_hello" extension is present, the server acts as a client-facing server and proceeds as described in Section 7.1 to extract a ClientHelloInner, if available.

* If only the "ech_is_inner" extension is present and the "encrypted_client_hello" extension is not present, the server acts as a backend server and proceeds as described in Section 7.2.

* If neither extension is present, the server completes the handshake normally, as described in [RFC8446].

7.1. Client-Facing Server

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

If the client offers the "ech_is_inner" extension (Section 6.1.1) in addition to the "encrypted_client_hello" extension, the server MUST abort with an "illegal_parameter" alert.

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

1. Compare ClientECH.config_id against identifiers of each known ECHConfig and select the ones that match, if any, as candidates.

2. Collect all known ECHConfig values as candidates, with trial decryption below determining the final selection.

Some uses of ECH, such as local discovery mode, may randomize the ClientECH.config_id since it can be used as a tracking vector. In such cases, the second method should be used for matching ClientECH to known ECHConfig. See Section 10.4. Unless specified by the application using (D)TLS or externally configured on both sides, implementations MUST use the first method.

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

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

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

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

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

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

Otherwise, if all candidate ECHConfig values fail to decrypt the extension, the client-facing server MUST ignore the extension and proceed with the connection using ClientHelloOuter. This connection proceeds as usual, except the server MUST include the "encrypted_client_hello" extension in its EncryptedExtensions with the "retry_configs" field set to one or more ECHConfig structures with up-to-date keys. Servers MAY supply multiple ECHConfig values of different versions. This allows a server to support multiple versions at once.

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

7.1.1. Handling HelloRetryRequest

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

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

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

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

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

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

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

[[OPEN ISSUE: If the client-facing server implements stateless HRR, it has no way to send a cookie, short of as-yet-unspecified integration with the backend server. Stateful HRR on the client-facing server works fine, however. See issue https://github.com/tlswg/draft-ietf-tls-esni/issues/333.]]

7.2. Backend Server

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

The backend server begins by generating a message ServerHelloECHConf, which is identical in content to a ServerHello message with the exception that ServerHelloECHConf.random is equal to 24 random bytes followed by 8 zero bytes. It then computes a string

```
accept_confirmation =
    Derive-Secret(Handshake Secret,
                  "ech accept confirmation",
                  ClientHelloInner...ServerHelloECHConf)
```

where Derive-Secret and Handshake Secret are as specified in [RFC8446], Section 7.1, and ClientHelloInner...ServerHelloECHConf refers to the sequence of handshake messages beginning with the first ClientHello and ending with ServerHelloECHConf. Finally, the backend server constructs its ServerHello message so that it is equal to ServerHelloECHConf but with the last 8 bytes of ServerHello.random set to the first 8 bytes of accept_confirmation.

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

The "ech_is_inner" is expected to have an empty payload. If the payload is non-empty (i.e., the length of the "extension_data" field is non-zero) then the backend server MUST abort the handshake with an "illegal_parameter" alert.

8. Compatibility Issues

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

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

8.1. Misconfiguration and Deployment Concerns

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

The retry mechanism repairs inconsistencies, provided the server is authoritative for the public name. If server and advertised keys mismatch, the server will respond with ech_retry_requested. If the server does not understand the "encrypted_client_hello" extension at all, it will ignore it as required by [RFC8446]; Section 4.1.2. Provided the server can present a certificate valid for the public name, the client can safely retry with updated settings, as described in Section 6.1.3.

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

8.2. Middleboxes

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

Depending on whether the client is configured to accept the proxy's certificate as authoritative for the public name, this may trigger the retry logic described in Section 6.1.3 or result in a connection failure. A proxy which is not authoritative for the public name cannot forge a signal to disable ECH.

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

9. Compliance Requirements

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

* KEM: DHKEM(X25519, HKDF-SHA256) (see [I-D.irtf-cfrg-hpke], Section 7.1)

* KDF: HKDF-SHA256 (see [I-D.irtf-cfrg-hpke], Section 7.2)

* AEAD: AES-128-GCM (see [I-D.irtf-cfrg-hpke], Section 7.3)

10. Security Considerations

10.1. Security and Privacy Goals

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

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

1. Use of ECH does not weaken the security properties of TLS without ECH.

2. TLS connection establishment to a host with a specific ECHConfig and TLS configuration is indistinguishable from a connection to any other host with the same ECHConfig and TLS configuration. (The set of hosts which share the same ECHConfig and TLS configuration is referred to as the anonymity set.)

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

Beyond these primary security and privacy goals, ECH also aims to hide, to some extent, (a) whether or not a specific server supports ECH and (b) whether or not ECH was accepted for a particular connection. ECH aims to achieve both properties, assuming the attacker is passive and does not know the set of ECH configurations offered by the client-facing server. It does not achieve these properties for active attackers. More specifically:

* Passive attackers with a known ECH configuration can distinguish between a connection that negotiates ECH with that configuration and one which does not, because the latter used a GREASE "encrypted_client_hello" extension (as specified in Section 6.2) or a different ECH configuration.

* Passive attackers without the ECH configuration cannot distinguish between a connection that negotiates ECH and one which uses a GREASE "encrypted_client_hello" extension.

* Active attackers can distinguish between a connection that negotiates ECH and one which uses a GREASE "encrypted_client_hello" extension.

See Section 10.8.4 for more discussion about the "do not stick out" criteria from [RFC8744].

10.2. Unauthenticated and Plaintext DNS

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

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

10.3. Client Tracking

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

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

10.4. Optional Configuration Identifiers and Trial Decryption

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

10.5. Outer ClientHello

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

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

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

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

Clients MAY include additional extensions in ClientHelloOuter to avoid signaling unusual behavior to passive observers, provided the choice of value and value itself are not sensitive. See Section 10.8.4.

10.6. Related Privacy Leaks

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

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

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

10.7. Attacks Exploiting Acceptance Confirmation

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

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

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

10.8. Comparison Against Criteria

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

10.8.1. Mitigate Cut-and-Paste Attacks

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

10.8.2. Avoid Widely Shared Secrets

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

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

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

10.8.4. Do Not Stick Out

The only explicit signal indicating possible use of ECH is the ClientHello "encrypted_client_hello" extension. Server handshake messages do not contain any signal indicating use or negotiation of ECH. Clients MAY GREASE the "encrypted_client_hello" extension, as described in Section 6.2, which helps ensure the ecosystem handles ECH correctly. Moreover, as more clients enable ECH support, e.g., as normal part of Web browser functionality, with keys supplied by shared hosting providers, the presence of ECH extensions becomes less unusual and part of typical client behavior. In other words, if all Web browsers start using ECH, the presence of this value will not signal unusual behavior to passive eavesdroppers.

10.8.5. Maintain Forward Secrecy

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

10.8.6. Enable Multi-party Security Contexts

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

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

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

10.8.7. Support Multiple Protocols

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

10.9. Padding Policy

Variations in the length of the ClientHelloInner ciphertext could leak information about the corresponding plaintext. Section 6.1.2 describes a RECOMMENDED padding mechanism for clients aimed at reducing potential information leakage.

10.10. Active Attack Mitigations

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

10.10.1. Client Reaction Attack Mitigation

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

```
 Client                         Attacker               Server
   ClientHello
   + key_share
   + ech         ------>      (intercept)     -----> X (drop)

                             ServerHello
                             + key_share
                   {EncryptedExtensions}
                   {CertificateRequest*}
                          {Certificate*}
                    {CertificateVerify*}
                 <------
   Alert
                 ------>
```

Figure 3: Client reaction attack

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

10.10.2. HelloRetryRequest Hijack Mitigation

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

```
 Client                          Attacker                     Server
   ClientHello
   + key_share
   + ech         ------>       (forward)        ------->
                                              HelloRetryRequest
                                                    + key_share
                              (intercept)       <-------

                              ClientHello
                              + key_share'
                              + ech'           ------->
                                                    ServerHello
                                                    + key_share
                                          {EncryptedExtensions}
                                          {CertificateRequest*}
                                                 {Certificate*}
                                           {CertificateVerify*}
                                                     {Finished}
                                                <-------
                         (process server flight)
```

Figure 4: HelloRetryRequest hijack attack

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

If the attacker could manipulate the second ClientHello, it might be possible for the server to act as an oracle if it required parameters from the first ClientHello to match that of the second ClientHello. For example, imagine the client's original SNI value in the inner ClientHello is "example.com", and the attacker's hijacked SNI value in its inner ClientHello is "test.com". A server which checks these for equality and changes behavior based on the result can be used as an oracle to learn the client's SNI.

10.10.3. ClientHello Malleability Mitigation

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

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

```
     Client              Attacker                        Server

                                   handshake and ticket
                                      for "example.com"
                                      <-------->

     ClientHello
     + key_share
     + ech
     + ech_outer_extensions(pre_shared_key)
     + pre_shared_key
                 -------->
                       (intercept)
                       ClientHello
                       + key_share
                       + ech
                          + ech_outer_extensions(pre_shared_key)
                       + pre_shared_key'
                                         -------->
                                                        Alert
                                                        -or-
                                                  ServerHello
                                                           ...
                                                     Finished
                                         <--------
```

Figure 5: Message flow for malleable ClientHello

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

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

11. IANA Considerations

11.1. Update of the TLS ExtensionType Registry

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

1. encrypted_client_hello(0xfe0a), with "TLS 1.3" column values set to "CH, EE", and "Recommended" column set to "Yes".

2. ech_is_inner (0xda09), with "TLS 1.3" column values set to "CH", and "Recommended" column set to "Yes".

3. ech_outer_extensions(0xfd00), with the "TLS 1.3" column values set to "", and "Recommended" column set to "Yes".

11.2. Update of the TLS Alert Registry

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

12. ECHConfig Extension Guidance

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

13. References

13.1. Normative References

[HTTPS-RR] Schwartz, B., Bishop, M., and E. Nygren, "Service binding and parameter specification via the DNS (DNS SVCB and HTTPS RRs)", Work in Progress, Internet-Draft, draft-ietf-dnsop-svcb-https-02, 2 November 2020, <http://www.ietf.org/internet-drafts/draft-ietf-dnsop-svcb-https-02.txt>.

[I-D.ietf-tls-exported-authenticator] Sullivan, N., "Exported Authenticators in TLS", Work in Progress, Internet-Draft, draft-ietf-tls-exported-authenticator-14, 25 January 2021, <http://www.ietf.org/internet-drafts/draft-ietf-tls-exported-authenticator-14.txt>.

[I-D.irtf-cfrg-hpke] Barnes, R., Bhargavan, K., Lipp, B., and C. Wood, "Hybrid Public Key Encryption", Work in Progress, Internet-Draft, draft-irtf-cfrg-hpke-07, 16 December 2020, <http://www.ietf.org/internet-drafts/draft-irtf-cfrg-hpke-07.txt>.

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

[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>.

[RFC7918] Langley, A., Modadugu, N., and B. Moeller, "Transport Layer Security (TLS) False Start", RFC 7918, DOI 10.17487/RFC7918, August 2016, <https://www.rfc-editor.org/info/rfc7918>.

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

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

13.2. Informative References

[I-D.kazuho-protected-sni] Oku, K., "TLS Extensions for Protecting SNI", Work in Progress, Internet-Draft, draft-kazuho-protected-sni-00, 18 July 2017, <http://www.ietf.org/internet-drafts/draft-kazuho-protected-sni-00.txt>.

[RFC7301] Friedl, S., Popov, A., Langley, A., and E. Stephan, "Transport Layer Security (TLS) Application-Layer Protocol Negotiation Extension", RFC 7301, DOI 10.17487/RFC7301, July 2014, <https://www.rfc-editor.org/info/rfc7301>.

[RFC7858] Hu, Z., Zhu, L., Heidemann, J., Mankin, A., Wessels, D., and P. Hoffman, "Specification for DNS over Transport Layer Security (TLS)", RFC 7858, DOI 10.17487/RFC7858, May 2016, <https://www.rfc-editor.org/info/rfc7858>.

[RFC7924] Santesson, S. and H. Tschofenig, "Transport Layer Security (TLS) Cached Information Extension", RFC 7924, DOI 10.17487/RFC7924, July 2016, <https://www.rfc-editor.org/info/rfc7924>.

[RFC8094] Reddy, T., Wing, D., and P. Patil, "DNS over Datagram Transport Layer Security (DTLS)", RFC 8094, DOI 10.17487/RFC8094, February 2017, <https://www.rfc-editor.org/info/rfc8094>.

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

[RFC8701] Benjamin, D., "Applying Generate Random Extensions And Sustain Extensibility (GREASE) to TLS Extensibility", RFC 8701, DOI 10.17487/RFC8701, January 2020, <https://www.rfc-editor.org/info/rfc8701>.

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

Appendix A. Alternative SNI Protection Designs

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

A.1. TLS-layer

A.1.1. TLS in Early Data

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

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

A.1.2. Combined Tickets

In this variant, client-facing and backend servers coordinate to produce "combined tickets" that are consumable by both. Clients offer combined tickets to client-facing servers. The latter parse them to determine the correct backend server to which the Client Hello should be forwarded. This approach is problematic due to non-trivial coordination between client-facing and backend servers for ticket construction and consumption. Moreover, it requires a bootstrapping step similar to that of the previous variant. In contrast, encrypted SNI requires no such coordination.

A.2. Application-layer

A.2.1. HTTP/2 CERTIFICATE Frames

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

Appendix B. Acknowledgements

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

Appendix C. Change Log

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

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

C.1. Since draft-ietf-tls-esni-09

* Finalize HPKE dependency (#390)

* Move from client-computed to server-chosen, one-byte config identifier (#376, #381)

* Rename ECHConfigs to ECHConfigList (#391)

* Clarify some security and privacy properties (#385, #383)

Authors' Addresses

Eric Rescorla
RTFM, Inc.

Email: ekr@rtfm.com

Kazuho Oku
Fastly

Email: kazuhooku@gmail.com

Nick Sullivan
Cloudflare

Email: nick@cloudflare.com

Christopher A. Wood
Cloudflare

Email: caw@heapingbits.net

Network Working Group   E. Rescorla
Internet-Draft   Mozilla
Obsoletes: 5077, 5246, 6961, 8446 (if approved)   19 February 2021
Updates: 4492, 5705, 6066, 7627 (if approved)
Intended status: Standards Track
Expires: 23 August 2021

The Transport Layer Security (TLS) Protocol Version 1.3
draft-ietf-tls-rfc8446bis-01

Abstract

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

This document updates RFCs 5705 and 6066 and 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/ekr/tls13-spec. Instructions are on that page as well.

The primary goal of TLS is to provide a secure channel between two communicating peers; the only requirement from the underlying transport is a reliable, in-order, data stream. Specifically, the secure channel should provide the following properties:

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

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

* Integrity: Data sent over the channel after establishment cannot be modified by attackers without detection.

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

TLS consists of two primary components:

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

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

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

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

This document supersedes and obsoletes previous versions of TLS, including version 1.2 [RFC5246]. It also obsoletes the TLS ticket mechanism defined in [RFC5077] and replaces it with the mechanism defined in Section 2.2. Because TLS 1.3 changes the way keys are derived, it updates [RFC5705] as described in Section 7.5. It also changes how Online Certificate Status Protocol (OCSP) messages are carried and therefore updates [RFC6066] and obsoletes [RFC6961] as described in Section 4.4.2.1.

1.1. Conventions and Terminology

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

The following terms are used:

client: The endpoint initiating the TLS connection.

connection: A transport-layer connection between two endpoints.

endpoint: Either the client or server of the connection.

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

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

receiver: An endpoint that is receiving records.

sender: An endpoint that is transmitting records.

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

1.2. Relationship to RFC 8446

TLS 1.3 was originally specified in [RFC8446]. This document is solely an editorial update. It contains updated text in areas which were found to be unclear as well as other editorial improvements. In addition, it removes the use of the term "master" as applied to secrets in favor of the term "main" or shorter names where no term was neccessary.

1.3. Major Differences from TLS 1.2

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

* The list of supported symmetric encryption algorithms has been pruned of all algorithms that are considered legacy. Those that remain are all Authenticated Encryption with Associated Data (AEAD) algorithms. The cipher suite concept has been changed to separate the authentication and key exchange mechanisms from the record protection algorithm (including secret key length) and a hash to be used with both the key derivation function and handshake message authentication code (MAC).

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

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

* All handshake messages after the ServerHello are now encrypted. The newly introduced EncryptedExtensions message allows various extensions previously sent in the clear in the ServerHello to also enjoy confidentiality protection.

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

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

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

* Other cryptographic improvements were made, including changing the RSA padding to use the RSA Probabilistic Signature Scheme (RSASSA-PSS), and the removal of compression, the Digital Signature Algorithm (DSA), and custom Ephemeral Diffie-Hellman (DHE) groups.

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

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

* References have been updated to point to the updated versions of RFCs, as appropriate (e.g., RFC 5280 rather than RFC 3280).

1.4. Updates Affecting TLS 1.2

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

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

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

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

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

* The term "master" as applied to secrets has been removed, and the "extended_master_secret" extension [RFC7627] has been renamed to "extended_main_secret".

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

2. Protocol Overview

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

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

TLS supports three basic key exchange modes:

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

* PSK-only

* PSK with (EC)DHE

Figure 1 below shows the basic full TLS handshake:

```
       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 list of Diffie-Hellman key shares (in the "key_share" (Section 4.2.8) extension), a list of pre-shared key labels (in the "pre_shared_key" (Section 4.2.11) extension), or both; and potentially additional extensions. Additional fields and/or messages may also be present for middlebox compatibility.

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

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

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

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

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

Certificate: The certificate of the endpoint and any per-certificate extensions. This message is omitted by the server if not authenticating with a certificate and by the client if the server did not send CertificateRequest (thus indicating that the client should not authenticate with a certificate). Note that if raw public keys [RFC7250] or the cached information extension [RFC7924] are in use, then this message will not contain a certificate but rather some other value corresponding to the server's long-term key. [Section 4.4.2]

CertificateVerify: A signature over the entire handshake using the private key corresponding to the public key in the Certificate message. This message is omitted if the endpoint is not authenticating via a certificate. [Section 4.4.3]

Finished: A MAC (Message Authentication Code) over the entire handshake. This message provides key confirmation, binds the endpoint's identity to the exchanged keys, and in PSK mode also authenticates the handshake. [Section 4.4.4]

Upon receiving the server's messages, the client responds with its Authentication messages, namely Certificate and CertificateVerify (if requested), and Finished.

At this point, the handshake is complete, and the client and server derive the keying material required by the record layer to exchange application-layer data protected through authenticated encryption. Application Data MUST NOT be sent prior to sending the Finished message, except as specified in Section 2.3. Note that while the server may send Application Data prior to receiving the client's Authentication messages, any data sent at that point is, of course, being sent to an unauthenticated peer.

2.1. Incorrect DHE Share

If the client has not provided a sufficient "key_share" extension (e.g., it includes only DHE or ECDHE groups unacceptable to or unsupported by the server), the server corrects the mismatch with a HelloRetryRequest and the client needs to restart the handshake with an appropriate "key_share" extension, as shown in Figure 2. If no common cryptographic parameters can be negotiated, the server MUST abort the handshake with an appropriate alert.

```
       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 externally, PSKs can also be established in a previous connection and then used to establish a new connection ("session resumption" or "resuming" with a PSK). Once a handshake has completed, the server can send the client a PSK identity that corresponds to a unique key derived from the initial handshake (see Section 4.6.1). The client can then use that PSK identity in future handshakes to negotiate the use of the associated PSK. If the server accepts the PSK, then the security context of the new connection is cryptographically tied to the original connection and the key derived from the initial handshake is used to bootstrap the cryptographic state instead of a full handshake. In TLS 1.2 and below, this functionality was provided by "session IDs" and "session tickets" [RFC5077]. Both mechanisms are obsoleted in TLS 1.3.

PSKs can be used with (EC)DHE key exchange in order to provide forward secrecy in combination with shared keys, or can be used alone, at the cost of losing forward secrecy for the application data.

Figure 3 shows a pair of handshakes in which the first handshake establishes a PSK and the second handshake uses it:

```
       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 a PSK, it SHOULD also supply a "key_share" extension to the server to allow the server to decline resumption and fall back to a full handshake, if needed. The server responds with a "pre_shared_key" extension to negotiate the use of PSK key establishment and can (as shown here) respond with a "key_share" extension to do (EC)DHE key establishment, thus providing forward secrecy.

When PSKs are provisioned externally, the PSK identity and the KDF hash algorithm to be used with the PSK MUST also be provisioned.

Note: When using an externally provisioned pre-shared secret, a critical consideration is using sufficient entropy during the key generation, as discussed in [RFC4086]. Deriving a shared secret from a password or other low-entropy sources is not secure. A low-entropy secret, or password, is subject to dictionary attacks based on the PSK binder. The specified PSK authentication is not a strong password-based authenticated key exchange even when used with Diffie-Hellman key establishment. Specifically, it does not prevent an attacker that can observe the handshake from performing a brute-force attack on the password/pre-shared key.

2.3. 0-RTT Data

When clients and servers share a PSK (either obtained externally or via a previous handshake), TLS 1.3 allows clients to send data on the first flight ("early data"). The client uses the PSK to authenticate the server and to encrypt the early data.

As shown in Figure 4, the 0-RTT data is just added to the 1-RTT handshake in the first flight. The rest of the handshake uses the same messages as for a 1-RTT handshake with PSK resumption.

```
       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 a 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 0-RTT 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_secret.

0-RTT data cannot be duplicated within a connection (i.e., the server will not process the same data twice for the same connection), and an attacker will not be able to make 0-RTT data appear to be 1-RTT data (because it is protected with different keys). Appendix F.5 contains a description of potential attacks, and Section 8 describes mechanisms which the server can use to limit the impact of replay.

3. Presentation Language

This document deals with the formatting of data in an external representation. The following very basic and somewhat casually defined presentation syntax will be used.

3.1. Basic Block Size

The representation of all data items is explicitly specified. The basic data block size is one byte (i.e., 8 bits). Multiple-byte data items are concatenations of bytes, from left to right, from top to bottom. From the byte stream, a multi-byte item (a numeric in the following example) is formed (using C notation) by:

```
value = (byte[0] << 8*(n-1)) | (byte[1] << 8*(n-2)) |
        ... | byte[n-1];
```

This byte ordering for multi-byte values is the commonplace network byte order or big-endian format.

3.2. Miscellaneous

Comments begin with "/*" and end with "*/".

Optional components are denoted by enclosing them in "[[ ]]" (double brackets).

Single-byte entities containing uninterpreted data are of type opaque.

A type alias T' for an existing type T is defined by:

```
T T';
```

3.3. Numbers

The basic numeric data type is an unsigned byte (uint8). All larger numeric data types are constructed from a fixed-length series of bytes concatenated as described in Section 3.1 and are also unsigned. The following numeric types are predefined.

```
uint8 uint16[2];
uint8 uint24[3];
uint8 uint32[4];
uint8 uint64[8];
```

All values, here and elsewhere in the specification, are transmitted in network byte (big-endian) order; the uint32 represented by the hex bytes 01 02 03 04 is equivalent to the decimal value 16909060.

3.4. Vectors

A vector (single-dimensioned array) is a stream of homogeneous data elements. For presentation purposes, this specification refers to vectors as lists. The size of the vector may be specified at documentation time or left unspecified until runtime. In either case, the length declares the number of bytes, not the number of elements, in the vector. The syntax for specifying a new type, T', that is a fixed-length vector of type T is

```
T T'[n];
```

Here, T' occupies n bytes in the data stream, where n is a multiple of the size of T. The length of the vector is not included in the encoded stream.

In the following example, Datum is defined to be three consecutive bytes that the protocol does not interpret, while Data is three consecutive Datum, consuming a total of nine bytes.

```
opaque Datum[3];      /* three uninterpreted bytes */
Datum Data[9];        /* three consecutive 3-byte vectors */
```

Variable-length vectors are defined by specifying a subrange of legal lengths, inclusively, using the notation <floor..ceiling>. When these are encoded, the actual length precedes the vector's contents in the byte stream. The length will be in the form of a number consuming as many bytes as required to hold the vector's specified maximum (ceiling) length. A variable-length vector with an actual length field of zero is referred to as an empty vector.

```
T T'<floor..ceiling>;
```

In the following example, "mandatory" is a vector that must contain between 300 and 400 bytes of type opaque. It can never be empty. The actual length field consumes two bytes, a uint16, which is sufficient to represent the value 400 (see Section 3.3). Similarly, "longer" can represent up to 800 bytes of data, or 400 uint16 elements, and it may be empty. Its encoding will include a two-byte actual length field prepended to the vector. The length of an encoded vector must be an exact multiple of the length of a single element (e.g., a 17-byte vector of uint16 would be illegal).

```
opaque mandatory<300..400>;
      /* length field is two bytes, cannot be empty */
uint16 longer<0..800>;
      /* zero to 400 16-bit unsigned integers */
```

3.5. Enumerateds

An additional sparse data type, called "enum" or "enumerated", is available. Each definition is a different type. Only enumerateds of the same type may be assigned or compared. Every element of an enumerated must be assigned a value, as demonstrated in the following example. Since the elements of the enumerated are not ordered, they can be assigned any unique value, in any order.

```
enum { e1(v1), e2(v2), ... , en(vn) [[, (n)]] } Te;
```

Future extensions or additions to the protocol may define new values. Implementations need to be able to parse and ignore unknown values unless the definition of the field states otherwise.

An enumerated occupies as much space in the byte stream as would its maximal defined ordinal value. The following definition would cause one byte to be used to carry fields of type Color.

```
enum { red(3), blue(5), white(7) } Color;
```

One may optionally specify a value without its associated tag to force the width definition without defining a superfluous element.

In the following example, Taste will consume two bytes in the data stream but can only assume the values 1, 2, or 4 in the current version of the protocol.

```
enum { sweet(1), sour(2), bitter(4), (32000) } Taste;
```

The names of the elements of an enumeration are scoped within the defined type. In the first example, a fully qualified reference to the second element of the enumeration would be Color.blue. Such qualification is not required if the target of the assignment is well specified.

```
Color color = Color.blue;     /* overspecified, legal */
Color color = blue;           /* correct, type implicit */
```

The names assigned to 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 used for definitions is much like that of C.

```
struct {
    T1 f1;
    T2 f2;
    ...
    Tn fn;
} T;
```

Fixed- and variable-length list (vector) fields are allowed using the standard list syntax. Structures V1 and V2 in the variants example (Section 3.8) demonstrate this.

The fields within a structure may be qualified using the type's name, with a syntax much like that available for 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 (below) specifies the type of that variant's field and an optional field label. The mechanism by which the variant is selected at runtime is not prescribed by the presentation language.

```
struct {
    T1 f1;
    T2 f2;
    ....
    Tn fn;
    select (E) {
        case e1: 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;             /* remaining bytes in message */
       select (Handshake.msg_type) {
           case client_hello:          ClientHello;
           case server_hello:          ServerHello;
           case end_of_early_data:     EndOfEarlyData;
           case encrypted_extensions:  EncryptedExtensions;
           case certificate_request:   CertificateRequest;
           case certificate:           Certificate;
           case certificate_verify:    CertificateVerify;
           case finished:              Finished;
           case new_session_ticket:    NewSessionTicket;
           case key_update:            KeyUpdate;
       };
   } Handshake;
```

Protocol messages MUST be sent in the order defined in Section 4.4.1 and shown in the diagrams in Section 2. A peer which receives a handshake message in an unexpected order MUST abort the handshake with an "unexpected_message" alert.

New handshake message types are assigned by IANA as described in Section 11.

4.1. Key Exchange Messages

The key exchange messages are used to determine the security capabilities of the client and the server and to establish shared secrets, including the traffic keys used to protect the rest of the handshake and the data.

4.1.1. Cryptographic Negotiation

In TLS, the cryptographic negotiation proceeds by the client offering the following four sets of options in its ClientHello:

* A list of cipher suites which indicates the AEAD algorithm/HKDF hash pairs which the client supports.

* A "supported_groups" (Section 4.2.7) extension which indicates the (EC)DHE groups which the client supports and a "key_share" (Section 4.2.8) extension which contains (EC)DHE shares for some or all of these groups.

* A "signature_algorithms" (Section 4.2.3) extension which indicates the signature algorithms which the client can accept. A "signature_algorithms_cert" extension (Section 4.2.3) may also be added to indicate certificate-specific signature algorithms.

* A "pre_shared_key" (Section 4.2.11) extension which contains a list of symmetric key identities known to the client and a "psk_key_exchange_modes" (Section 4.2.9) extension which indicates the key exchange modes that may be used with PSKs.

If the server does not select a PSK, then the first three of these options are entirely orthogonal: the server independently selects a cipher suite, an (EC)DHE group and key share for key establishment, and a signature algorithm/certificate pair to authenticate itself to the client. If there is no overlap between the received "supported_groups" and the groups supported by the server, then the server MUST abort the handshake with a "handshake_failure" or an "insufficient_security" alert.

If the server selects a PSK, then it MUST also select a key establishment mode from the list indicated by the client's "psk_key_exchange_modes" extension (at present, PSK alone or with (EC)DHE). Note that if the PSK can be used without (EC)DHE, then non-overlap in the "supported_groups" parameters need not be fatal, as it is in the non-PSK case discussed in the previous paragraph.

If the server selects an (EC)DHE group and the client did not offer a compatible "key_share" extension in the initial ClientHello, the server MUST respond with a HelloRetryRequest (Section 4.1.4) message.

If the server successfully selects parameters and does not require a HelloRetryRequest, it indicates the selected parameters in the ServerHello as follows:

* If PSK is being used, then the server will send a "pre_shared_key" extension indicating the selected key.

* When (EC)DHE is in use, the server will also provide a "key_share" extension. If PSK is not being used, then (EC)DHE and certificate-based authentication are always used.

* When authenticating via a certificate, the server will send the Certificate (Section 4.4.2) and CertificateVerify (Section 4.4.3) messages. In TLS 1.3 as defined by this document, either a PSK or a certificate is always used, but not both. Future documents may define how to use them together.

If the server is unable to negotiate a supported set of parameters (i.e., there is no overlap between the client and server parameters), it MUST abort the handshake with either a "handshake_failure" or "insufficient_security" fatal alert (see Section 6).

4.1.2. Client Hello

When a client first connects to a server, it is REQUIRED to send the ClientHello as its first TLS message. The client will also send a ClientHello when the server has responded to its ClientHello with a HelloRetryRequest. In that case, the client MUST send the same ClientHello without modification, except as follows:

* If a "key_share" extension was supplied in the HelloRetryRequest, replacing the list of shares with a list containing a single KeyShareEntry from the indicated group.

* Removing the "early_data" extension (Section 4.2.10) if one was present. Early data is not permitted after a HelloRetryRequest.

* Including a "cookie" extension if one was provided in the HelloRetryRequest.

* Updating the "pre_shared_key" extension if present by recomputing the "obfuscated_ticket_age" and binder values and (optionally) removing any PSKs which are incompatible with the server's indicated cipher suite.

* Optionally adding, removing, or changing the length of the "padding" extension [RFC7685].

* Other modifications that may be allowed by an extension defined in the future and present in the HelloRetryRequest.

Because TLS 1.3 forbids renegotiation, if a server has negotiated TLS 1.3 and receives a ClientHello at any other time, it MUST terminate the connection with an "unexpected_message" alert.

If a server established a TLS connection with a previous version of TLS and receives a TLS 1.3 ClientHello in a renegotiation, it MUST retain the previous protocol version. In particular, it MUST NOT negotiate TLS 1.3.

Structure of this message:

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

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

struct {
    ProtocolVersion legacy_version = 0x0303;    /* TLS v1.2 */
    Random random;
    opaque legacy_session_id<0..32>;
    CipherSuite cipher_suites<2..2^16-2>;
    opaque legacy_compression_methods<1..2^8-1>;
    Extension extensions<8..2^16-1>;
} ClientHello;
```

legacy_version: In previous versions of TLS, this field was used for version negotiation and represented the highest version number supported by the client. Experience has shown that many servers do not properly implement version negotiation, leading to "version intolerance" in which the server rejects an otherwise acceptable ClientHello with a version number higher than it supports. In TLS 1.3, the client indicates its version preferences in the "supported_versions" extension (Section 4.2.1) and the legacy_version field MUST be set to 0x0303, which is the version number for TLS 1.2. TLS 1.3 ClientHellos are identified as having a legacy_version of 0x0303 and a supported_versions extension present with 0x0304 as the highest version indicated therein. (See Appendix E for details about backward compatibility.) A server which receives a legacy_version value not equal to 0x0303 MUST abort the handshake with an "illegal_parameter" alert.

random: 32 bytes generated by a secure random number generator. See

Appendix C for additional information.

legacy_session_id: Versions of TLS before TLS 1.3 supported a "session resumption" feature which has been merged with pre-shared keys in this version (see Section 2.2). A client which has a cached session ID set by a pre-TLS 1.3 server SHOULD set this field to that value. In compatibility mode (see Appendix E.4), this field MUST be non-empty, so a client not offering a pre-TLS 1.3 session MUST generate a new 32-byte value. This value need not be random but SHOULD be unpredictable to avoid implementations fixating on a specific value (also known as ossification). Otherwise, it MUST be set as a zero-length list (i.e., a zero-valued single byte length field).

cipher_suites: A list of the symmetric cipher options supported by the client, specifically the record protection algorithm (including secret key length) and a hash to be used with HKDF, in descending order of client preference. Values are defined in Appendix B.4. If the list contains cipher suites that the server does not recognize, support, or wish to use, the server MUST ignore those cipher suites and process the remaining ones as usual. If the client is attempting a PSK key establishment, it SHOULD advertise at least one cipher suite indicating a Hash associated with the PSK.

legacy_compression_methods: Versions of TLS before 1.3 supported compression with the list of supported compression methods being sent in this field. For every TLS 1.3 ClientHello, this list MUST contain exactly one byte, set to zero, which corresponds to the "null" compression method in prior versions of TLS. If a TLS 1.3 ClientHello is received with any other value in this field, the server MUST abort the handshake with an "illegal_parameter" alert. Note that TLS 1.3 servers might receive TLS 1.2 or prior ClientHellos which contain other compression methods and (if negotiating such a prior version) MUST follow the procedures for the appropriate prior version of TLS.

extensions: Clients request extended functionality from servers by sending data in the extensions field. The actual "Extension" format is defined in Section 4.2. In TLS 1.3, the use of certain extensions is mandatory, as functionality has moved into extensions to preserve ClientHello compatibility with previous versions of TLS. Servers MUST ignore unrecognized extensions.

All versions of TLS allow an extensions field to optionally follow the compression_methods field. TLS 1.3 ClientHello messages always contain extensions (minimally "supported_versions", otherwise, they will be interpreted as TLS 1.2 ClientHello messages). However, TLS

1.3 servers might receive ClientHello messages without an extensions field from prior versions of TLS. The presence of extensions can be detected by determining whether there are bytes following the compression_methods field at the end of the ClientHello. Note that this method of detecting optional data differs from the normal TLS method of having a variable-length field, but it is used for compatibility with TLS before extensions were defined. TLS 1.3 servers will need to perform this check first and only attempt to negotiate TLS 1.3 if the "supported_versions" extension is present. If negotiating a version of TLS prior to 1.3, a server MUST check that the message either contains no data after legacy_compression_methods or that it contains a valid extensions block with no data following. If not, then it MUST abort the handshake with a "decode_error" alert.

In the event that a client requests additional functionality using extensions and this functionality is not supplied by the server, the client MAY abort the handshake.

After sending the ClientHello message, the client waits for a ServerHello or HelloRetryRequest message. If early data is in use, the client may transmit early Application Data (Section 2.3) while waiting for the next handshake message.

4.1.3. Server Hello

The server will send this message in response to a ClientHello message to proceed with the handshake if it is able to negotiate an acceptable set of handshake parameters based on the ClientHello.

Structure of this message:

```
struct {
    ProtocolVersion legacy_version = 0x0303;    /* TLS v1.2 */
    Random random;
    opaque legacy_session_id_echo<0..32>;
    CipherSuite cipher_suite;
    uint8 legacy_compression_method = 0;
    Extension extensions<6..2^16-1>;
} ServerHello;
```

legacy_version: In previous versions of TLS, this field was used for version negotiation and represented the selected version number for the connection. Unfortunately, some middleboxes fail when presented with new values. In TLS 1.3, the TLS server indicates its version using the "supported_versions" extension (Section 4.2.1), and the legacy_version field MUST be set to 0x0303, which is the version number for TLS 1.2. (See Appendix E for details about backward compatibility.)

random: 32 bytes generated by a secure random number generator. See Appendix C for additional information. The last 8 bytes MUST be overwritten as described below if negotiating TLS 1.2 or TLS 1.1, but the remaining bytes MUST be random. This structure is generated by the server and MUST be generated independently of the ClientHello.random.

legacy_session_id_echo: The contents of the client's legacy_session_id field. Note that this field is echoed even if the client's value corresponded to a cached pre-TLS 1.3 session which the server has chosen not to resume. A client which receives a legacy_session_id_echo field that does not match what it sent in the ClientHello MUST abort the handshake with an "illegal_parameter" alert.

cipher_suite: The single cipher suite selected by the server from the ClientHello.cipher_suites list. A client which receives a cipher suite that was not offered MUST abort the handshake with an "illegal_parameter" alert.

legacy_compression_method: A single byte which MUST have the value 0.

extensions: A list of extensions. The ServerHello MUST only include extensions which are required to establish the cryptographic context and negotiate the protocol version. All TLS 1.3 ServerHello messages MUST contain the "supported_versions" extension. Current ServerHello messages additionally contain either the "pre_shared_key" extension or the "key_share" extension, or both (when using a PSK with (EC)DHE key establishment). Other extensions (see Section 4.2) are sent separately in the EncryptedExtensions message.

For reasons of backward compatibility with middleboxes (see Appendix E.4), the HelloRetryRequest message uses the same structure as the ServerHello, but with Random set to the special value of the SHA-256 of "HelloRetryRequest":

```
CF 21 AD 74 E5 9A 61 11 BE 1D 8C 02 1E 65 B8 91
C2 A2 11 16 7A BB 8C 5E 07 9E 09 E2 C8 A8 33 9C
```

Upon receiving a message with type server_hello, implementations MUST first examine the Random value and, if it matches this value, process it as described in Section 4.1.4).

TLS 1.3 has a downgrade protection mechanism embedded in the server's random value. TLS 1.3 servers which negotiate TLS 1.2 or below in response to a ClientHello MUST set the last 8 bytes of their Random value specially in their ServerHello.

If negotiating TLS 1.2, TLS 1.3 servers MUST set the last 8 bytes of their Random value to the bytes:

```
44 4F 57 4E 47 52 44 01
```

If negotiating TLS 1.1 or below, TLS 1.3 servers MUST, and TLS 1.2 servers SHOULD, set the last 8 bytes of their ServerHello.Random value to the bytes:

```
44 4F 57 4E 47 52 44 00
```

TLS 1.3 clients receiving a ServerHello indicating TLS 1.2 or below MUST check that the last 8 bytes are not equal to either of these values. TLS 1.2 clients SHOULD also check that the last 8 bytes are not equal to the second value if the ServerHello indicates TLS 1.1 or below. If a match is found, the client MUST abort the handshake with an "illegal_parameter" alert. This mechanism provides limited protection against downgrade attacks over and above what is provided by the Finished exchange: because the ServerKeyExchange, a message present in TLS 1.2 and below, includes a signature over both random values, it is not possible for an active attacker to modify the random values without detection as long as ephemeral ciphers are used. It does not provide downgrade protection when static RSA is used.

Note: This is a change from [RFC5246], so in practice many TLS 1.2 clients and servers will not behave as specified above.

A legacy TLS client performing renegotiation with TLS 1.2 or prior and which receives a TLS 1.3 ServerHello during renegotiation MUST abort the handshake with a "protocol_version" alert. Note that renegotiation is not possible when TLS 1.3 has been negotiated.

4.1.4. Hello Retry Request

The server will send this message in response to a ClientHello message if it is able to find an acceptable set of parameters but the ClientHello does not contain sufficient information to proceed with the handshake. As discussed in Section 4.1.3, the HelloRetryRequest has the same format as a ServerHello message, and the legacy_version, legacy_session_id_echo, cipher_suite, and legacy_compression_method fields have the same meaning. However, for convenience we discuss "HelloRetryRequest" throughout this document as if it were a distinct message.

The server's extensions MUST contain "supported_versions". Additionally, it SHOULD contain the minimal set of extensions necessary for the client to generate a correct ClientHello pair. As with the ServerHello, a HelloRetryRequest MUST NOT contain any extensions that were not first offered by the client in its ClientHello, with the exception of optionally the "cookie" (see Section 4.2.2) extension.

Upon receipt of a HelloRetryRequest, the client MUST check the legacy_version, legacy_session_id_echo, cipher_suite, and legacy_compression_method as specified in Section 4.1.3 and then process the extensions, starting with determining the version using "supported_versions". Clients MUST abort the handshake with an "illegal_parameter" alert if the HelloRetryRequest would not result in any change in the ClientHello. If a client receives a second HelloRetryRequest in the same connection (i.e., where the ClientHello was itself in response to a HelloRetryRequest), it MUST abort the handshake with an "unexpected_message" alert.

Otherwise, the client MUST process all extensions in the HelloRetryRequest and send a second updated ClientHello. The HelloRetryRequest extensions defined in this specification are:

* supported_versions (see Section 4.2.1)

* cookie (see Section 4.2.2)

* key_share (see Section 4.2.8)

A client which receives a cipher suite that was not offered MUST abort the handshake. Servers MUST ensure that they negotiate the same cipher suite when receiving a conformant updated ClientHello (if the server selects the cipher suite as the first step in the negotiation, then this will happen automatically). Upon receiving the ServerHello, clients MUST check that the cipher suite supplied in the ServerHello is the same as that in the HelloRetryRequest and otherwise abort the handshake with an "illegal_parameter" alert.

In addition, in its updated ClientHello, the client SHOULD NOT offer any pre-shared keys associated with a hash other than that of the selected cipher suite. This allows the client to avoid having to compute partial hash transcripts for multiple hashes in the second ClientHello.

The value of selected_version in the HelloRetryRequest "supported_versions" extension MUST be retained in the ServerHello, and a client MUST abort the handshake with an "illegal_parameter" alert if the value changes.

4.2. Extensions

A number of TLS messages contain tag-length-value encoded extensions structures.

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

    enum {
        server_name(0),                             /* RFC 6066 */
        max_fragment_length(1),                     /* RFC 6066 */
        status_request(5),                          /* RFC 6066 */
        supported_groups(10),                       /* RFC 8422, 7919 */
        signature_algorithms(13),                   /* RFC 8446 */
        use_srtp(14),                               /* RFC 5764 */
        heartbeat(15),                              /* RFC 6520 */
        application_layer_protocol_negotiation(16), /* RFC 7301 */
        signed_certificate_timestamp(18),           /* RFC 6962 */
        client_certificate_type(19),                /* RFC 7250 */
        server_certificate_type(20),                /* RFC 7250 */
        padding(21),                                /* RFC 7685 */
        pre_shared_key(41),                         /* RFC 8446 */
        early_data(42),                             /* RFC 8446 */
        supported_versions(43),                     /* RFC 8446 */
        cookie(44),                                 /* RFC 8446 */
        psk_key_exchange_modes(45),                 /* RFC 8446 */
        certificate_authorities(47),                /* RFC 8446 */
        oid_filters(48),                            /* RFC 8446 */
        post_handshake_auth(49),                    /* RFC 8446 */
        signature_algorithms_cert(50),              /* RFC 8446 */
        key_share(51),                              /* RFC 8446 */
        (65535)
    } ExtensionType;
```

Here:

* "extension_type" identifies the particular extension type.

* "extension_data" contains information specific to the particular extension type.

The 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 the HelloRetryRequest. Upon receiving such an extension, an endpoint MUST abort the handshake with an "unsupported_extension" alert.

The table below indicates the messages where a given extension may appear, using the following notation: CH (ClientHello), SH (ServerHello), EE (EncryptedExtensions), CT (Certificate), CR (CertificateRequest), NST (NewSessionTicket), and HRR (HelloRetryRequest). If an implementation receives an extension which it recognizes and which is not specified for the message in which it appears, it MUST abort the handshake with an "illegal_parameter" alert.

Extension	TLS 1.3
server_name [RFC6066]	CH, EE
max_fragment_length [RFC6066]	CH, EE
status_request [RFC6066]	CH, CR, CT
supported_groups [RFC7919]	CH, EE
signature_algorithms (RFC8446)	CH, CR
use_srtp [RFC5764]	CH, EE
heartbeat [RFC6520]	CH, EE
application_layer_protocol_negotiation [RFC7301]	CH, EE
signed_certificate_timestamp [RFC6962]	CH, CR, CT
client_certificate_type [RFC7250]	CH, EE
server_certificate_type [RFC7250]	CH, EE
padding [RFC7685]	CH
key_share (RFC 8446)	CH, SH, HRR

pre_shared_key (RFC 8446)	CH, SH
psk_key_exchange_modes (RFC 8446)	CH
early_data (RFC 8446)	CH, EE, NST
cookie (RFC 8446)	CH, HRR
supported_versions (RFC 8446)	CH, SH, HRR
certificate_authorities (RFC 8446)	CH, CR
oid_filters (RFC 8446)	CR
post_handshake_auth (RFC 8446)	CH
signature_algorithms_cert (RFC 8446)	CH, CR

Table 1: TLS Extensions

When multiple extensions of different types are present, the extensions MAY appear in any order, with the exception of "pre_shared_key" (Section 4.2.11) which MUST be the last extension in the ClientHello (but can appear anywhere in the ServerHello extensions block). There MUST NOT be more than one extension of the same type in a given extension block.

In TLS 1.3, unlike TLS 1.2, extensions are negotiated for each handshake even when in resumption-PSK mode. However, 0-RTT parameters are those negotiated in the previous handshake; mismatches may require rejecting 0-RTT (see Section 4.2.10).

There are subtle (and not so subtle) interactions that may occur in this protocol between new features and existing features which may result in a significant reduction in overall security. The following considerations should be taken into account when designing new extensions:

* Some cases where a server does not agree to an extension are error conditions (e.g., the handshake cannot continue), and some are simply refusals to support particular features. In general, error alerts should be used for the former and a field in the server extension response for the latter.

* Extensions should, as far as possible, be designed to prevent any attack that forces use (or non-use) of a particular feature by manipulation of handshake messages. This principle should be

followed regardless of whether the feature is believed to cause a security problem. Often the fact that the extension fields are included in the inputs to the Finished message hashes will be sufficient, but extreme care is needed when the extension changes the meaning of messages sent in the handshake phase. Designers and implementors should be aware of the fact that until the handshake has been authenticated, active attackers can modify messages and insert, remove, or replace extensions.

4.2.1. Supported Versions

```
struct {
    select (Handshake.msg_type) {
        case client_hello:
             ProtocolVersion versions<2..254>;

        case server_hello: /* and HelloRetryRequest */
             ProtocolVersion selected_version;
    };
} SupportedVersions;
```

The "supported_versions" extension is used by the client to indicate which versions of TLS it supports and by the server to indicate which version it is using. The extension contains a list of supported versions in preference order, with the most preferred version first. Implementations of this specification MUST send this extension in the ClientHello containing all versions of TLS which they are prepared to negotiate (for this specification, that means minimally 0x0304, but if previous versions of TLS are allowed to be negotiated, they MUST be present as well).

If this extension is not present, servers which are compliant with this specification and which also support TLS 1.2 MUST negotiate TLS 1.2 or prior as specified in [RFC5246], even if ClientHello.legacy_version is 0x0304 or later. Servers MAY abort the handshake upon receiving a ClientHello with legacy_version 0x0304 or later.

If this extension is present in the ClientHello, servers MUST NOT use the ClientHello.legacy_version value for version negotiation and MUST use only the "supported_versions" extension to determine client preferences. Servers MUST only select a version of TLS present in that extension and MUST ignore any unknown versions that are present in that extension. Note that this mechanism makes it possible to negotiate a version prior to TLS 1.2 if one side supports a sparse range. Implementations of TLS 1.3 which choose to support prior versions of TLS SHOULD support TLS 1.2. Servers MUST be prepared to receive ClientHellos that include this extension but do not include 0x0304 in the list of versions.

A server which negotiates a version of TLS prior to TLS 1.3 MUST set ServerHello.version and MUST NOT send the "supported_versions" extension. A server which negotiates TLS 1.3 MUST respond by sending a "supported_versions" extension containing the selected version value (0x0304). It MUST set the ServerHello.legacy_version field to 0x0303 (TLS 1.2).

After checking ServerHello.random to determine if the server handshake message is a ServerHello or HelloRetryRequest, clients MUST check for this extension prior to processing the rest of the ServerHello. This will require clients to parse the ServerHello in order to read the extension. If this extension is present, clients MUST ignore the ServerHello.legacy_version value and MUST use only the "supported_versions" extension to determine the selected version. If the "supported_versions" extension in the ServerHello contains a version not offered by the client or contains a version prior to TLS 1.3, the client MUST abort the handshake with an "illegal_parameter" alert.

4.2.2. Cookie

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

Cookies serve two primary purposes:

* Allowing the server to force the client to demonstrate reachability at their apparent network address (thus providing a measure of DoS protection). This is primarily useful for non-connection-oriented transports (see [RFC6347] for an example of this).

* Allowing the server to offload state to the client, thus allowing it to send a HelloRetryRequest without storing any state. The server can do this by storing the hash of the ClientHello in the HelloRetryRequest cookie (protected with some suitable integrity protection algorithm).

When sending a HelloRetryRequest, the server MAY provide a "cookie" extension to the client (this is an exception to the usual rule that the only extensions that may be sent are those that appear in the ClientHello). When sending the new ClientHello, the client MUST copy the contents of the extension received in the HelloRetryRequest into a "cookie" extension in the new ClientHello. Clients MUST NOT use cookies in their initial ClientHello in subsequent connections.

When a server is operating statelessly, it may receive an unprotected record of type change_cipher_spec between the first and second ClientHello (see Section 5). Since the server is not storing any state, this will appear as if it were the first message to be received. Servers operating statelessly MUST ignore these records.

4.2.3. Signature Algorithms

TLS 1.3 provides two extensions for indicating which signature algorithms may be used in digital signatures. The "signature_algorithms_cert" extension applies to signatures in certificates, and the "signature_algorithms" extension, which originally appeared in TLS 1.2, applies to signatures in CertificateVerify messages. The keys found in certificates MUST also be of appropriate type for the signature algorithms they are used with. This is a particular issue for RSA keys and PSS signatures, as described below. If no "signature_algorithms_cert" extension is present, then the "signature_algorithms" extension also applies to signatures appearing in certificates. Clients which desire the server to authenticate itself via a certificate MUST send the "signature_algorithms" extension. If a server is authenticating via a certificate and the client has not sent a "signature_algorithms" extension, then the server MUST abort the handshake with a "missing_extension" alert (see Section 9.2).

The "signature_algorithms_cert" extension was added to allow implementations which supported different sets of algorithms for certificates and in TLS itself to clearly signal their capabilities. TLS 1.2 implementations SHOULD also process this extension. Implementations which have the same policy in both cases MAY omit the "signature_algorithms_cert" extension.

The "extension_data" field of these extensions contains a SignatureSchemeList value:

```
      enum {
          /* RSASSA-PKCS1-v1_5 algorithms */
          rsa_pkcs1_sha256(0x0401),
          rsa_pkcs1_sha384(0x0501),
          rsa_pkcs1_sha512(0x0601),

          /* ECDSA algorithms */
          ecdsa_secp256r1_sha256(0x0403),
          ecdsa_secp384r1_sha384(0x0503),
          ecdsa_secp521r1_sha512(0x0603),

          /* RSASSA-PSS algorithms with public key OID rsaEncryption */
          rsa_pss_rsae_sha256(0x0804),
          rsa_pss_rsae_sha384(0x0805),
          rsa_pss_rsae_sha512(0x0806),

          /* EdDSA algorithms */
          ed25519(0x0807),
          ed448(0x0808),

          /* RSASSA-PSS algorithms with public key OID RSASSA-PSS */
          rsa_pss_pss_sha256(0x0809),
          rsa_pss_pss_sha384(0x080a),
          rsa_pss_pss_sha512(0x080b),

          /* Legacy algorithms */
          rsa_pkcs1_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 [ECDSA] and FIPS 186-4 [DSS], and the corresponding hash algorithm as defined in [SHS]. The signature is represented as a DER-encoded [X690] ECDSA-Sig-Value structure as defined in [RFC4492].

RSASSA-PSS RSAE algorithms: Indicates a signature algorithm using RSASSA-PSS [RFC8017] with mask generation function 1. The digest used in the mask generation function and the digest being signed are both the corresponding hash algorithm as defined in [SHS]. The length of the Salt MUST be equal to the length of the output of the digest algorithm. If the public key is carried in an X.509 certificate, it MUST use the rsaEncryption OID [RFC5280].

EdDSA algorithms: Indicates a signature algorithm using EdDSA as defined in [RFC8032] or its successors. Note that these correspond to the "PureEdDSA" algorithms and not the "prehash" variants.

RSASSA-PSS PSS algorithms: Indicates a signature algorithm using RSASSA-PSS [RFC8017] with mask generation function 1. The digest used in the mask generation function and the digest being signed are both the corresponding hash algorithm as defined in [SHS]. The length of the Salt MUST be equal to the length of the digest algorithm. If the public key is carried in an X.509 certificate, it MUST use the RSASSA-PSS OID [RFC5756]. When used in certificate signatures, the algorithm parameters MUST be DER encoded. If the corresponding public key's parameters are present, then the parameters in the signature MUST be identical to those in the public key.

Legacy algorithms: Indicates algorithms which are being deprecated because they use algorithms with known weaknesses, specifically SHA-1 which is used in this context with either (1) RSA using RSASSA-PKCS1-v1_5 or (2) ECDSA. These values refer solely to signatures which appear in certificates (see Section 4.4.2.2) and are not defined for use in signed TLS handshake messages, although they MAY appear in "signature_algorithms" and "signature_algorithms_cert" for backward compatibility with TLS 1.2. Endpoints SHOULD NOT negotiate these algorithms but are permitted to do so solely for backward compatibility. Clients offering these values MUST list them as the lowest priority (listed after all other algorithms in SignatureSchemeList). TLS 1.3 servers MUST NOT offer a SHA-1 signed certificate unless no valid certificate chain can be produced without it (see Section 4.4.2.2).

The signatures on certificates that are self-signed or certificates that are trust anchors are not validated, since they begin a certification path (see [RFC5280], Section 3.2). A certificate that begins a certification path MAY use a signature algorithm that is not advertised as being supported in the "signature_algorithms" 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 (CAs) which an endpoint supports and which SHOULD be used by the receiving endpoint to guide certificate selection.

The body of the "certificate_authorities" extension consists of a CertificateAuthoritiesExtension structure.

```
opaque DistinguishedName<1..2^16-1>;

struct {
    DistinguishedName authorities<3..2^16-1>;
} CertificateAuthoritiesExtension;
```

authorities: A list of the distinguished names [X501] of acceptable certificate authorities, represented in DER-encoded [X690] format. These distinguished names specify a desired distinguished name for a trust anchor or subordinate CA; thus, this message can be used to describe known trust anchors as well as a desired authorization space.

The client MAY send the "certificate_authorities" extension in the ClientHello message. The server MAY send it in the CertificateRequest message.

The "trusted_ca_keys" extension [RFC6066], which serves a similar purpose, but is more complicated, is not used in TLS 1.3 (although it may appear in ClientHello messages from clients which are offering prior versions of TLS).

4.2.5. OID Filters

The "oid_filters" extension allows servers to provide a list of OID/value pairs which it would like the client's certificate to match. This extension, if provided by the server, MUST only be sent in the CertificateRequest message.

```
struct {
    opaque certificate_extension_oid<1..2^8-1>;
    opaque certificate_extension_values<0..2^16-1>;
} OIDFilter;

struct {
    OIDFilter filters<0..2^16-1>;
} OIDFilterExtension;
```

filters: A list of certificate extension OIDs [RFC5280] with their allowed value(s) and represented in DER-encoded [X690] format. Some certificate extension OIDs allow multiple values (e.g., Extended Key Usage). If the server has included a non-empty filters list, the client certificate included in the response MUST contain all of the specified extension OIDs that the client recognizes. For each extension OID recognized by the client, all of the specified values MUST be present in the client certificate (but the certificate MAY have other values as well). However, the client MUST ignore and skip any unrecognized certificate extension OIDs. If the client ignored some of the required certificate extension OIDs and supplied a certificate that does not satisfy the request, the server MAY at its discretion either continue the connection without client authentication or abort the handshake with an "unsupported_certificate" alert. Any given OID MUST NOT appear more than once in the filters list.

PKIX RFCs define a variety of certificate extension OIDs and their corresponding value types. Depending on the type, matching certificate extension values are not necessarily bitwise-equal. It is expected that TLS implementations will rely on their PKI libraries to perform certificate selection using certificate extension OIDs.

This document defines matching rules for two standard certificate extensions defined in [RFC5280]:

* The Key Usage extension in a certificate matches the request when all key usage bits asserted in the request are also asserted in the Key Usage certificate extension.

* The Extended Key Usage extension in a certificate matches the request when all key purpose OIDs present in the request are also found in the Extended Key Usage certificate extension. The special anyExtendedKeyUsage OID MUST NOT be used in the request.

Separate specifications may define matching rules for other certificate extensions.

4.2.6. Post-Handshake Certificate-Based Client Authentication

The "post_handshake_auth" extension is used to indicate that a client is willing to perform post-handshake authentication (Section 4.6.2). Servers MUST NOT send a post-handshake CertificateRequest to clients which do not offer this extension. Servers MUST NOT send this extension.

```
struct {} PostHandshakeAuth;
```

The "extension_data" field of the "post_handshake_auth" extension is zero length.

4.2.7. Supported Groups

When sent by the client, the "supported_groups" extension indicates the named groups which the client supports for key exchange, ordered from most preferred to least preferred.

Note: In versions of TLS prior to TLS 1.3, this extension was named "elliptic_curves" and only contained elliptic curve groups. See [RFC8422] and [RFC7919]. This extension was also used to negotiate ECDSA curves. Signature algorithms are now negotiated independently (see Section 4.2.3).

The "extension_data" field of this extension contains a "NamedGroupList" value:

```
enum {

    /* Elliptic Curve Groups (ECDHE) */
    secp256r1(0x0017), secp384r1(0x0018), secp521r1(0x0019),
    x25519(0x001D), x448(0x001E),

    /* Finite Field Groups (DHE) */
    ffdhe2048(0x0100), ffdhe3072(0x0101), ffdhe4096(0x0102),
    ffdhe6144(0x0103), ffdhe8192(0x0104),

    /* Reserved Code Points */
    ffdhe_private_use(0x01FC..0x01FF),
    ecdhe_private_use(0xFE00..0xFEFF),
    (0xFFFF)
} NamedGroup;

struct {
    NamedGroup named_group_list<2..2^16-1>;
} NamedGroupList;
```

Elliptic Curve Groups (ECDHE): Indicates support for the corresponding named curve, defined in either FIPS 186-4 [DSS] or in [RFC7748]. Values 0xFE00 through 0xFEFF are reserved for Private Use [RFC8126].

Finite Field Groups (DHE): Indicates support for the corresponding finite field group, defined in [RFC7919]. Values 0x01FC through 0x01FF are reserved for Private Use.

Items in "named_group_list" are ordered according to the sender's preferences (most preferred choice first).

As of TLS 1.3, servers are permitted to send the "supported_groups" extension to the client. Clients MUST NOT act upon any information found in "supported_groups" prior to successful completion of the handshake but MAY use the information learned from a successfully completed handshake to change what groups they use in their "key_share" extension in subsequent connections. If the server has a group it prefers to the ones in the "key_share" extension but is still willing to accept the ClientHello, it SHOULD send "supported_groups" to update the client's view of its preferences; this extension SHOULD contain all groups the server supports, regardless of whether they are currently supported by the client.

4.2.8. Key Share

The "key_share" extension contains the endpoint's cryptographic parameters.

Clients MAY send an empty client_shares list in order to request group selection from the server, at the cost of an additional round trip (see Section 4.1.4).

```
struct {
    NamedGroup group;
    opaque key_exchange<1..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 [DH76] parameters are described in Section 4.2.8.1; Elliptic Curve Diffie-Hellman parameters are described in Section 4.2.8.2.

In the ClientHello message, the "extension_data" field of this extension contains a "KeyShareClientHello" value:

```
struct {
    KeyShareEntry client_shares<0..2^16-1>;
} KeyShareClientHello;
```

client_shares: A list of offered KeyShareEntry values in descending order of client preference.

This list MAY be empty if the client is requesting a HelloRetryRequest. Each KeyShareEntry value MUST correspond to a group offered in the "supported_groups" extension and MUST appear in the same order. However, the values MAY be a non-contiguous subset of the "supported_groups" extension and MAY omit the most preferred groups. Such a situation could arise if the most preferred groups are new and unlikely to be supported in enough places to make pregenerating key shares for them efficient.

Clients can offer as many KeyShareEntry values as the number of supported groups it is offering, each representing a single set of key exchange parameters. For instance, a client might offer shares for several elliptic curves or multiple FFDHE groups. The key_exchange values for each KeyShareEntry MUST be generated independently. Clients MUST NOT offer multiple KeyShareEntry values for the same group. Clients MUST NOT offer any KeyShareEntry values for groups not listed in the client's "supported_groups" extension. Servers MAY check for violations of these rules and abort the handshake with an "illegal_parameter" alert if one is violated.

In a HelloRetryRequest message, the "extension_data" field of this extension contains a KeyShareHelloRetryRequest value:

```
struct {
    NamedGroup selected_group;
} KeyShareHelloRetryRequest;
```

selected_group: The mutually supported group the server intends to negotiate and is requesting a retried ClientHello/KeyShare for.

Upon receipt of this extension in a HelloRetryRequest, the client MUST verify that (1) the selected_group field corresponds to a group which was provided in the "supported_groups" extension in the original ClientHello and (2) the selected_group field does not correspond to a group which was provided in the "key_share" extension in the original ClientHello. If either of these checks fails, then the client MUST abort the handshake with an "illegal_parameter" alert. Otherwise, when sending the new ClientHello, the client MUST replace the original "key_share" extension with one containing only a new KeyShareEntry for the group indicated in the selected_group field of the triggering HelloRetryRequest.

In a ServerHello message, the "extension_data" field of this extension contains a KeyShareServerHello value:

```
struct {
    KeyShareEntry server_share;
} KeyShareServerHello;
```

server_share: A single KeyShareEntry value that is in the same group as one of the client's shares.

If using (EC)DHE key establishment, servers offer exactly one KeyShareEntry in the ServerHello. This value MUST be in the same group as the KeyShareEntry value offered by the client that the server has selected for the negotiated key exchange. Servers MUST NOT send a KeyShareEntry for any group not indicated in the client's "supported_groups" extension and MUST NOT send a KeyShareEntry when using the "psk_ke" PskKeyExchangeMode. If using (EC)DHE key establishment and a HelloRetryRequest containing a "key_share" extension was received by the client, the client MUST verify that the selected NamedGroup in the ServerHello is the same as that in the HelloRetryRequest. If this check fails, the client MUST abort the handshake with an "illegal_parameter" alert.

4.2.8.1. Diffie-Hellman Parameters

Diffie-Hellman [DH76] parameters for both clients and servers are encoded in the opaque key_exchange field of a KeyShareEntry in a KeyShare structure. The opaque value contains the Diffie-Hellman public value (Y = g^X 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. For P-521, they take 66 octets each.

For the curves secp256r1, secp384r1, and secp521r1, peers MUST validate each other's public value Q by ensuring that the point is a valid point on the elliptic curve. The appropriate validation procedures are defined in Section 4.3.7 of [ECDSA] and alternatively in Section 5.6.2.3 of [KEYAGREEMENT]. This process consists of three steps: (1) verify that Q is not the point at infinity (O), (2) verify that for Q = (x, y) both integers x and y are in the correct interval, and (3) ensure that (x, y) is a correct solution to the elliptic curve equation. For these curves, implementors do not need to verify membership in the correct subgroup.

For X25519 and X448, the contents of the public value is the K_A or K_B value described in Section 6 of [RFC7748]. This is 32 bytes for X25519 and 56 bytes for X448.

Note: Versions of TLS prior to 1.3 permitted point format negotiation; TLS 1.3 removes this feature in favor of a single point format for each curve.

4.2.9. Pre-Shared Key Exchange Modes

In order to use PSKs, clients MUST also send a "psk_key_exchange_modes" extension. The semantics of this extension are that the client only supports the use of PSKs with these modes, which restricts both the use of PSKs offered in this ClientHello and those which the server might supply via NewSessionTicket.

A client MUST provide a "psk_key_exchange_modes" extension if it offers a "pre_shared_key" extension. If clients offer "pre_shared_key" without a "psk_key_exchange_modes" extension, servers MUST abort the handshake. Servers MUST NOT select a key exchange mode that is not listed by the client. This extension also restricts the modes for use with PSK resumption. Servers SHOULD NOT send NewSessionTicket with tickets that are not compatible with the advertised modes; however, if a server does so, the impact will just be that the client's attempts at resumption fail.

The server MUST NOT send a "psk_key_exchange_modes" extension.

```
enum { psk_ke(0), psk_dhe_ke(1), (255) } PskKeyExchangeMode;

struct {
    PskKeyExchangeMode ke_modes<1..255>;
} PskKeyExchangeModes;
```

psk_ke: PSK-only key establishment. In this mode, the server MUST NOT supply a "key_share" value.

psk_dhe_ke: PSK with (EC)DHE key establishment. In this mode, the client and server MUST supply "key_share" values as described in Section 4.2.8.

Any future values that are allocated must ensure that the transmitted protocol messages unambiguously identify which mode was selected by the server; at present, this is indicated by the presence of the "key_share" in the ServerHello.

4.2.10. Early Data Indication

When a PSK is used and early data is allowed for that PSK (see for instance Appendix B.3.4), the client can send Application Data in its first flight of messages. If the client opts to do so, it MUST supply both the "pre_shared_key" and "early_data" extensions.

The "extension_data" field of this extension contains an "EarlyDataIndication" value.

```
struct {} Empty;

struct {
    select (Handshake.msg_type) {
        case new_session_ticket:   uint32 max_early_data_size;
        case client_hello:         Empty;
        case encrypted_extensions: Empty;
    };
} EarlyDataIndication;
```

See Section 4.6.1 for details regarding the use of the max_early_data_size field.

The parameters for the 0-RTT data (version, symmetric cipher suite, Application-Layer Protocol Negotiation (ALPN) [RFC7301] protocol, etc.) are those associated with the PSK in use. For externally provisioned PSKs, the associated values are those provisioned along with the key. For PSKs established via a NewSessionTicket message, the associated values are those which were negotiated in the connection which established the PSK. The PSK used to encrypt the early data MUST be the first PSK listed in the client's "pre_shared_key" extension.

For PSKs provisioned via NewSessionTicket, a server MUST validate that the ticket age for the selected PSK identity (computed by subtracting ticket_age_add from PskIdentity.obfuscated_ticket_age modulo 2^32) is within a small tolerance of the time since the ticket was issued (see Section 8). If it is not, the server SHOULD proceed with the handshake but reject 0-RTT, and SHOULD NOT take any other action that assumes that this ClientHello is fresh.

0-RTT messages sent in the first flight have the same (encrypted) content types as messages of the same type sent in other flights (handshake and application_data) but are protected under different keys. After receiving the server's Finished message, if the server has accepted early data, an EndOfEarlyData message will be sent to indicate the key change. This message will be encrypted with the 0-RTT traffic keys.

A server which receives an "early_data" extension MUST behave in one of three ways:

* Ignore the extension and return a regular 1-RTT response. The server then skips past early data by attempting to deprotect received records using the handshake traffic key, discarding records which fail deprotection (up to the configured max_early_data_size). Once a record is deprotected successfully, it is treated as the start of the client's second flight and the server proceeds as with an ordinary 1-RTT handshake.

* Request that the client send another ClientHello by responding with a HelloRetryRequest. A client MUST NOT include the "early_data" extension in its followup ClientHello. The server then ignores early data by skipping all records with an external content type of "application_data" (indicating that they are encrypted), up to the configured max_early_data_size.

* Return its own "early_data" extension in EncryptedExtensions, indicating that it intends to process the early data. It is not possible for the server to accept only a subset of the early data messages. Even though the server sends a message accepting early data, the actual early data itself may already be in flight by the time the server generates this message.

In order to accept early data, the server MUST have selected the first key offered in the client's "pre_shared_key" extension. In addition, it MUST verify that the following values are the same as those associated with the selected PSK:

* The selected TLS version number

* The selected cipher suite

* The selected ALPN [RFC7301] protocol, if any

These requirements are a superset of those needed to perform a 1-RTT handshake using the PSK in question.

Future extensions MUST define their interaction with 0-RTT.

If any of these checks fail, the server MUST NOT respond with the extension and must discard all the first-flight data using one of the first two mechanisms listed above (thus falling back to 1-RTT or 2-RTT). If the client attempts a 0-RTT handshake but the server rejects it, the server will generally not have the 0-RTT record protection keys and must instead use trial decryption (either with the 1-RTT handshake keys or by looking for a cleartext ClientHello in the case of a HelloRetryRequest) to find the first non-0-RTT message.

If the server chooses to accept the "early_data" extension, then it MUST comply with the same error-handling requirements specified for all records when processing early data records. Specifically, if the server fails to decrypt a 0-RTT record following an accepted "early_data" extension, it MUST terminate the connection with a "bad_record_mac" alert as per Section 5.2.

If the server rejects the "early_data" extension, the client application MAY opt to retransmit the Application Data previously sent in early data once the handshake has been completed. Note that automatic retransmission of early data could result in incorrect assumptions regarding the status of the connection. For instance, when the negotiated connection selects a different ALPN protocol from what was used for the early data, an application might need to construct different messages. Similarly, if early data assumes anything about the connection state, it might be sent in error after the handshake completes.

A TLS implementation SHOULD NOT automatically resend early data; applications are in a better position to decide when retransmission is appropriate. A TLS implementation MUST NOT automatically resend early data unless the negotiated connection selects the same ALPN protocol.

4.2.11. Pre-Shared Key Extension

The "pre_shared_key" extension is used to negotiate the identity of the pre-shared key to be used with a given handshake in association with PSK key establishment.

The "extension_data" field of this extension contains a "PreSharedKeyExtension" value:

```
   struct {
       opaque identity<1..2^16-1>;
       uint32 obfuscated_ticket_age;
   } PskIdentity;

   opaque PskBinderEntry<32..255>;

   struct {
       PskIdentity identities<7..2^16-1>;
       PskBinderEntry binders<33..2^16-1>;
   } OfferedPsks;

   struct {
       select (Handshake.msg_type) {
           case client_hello: OfferedPsks;
           case server_hello: uint16 selected_identity;
       };
   } PreSharedKeyExtension;
```

identity: A label for a key. For instance, a ticket (as defined in Appendix B.3.4) or a label for a pre-shared key established externally.

obfuscated_ticket_age: An obfuscated version of the age of the key. Section 4.2.11.1 describes how to form this value for identities established via the NewSessionTicket message. For identities established externally, an obfuscated_ticket_age of 0 SHOULD be used, and servers MUST ignore the value.

identities: A list of the identities that the client is willing to negotiate with the server. If sent alongside the "early_data" extension (see Section 4.2.10), the first identity is the one used for 0-RTT data.

binders: A series of HMAC values, one for each value in the identities list and in the same order, computed as described below.

selected_identity: The server's chosen identity expressed as a (0-based) index into the identities in the client's "OfferedPsks.identities" list.

Each PSK is associated with a single Hash algorithm. For PSKs established via the ticket mechanism (Section 4.6.1), this is the KDF Hash algorithm on the connection where the ticket was established. For externally established PSKs, the Hash algorithm MUST be set when the PSK is established or default to SHA-256 if no such algorithm is defined. The server MUST ensure that it selects a compatible PSK (if any) and cipher suite.

In TLS versions prior to TLS 1.3, the Server Name Indication (SNI) value was intended to be associated with the session (Section 3 of [RFC6066]), with the server being required to enforce that the SNI value associated with the session matches the one specified in the resumption handshake. However, in reality the implementations were not consistent on which of two supplied SNI values they would use, leading to the consistency requirement being de facto enforced by the clients. In TLS 1.3, the SNI value is always explicitly specified in the resumption handshake, and there is no need for the server to associate an SNI value with the ticket. Clients, however, SHOULD store the SNI with the PSK to fulfill the requirements of Section 4.6.1.

Implementor's note: When session resumption is the primary use case of PSKs, the most straightforward way to implement the PSK/cipher suite matching requirements is to negotiate the cipher suite first and then exclude any incompatible PSKs. Any unknown PSKs (e.g., ones not in the PSK database or encrypted with an unknown key) SHOULD simply be ignored. If no acceptable PSKs are found, the server SHOULD perform a non-PSK handshake if possible. If backward compatibility is important, client-provided, externally established PSKs SHOULD influence cipher suite selection.

Prior to accepting PSK key establishment, the server MUST validate the corresponding binder value (see Section 4.2.11.2 below). If this value is not present or does not validate, the server MUST abort the handshake. Servers SHOULD NOT attempt to validate multiple binders; rather, they SHOULD select a single PSK and validate solely the binder that corresponds to that PSK. See Section 8.2 and Appendix F.6 for the security rationale for this requirement. In order to accept PSK key establishment, the server sends a "pre_shared_key" extension indicating the selected identity.

Clients MUST verify that the server's selected_identity is within the range supplied by the client, that the server selected a cipher suite indicating a Hash associated with the PSK, and that a server "key_share" extension is present if required by the ClientHello "psk_key_exchange_modes" extension. If these values are not consistent, the client MUST abort the handshake with an "illegal_parameter" alert.

If the server supplies an "early_data" extension, the client MUST verify that the server's selected_identity is 0. If any other value is returned, the client MUST abort the handshake with an "illegal_parameter" alert.

The "pre_shared_key" extension MUST be the last extension in the ClientHello (this facilitates implementation as described below). Servers MUST check that it is the last extension and otherwise fail the handshake with an "illegal_parameter" alert.

4.2.11.1. Ticket Age

The client's view of the age of a ticket is the time since the receipt of the NewSessionTicket message. Clients MUST NOT attempt to use tickets which have ages greater than the "ticket_lifetime" value which was provided with the ticket. The "obfuscated_ticket_age" field of each PskIdentity contains an obfuscated version of the ticket age formed by taking the age in milliseconds and adding the "ticket_age_add" value that was included with the ticket (see Section 4.6.1), modulo 2^32. This addition prevents passive observers from correlating connections unless tickets are reused. Note that the "ticket_lifetime" field in the NewSessionTicket message is in seconds but the "obfuscated_ticket_age" is in milliseconds. Because ticket lifetimes are restricted to a week, 32 bits is enough to represent any plausible age, even in milliseconds.

4.2.11.2. PSK Binder

The PSK binder value forms a binding between a PSK and the current handshake, as well as a binding between the handshake in which the PSK was generated (if via a NewSessionTicket message) and the current handshake. Each entry in the binders list is computed as an HMAC over a transcript hash (see Section 4.4.1) containing a partial ClientHello up to and including the PreSharedKeyExtension.identities field. That is, it includes all of the ClientHello but not the binders list itself. The length fields for the message (including the overall length, the length of the extensions block, and the length of the "pre_shared_key" extension) are all set as if binders of the correct lengths were present.

The PskBinderEntry is computed in the same way as the Finished message (Section 4.4.4) but with the BaseKey being the binder_key derived via the key schedule from the corresponding PSK which is being offered (see Section 7.1).

If the handshake includes a HelloRetryRequest, the initial ClientHello and HelloRetryRequest are included in the transcript along with the new ClientHello. For instance, if the client sends ClientHello1, its binder will be computed over:

```
Transcript-Hash(Truncate(ClientHello1))
```

Where Truncate() removes the binders list from the ClientHello.

If the server responds with a HelloRetryRequest and the client then sends ClientHello2, its binder will be computed over:

```
Transcript-Hash(ClientHello1,
                HelloRetryRequest,
                Truncate(ClientHello2))
```

The full ClientHello1/ClientHello2 is included in all other handshake hash computations. Note that in the first flight, Truncate(ClientHello1) is hashed directly, but in the second flight, ClientHello1 is hashed and then reinjected as a "message_hash" message, as described in Section 4.4.1.

4.2.11.3. Processing Order

Clients are permitted to "stream" 0-RTT data until they receive the server's Finished, only then sending the EndOfEarlyData message, followed by the rest of the handshake. In order to avoid deadlocks, when accepting "early_data", servers MUST process the client's ClientHello and then immediately send their flight of messages, rather than waiting for the client's EndOfEarlyData message before sending its ServerHello.

4.3. Server Parameters

The next two messages from the server, EncryptedExtensions and CertificateRequest, contain information from the server that determines the rest of the handshake. These messages are encrypted with keys derived from the server_handshake_traffic_secret.

4.3.1. Encrypted Extensions

In all handshakes, the server MUST send the EncryptedExtensions message immediately after the ServerHello message. This is the first message that is encrypted under keys derived from the server_handshake_traffic_secret.

The EncryptedExtensions message contains extensions that can be protected, i.e., any which are not needed to establish the cryptographic context but which are not associated with individual certificates. The client MUST check EncryptedExtensions for the presence of any forbidden extensions and if any are found MUST abort the handshake with an "illegal_parameter" alert.

Structure of this message:

```
struct {
    Extension extensions<0..2^16-1>;
} EncryptedExtensions;
```

extensions: A list of extensions. For more information, see the table in Section 4.2.

4.3.2. Certificate Request

A server which is authenticating with a certificate MAY optionally request a certificate from the client. This message, if sent, MUST follow EncryptedExtensions.

Structure of this message:

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

certificate_request_context: An opaque string which identifies the certificate request and which will be echoed in the client's Certificate message. The certificate_request_context MUST be unique within the scope of this connection (thus preventing replay of client CertificateVerify messages). This field SHALL be zero length unless used for the post-handshake authentication exchanges described in Section 4.6.2. When requesting post-handshake authentication, the server SHOULD make the context unpredictable to the client (e.g., by randomly generating it) in order to prevent an attacker who has temporary access to the client's private key from pre-computing valid CertificateVerify messages.

extensions: A list of extensions describing the parameters of the certificate being requested. The "signature_algorithms" extension MUST be specified, and other extensions may optionally be included if defined for this message. Clients MUST ignore unrecognized extensions.

In prior versions of TLS, the CertificateRequest message carried a list of signature algorithms and certificate authorities which the server would accept. In TLS 1.3, the former is expressed by sending the "signature_algorithms" and optionally "signature_algorithms_cert" extensions. The latter is expressed by sending the "certificate_authorities" extension (see Section 4.2.4).

Servers which are authenticating with a resumption PSK MUST NOT send the CertificateRequest message in the main handshake, though they MAY send it in post-handshake authentication (see Section 4.6.2) provided that the client has sent the "post_handshake_auth" extension (see Section 4.2.6). Servers which are authenticating with an external PSK MUST NOT send the CertificateRequest message either in the main handshake or request post-handshake authentication. Future specifications MAY provide an extension to permit this.

4.4. Authentication Messages

As discussed in Section 2, TLS generally uses a common set of messages for authentication, key confirmation, and handshake integrity: Certificate, CertificateVerify, and Finished. (The PSK binders also perform key confirmation, in a similar fashion.) These three messages are always sent as the last messages in their handshake flight. The Certificate and CertificateVerify messages are only sent under certain circumstances, as defined below. The

Finished message is always sent as part of the Authentication Block. These messages are encrypted under keys derived from the [sender]_handshake_traffic_secret.

The computations for the Authentication messages all uniformly take the following inputs:

* The certificate and signing key to be used.

* A Handshake Context consisting of the list of messages to be included in the transcript hash.

* A Base Key to be used to compute a MAC key.

Based on these inputs, the messages then contain:

Certificate The certificate to be used for authentication, and any supporting certificates in the chain. Note that certificate-based client authentication is not available in PSK handshake flows (including 0-RTT).

CertificateVerify: A signature over the value Transcript-Hash(Handshake Context, Certificate)

Finished: A MAC over the value Transcript-Hash(Handshake Context, Certificate, CertificateVerify) using a MAC key derived from the Base Key.

The following table defines the Handshake Context and MAC Base Key for each scenario:

Mode	Handshake Context	Base Key
Server	ClientHello ... later of EncryptedExtensions/ CertificateRequest	server_handshake_traffic_secret
Client	ClientHello ... later of server Finished/ EndOfEarlyData	client_handshake_traffic_secret
Post-Handshake	ClientHello ... client Finished + CertificateRequest	client_application_traffic_secret_N

Table 2: Authentication Inputs

4.4.1. The Transcript Hash

Many of the cryptographic computations in TLS make use of a transcript hash. This value is computed by hashing the concatenation of each included handshake message, including the handshake message header carrying the handshake message type and length fields, but not including record layer headers. I.e.,

```
 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 certificate-based client authentication via a CertificateRequest message (Section 4.3.2). If the server requests certificate-based client authentication but no suitable certificate is available, the client MUST send a Certificate message containing no certificates (i.e., with the "certificate_list" field having length 0). A Finished message MUST be sent regardless of whether the Certificate message is empty.

Structure of this message:

```
      enum {
          X509(0),
          RawPublicKey(2),
          (255)
      } CertificateType;

      struct {
          select (certificate_type) {
              case RawPublicKey:
                /* From RFC 7250 ASN.1_subjectPublicKeyInfo */
                opaque ASN1_subjectPublicKeyInfo<1..2^24-1>;

              case X509:
                opaque cert_data<1..2^24-1>;
          };
          Extension extensions<0..2^16-1>;
      } CertificateEntry;

      struct {
          opaque certificate_request_context<0..2^8-1>;
          CertificateEntry certificate_list<0..2^24-1>;
      } Certificate;
```

certificate_request_context: If this message is in response to a CertificateRequest, the value of certificate_request_context in that message. Otherwise (in the case of server authentication), this field SHALL be zero length.

certificate_list: A list (chain) of CertificateEntry structures, each containing a single certificate and list of extensions.

extensions: A list of extension values for the CertificateEntry. The "Extension" format is defined in Section 4.2. Valid extensions for server certificates at present include the OCSP Status extension [RFC6066] and the SignedCertificateTimestamp extension [RFC6962]; future extensions may be defined for this message as well. Extensions in the Certificate message from the server MUST correspond to ones from the ClientHello message. Extensions in the Certificate message from the client MUST correspond to extensions in the CertificateRequest message from the server. If an extension applies to the entire chain, it SHOULD be included in the first CertificateEntry.

If the corresponding certificate type extension ("server_certificate_type" or "client_certificate_type") was not negotiated in EncryptedExtensions, or the X.509 certificate type was negotiated, then each CertificateEntry contains a DER-encoded X.509 certificate. The sender's certificate MUST come in the first

CertificateEntry in the list. Each following certificate SHOULD directly certify the one immediately preceding it. Because certificate validation requires that trust anchors be distributed independently, a certificate that specifies a trust anchor MAY be omitted from the chain, provided that supported peers are known to possess any omitted certificates.

Note: Prior to TLS 1.3, "certificate_list" ordering required each certificate to certify the one immediately preceding it; however, some implementations allowed some flexibility. Servers sometimes send both a current and deprecated intermediate for transitional purposes, and others are simply configured incorrectly, but these cases can nonetheless be validated properly. For maximum compatibility, all implementations SHOULD be prepared to handle potentially extraneous certificates and arbitrary orderings from any TLS version, with the exception of the end-entity certificate which MUST be first.

If the RawPublicKey certificate type was negotiated, then the certificate_list MUST contain no more than one CertificateEntry, which contains an ASN1_subjectPublicKeyInfo value as defined in [RFC7250], Section 3.

The OpenPGP certificate type [RFC6091] MUST NOT be used with TLS 1.3.

The server's certificate_list MUST always be non-empty. A client will send an empty certificate_list if it does not have an appropriate certificate to send in response to the server's authentication request.

4.4.2.1. OCSP Status and SCT Extensions

[RFC6066] and [RFC6961] provide extensions to negotiate the server sending OCSP responses to the client. In TLS 1.2 and below, the server replies with an empty extension to indicate negotiation of this extension and the OCSP information is carried in a CertificateStatus message. In TLS 1.3, the server's OCSP information is carried in an extension in the CertificateEntry containing the associated certificate. Specifically, the body of the "status_request" extension from the server MUST be a CertificateStatus structure as defined in [RFC6066], which is interpreted as defined in [RFC6960].

Note: The status_request_v2 extension [RFC6961] is deprecated. TLS 1.3 servers MUST NOT act upon its presence or information in it when processing ClientHello messages; in particular, they MUST NOT send the status_request_v2 extension in the EncryptedExtensions, CertificateRequest, or Certificate messages. TLS 1.3 servers MUST be able to process ClientHello messages that include it, as it MAY be sent by clients that wish to use it in earlier protocol versions.

A server MAY request that a client present an OCSP response with its certificate by sending an empty "status_request" extension in its CertificateRequest message. If the client opts to send an OCSP response, the body of its "status_request" extension MUST be a CertificateStatus structure as defined in [RFC6066].

Similarly, [RFC6962] provides a mechanism for a server to send a Signed Certificate Timestamp (SCT) as an extension in the ServerHello in TLS 1.2 and below. In TLS 1.3, the server's SCT information is carried in an extension in the CertificateEntry.

4.4.2.2. Server Certificate Selection

The following rules apply to the certificates sent by the server:

* The certificate type MUST be X.509v3 [RFC5280], unless explicitly negotiated otherwise (e.g., [RFC7250]).

* The 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 the server is identified by name.

All certificates provided by the server MUST be signed by a signature algorithm advertised by the client, if it is able to provide such a chain (see Section 4.2.3). Certificates that are self-signed or certificates that are expected to be trust anchors are not validated as part of the chain and therefore MAY be signed with any algorithm.

If the server cannot produce a certificate chain that is signed only via the indicated supported algorithms, then it SHOULD continue the handshake by sending the client a certificate chain of its choice that may include algorithms that are not known to be supported by the client. This fallback chain SHOULD NOT use the deprecated SHA-1 hash algorithm in general, but MAY do so if the client's advertisement permits it, and MUST NOT do so otherwise.

If the client cannot construct an acceptable chain using the provided certificates and decides to abort the handshake, then it MUST abort the handshake with an appropriate certificate-related alert (by default, "unsupported_certificate"; see Section 6.2 for more information).

If the server has multiple certificates, it chooses one of them based on the above-mentioned criteria (in addition to other criteria, such as transport-layer endpoint, local configuration, and preferences).

4.4.2.3. Client Certificate Selection

The following rules apply to certificates sent by the client:

* The certificate type MUST be X.509v3 [RFC5280], unless explicitly negotiated otherwise (e.g., [RFC7250]).

* If the "certificate_authorities" extension in the CertificateRequest message was present, at least one of the certificates in the certificate chain SHOULD be issued by one of the listed CAs.

* The certificates MUST be signed using an acceptable signature algorithm, as described in Section 4.3.2. Note that this relaxes the constraints on certificate-signing algorithms found in prior versions of TLS.

* If the CertificateRequest message contained a non-empty "oid_filters" extension, the end-entity certificate MUST match the extension OIDs that are recognized by the client, as described in Section 4.2.5.

4.4.2.4. Receiving a Certificate Message

In general, detailed certificate validation procedures are out of scope for TLS (see [RFC5280]). This section provides TLS-specific requirements.

If the server supplies an empty Certificate message, the client MUST abort the handshake with a "decode_error" alert.

If the client does not send any certificates (i.e., it sends an empty Certificate message), the server MAY at its discretion either continue the handshake without client authentication, or abort the handshake with a "certificate_required" alert. Also, if some aspect of the certificate chain was unacceptable (e.g., it was not signed by a known, trusted CA), the server MAY at its discretion either continue the handshake (considering the client unauthenticated) or abort the handshake.

Any endpoint receiving any certificate which it would need to validate using any signature algorithm using an MD5 hash MUST abort the handshake with a "bad_certificate" alert. SHA-1 is deprecated and it is RECOMMENDED that any endpoint receiving any certificate which it would need to validate using any signature algorithm using a SHA-1 hash abort the handshake with a "bad_certificate" alert. For clarity, this means that endpoints can accept these algorithms for certificates that are self-signed or are trust anchors.

All endpoints are RECOMMENDED to transition to SHA-256 or better as soon as possible to maintain interoperability with implementations currently in the process of phasing out SHA-1 support.

Note that a certificate containing a key for one signature algorithm MAY be signed using a different signature algorithm (for instance, an RSA key signed with an ECDSA key).

4.4.3. Certificate Verify

This message is used to provide explicit proof that an endpoint possesses the private key corresponding to its certificate. The CertificateVerify message also provides integrity for the handshake up to this point. Servers MUST send this message when authenticating via a certificate. Clients MUST send this message whenever authenticating via a certificate (i.e., when the Certificate message is non-empty). When sent, this message MUST appear immediately after the Certificate message and immediately prior to the Finished message.

Structure of this message:

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

The algorithm field specifies the signature algorithm used (see Section 4.2.3 for the definition of this type). The signature is a digital signature using that algorithm. The content that is covered under the signature is the hash output as described in Section 4.4.1, namely:

```
Transcript-Hash(Handshake Context, Certificate)
```

The digital signature is then computed over the concatenation of:

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

* The context string (defined below)

* A single 0 byte which serves as the separator

* The content to be signed

This structure is intended to prevent an attack on previous versions of TLS in which the ServerKeyExchange format meant that attackers could obtain a signature of a message with a chosen 32-byte prefix (ClientHello.random). The initial 64-byte pad clears that prefix along with the server-controlled ServerHello.random.

The context string for a server signature is "TLS 1.3, server CertificateVerify" The context string for a client signature is "TLS 1.3, client CertificateVerify" It is used to provide separation between signatures made in different contexts, helping against potential cross-protocol attacks.

For example, if the transcript hash was 32 bytes of 01 (this length would make sense for SHA-256), the content covered by the digital signature for a server CertificateVerify would be:

```
2020202020202020202020202020202020202020202020202020202020202020
2020202020202020202020202020202020202020202020202020202020202020
544c5320312e332c207365727665722043657274696669636174655665726966
79
00
0101010101010101010101010101010101010101010101010101010101010101
```

On the sender side, the process for computing the signature field of the CertificateVerify message takes as input:

* The content covered by the digital signature

* The private signing key corresponding to the certificate sent in the previous message

If the CertificateVerify message is sent by a server, the signature algorithm MUST be one offered in the client's "signature_algorithms" extension unless no valid certificate chain can be produced without unsupported algorithms (see Section 4.2.3).

If sent by a client, the signature algorithm used in the signature MUST be one of those present in the supported_signature_algorithms field of the "signature_algorithms" extension in the CertificateRequest message.

In addition, the signature algorithm MUST be compatible with the key in the sender's end-entity certificate. RSA signatures MUST use an RSASSA-PSS algorithm, regardless of whether RSASSA-PKCS1-v1_5 algorithms appear in "signature_algorithms". The SHA-1 algorithm MUST NOT be used in any signatures of CertificateVerify messages. All SHA-1 signature algorithms in this specification are defined solely for use in legacy certificates and are not valid for CertificateVerify signatures.

The receiver of a CertificateVerify message MUST verify the signature field. The verification process takes as input:

* The content covered by the digital signature

* The public key contained in the end-entity certificate found in the associated Certificate message

* The digital signature received in the signature field of the CertificateVerify message

If the verification fails, the receiver MUST terminate the handshake with a "decrypt_error" alert.

4.4.4. Finished

The Finished message is the final message in the Authentication Block. It is essential for providing authentication of the handshake and of the computed keys.

Recipients of Finished messages MUST verify that the contents are correct and if incorrect MUST terminate the connection with a "decrypt_error" alert.

Once a side has sent its Finished message and has received and validated the Finished message from its peer, it may begin to send and receive Application Data over the connection. There are two settings in which it is permitted to send data prior to receiving the peer's Finished:

1. Clients sending 0-RTT data as described in Section 4.2.10.

2. Servers MAY send data after sending their first flight, but because the handshake is not yet complete, they have no assurance of either the peer's identity or its liveness (i.e., the ClientHello might have been replayed).

The key used to compute the Finished message is computed from the Base Key defined in Section 4.4 using HKDF (see Section 7.1). Specifically:

```
finished_key =
    HKDF-Expand-Label(BaseKey, "finished", "", Hash.length)
```

Structure of this message:

```
struct {
    opaque verify_data[Hash.length];
} Finished;
```

The verify_data value is computed as follows:

```
verify_data =
    HMAC(finished_key,
         Transcript-Hash(Handshake Context,
                         Certificate*, CertificateVerify*))

* Only included if present.
```

HMAC [RFC2104] uses the Hash algorithm for the handshake. As noted above, the HMAC input can generally be implemented by a running hash, i.e., just the handshake hash at this point.

In previous versions of TLS, the verify_data was always 12 octets long. In TLS 1.3, it is the size of the HMAC output for the Hash used for the handshake.

Note: Alerts and any other non-handshake record types are not handshake messages and are not included in the hash computations.

Any records following a Finished message MUST be encrypted under the appropriate application traffic key as described in Section 7.2. In particular, this includes any alerts sent by the server in response to client Certificate and CertificateVerify messages.

4.5. End of Early Data

```
struct {} EndOfEarlyData;
```

If the server sent an "early_data" extension in EncryptedExtensions, the client MUST send an EndOfEarlyData message after receiving the server Finished. If the server does not send an "early_data" extension in EncryptedExtensions, then the client MUST NOT send an EndOfEarlyData message. This message indicates that all 0-RTT application_data messages, if any, have been transmitted and that the following records are protected under handshake traffic keys. Servers MUST NOT send this message, and clients receiving it MUST terminate the connection with an "unexpected_message" alert. This message is encrypted under keys derived from the client_early_traffic_secret.

4.6. Post-Handshake Messages

TLS also allows other messages to be sent after the main handshake. These messages use a handshake content type and are encrypted under the appropriate application traffic key.

4.6.1. New Session Ticket Message

At any time after the server has received the client Finished message, it MAY send a NewSessionTicket message. This message creates a unique association between the ticket value and a secret PSK derived from the resumption secret (see Section 7).

The client MAY use this PSK for future handshakes by including the ticket value in the "pre_shared_key" extension in its ClientHello (Section 4.2.11). Servers MAY send multiple tickets on a single connection, either immediately after each other or after specific events (see Appendix C.4). For instance, the server might send a new ticket after post-handshake authentication in order to encapsulate the additional client authentication state. Multiple tickets are useful for clients for a variety of purposes, including:

* Opening multiple parallel HTTP connections.

* Performing connection racing across interfaces and address families via (for example) Happy Eyeballs [RFC8305] or related techniques.

Any ticket MUST only be resumed with a cipher suite that has the same KDF hash algorithm as that used to establish the original connection.

Clients MUST only resume if the new SNI value is valid for the server certificate presented in the original session, and SHOULD only resume if the SNI value matches the one used in the original session. The latter is a performance optimization: normally, there is no reason to expect that different servers covered by a single certificate would be able to accept each other's tickets; hence, attempting resumption in that case would waste a single-use ticket. If such an indication is provided (externally or by any other means), clients MAY resume with a different SNI value.

On resumption, if reporting an SNI value to the calling application, implementations MUST use the value sent in the resumption ClientHello rather than the value sent in the previous session. Note that if a server implementation declines all PSK identities with different SNI values, these two values are always the same.

Note: Although the resumption ecret depends on the client's second flight, a server which does not request certificate-based client authentication MAY compute the remainder of the transcript independently and then send a NewSessionTicket immediately upon sending its Finished rather than waiting for the client Finished. This might be appropriate in cases where the client is expected to open multiple TLS connections in parallel and would benefit from the reduced overhead of a resumption handshake, for example.

```
struct {
    uint32 ticket_lifetime;
    uint32 ticket_age_add;
    opaque ticket_nonce<0..255>;
    opaque ticket<1..2^16-1>;
    Extension extensions<0..2^16-2>;
} 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 use tickets for longer than 7 days after issuance, regardless of the ticket_lifetime, and MAY delete tickets earlier based on local policy. A server MAY treat a ticket as valid for a shorter period of time than what is stated in the ticket_lifetime.

ticket_age_add: A securely generated, random 32-bit value that is used to obscure the age of the ticket that the client includes in the "pre_shared_key" extension. The client-side ticket age is added to this value modulo 2^32 to obtain the value that is transmitted by the client. The server MUST generate a fresh value for each ticket it sends.

ticket_nonce: A per-ticket value that is unique across all tickets issued on this connection.

ticket: The value of the ticket to be used as the PSK identity. The ticket itself is an opaque label. It MAY be either a database lookup key or a self-encrypted and self-authenticated value.

extensions: A list of extension values for the ticket. The "Extension" format is defined in Section 4.2. Clients MUST ignore unrecognized extensions.

The sole extension currently defined for NewSessionTicket is "early_data", indicating that the ticket may be used to send 0-RTT data (Section 4.2.10). It contains the following value:

max_early_data_size: The maximum amount of 0-RTT data that the client is allowed to send when using this ticket, in bytes. Only Application Data payload (i.e., plaintext but not padding or the inner content type byte) is counted. A server receiving more than max_early_data_size bytes of 0-RTT data SHOULD terminate the connection with an "unexpected_message" alert. Note that servers that reject early data due to lack of cryptographic material will be unable to differentiate padding from content, so clients SHOULD NOT depend on being able to send large quantities of padding in early data records.

The PSK associated with the ticket is computed as:

```
HKDF-Expand-Label(resumption_secret,
                  "resumption", ticket_nonce, Hash.length)
```

Because the ticket_nonce value is distinct for each NewSessionTicket message, a different PSK will be derived for each ticket.

Note that in principle it is possible to continue issuing new tickets which indefinitely extend the lifetime of the keying material originally derived from an initial non-PSK handshake (which was most likely tied to the peer's certificate). It is RECOMMENDED that implementations place limits on the total lifetime of such keying material; these limits should take into account the lifetime of the peer's certificate, the likelihood of intervening revocation, and the time since the peer's online CertificateVerify signature.

4.6.2. Post-Handshake Authentication

When the client has sent the "post_handshake_auth" extension (see Section 4.2.6), a server MAY request certificate-based client authentication at any time after the handshake has completed by sending a CertificateRequest message. The client MUST respond with the appropriate Authentication messages (see Section 4.4). If the client chooses to authenticate, it MUST send Certificate, CertificateVerify, and Finished. If it declines, it MUST send a Certificate message containing no certificates followed by Finished. All of the client's messages for a given response MUST appear consecutively on the wire with no intervening messages of other types.

A client that receives a CertificateRequest message without having sent the "post_handshake_auth" extension MUST send an "unexpected_message" fatal alert.

Note: Because certificate-based client authentication could involve prompting the user, servers MUST be prepared for some delay, including receiving an arbitrary number of other messages between sending the CertificateRequest and receiving a response. In addition, clients which receive multiple CertificateRequests in close succession MAY respond to them in a different order than they were received (the certificate_request_context value allows the server to disambiguate the responses).

4.6.3. Key and Initialization Vector Update

The KeyUpdate handshake message is used to indicate that the sender is updating its sending cryptographic keys. This message can be sent by either peer after it has sent a Finished message. Implementations that receive a KeyUpdate message prior to receiving a Finished message MUST terminate the connection with an "unexpected_message" alert. After sending a KeyUpdate message, the sender SHALL send all its traffic using the next generation of keys, computed as described in Section 7.2. Upon receiving a KeyUpdate, the receiver MUST update its receiving keys.

```
enum {
    update_not_requested(0), update_requested(1), (255)
} KeyUpdateRequest;

struct {
    KeyUpdateRequest request_update;
} KeyUpdate;
```

request_update: Indicates whether the recipient of the KeyUpdate should respond with its own KeyUpdate. If an implementation receives any other value, it MUST terminate the connection with an "illegal_parameter" alert.

If the request_update field is set to "update_requested", then the receiver MUST send a KeyUpdate of its own with request_update set to "update_not_requested" prior to sending its next Application Data record. This mechanism allows either side to force an update to the entire connection, but causes an implementation which receives multiple KeyUpdates while it is silent to respond with a single update. Note that implementations may receive an arbitrary number of messages between sending a KeyUpdate with request_update set to "update_requested" and receiving the peer's KeyUpdate, because those messages may already be in flight. However, because send and receive keys are derived from independent traffic secrets, retaining the receive traffic secret does not threaten the forward secrecy of data sent before the sender changed keys.

If implementations independently send their own KeyUpdates with request_update set to "update_requested", and they cross in flight, then each side will also send a response, with the result that each side increments by two generations.

Both sender and receiver MUST encrypt their KeyUpdate messages with the old keys. Additionally, both sides MUST enforce that a KeyUpdate with the old key is received before accepting any messages encrypted with the new key. Failure to do so may allow message truncation attacks.

5. Record Protocol

The TLS record protocol takes messages to be transmitted, fragments the data into manageable blocks, protects the records, and transmits the result. Received data is verified, decrypted, reassembled, and then delivered to higher-level clients.

TLS records are typed, which allows multiple higher-level protocols to be multiplexed over the same record layer. This document specifies four content types: handshake, application_data, alert, and change_cipher_spec. The change_cipher_spec record is used only for compatibility purposes (see Appendix E.4).

An implementation may receive an unencrypted record of type change_cipher_spec consisting of the single byte value 0x01 at any time after the first ClientHello message has been sent or received and before the peer's Finished message has been received and MUST simply drop it without further processing. Note that this record may appear at a point at the handshake where the implementation is expecting protected records, and so it is necessary to detect this condition prior to attempting to deprotect the record. An implementation which receives any other change_cipher_spec value or which receives a protected change_cipher_spec record MUST abort the handshake with an "unexpected_message" alert. If an implementation detects a change_cipher_spec record received before the first ClientHello message or after the peer's Finished message, it MUST be treated as an unexpected record type (though stateless servers may not be able to distinguish these cases from allowed cases).

Implementations MUST NOT send record types not defined in this document unless negotiated by some extension. If a TLS implementation receives an unexpected record type, it MUST terminate the connection with an "unexpected_message" alert. New record content type values are assigned by IANA in the TLS ContentType registry as described in Section 11.

5.1. Record Layer

The record layer fragments information blocks into TLSPlaintext records carrying data in chunks of 2^14 bytes or less. Message boundaries are handled differently depending on the underlying ContentType. Any future content types MUST specify appropriate rules. Note that these rules are stricter than what was enforced in TLS 1.2.

Handshake messages MAY be coalesced into a single TLSPlaintext record or fragmented across several records, provided that:

* Handshake messages MUST NOT be interleaved with other record types. That is, if a handshake message is split over two or more records, there MUST NOT be any other records between them.

* Handshake messages MUST NOT span key changes. Implementations MUST verify that all messages immediately preceding a key change align with a record boundary; if not, then they MUST terminate the connection with an "unexpected_message" alert. Because the ClientHello, EndOfEarlyData, ServerHello, Finished, and KeyUpdate messages can immediately precede a key change, implementations MUST send these messages in alignment with a record boundary.

Implementations MUST NOT send zero-length fragments of Handshake types, even if those fragments contain padding.

Alert messages (Section 6) MUST NOT be fragmented across records, and multiple alert messages MUST NOT be coalesced into a single TLSPlaintext record. In other words, a record with an Alert type MUST contain exactly one message.

Application Data messages contain data that is opaque to TLS. Application Data messages are always protected. Zero-length fragments of Application Data MAY be sent, as they are potentially useful as a traffic analysis countermeasure. Application Data fragments MAY be split across multiple records or coalesced into a single record.

```
   enum {
       invalid(0),
       change_cipher_spec(20),
       alert(21),
       handshake(22),
       application_data(23),
       (255)
   } ContentType;

   struct {
       ContentType type;
       ProtocolVersion legacy_record_version;
       uint16 length;
       opaque fragment[TLSPlaintext.length];
   } TLSPlaintext;
```

type: The higher-level protocol used to process the enclosed fragment.

legacy_record_version: MUST be set to 0x0303 for all records generated by a TLS 1.3 implementation other than an initial ClientHello (i.e., one not generated after a HelloRetryRequest), where it MAY also be 0x0301 for compatibility purposes. This field is deprecated and MUST be ignored for all purposes. Previous versions of TLS would use other values in this field under some circumstances.

length: The length (in bytes) of the following TLSPlaintext.fragment. The length MUST NOT exceed 2^14 bytes. An endpoint that receives a record that exceeds this length MUST terminate the connection with a "record_overflow" alert.

fragment The data being transmitted. This value is transparent and is treated as an independent block to be dealt with by the higher-level protocol specified by the type field.

This document describes TLS 1.3, which uses the version 0x0304. This version value is historical, deriving from the use of 0x0301 for TLS 1.0 and 0x0300 for SSL 3.0. In order to maximize backward compatibility, a record containing an initial ClientHello SHOULD have version 0x0301 (reflecting TLS 1.0) and a record containing a second

ClientHello or a ServerHello MUST have version 0x0303 (reflecting TLS 1.2). When negotiating prior versions of TLS, endpoints follow the procedure and requirements provided in Appendix E.

When record protection has not yet been engaged, TLSPlaintext structures are written directly onto the wire. Once record protection has started, TLSPlaintext records are protected and sent as described in the following section. Note that Application Data records MUST NOT be written to the wire unprotected (see Section 2 for details).

5.2. Record Payload Protection

The record protection functions translate a TLSPlaintext structure into a TLSCiphertext structure. The deprotection functions reverse the process. In TLS 1.3, as opposed to previous versions of TLS, all ciphers are modeled as "Authenticated Encryption with Associated Data" (AEAD) [RFC5116]. AEAD functions provide a unified encryption and authentication operation which turns plaintext into authenticated ciphertext and back again. Each encrypted record consists of a plaintext header followed by an encrypted body, which itself contains a type and optional padding.

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

struct {
    ContentType opaque_type = application_data; /* 23 */
    ProtocolVersion legacy_record_version = 0x0303; /* TLS v1.2 */
    uint16 length;
    opaque encrypted_record[TLSCiphertext.length];
} TLSCiphertext;
```

content: The TLSPlaintext.fragment value, containing the byte encoding of a handshake or an alert message, or the raw bytes of the application’s data to send.

type: The 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)
```

The encrypted_record field of TLSCiphertext is set to AEADEncrypted.

In order to decrypt and verify, the cipher takes as input the key, nonce, additional data, and the AEADEncrypted value. The output is either the plaintext or an error indicating that the decryption failed. There is no separate integrity check. Symbolically,

```
plaintext of encrypted_record =
    AEAD-Decrypt(peer_write_key, nonce, additional_data, AEADEncrypted)
```

If the decryption fails, the receiver MUST terminate the connection with a "bad_record_mac" alert.

An AEAD algorithm used in TLS 1.3 MUST NOT produce an expansion greater than 255 octets. An endpoint that receives a record from its peer with TLSCiphertext.length larger than 2^14 + 256 octets MUST terminate the connection with a "record_overflow" alert. This limit is derived from the maximum TLSInnerPlaintext length of 2^14 octets + 1 octet for ContentType + the maximum AEAD expansion of 255 octets.

5.3. Per-Record Nonce

A 64-bit sequence number is maintained separately for reading and writing records. The appropriate sequence number is incremented by one after reading or writing each record. Each sequence number is set to zero at the beginning of a connection and whenever the key is changed; the first record transmitted under a particular traffic key MUST use sequence number 0.

Because the size of sequence numbers is 64-bit, they should not wrap. If a TLS implementation would need to wrap a sequence number, it MUST either rekey (Section 4.6.3) or terminate the connection.

Each AEAD algorithm will specify a range of possible lengths for the per-record nonce, from N_MIN bytes to N_MAX bytes of input [RFC5116]. The length of the TLS per-record nonce (iv_length) is set to the larger of 8 bytes and N_MIN for the AEAD algorithm (see [RFC5116], Section 4). An AEAD algorithm where N_MAX is less than 8 bytes MUST NOT be used with TLS. The per-record nonce for the AEAD construction is formed as follows:

1. The 64-bit record sequence number is encoded in network byte order and padded to the left with zeros to iv_length.

2. The padded sequence number is XORed with either the static client_write_iv or server_write_iv (depending on the role).

The resulting quantity (of length iv_length) is used as the per-record nonce.

Note: This is a different construction from that in TLS 1.2, which specified a partially explicit nonce.

5.4. Record Padding

All encrypted TLS records can be padded to inflate the size of the TLSCiphertext. This allows the sender to hide the size of the traffic from an observer.

When generating a TLSCiphertext record, implementations MAY choose to pad. An unpadded record is just a record with a padding length of zero. Padding is a string of zero-valued bytes appended to the ContentType field before encryption. Implementations MUST set the padding octets to all zeros before encrypting.

Application Data records may contain a zero-length TLSInnerPlaintext.content if the sender desires. This permits generation of plausibly sized cover traffic in contexts where the presence or absence of activity may be sensitive. Implementations MUST NOT send Handshake and Alert records that have a zero-length TLSInnerPlaintext.content; if such a message is received, the receiving implementation MUST terminate the connection with an "unexpected_message" alert.

The padding sent is automatically verified by the record protection mechanism; upon successful decryption of a TLSCiphertext.encrypted_record, the receiving implementation scans the field from the end toward the beginning until it finds a non-zero octet. This non-zero octet is the content type of the message. This padding scheme was selected because it allows padding of any encrypted TLS record by an arbitrary size (from zero up to TLS record size limits) without introducing new content types. The design also enforces all-zero padding octets, which allows for quick detection of padding errors.

Implementations MUST limit their scanning to the cleartext returned from the AEAD decryption. If a receiving implementation does not find a non-zero octet in the cleartext, it MUST terminate the connection with an "unexpected_message" alert.

The presence of padding does not change the overall record size limitations: the full encoded TLSInnerPlaintext MUST NOT exceed 2^14 + 1 octets. If the maximum fragment length is reduced - as for example by the record_size_limit extension from [RFC8449] - then the reduced limit applies to the full plaintext, including the content type and padding.

Selecting a padding policy that suggests when and how much to pad is a complex topic and is beyond the scope of this specification. If the application-layer protocol on top of TLS has its own padding, it may be preferable to pad Application Data TLS records within the application layer. Padding for encrypted Handshake or Alert records must still be handled at the TLS layer, though. Later documents may define padding selection algorithms or define a padding policy request mechanism through TLS extensions or some other means.

5.5. Limits on Key Usage

There are cryptographic limits on the amount of plaintext which can be safely encrypted under a given set of keys. [AEAD-LIMITS] provides an analysis of these limits under the assumption that the underlying primitive (AES or ChaCha20) has no weaknesses. Implementations SHOULD do a key update as described in Section 4.6.3 prior to reaching these limits.

For AES-GCM, up to 2^24.5 full-size records (about 24 million) may be encrypted on a given connection while keeping a safety margin of approximately 2^-57 for Authenticated Encryption (AE) security. For ChaCha20/Poly1305, the record sequence number would wrap before the safety limit is reached.

6. Alert Protocol

TLS provides an Alert content type to indicate closure information and errors. Like other messages, alert messages are encrypted as specified by the current connection state.

Alert messages convey a description of the alert and a legacy field that conveyed the severity level of the message in previous versions of TLS. Alerts are divided into two classes: closure alerts and error alerts. In TLS 1.3, the severity is implicit in the type of alert being sent, and the "level" field can safely be ignored. The "close_notify" alert is used to indicate orderly closure of one direction of the connection. Upon receiving such an alert, the TLS implementation SHOULD indicate end-of-data to the application.

Error alerts indicate abortive closure of the connection (see Section 6.2). Upon receiving an error alert, the TLS implementation SHOULD indicate an error to the application and MUST NOT allow any further data to be sent or received on the connection. Servers and clients MUST forget the secret values and keys established in failed connections, with the exception of the PSKs associated with session tickets, which SHOULD be discarded if possible.

All the alerts listed in Section 6.2 MUST be sent with AlertLevel=fatal and MUST be treated as error alerts when received regardless of the AlertLevel in the message. Unknown Alert types MUST be treated as error alerts.

Note: TLS defines two generic alerts (see Section 6) to use upon failure to parse a message. Peers which receive a message which cannot be parsed according to the syntax (e.g., have a length extending beyond the message boundary or contain an out-of-range length) MUST terminate the connection with a "decode_error" alert. Peers which receive a message which is syntactically correct but semantically invalid (e.g., a DHE share of p - 1, or an invalid enum) MUST terminate the connection with an "illegal_parameter" alert.

```
      enum { warning(1), fatal(2), (255) } AlertLevel;

      enum {
          close_notify(0),
          unexpected_message(10),
          bad_record_mac(20),
          record_overflow(22),
          handshake_failure(40),
          bad_certificate(42),
          unsupported_certificate(43),
          certificate_revoked(44),
          certificate_expired(45),
          certificate_unknown(46),
          illegal_parameter(47),
          unknown_ca(48),
          access_denied(49),
          decode_error(50),
          decrypt_error(51),
          protocol_version(70),
          insufficient_security(71),
          internal_error(80),
          inappropriate_fallback(86),
          user_canceled(90),
          missing_extension(109),
          unsupported_extension(110),
          unrecognized_name(112),
          bad_certificate_status_response(113),
          unknown_psk_identity(115),
          certificate_required(116),
          no_application_protocol(120),
          (255)
      } AlertDescription;

      struct {
          AlertLevel level;
          AlertDescription description;
      } Alert;
```

6.1. Closure Alerts

The client and the server must share knowledge that the connection is ending in order to avoid a truncation attack.

close_notify: This alert notifies the recipient that the sender will not send any more messages on this connection. Any data received after a closure alert has been received MUST be ignored.

user_canceled: This alert notifies the recipient that the sender is canceling the handshake for some reason unrelated to a protocol failure. If a user cancels an operation after the handshake is complete, just closing the connection by sending a "close_notify" is more appropriate. This alert SHOULD be followed by a "close_notify". This alert generally has AlertLevel=warning.

Either party MAY initiate a close of its write side of the connection by sending a "close_notify" alert. Any data received after a closure alert has been received MUST be ignored. If a transport-level close is received prior to a "close_notify", the receiver cannot know that all the data that was sent has been received.

Each party MUST send a "close_notify" alert before closing its write side of the connection, unless it has already sent some error alert. This does not have any effect on its read side of the connection. Note that this is a change from versions of TLS prior to TLS 1.3 in which implementations were required to react to a "close_notify" by discarding pending writes and sending an immediate "close_notify" alert of their own. That previous requirement could cause truncation in the read side. Both parties need not wait to receive a "close_notify" alert before closing their read side of the connection, though doing so would introduce the possibility of truncation.

If the application protocol using TLS provides that any data may be carried over the underlying transport after the TLS connection is closed, the TLS implementation MUST receive a "close_notify" alert before indicating end-of-data to the application layer. No part of this standard should be taken to dictate the manner in which a usage profile for TLS manages its data transport, including when connections are opened or closed.

Note: It is assumed that closing the write side of a connection reliably delivers pending data before destroying the transport.

6.2. Error Alerts

Error handling in TLS is very simple. When an error is detected, the detecting party sends a message to its peer. Upon transmission or receipt of a fatal alert message, both parties MUST immediately close the connection.

Whenever an implementation encounters a fatal error condition, it SHOULD send an appropriate fatal alert and MUST close the connection without sending or receiving any additional data. Throughout this specification, when the phrases "terminate the connection" and "abort the handshake" are used without a specific alert it means that the implementation SHOULD send the alert indicated by the descriptions below. The phrases "terminate the connection with an X alert" and "abort the handshake with an X alert" mean that the implementation MUST send alert X if it sends any alert. All alerts defined below in this section, as well as all unknown alerts, are universally considered fatal as of TLS 1.3 (see Section 6). The implementation SHOULD provide a way to facilitate logging the sending and receiving of alerts.

The following error alerts are defined:

unexpected_message: An inappropriate message (e.g., the wrong handshake message, premature Application Data, etc.) was received. This alert should never be observed in communication between proper implementations.

bad_record_mac: This alert is returned if a record is received which cannot be deprotected. Because AEAD algorithms combine decryption and verification, and also to avoid side-channel attacks, this alert is used for all deprotection failures. This alert should never be observed in communication between proper implementations, except when messages were corrupted in the network.

record_overflow: A TLSCiphertext record was received that had a length more than 2^14 + 256 bytes, or a record decrypted to a TLSPlaintext record with more than 2^14 bytes (or some other negotiated limit). This alert should never be observed in communication between proper implementations, except when messages were corrupted in the network.

handshake_failure: Receipt of a "handshake_failure" alert message indicates that the sender was unable to negotiate an acceptable set of security parameters given the options available.

bad_certificate: A certificate was corrupt, contained signatures that did not verify correctly, etc.

unsupported_certificate: A certificate was of an unsupported type.

certificate_revoked: A certificate was revoked by its signer.

certificate_expired: A certificate has expired or is not currently valid.

certificate_unknown: Some other (unspecified) issue arose in processing the certificate, rendering it unacceptable.

illegal_parameter: A field in the handshake was incorrect or inconsistent with other fields. This alert is used for errors which conform to the formal protocol syntax but are otherwise incorrect.

unknown_ca: A valid certificate chain or partial chain was received, but the certificate was not accepted because the CA certificate could not be located or could not be matched with a known trust anchor.

access_denied: A valid certificate or PSK was received, but when access control was applied, the sender decided not to proceed with negotiation.

decode_error: A message could not be decoded because some field was out of the specified range or the length of the message was incorrect. This alert is used for errors where the message does not conform to the formal protocol syntax. This alert should never be observed in communication between proper implementations, except when messages were corrupted in the network.

decrypt_error: A handshake (not record layer) cryptographic operation failed, including being unable to correctly verify a signature or validate a Finished message or a PSK binder.

protocol_version: The protocol version the peer has attempted to negotiate is recognized but not supported (see Appendix E).

insufficient_security: Returned instead of "handshake_failure" when a negotiation has failed specifically because the server requires parameters more secure than those supported by the client.

internal_error: An internal error unrelated to the peer or the correctness of the protocol (such as a memory allocation failure) makes it impossible to continue.

inappropriate_fallback: Sent by a server in response to an invalid connection retry attempt from a client (see [RFC7507]).

missing_extension: Sent by endpoints that receive a handshake message not containing an extension that is mandatory to send for the offered TLS version or other negotiated parameters.

unsupported_extension: Sent by endpoints receiving any handshake message containing an extension known to be prohibited for inclusion in the given handshake message, or including any extensions in a ServerHello or Certificate not first offered in the corresponding ClientHello or CertificateRequest.

unrecognized_name: Sent by servers when no server exists identified by the name provided by the client via the "server_name" extension (see [RFC6066]).

bad_certificate_status_response: Sent by clients when an invalid or unacceptable OCSP response is provided by the server via the "status_request" extension (see [RFC6066]).

unknown_psk_identity: Sent by servers when PSK key establishment is desired but no acceptable PSK identity is provided by the client. Sending this alert is OPTIONAL; servers MAY instead choose to send a "decrypt_error" alert to merely indicate an invalid PSK identity.

certificate_required: Sent by servers when a client certificate is desired but none was provided by the client.

no_application_protocol: Sent by servers when a client "application_layer_protocol_negotiation" extension advertises only protocols that the server does not support (see [RFC7301]).

New Alert values are assigned by IANA as described in Section 11.

7. Cryptographic Computations

The TLS handshake establishes one or more input secrets which are combined to create the actual working keying material, as detailed below. The key derivation process incorporates both the input secrets and the handshake transcript. Note that because the handshake transcript includes the random values from the Hello messages, any given handshake will have different traffic secrets, even if the same input secrets are used, as is the case when the same PSK is used for multiple connections.

7.1. Key Schedule

The key derivation process makes use of the HKDF-Extract and HKDF-Expand functions as defined for HKDF [RFC5869], as well as the functions defined below:

```
HKDF-Expand-Label(Secret, Label, Context, Length) =
     HKDF-Expand(Secret, HkdfLabel, Length)

Where HkdfLabel is specified as:

struct {
    uint16 length = Length;
    opaque label<7..255> = "tls13 " + Label;
    opaque context<0..255> = Context;
} HkdfLabel;

Derive-Secret(Secret, Label, Messages) =
     HKDF-Expand-Label(Secret, Label,
                       Transcript-Hash(Messages), Hash.length)
```

The Hash function used by Transcript-Hash and HKDF is the cipher suite hash algorithm. Hash.length is its output length in bytes. Messages is the concatenation of the indicated handshake messages, including the handshake message type and length fields, but not including record layer headers. Note that in some cases a zero-length Context (indicated by "") is passed to HKDF-Expand-Label. The labels specified in this document are all ASCII strings and do not include a trailing NUL byte.

Note: With common hash functions, any label longer than 12 characters requires an additional iteration of the hash function to compute. The labels in this specification have all been chosen to fit within this limit.

Keys are derived from two input secrets using the HKDF-Extract and Derive-Secret functions. The general pattern for adding a new secret is to use HKDF-Extract with the Salt being the current secret state and the Input Keying Material (IKM) being the new secret to be added. In this version of TLS 1.3, the two input secrets are:

* PSK (a pre-shared key established externally or derived from the resumption_secret value from a previous connection)

* (EC)DHE shared secret (Section 7.4)

This produces a full key derivation schedule shown in the diagram below. In this diagram, the following formatting conventions apply:

* HKDF-Extract is drawn as taking the Salt argument from the top and the IKM argument from the left, with its output to the bottom and the name of the output on the right.

* Derive-Secret's Secret argument is indicated by the incoming arrow. For instance, the Early Secret is the Secret for generating the client_early_traffic_secret.

* "0" indicates a string of Hash.length bytes set to zero.

Note: the key derivation labels use the string "master" even though the values are referred to as "main" secrets. This mismatch is a result of renaming the values while retaining compatibility.

[[OPEN ISSUE: Replace the strings with hex value?]]

```
                 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_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 = Main 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_secret
                 |
                 +-----> Derive-Secret(., "res master",
                                       ClientHello...client Finished)
                                       = resumption_secret
```

The general pattern here is that the secrets shown down the left side of the diagram are just raw entropy without context, whereas the secrets down the right side include Handshake Context and therefore can be used to derive working keys without additional context. Note that the different calls to Derive-Secret may take different Messages arguments, even with the same secret. In a 0-RTT exchange, Derive-Secret is called with four distinct transcripts; in a 1-RTT-only exchange, it is called with three distinct transcripts.

If a given secret is not available, then the 0-value consisting of a string of Hash.length bytes set to zeros is used. Note that this does not mean skipping rounds, so if PSK is not in use, Early Secret will still be HKDF-Extract(0, 0). For the computation of the binder_key, the label is "ext binder" for external PSKs (those provisioned outside of TLS) and "res binder" for resumption PSKs (those provisioned as the resumption secret of a previous handshake). The different labels prevent the substitution of one type of PSK for the other.

There are multiple potential Early Secret values, depending on which PSK the server ultimately selects. The client will need to compute one for each potential PSK; if no PSK is selected, it will then need to compute the Early Secret corresponding to the zero PSK.

Once all the values which are to be derived from a given secret have been computed, that secret SHOULD be erased.

7.2. Updating Traffic Secrets

Once the handshake is complete, it is possible for either side to update its sending traffic keys using the KeyUpdate handshake message defined in Section 4.6.3. The next generation of traffic keys is computed by generating client_/server_application_traffic_secret_N+1 from client_/server_application_traffic_secret_N as described in this section and then re-deriving the traffic keys as described in Section 7.3.

The next-generation application_traffic_secret is computed as:

```
    application_traffic_secret_N+1 =
        HKDF-Expand-Label(application_traffic_secret_N,
                          "traffic upd", "", Hash.length)
```

Once client_/server_application_traffic_secret_N+1 and its associated traffic keys have been computed, implementations SHOULD delete client_/server_application_traffic_secret_N and its associated traffic keys.

7.3. Traffic Key Calculation

The traffic keying material is generated from the following input values:

* A secret value

* A purpose value indicating the specific value being generated

* The length of the key being generated

The traffic keying material is generated from an input traffic secret value using:

```
 [sender]_write_key = HKDF-Expand-Label(Secret, "key", "", key_length)
 [sender]_write_iv  = HKDF-Expand-Label(Secret, "iv", "", iv_length)
```

[sender] denotes the sending side. The value of Secret for each 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

Table 3: Secrets for Traffic Keys

All the traffic keying material is recomputed whenever the underlying Secret changes (e.g., when changing from the handshake to Application Data keys or upon a key update).

7.4. (EC)DHE Shared Secret Calculation

7.4.1. Finite Field Diffie-Hellman

For finite field groups, a conventional Diffie-Hellman [DH76] computation is performed. The negotiated key (Z) is converted to a byte string by encoding in big-endian form and left-padded with zeros up to the size of the prime. This byte string is used as the shared secret in the key schedule as specified above.

Note that this construction differs from previous versions of TLS which remove leading zeros.

7.4.2. Elliptic Curve Diffie-Hellman

For secp256r1, secp384r1 and secp521r1, ECDH calculations (including parameter and key generation as well as the shared secret calculation) are performed according to [IEEE1363] using the ECKAS-DH1 scheme with the identity map as the key derivation function (KDF), so that the shared secret is the x-coordinate of the ECDH shared secret elliptic curve point represented as an octet string. Note that this octet string ("Z" in IEEE 1363 terminology) as output by FE2OSP (the Field Element to Octet String Conversion Primitive) has constant length for any given field; leading zeros found in this octet string MUST NOT be truncated.

(Note that this use of the identity KDF is a technicality. The complete picture is that ECDH is employed with a non-trivial KDF because TLS does not directly use this secret for anything other than for computing other secrets.)

For X25519 and X448, the ECDH calculations are as follows:

* The public key to put into the KeyShareEntry.key_exchange structure is the result of applying the ECDH scalar multiplication function to the secret key of appropriate length (into scalar input) and the standard public basepoint (into u-coordinate point input).

* The ECDH shared secret is the result of applying the ECDH scalar multiplication function to the secret key (into scalar input) and the peer's public key (into u-coordinate point input). The output is used raw, with no processing.

For these curves, implementations SHOULD use the approach specified in [RFC7748] to calculate the Diffie-Hellman shared secret. Implementations MUST check whether the computed Diffie-Hellman shared secret is the all-zero value and abort if so, as described in Section 6 of [RFC7748]. If implementors use an alternative implementation of these elliptic curves, they SHOULD perform the additional checks specified in Section 7 of [RFC7748].

7.5. Exporters

[RFC5705] defines keying material exporters for TLS in terms of the TLS pseudorandom function (PRF). This document replaces the PRF with HKDF, thus requiring a new construction. The exporter interface remains the same.

The exporter value is computed as:

```
TLS-Exporter(label, context_value, key_length) =
    HKDF-Expand-Label(Derive-Secret(Secret, label, ""),
                      "exporter", Hash(context_value), key_length)
```

Where Secret is either the early_exporter_secret or the exporter_secret. Implementations MUST use the exporter_secret unless explicitly specified by the application. The early_exporter_secret is defined for use in settings where an exporter is needed for 0-RTT data. A separate interface for the early exporter is RECOMMENDED; this avoids the exporter user accidentally using an early exporter when a regular one is desired or vice versa.

If no context is provided, the context_value is zero length. Consequently, providing no context computes the same value as providing an empty context. This is a change from previous versions of TLS where an empty context produced a different output than an absent context. As of this document's publication, no allocated exporter label is used both with and without a context. Future specifications MUST NOT define a use of exporters that permit both an empty context and no context with the same label. New uses of exporters SHOULD provide a context in all exporter computations, though the value could be empty.

Requirements for the format of exporter labels are defined in Section 4 of [RFC5705].

8. 0-RTT and Anti-Replay

As noted in Section 2.3 and Appendix F.5, TLS does not provide inherent replay protections for 0-RTT data. There are two potential threats to be concerned with:

* Network attackers who mount a replay attack by simply duplicating a flight of 0-RTT data.

* Network attackers who take advantage of client retry behavior to arrange for the server to receive multiple copies of an application message. This threat already exists to some extent because clients that value robustness respond to network errors by attempting to retry requests. However, 0-RTT adds an additional dimension for any server system which does not maintain globally consistent server state. Specifically, if a server system has multiple zones where tickets from zone A will not be accepted in zone B, then an attacker can duplicate a ClientHello and early data intended for A to both A and B. At A, the data will be accepted in 0-RTT, but at B the server will reject 0-RTT data and instead force a full handshake. If the attacker blocks the ServerHello from A, then the client will complete the handshake with B and probably retry the request, leading to duplication on the server system as a whole.

The first class of attack can be prevented by sharing state to guarantee that the 0-RTT data is accepted at most once. Servers SHOULD provide that level of replay safety by implementing one of the methods described in this section or by equivalent means. It is understood, however, that due to operational concerns not all deployments will maintain state at that level. Therefore, in normal operation, clients will not know which, if any, of these mechanisms servers actually implement and hence MUST only send early data which they deem safe to be replayed.

In addition to the direct effects of replays, there is a class of attacks where even operations normally considered idempotent could be exploited by a large number of replays (timing attacks, resource limit exhaustion and others, as described in Appendix F.5). Those can be mitigated by ensuring that every 0-RTT payload can be replayed only a limited number of times. The server MUST ensure that any instance of it (be it a machine, a thread, or any other entity within the relevant serving infrastructure) would accept 0-RTT for the same 0-RTT handshake at most once; this limits the number of replays to the number of server instances in the deployment. Such a guarantee can be accomplished by locally recording data from recently received ClientHellos and rejecting repeats, or by any other method that provides the same or a stronger guarantee. The "at most once per server instance" guarantee is a minimum requirement; servers SHOULD limit 0-RTT replays further when feasible.

The second class of attack cannot be prevented at the TLS layer and MUST be dealt with by any application. Note that any application whose clients implement any kind of retry behavior already needs to implement some sort of anti-replay defense.

8.1. Single-Use Tickets

The simplest form of anti-replay defense is for the server to only allow each session ticket to be used once. For instance, the server can maintain a database of all outstanding valid tickets, deleting each ticket from the database as it is used. If an unknown ticket is provided, the server would then fall back to a full handshake.

If the tickets are not self-contained but rather are database keys, and the corresponding PSKs are deleted upon use, then connections established using PSKs enjoy 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 in Section 4.2.11. It then computes the expected_arrival_time as described in the next section and rejects 0-RTT if it is outside the recording window, falling back to the 1-RTT handshake.

If the expected_arrival_time is in the window, then the server checks to see if it has recorded a matching ClientHello. If one is found, it either aborts the handshake with an "illegal_parameter" alert or accepts the PSK but rejects 0-RTT. If no matching ClientHello is found, then it accepts 0-RTT and then stores the ClientHello for as long as the expected_arrival_time is inside the window. Servers MAY also implement data stores with false positives, such as Bloom filters, in which case they MUST respond to apparent replay by rejecting 0-RTT but MUST NOT abort the handshake.

The server MUST derive the storage key only from validated sections of the ClientHello. If the ClientHello contains multiple PSK identities, then an attacker can create multiple ClientHellos with different binder values for the less-preferred identity on the assumption that the server will not verify it (as recommended by Section 4.2.11). I.e., if the client sends PSKs A and B but the server prefers A, then the attacker can change the binder for B without affecting the binder for A. If the binder for B is part of the storage key, then this ClientHello will not appear as a duplicate, which will cause the ClientHello to be accepted, and may cause side effects such as replay cache pollution, although any 0-RTT data will not be decryptable because it will use different keys. If the validated binder or the ClientHello.random is used as the storage key, then this attack is not possible.

Because this mechanism does not require storing all outstanding tickets, it may be easier to implement in distributed systems with high rates of resumption and 0-RTT, at the cost of potentially weaker anti-replay defense because of the difficulty of reliably storing and retrieving the received ClientHello messages. In many such systems, it is impractical to have globally consistent storage of all the received ClientHellos. In this case, the best anti-replay protection is provided by having a single storage zone be authoritative for a given ticket and refusing 0-RTT for that ticket in any other zone. This approach prevents simple replay by the attacker because only one zone will accept 0-RTT data. A weaker design is to implement separate storage for each zone but allow 0-RTT in any zone. This approach limits the number of replays to once per zone. Application message duplication of course remains possible with either design.

When implementations are freshly started, they SHOULD reject 0-RTT as long as any portion of their recording window overlaps the startup time. Otherwise, they run the risk of accepting replays which were originally sent during that period.

Note: If the client's clock is running much faster than the server's, then a ClientHello may be received that is outside the window in the future, in which case it might be accepted for 1-RTT, causing a client retry, and then acceptable later for 0-RTT. This is another variant of the second form of attack described in Section 8.

8.3. Freshness Checks

Because the ClientHello indicates the time at which the client sent it, it is possible to efficiently determine whether a ClientHello was likely sent reasonably recently and only accept 0-RTT for such a ClientHello, otherwise falling back to a 1-RTT handshake. This is necessary for the ClientHello storage mechanism described in Section 8.2 because otherwise the server needs to store an unlimited number of ClientHellos, and is a useful optimization for self-contained single-use tickets because it allows efficient rejection of ClientHellos which cannot be used for 0-RTT.

In order to implement this mechanism, a server needs to store the time that the server generated the session ticket, offset by an estimate of the round-trip time between client and server. I.e.,

```
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 subsequent early Application Data records are received. After early data is accepted, records may continue to be streamed to the server over a longer time period.

9. Compliance Requirements

9.1. Mandatory-to-Implement Cipher Suites

In the absence of an application profile standard specifying otherwise:

A TLS-compliant application MUST implement the TLS_AES_128_GCM_SHA256 [GCM] cipher suite and SHOULD implement the TLS_AES_256_GCM_SHA384 [GCM] and TLS_CHACHA20_POLY1305_SHA256 [RFC8439] cipher suites (see Appendix B.4).

A TLS-compliant application MUST support digital signatures with rsa_pkcs1_sha256 (for certificates), rsa_pss_rsae_sha256 (for CertificateVerify and certificates), and ecdsa_secp256r1_sha256. A TLS-compliant application MUST support key exchange with secp256r1 (NIST P-256) and SHOULD support key exchange with X25519 [RFC7748].

9.2. Mandatory-to-Implement Extensions

In the absence of an application profile standard specifying otherwise, a TLS-compliant application MUST implement the following TLS extensions:

* Supported Versions ("supported_versions"; Section 4.2.1)

* Cookie ("cookie"; Section 4.2.2)

* Signature Algorithms ("signature_algorithms"; Section 4.2.3)

* Signature Algorithms Certificate ("signature_algorithms_cert"; Section 4.2.3)

* Negotiated Groups ("supported_groups"; Section 4.2.7)

* Key Share ("key_share"; Section 4.2.8)

* Server Name Indication ("server_name"; Section 3 of [RFC6066])

All implementations MUST send and use these extensions when offering applicable features:

* "supported_versions" is REQUIRED for all ClientHello, ServerHello, and HelloRetryRequest messages.

* "signature_algorithms" is REQUIRED for certificate authentication.

* "supported_groups" is REQUIRED for ClientHello messages using DHE or ECDHE key exchange.

* "key_share" is REQUIRED for DHE or ECDHE key exchange.

* "pre_shared_key" is REQUIRED for PSK key agreement.

* "psk_key_exchange_modes" is REQUIRED for PSK key agreement.

A client is considered to be attempting to negotiate using this specification if the ClientHello contains a "supported_versions" extension with 0x0304 contained in its body. Such a ClientHello message MUST meet the following requirements:

* If not containing a "pre_shared_key" extension, it MUST contain both a "signature_algorithms" extension and a "supported_groups" extension.

* If containing a "supported_groups" extension, it MUST also contain a "key_share" extension, and vice versa. An empty KeyShare.client_shares list is permitted.

Servers receiving a ClientHello which does not conform to these requirements MUST abort the handshake with a "missing_extension" alert.

Additionally, all implementations MUST support the use of the "server_name" extension with applications capable of using it. Servers MAY require clients to send a valid "server_name" extension. Servers requiring this extension SHOULD respond to a ClientHello lacking a "server_name" extension by terminating the connection with a "missing_extension" alert.

9.3. Protocol Invariants

This section describes invariants that TLS endpoints and middleboxes MUST follow. It also applies to earlier versions of TLS.

TLS is designed to be securely and compatibly extensible. Newer clients or servers, when communicating with newer peers, should negotiate the most preferred common parameters. The TLS handshake provides downgrade protection: Middleboxes passing traffic between a newer client and newer server without terminating TLS should be unable to influence the handshake (see Appendix F.1). At the same time, deployments update at different rates, so a newer client or server MAY continue to support older parameters, which would allow it to interoperate with older endpoints.

For this to work, implementations MUST correctly handle extensible fields:

* A client sending a ClientHello MUST support all parameters advertised in it. Otherwise, the server may fail to interoperate by selecting one of those parameters.

* A server receiving a ClientHello MUST correctly ignore all unrecognized cipher suites, extensions, and other parameters. Otherwise, it may fail to interoperate with newer clients. In TLS 1.3, a client receiving a CertificateRequest or NewSessionTicket MUST also ignore all unrecognized extensions.

* A middlebox which terminates a TLS connection MUST behave as a compliant TLS server (to the original client), including having a certificate which the client is willing to accept, and also as a compliant TLS client (to the original server), including verifying the original server's certificate. In particular, it MUST

generate its own ClientHello containing only parameters it understands, and it MUST generate a fresh ServerHello random value, rather than forwarding the endpoint's value.

Note that TLS's protocol requirements and security analysis only apply to the two connections separately. Safely deploying a TLS terminator requires additional security considerations which are beyond the scope of this document.

* A middlebox which forwards ClientHello parameters it does not understand MUST NOT process any messages beyond that ClientHello. It MUST forward all subsequent traffic unmodified. Otherwise, it may fail to interoperate with newer clients and servers.

  Forwarded ClientHellos may contain advertisements for features not supported by the middlebox, so the response may include future TLS additions the middlebox does not recognize. These additions MAY change any message beyond the ClientHello arbitrarily. In particular, the values sent in the ServerHello might change, the ServerHello format might change, and the TLSCiphertext format might change.

The design of TLS 1.3 was constrained by widely deployed non-compliant TLS middleboxes (see Appendix E.4); however, it does not relax the invariants. Those middleboxes continue to be non-compliant.

10. Security Considerations

Security issues are discussed throughout this memo, especially in Appendix C, Appendix E, and Appendix F.

11. IANA Considerations

[[OPEN ISSUE: Should we remove this? I am reluctant to create a situation where one needs to read 8446 to process this document.]]

[[OPEN ISSUE: Add some text to rename the extended_master_secret entry in the extensions registry to extended_main_secret, after the above is resolved.]]

This document uses several registries that were originally created in [RFC4346] and updated in [RFC8447]. IANA has updated these to reference this document. The registries and their allocation policies are below:

* TLS Cipher Suites registry: values with the first byte in the range 0-254 (decimal) are assigned via Specification Required [RFC8126]. Values with the first byte 255 (decimal) are reserved for Private Use [RFC8126].

  IANA has added the cipher suites listed in Appendix B.4 to the registry. The "Value" and "Description" columns are taken from the table. The "DTLS-OK" and "Recommended" columns are both marked as "Y" for each new cipher suite.

* TLS ContentType registry: Future values are allocated via Standards Action [RFC8126].

* TLS Alerts registry: Future values are allocated via Standards Action [RFC8126]. IANA has populated this registry with the values from Appendix B.2. The "DTLS-OK" column is marked as "Y" for all such values. Values marked as "_RESERVED" have comments describing their previous usage.

* TLS HandshakeType registry: Future values are allocated via Standards Action [RFC8126]. IANA has updated this registry to rename item 4 from "NewSessionTicket" to "new_session_ticket" and populated this registry with the values from Appendix B.3. The "DTLS-OK" column is marked as "Y" for all such values. Values marked "_RESERVED" have comments describing their previous or temporary usage.

This document also uses the TLS ExtensionType Values registry originally created in [RFC4366]. IANA has updated it to reference this document. Changes to the registry follow:

* IANA has updated the registration policy as follows:

  Values with the first byte in the range 0-254 (decimal) are assigned via Specification Required [RFC8126]. Values with the first byte 255 (decimal) are reserved for Private Use [RFC8126].

* IANA has updated this registry to include the "key_share", "pre_shared_key", "psk_key_exchange_modes", "early_data", "cookie", "supported_versions", "certificate_authorities", "oid_filters", "post_handshake_auth", and "signature_algorithms_cert" extensions with the values defined in this document and the "Recommended" value of "Y".

* IANA has updated this registry to include a "TLS 1.3" column which lists the messages in which the extension may appear. This column has been initially populated from the table in Section 4.2, with any extension not listed there marked as "-" to indicate that it is not used by TLS 1.3.

This document updates an entry in the TLS Certificate Types registry originally created in [RFC6091] and updated in [RFC8447]. IANA has updated the entry for value 1 to have the name "OpenPGP_RESERVED", "Recommended" value "N", and comment "Used in TLS versions prior to 1.3." IANA has updated the entry for value 0 to have the name "X509", "Recommended" value "Y", and comment "Was X.509 before TLS 1.3".

This document updates an entry in the TLS Certificate Status Types registry originally created in [RFC6961]. IANA has updated the entry for value 2 to have the name "ocsp_multi_RESERVED" and comment "Used in TLS versions prior to 1.3".

This document updates two entries in the TLS Supported Groups registry (created under a different name by [RFC4492]; now maintained by [RFC8422]) and updated by [RFC7919] and [RFC8447]. The entries for values 29 and 30 (x25519 and x448) have been updated to also refer to this document.

In addition, this document defines two new registries that are maintained by IANA:

* TLS SignatureScheme registry: Values with the first byte in the range 0-253 (decimal) are assigned via Specification Required [RFC8126]. Values with the first byte 254 or 255 (decimal) are reserved for Private Use [RFC8126]. Values with the first byte in the range 0-6 or with the second byte in the range 0-3 that are not currently allocated are reserved for backward compatibility. This registry has a "Recommended" column. The registry has been initially populated with the values described in Section 4.2.3. The following values are marked as "Recommended": ecdsa_secp256r1_sha256, ecdsa_secp384r1_sha384, rsa_pss_rsae_sha256, rsa_pss_rsae_sha384, rsa_pss_rsae_sha512, rsa_pss_pss_sha256, rsa_pss_pss_sha384, rsa_pss_pss_sha512, and ed25519. The "Recommended" column is assigned a value of "N" unless explicitly requested, and adding a value with a "Recommended" value of "Y" requires Standards Action [RFC8126]. IESG Approval is REQUIRED for a Y->N transition.

* TLS PskKeyExchangeMode registry: Values in the range 0-253 (decimal) are assigned via Specification Required [RFC8126]. The values 254 and 255 (decimal) are reserved for Private Use

[RFC8126]. This registry has a "Recommended" column. The registry has been initially populated with psk_ke (0) and psk_dhe_ke (1). Both are marked as "Recommended". The "Recommended" column is assigned a value of "N" unless explicitly requested, and adding a value with a "Recommended" value of "Y" requires Standards Action [RFC8126]. IESG Approval is REQUIRED for a Y->N transition.

12. References

12.1. Normative References

[DH76] Diffie, W. and M. Hellman, "New directions in cryptography", IEEE Transactions on Information Theory Vol. 22, pp. 644-654, DOI 10.1109/tit.1976.1055638, November 1976, <https://doi.org/10.1109/tit.1976.1055638>.

[ECDSA] American National Standards Institute, "Public Key Cryptography for the Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA)", ANSI ANS X9.62-2005, November 2005.

[GCM] Dworkin, M., "Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC", NIST Special Publication 800-38D, November 2007.

[IEEE1363] "IEEE Standard Specifications for Public-Key Cryptography", IEEE standard, DOI 10.1109/ieeestd.2000.92292, n.d., <https://doi.org/10.1109/ieeestd.2000.92292>.

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

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

[RFC5116] McGrew, D., "An Interface and Algorithms for Authenticated Encryption", RFC 5116, DOI 10.17487/RFC5116, January 2008, <https://www.rfc-editor.org/info/rfc5116>.

[RFC5280] Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R., and W. Polk, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", RFC 5280, DOI 10.17487/RFC5280, May 2008, <https://www.rfc-editor.org/info/rfc5280>.

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

[RFC5756] Turner, S., Brown, D., Yiu, K., Housley, R., and T. Polk, "Updates for RSAES-OAEP and RSASSA-PSS Algorithm Parameters", RFC 5756, DOI 10.17487/RFC5756, January 2010, <https://www.rfc-editor.org/info/rfc5756>.

[RFC5869] Krawczyk, H. and P. Eronen, "HMAC-based Extract-and-Expand Key Derivation Function (HKDF)", RFC 5869, DOI 10.17487/RFC5869, May 2010, <https://www.rfc-editor.org/info/rfc5869>.

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

[RFC6655] McGrew, D. and D. Bailey, "AES-CCM Cipher Suites for Transport Layer Security (TLS)", RFC 6655, DOI 10.17487/RFC6655, July 2012, <https://www.rfc-editor.org/info/rfc6655>.

[RFC6960] Santesson, S., Myers, M., Ankney, R., Malpani, A., Galperin, S., and C. Adams, "X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP", RFC 6960, DOI 10.17487/RFC6960, June 2013, <https://www.rfc-editor.org/info/rfc6960>.

[RFC6961] Pettersen, Y., "The Transport Layer Security (TLS) Multiple Certificate Status Request Extension", RFC 6961, DOI 10.17487/RFC6961, June 2013, <https://www.rfc-editor.org/info/rfc6961>.

[RFC6962] Laurie, B., Langley, A., and E. Kasper, "Certificate Transparency", RFC 6962, DOI 10.17487/RFC6962, June 2013, <https://www.rfc-editor.org/info/rfc6962>.

[RFC6979] Pornin, T., "Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA)", RFC 6979, DOI 10.17487/RFC6979, August 2013, <https://www.rfc-editor.org/info/rfc6979>.

[RFC7301] Friedl, S., Popov, A., Langley, A., and E. Stephan, "Transport Layer Security (TLS) Application-Layer Protocol Negotiation Extension", RFC 7301, DOI 10.17487/RFC7301, July 2014, <https://www.rfc-editor.org/info/rfc7301>.

[RFC7507] Moeller, B. and A. Langley, "TLS Fallback Signaling Cipher Suite Value (SCSV) for Preventing Protocol Downgrade Attacks", RFC 7507, DOI 10.17487/RFC7507, April 2015, <https://www.rfc-editor.org/info/rfc7507>.

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

[RFC7748] Langley, A., Hamburg, M., and S. Turner, "Elliptic Curves for Security", RFC 7748, DOI 10.17487/RFC7748, January 2016, <https://www.rfc-editor.org/info/rfc7748>.

[RFC7919] Gillmor, D., "Negotiated Finite Field Diffie-Hellman Ephemeral Parameters for Transport Layer Security (TLS)", RFC 7919, DOI 10.17487/RFC7919, August 2016, <https://www.rfc-editor.org/info/rfc7919>.

[RFC8017] Moriarty, K., Ed., Kaliski, B., Jonsson, J., and A. Rusch, "PKCS #1: RSA Cryptography Specifications Version 2.2", RFC 8017, DOI 10.17487/RFC8017, November 2016, <https://www.rfc-editor.org/info/rfc8017>.

[RFC8032] Josefsson, S. and I. Liusvaara, "Edwards-Curve Digital Signature Algorithm (EdDSA)", RFC 8032, DOI 10.17487/RFC8032, January 2017, <https://www.rfc-editor.org/info/rfc8032>.

[RFC8126] Cotton, M., Leiba, B., and T. Narten, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 8126, DOI 10.17487/RFC8126, June 2017, <https://www.rfc-editor.org/info/rfc8126>.

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

[RFC8439] Nir, Y. and A. Langley, "ChaCha20 and Poly1305 for IETF Protocols", RFC 8439, DOI 10.17487/RFC8439, June 2018, <https://www.rfc-editor.org/info/rfc8439>.

[SHS] Dang, Q., "Secure Hash Standard", National Institute of Standards and Technology report, DOI 10.6028/nist.fips.180-4, July 2015, <https://doi.org/10.6028/nist.fips.180-4>.

[X690] ITU-T, "Information technology - ASN.1 encoding Rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)", ISO/IEC 8825-1:2002, November 2015.

12.2. Informative References

[AEAD-LIMITS] Luykx, A. and K. Paterson, "Limits on Authenticated Encryption Use in TLS", August 2017, <http://www.isg.rhul.ac.uk/~kp/TLS-AEbounds.pdf>.

[BBFGKZ16] Bhargavan, K., Brzuska, C., Fournet, C., Green, M., Kohlweiss, M., and S. Zanella-Beguelin, "Downgrade Resilience in Key-Exchange Protocols", 2016 IEEE Symposium on Security and Privacy (SP), DOI 10.1109/sp.2016.37, May 2016, <https://doi.org/10.1109/sp.2016.37>.

[BBK17] Bhargavan, K., Blanchet, B., and N. Kobeissi, "Verified Models and Reference Implementations for the TLS 1.3 Standard Candidate", 2017 IEEE Symposium on Security and Privacy (SP), DOI 10.1109/sp.2017.26, May 2017, <https://doi.org/10.1109/sp.2017.26>.

[BDFKPPRSZZ16] Bhargavan, K., Delignat-Lavaud, A., Fournet, C., Kohlweiss, M., Pan, J., Protzenko, J., Rastogi, A., Swamy, N., Zanella-Beguelin, S., and J. Zinzindohoue, "Implementing and Proving the TLS 1.3 Record Layer", Proceedings of IEEE Symposium on Security and Privacy (San Jose) 2017 , December 2016, <https://eprint.iacr.org/2016/1178>.

[Ben17a] Benjamin, D., "Presentation before the TLS WG at IETF 100", 2017, <https://datatracker.ietf.org/meeting/100/materials/slides-100-tls-sessa-tls13/>.

[Ben17b] Benjamin, D., "Additional TLS 1.3 results from Chrome", 2017, <https://www.ietf.org/mail-archive/web/tls/current/msg25168.html>.

[Blei98] Bleichenbacher, D., "Chosen Ciphertext Attacks against Protocols Based on RSA Encryption Standard PKCS #1", Proceedings of CRYPTO '98 , 1998.

[BMMRT15] Badertscher, C., Matt, C., Maurer, U., Rogaway, P., and B. Tackmann, "Augmented Secure Channels and the Goal of the TLS 1.3 Record Layer", ProvSec 2015 , September 2015, <https://eprint.iacr.org/2015/394>.

[BT16] Bellare, M. and B. Tackmann, "The Multi-User Security of Authenticated Encryption: AES-GCM in TLS 1.3", Proceedings of CRYPTO 2016 , July 2016, <https://eprint.iacr.org/2016/564>.

[CCG16] Cohn-Gordon, K., Cremers, C., and L. Garratt, "On Post-compromise Security", 2016 IEEE 29th Computer Security Foundations Symposium (CSF), DOI 10.1109/csf.2016.19, June 2016, <https://doi.org/10.1109/csf.2016.19>.

[CHECKOWAY]
Checkoway, S., Maskiewicz, J., Garman, C., Fried, J., Cohney, S., Green, M., Heninger, N., Weinmann, R., Rescorla, E., and H. Shacham, "A Systematic Analysis of the Juniper Dual EC Incident", Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, DOI 10.1145/2976749.2978395, October 2016, <https://doi.org/10.1145/2976749.2978395>.

[CHHSV17] Cremers, C., Horvat, M., Hoyland, J., van der Merwe, T., and S. Scott, "Awkward Handshake: Possible mismatch of client/server view on client authentication in post-handshake mode in Revision 18", message to the TLS mailing list , February 2017, <https://www.ietf.org/mail-archive/web/tls/current/msg22382.html>.

[CHSV16] Cremers, C., Horvat, M., Scott, S., and T. van der Merwe, "Automated Analysis and Verification of TLS 1.3: 0-RTT, Resumption and Delayed Authentication", 2016 IEEE Symposium on Security and Privacy (SP), DOI 10.1109/sp.2016.35, May 2016, <https://doi.org/10.1109/sp.2016.35>.

[CK01] Canetti, R. and H. Krawczyk, "Analysis of Key-Exchange Protocols and Their Use for Building Secure Channels", Lecture Notes in Computer Science pp. 453-474, DOI 10.1007/3-540-44987-6_28, 2001, <https://doi.org/10.1007/3-540-44987-6_28>.

[CLINIC] Miller, B., Huang, L., Joseph, A., and J. Tygar, "I Know Why You Went to the Clinic: Risks and Realization of HTTPS Traffic Analysis", Privacy Enhancing Technologies pp. 143-163, DOI 10.1007/978-3-319-08506-7_8, 2014, <https://doi.org/10.1007/978-3-319-08506-7_8>.

[DFGS15] Dowling, B., Fischlin, M., Guenther, F., and D. Stebila, "A Cryptographic Analysis of the TLS 1.3 draft-10 Full and Pre-shared Key Handshake Protocol", Proceedings of ACM CCS 2015 , October 2016, <https://eprint.iacr.org/2015/914>.

[DFGS16] Dowling, B., Fischlin, M., Guenther, F., and D. Stebila, "A Cryptographic Analysis of the TLS 1.3 draft-10 Full and Pre-shared Key Handshake Protocol", TRON 2016 , February 2016, <https://eprint.iacr.org/2016/081>.

[DOW92] Diffie, W., Van Oorschot, P., and M. Wiener, "Authentication and authenticated key exchanges", Designs, Codes and Cryptography Vol. 2, pp. 107-125, DOI 10.1007/bf00124891, June 1992, <https://doi.org/10.1007/bf00124891>.

[DSS] "Digital Signature Standard (DSS)", National Institute of Standards and Technology report, DOI 10.6028/nist.fips.186-4, July 2013, <https://doi.org/10.6028/nist.fips.186-4>.

[FETCH] WHATWG, "Fetch Standard", February 2021, <https://fetch.spec.whatwg.org/>.

[FG17] Fischlin, M. and F. Guenther, "Replay Attacks on Zero Round-Trip Time: The Case of the TLS 1.3 Handshake Candidates", Proceedings of Euro S&P 2017 , 2017, <https://eprint.iacr.org/2017/082>.

[FGSW16] Fischlin, M., Guenther, F., Schmidt, B., and B. Warinschi, "Key Confirmation in Key Exchange: A Formal Treatment and Implications for TLS 1.3", Proceedings of IEEE Symposium on Security and Privacy (Oakland) 2016 , 2016, <http://ieeexplore.ieee.org/document/7546517/>.

[FW15] Weimer, F., "Factoring RSA Keys With TLS Perfect Forward Secrecy", September 2015.

[HCJC16] Husák, M., ermák, M., Jirsík, T., and P. eleda, "HTTPS traffic analysis and client identification using passive SSL/TLS fingerprinting", EURASIP Journal on Information Security Vol. 2016, DOI 10.1186/s13635-016-0030-7, February 2016, <https://doi.org/10.1186/s13635-016-0030-7>.

[HGFS15] Hlauschek, C., Gruber, M., Fankhauser, F., and C. Schanes, "Prying Open Pandora's Box: KCI Attacks against TLS", Proceedings of USENIX Workshop on Offensive Technologies , 2015.

[JSS15] Jager, T., Schwenk, J., and J. Somorovsky, "On the Security of TLS 1.3 and QUIC Against Weaknesses in PKCS#1 v1.5 Encryption", Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, DOI 10.1145/2810103.2813657, October 2015, <https://doi.org/10.1145/2810103.2813657>.

[KEYAGREEMENT] Barker, E., Chen, L., Roginsky, A., and M. Smid, "Recommendation for Pair-Wise Key Establishment Schemes Using Discrete Logarithm Cryptography", National Institute of Standards and Technology report, DOI 10.6028/nist.sp.800-56ar2, May 2013, <https://doi.org/10.6028/nist.sp.800-56ar2>.

[Kraw10] Krawczyk, H., "Cryptographic Extraction and Key Derivation: The HKDF Scheme", Proceedings of CRYPTO 2010 , 2010, <https://eprint.iacr.org/2010/264>.

[Kraw16] Krawczyk, H., "A Unilateral-to-Mutual Authentication Compiler for Key Exchange (with Applications to Client Authentication in TLS 1.3", Proceedings of ACM CCS 2016 , October 2016, <https://eprint.iacr.org/2016/711>.

[KW16] Krawczyk, H. and H. Wee, "The OPTLS Protocol and TLS 1.3", Proceedings of Euro S&P 2016 , 2016, <https://eprint.iacr.org/2015/978>.

[LXZFH16] Li, X., Xu, J., Zhang, Z., Feng, D., and H. Hu, "Multiple Handshakes Security of TLS 1.3 Candidates", 2016 IEEE Symposium on Security and Privacy (SP), DOI 10.1109/sp.2016.36, May 2016, <https://doi.org/10.1109/sp.2016.36>.

[Mac17] MacCarthaigh, C., "Security Review of TLS1.3 0-RTT", March 2017, <https://github.com/tlswg/tls13-spec/issues/1001>.

[PS18] Patton, C. and T. Shrimpton, "Partially specified channels: The TLS 1.3 record layer without elision", 2018, <https://eprint.iacr.org/2018/634>.

[PSK-FINISHED]
Cremers, C., Horvat, M., van der Merwe, T., and S. Scott, "Revision 10: possible attack if client authentication is allowed during PSK", message to the TLS mailing list, , 2015, <https://www.ietf.org/mail-archive/web/tls/current/msg18215.html>.

[REKEY] Abdalla, M. and M. Bellare, "Increasing the Lifetime of a Key: A Comparative Analysis of the Security of Re-keying Techniques", Advances in Cryptology - ASIACRYPT 2000 pp. 546-559, DOI 10.1007/3-540-44448-3_42, 2000, <https://doi.org/10.1007/3-540-44448-3_42>.

[Res17a] Rescorla, E., "Preliminary data on Firefox TLS 1.3 Middlebox experiment", message to the TLS mailing list , 2017, <https://www.ietf.org/mail-archive/web/tls/current/msg25091.html>.

[Res17b] Rescorla, E., "More compatibility measurement results", message to the TLS mailing list , December 2017, <https://www.ietf.org/mail-archive/web/tls/current/msg25179.html>.

[RFC3552] Rescorla, E. and B. Korver, "Guidelines for Writing RFC Text on Security Considerations", BCP 72, RFC 3552, DOI 10.17487/RFC3552, July 2003, <https://www.rfc-editor.org/info/rfc3552>.

[RFC4086] Eastlake 3rd, D., Schiller, J., and S. Crocker, "Randomness Requirements for Security", BCP 106, RFC 4086, DOI 10.17487/RFC4086, June 2005, <https://www.rfc-editor.org/info/rfc4086>.

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

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

[RFC4492] Blake-Wilson, S., Bolyard, N., Gupta, V., Hawk, C., and B. Moeller, "Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS)", RFC 4492, DOI 10.17487/RFC4492, May 2006, <https://www.rfc-editor.org/info/rfc4492>.

[RFC5077] Salowey, J., Zhou, H., Eronen, P., and H. Tschofenig, "Transport Layer Security (TLS) Session Resumption without Server-Side State", RFC 5077, DOI 10.17487/RFC5077, January 2008, <https://www.rfc-editor.org/info/rfc5077>.

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

[RFC5764] McGrew, D. and E. Rescorla, "Datagram Transport Layer Security (DTLS) Extension to Establish Keys for the Secure Real-time Transport Protocol (SRTP)", RFC 5764, DOI 10.17487/RFC5764, May 2010, <https://www.rfc-editor.org/info/rfc5764>.

[RFC5929] Altman, J., Williams, N., and L. Zhu, "Channel Bindings for TLS", RFC 5929, DOI 10.17487/RFC5929, July 2010, <https://www.rfc-editor.org/info/rfc5929>.

[RFC6091] Mavrogiannopoulos, N. and D. Gillmor, "Using OpenPGP Keys for Transport Layer Security (TLS) Authentication", RFC 6091, DOI 10.17487/RFC6091, February 2011, <https://www.rfc-editor.org/info/rfc6091>.

[RFC6101] Freier, A., Karlton, P., and P. Kocher, "The Secure Sockets Layer (SSL) Protocol Version 3.0", RFC 6101, DOI 10.17487/RFC6101, August 2011, <https://www.rfc-editor.org/info/rfc6101>.

[RFC6176] Turner, S. and T. Polk, "Prohibiting Secure Sockets Layer (SSL) Version 2.0", RFC 6176, DOI 10.17487/RFC6176, March 2011, <https://www.rfc-editor.org/info/rfc6176>.

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

[RFC6520] Seggelmann, R., Tuexen, M., and M. Williams, "Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS) Heartbeat Extension", RFC 6520, DOI 10.17487/RFC6520, February 2012, <https://www.rfc-editor.org/info/rfc6520>.

[RFC7230] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing", RFC 7230, DOI 10.17487/RFC7230, June 2014, <https://www.rfc-editor.org/info/rfc7230>.

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

[RFC7465] Popov, A., "Prohibiting RC4 Cipher Suites", RFC 7465, DOI 10.17487/RFC7465, February 2015, <https://www.rfc-editor.org/info/rfc7465>.

[RFC7568] Barnes, R., Thomson, M., Pironti, A., and A. Langley, "Deprecating Secure Sockets Layer Version 3.0", RFC 7568, DOI 10.17487/RFC7568, June 2015, <https://www.rfc-editor.org/info/rfc7568>.

[RFC7685] Langley, A., "A Transport Layer Security (TLS) ClientHello Padding Extension", RFC 7685, DOI 10.17487/RFC7685, October 2015, <https://www.rfc-editor.org/info/rfc7685>.

[RFC7924] Santesson, S. and H. Tschofenig, "Transport Layer Security (TLS) Cached Information Extension", RFC 7924, DOI 10.17487/RFC7924, July 2016, <https://www.rfc-editor.org/info/rfc7924>.

[RFC8305] Schinazi, D. and T. Pauly, "Happy Eyeballs Version 2: Better Connectivity Using Concurrency", RFC 8305, DOI 10.17487/RFC8305, December 2017, <https://www.rfc-editor.org/info/rfc8305>.

[RFC8422] Nir, Y., Josefsson, S., and M. Pegourie-Gonnard, "Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS) Versions 1.2 and Earlier", RFC 8422, DOI 10.17487/RFC8422, August 2018, <https://www.rfc-editor.org/info/rfc8422>.

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

[RFC8447] Salowey, J. and S. Turner, "IANA Registry Updates for TLS and DTLS", RFC 8447, DOI 10.17487/RFC8447, August 2018, <https://www.rfc-editor.org/info/rfc8447>.

[RFC8448] Thomson, M., "Example Handshake Traces for TLS 1.3", RFC 8448, DOI 10.17487/RFC8448, January 2019, <https://www.rfc-editor.org/info/rfc8448>.

[RFC8449] Thomson, M., "Record Size Limit Extension for TLS", RFC 8449, DOI 10.17487/RFC8449, August 2018, <https://www.rfc-editor.org/info/rfc8449>.

[RSA] Rivest, R., Shamir, A., and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems", Communications of the ACM Vol. 21, pp. 120-126, DOI 10.1145/359340.359342, February 1978, <https://doi.org/10.1145/359340.359342>.

[SIGMA] Krawczyk, H., "SIGMA: The SIGn-and-MAc Approach to Authenticated Diffie-Hellman and Its Use in the IKE Protocols", Advances in Cryptology - CRYPTO 2003 pp. 400-425, DOI 10.1007/978-3-540-45146-4_24, 2003, <https://doi.org/10.1007/978-3-540-45146-4_24>.

[SLOTH] Bhargavan, K. and G. Leurent, "Transcript Collision Attacks: Breaking Authentication in TLS, IKE, and SSH", Proceedings 2016 Network and Distributed System Security Symposium, DOI 10.14722/ndss.2016.23418, 2016, <https://doi.org/10.14722/ndss.2016.23418>.

[SSL2] Hickman, K., "The SSL Protocol", 9 February 1995.

[TIMING] Boneh, D. and D. Brumley, "Remote Timing Attacks Are Practical", USENIX Security Symposium, 2003.

[X501] "Information Technology - Open Systems Interconnection - The Directory: Models", ITU-T X.501, 1993.

Appendix A. State Machine

This appendix provides a summary of the legal state transitions for the client and server handshakes. State names (in all capitals, e.g., START) have no formal meaning but are provided for ease of comprehension. Actions which are taken only in certain circumstances are indicated in []. The notation "K_{send,recv} = foo" means "set the send/recv key to the given key".

A.1. Client

```
                              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 |                 | Cert-based 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 appendix provides the normative protocol types and the definitions for constants. Values listed as "_RESERVED" were used in previous versions of TLS and are listed here for completeness. TLS 1.3 implementations MUST NOT send them but might receive them from older TLS implementations.

B.1. Record Layer

```
enum {
    invalid(0),
    change_cipher_spec(20),
    alert(21),
    handshake(22),
    application_data(23),
    (255)
} ContentType;

struct {
    ContentType type;
    ProtocolVersion legacy_record_version;
    uint16 length;
    opaque fragment[TLSPlaintext.length];
} TLSPlaintext;

struct {
    opaque content[TLSPlaintext.length];
    ContentType type;
    uint8 zeros[length_of_padding];
} TLSInnerPlaintext;

struct {
    ContentType opaque_type = application_data; /* 23 */
    ProtocolVersion legacy_record_version = 0x0303; /* TLS v1.2 */
    uint16 length;
    opaque encrypted_record[TLSCiphertext.length];
} TLSCiphertext;
```

B.2. Alert Messages

```
   enum { warning(1), fatal(2), (255) } AlertLevel;

   enum {
       close_notify(0),
       unexpected_message(10),
       bad_record_mac(20),
       decryption_failed_RESERVED(21),
       record_overflow(22),
       decompression_failure_RESERVED(30),
       handshake_failure(40),
       no_certificate_RESERVED(41),
       bad_certificate(42),
       unsupported_certificate(43),
       certificate_revoked(44),
       certificate_expired(45),
       certificate_unknown(46),
       illegal_parameter(47),
       unknown_ca(48),
       access_denied(49),
       decode_error(50),
       decrypt_error(51),
       export_restriction_RESERVED(60),
       protocol_version(70),
       insufficient_security(71),
       internal_error(80),
       inappropriate_fallback(86),
       user_canceled(90),
       no_renegotiation_RESERVED(100),
       missing_extension(109),
       unsupported_extension(110),
       certificate_unobtainable_RESERVED(111),
       unrecognized_name(112),
       bad_certificate_status_response(113),
       bad_certificate_hash_value_RESERVED(114),
       unknown_psk_identity(115),
       certificate_required(116),
       no_application_protocol(120),
       (255)
   } AlertDescription;

   struct {
       AlertLevel level;
       AlertDescription description;
   } Alert;
```

B.3. Handshake Protocol

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

      struct {
          HandshakeType msg_type;    /* handshake type */
          uint24 length;             /* remaining bytes in message */
          select (Handshake.msg_type) {
              case client_hello:          ClientHello;
              case server_hello:          ServerHello;
              case end_of_early_data:     EndOfEarlyData;
              case encrypted_extensions:  EncryptedExtensions;
              case certificate_request:   CertificateRequest;
              case certificate:           Certificate;
              case certificate_verify:    CertificateVerify;
              case finished:              Finished;
              case new_session_ticket:    NewSessionTicket;
              case key_update:            KeyUpdate;
          };
      } Handshake;
```

B.3.1. Key Exchange Messages

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

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

struct {
    ProtocolVersion legacy_version = 0x0303;    /* TLS v1.2 */
    Random random;
    opaque legacy_session_id<0..32>;
    CipherSuite cipher_suites<2..2^16-2>;
    opaque legacy_compression_methods<1..2^8-1>;
    Extension extensions<8..2^16-1>;
} ClientHello;

struct {
    ProtocolVersion legacy_version = 0x0303;    /* TLS v1.2 */
    Random random;
    opaque legacy_session_id_echo<0..32>;
    CipherSuite cipher_suite;
    uint8 legacy_compression_method = 0;
    Extension extensions<6..2^16-1>;
} ServerHello;

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

enum {
    server_name(0),                             /* RFC 6066 */
    max_fragment_length(1),                     /* RFC 6066 */
    status_request(5),                          /* RFC 6066 */
    supported_groups(10),                       /* RFC 8422, 7919 */
    signature_algorithms(13),                   /* RFC 8446 */
    use_srtp(14),                               /* RFC 5764 */
    heartbeat(15),                              /* RFC 6520 */
    application_layer_protocol_negotiation(16), /* RFC 7301 */
    signed_certificate_timestamp(18),           /* RFC 6962 */
    client_certificate_type(19),                /* RFC 7250 */
    server_certificate_type(20),                /* RFC 7250 */
    padding(21),                                /* RFC 7685 */
    pre_shared_key(41),                         /* RFC 8446 */
    early_data(42),                             /* RFC 8446 */
    supported_versions(43),                     /* RFC 8446 */
    cookie(44),                                 /* RFC 8446 */
    psk_key_exchange_modes(45),                 /* RFC 8446 */
    certificate_authorities(47),                /* RFC 8446 */
    oid_filters(48),                            /* RFC 8446 */
```

```
        post_handshake_auth(49),                   /* RFC 8446 */
        signature_algorithms_cert(50),             /* RFC 8446 */
        key_share(51),                             /* RFC 8446 */
        (65535)
    } ExtensionType;

    struct {
        NamedGroup group;
        opaque key_exchange<1..2^16-1>;
    } KeyShareEntry;

    struct {
        KeyShareEntry client_shares<0..2^16-1>;
    } KeyShareClientHello;

    struct {
        NamedGroup selected_group;
    } KeyShareHelloRetryRequest;

    struct {
        KeyShareEntry server_share;
    } KeyShareServerHello;

    struct {
        uint8 legacy_form = 4;
        opaque X[coordinate_length];
        opaque Y[coordinate_length];
    } UncompressedPointRepresentation;

    enum { psk_ke(0), psk_dhe_ke(1), (255) } PskKeyExchangeMode;

    struct {
        PskKeyExchangeMode ke_modes<1..255>;
    } PskKeyExchangeModes;

    struct {} Empty;

    struct {
        select (Handshake.msg_type) {
            case new_session_ticket:   uint32 max_early_data_size;
            case client_hello:         Empty;
            case encrypted_extensions: Empty;
        };
    } EarlyDataIndication;

    struct {
        opaque identity<1..2^16-1>;
        uint32 obfuscated_ticket_age;
```

```
    } PskIdentity;

    opaque PskBinderEntry<32..255>;

    struct {
        PskIdentity identities<7..2^16-1>;
        PskBinderEntry binders<33..2^16-1>;
    } OfferedPsks;

    struct {
        select (Handshake.msg_type) {
            case client_hello: OfferedPsks;
            case server_hello: uint16 selected_identity;
        };
    } PreSharedKeyExtension;
```

B.3.1.1. Version Extension

```
      struct {
          select (Handshake.msg_type) {
              case client_hello:
                   ProtocolVersion versions<2..254>;

              case server_hello: /* and HelloRetryRequest */
                   ProtocolVersion selected_version;
          };
      } SupportedVersions;
```

B.3.1.2. Cookie Extension

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

B.3.1.3. Signature Algorithm Extension

```
   enum {
       /* RSASSA-PKCS1-v1_5 algorithms */
       rsa_pkcs1_sha256(0x0401),
       rsa_pkcs1_sha384(0x0501),
       rsa_pkcs1_sha512(0x0601),

       /* ECDSA algorithms */
       ecdsa_secp256r1_sha256(0x0403),
       ecdsa_secp384r1_sha384(0x0503),
       ecdsa_secp521r1_sha512(0x0603),

       /* RSASSA-PSS algorithms with public key OID rsaEncryption */
       rsa_pss_rsae_sha256(0x0804),
       rsa_pss_rsae_sha384(0x0805),
       rsa_pss_rsae_sha512(0x0806),

       /* EdDSA algorithms */
       ed25519(0x0807),
       ed448(0x0808),

       /* RSASSA-PSS algorithms with public key OID RSASSA-PSS */
       rsa_pss_pss_sha256(0x0809),
       rsa_pss_pss_sha384(0x080a),
       rsa_pss_pss_sha512(0x080b),

       /* Legacy algorithms */
       rsa_pkcs1_sha1(0x0201),
       ecdsa_sha1(0x0203),

       /* Reserved Code Points */
       obsolete_RESERVED(0x0000..0x0200),
       dsa_sha1_RESERVED(0x0202),
       obsolete_RESERVED(0x0204..0x0400),
       dsa_sha256_RESERVED(0x0402),
       obsolete_RESERVED(0x0404..0x0500),
       dsa_sha384_RESERVED(0x0502),
       obsolete_RESERVED(0x0504..0x0600),
       dsa_sha512_RESERVED(0x0602),
       obsolete_RESERVED(0x0604..0x06FF),
       private_use(0xFE00..0xFFFF),
       (0xFFFF)
   } SignatureScheme;

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

B.3.1.4. Supported Groups Extension

```
enum {
    unallocated_RESERVED(0x0000),

    /* Elliptic Curve Groups (ECDHE) */
    obsolete_RESERVED(0x0001..0x0016),
    secp256r1(0x0017), secp384r1(0x0018), secp521r1(0x0019),
    obsolete_RESERVED(0x001A..0x001C),
    x25519(0x001D), x448(0x001E),

    /* Finite Field Groups (DHE) */
    ffdhe2048(0x0100), ffdhe3072(0x0101), ffdhe4096(0x0102),
    ffdhe6144(0x0103), ffdhe8192(0x0104),

    /* Reserved Code Points */
    ffdhe_private_use(0x01FC..0x01FF),
    ecdhe_private_use(0xFE00..0xFEFF),
    obsolete_RESERVED(0xFF01..0xFF02),
    (0xFFFF)
} NamedGroup;

struct {
    NamedGroup named_group_list<2..2^16-1>;
} NamedGroupList;
```

Values within "obsolete_RESERVED" ranges are used in previous versions of TLS and MUST NOT be offered or negotiated by TLS 1.3 implementations. The obsolete curves have various known/theoretical weaknesses or have had very little usage, in some cases only due to unintentional server configuration issues. They are no longer considered appropriate for general use and should be assumed to be potentially unsafe. The set of curves specified here is sufficient for interoperability with all currently deployed and properly configured TLS implementations.

B.3.2. Server Parameters Messages

```
opaque DistinguishedName<1..2^16-1>;

struct {
    DistinguishedName authorities<3..2^16-1>;
} CertificateAuthoritiesExtension;

struct {
    opaque certificate_extension_oid<1..2^8-1>;
    opaque certificate_extension_values<0..2^16-1>;
} OIDFilter;

struct {
    OIDFilter filters<0..2^16-1>;
} OIDFilterExtension;

struct {} PostHandshakeAuth;

struct {
    Extension extensions<0..2^16-1>;
} EncryptedExtensions;

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

B.3.3. Authentication Messages

```
      enum {
          X509(0),
          OpenPGP_RESERVED(1),
          RawPublicKey(2),
          (255)
      } CertificateType;

      struct {
          select (certificate_type) {
              case RawPublicKey:
                /* From RFC 7250 ASN.1_subjectPublicKeyInfo */
                opaque ASN1_subjectPublicKeyInfo<1..2^24-1>;

              case X509:
                opaque cert_data<1..2^24-1>;
          };
          Extension extensions<0..2^16-1>;
      } CertificateEntry;

      struct {
          opaque certificate_request_context<0..2^8-1>;
          CertificateEntry certificate_list<0..2^24-1>;
      } Certificate;

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

      struct {
          opaque verify_data[Hash.length];
      } Finished;
```

B.3.4. Ticket Establishment

```
      struct {
          uint32 ticket_lifetime;
          uint32 ticket_age_add;
          opaque ticket_nonce<0..255>;
          opaque ticket<1..2^16-1>;
          Extension extensions<0..2^16-2>;
      } 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 cipher suite defines the pair of the AEAD algorithm and hash algorithm to be used with HKDF. Cipher suite names follow the naming convention:

```
CipherSuite TLS_AEAD_HASH = VALUE;
```

Component	Contents
TLS	The string "TLS"
AEAD	The AEAD algorithm used for record protection
HASH	The hash algorithm used with HKDF
VALUE	The two byte ID assigned for this cipher suite

Table 4: Cipher Suite Name Structure

This specification defines the following cipher suites for use with TLS 1.3.

Description	Value
TLS_AES_128_GCM_SHA256	{0x13,0x01}
TLS_AES_256_GCM_SHA384	{0x13,0x02}
TLS_CHACHA20_POLY1305_SHA256	{0x13,0x03}
TLS_AES_128_CCM_SHA256	{0x13,0x04}
TLS_AES_128_CCM_8_SHA256	{0x13,0x05}

Table 5: Cipher Suite List

The corresponding AEAD algorithms AEAD_AES_128_GCM, AEAD_AES_256_GCM, and AEAD_AES_128_CCM are defined in [RFC5116]. AEAD_CHACHA20_POLY1305 is defined in [RFC8439]. AEAD_AES_128_CCM_8 is defined in [RFC6655]. The corresponding hash algorithms are defined in [SHS].

Although TLS 1.3 uses the same cipher suite space as previous versions of TLS, TLS 1.3 cipher suites are defined differently, only specifying the symmetric ciphers, and cannot be used for TLS 1.2. Similarly, cipher suites for TLS 1.2 and lower cannot be used with TLS 1.3.

New cipher suite values are assigned by IANA as described in Section 11.

Appendix C. Implementation Notes

The TLS protocol cannot prevent many common security mistakes. This appendix provides several recommendations to assist implementors. [RFC8448] provides test vectors for TLS 1.3 handshakes.

C.1. Random Number Generation and Seeding

TLS requires a cryptographically secure pseudorandom number generator (CSPRNG). In most cases, the operating system provides an appropriate facility such as /dev/urandom, which should be used absent other (e.g., performance) concerns. It is RECOMMENDED to use an existing CSPRNG implementation in preference to crafting a new one. Many adequate cryptographic libraries are already available under favorable license terms. Should those prove unsatisfactory, [RFC4086] provides guidance on the generation of random values.

TLS uses random values (1) in public protocol fields such as the public Random values in the ClientHello and ServerHello and (2) to generate keying material. With a properly functioning CSPRNG, this does not present a security problem, as it is not feasible to determine the CSPRNG state from its output. However, with a broken CSPRNG, it may be possible for an attacker to use the public output to determine the CSPRNG internal state and thereby predict the keying material, as documented in [CHECKOWAY]. 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)? Do you correctly handle corner cases like a ClientHello that is split into several small fragments? Do you fragment handshake messages that exceed the maximum fragment size? In particular, the Certificate and CertificateRequest handshake messages can be large enough to require fragmentation.

* Do you ignore the TLS record layer version number in all unencrypted TLS records (see Appendix E)?

* Have you ensured that all support for SSL, RC4, EXPORT ciphers, and MD5 (via the "signature_algorithms" extension) is completely removed from all possible configurations that support TLS 1.3 or later, and that attempts to use these obsolete capabilities fail correctly? (see Appendix E)?

* Do you handle TLS extensions in ClientHellos correctly, including unknown extensions?

* When the server has requested a client certificate but no suitable certificate is available, do you correctly send an empty Certificate message, instead of omitting the whole message (see Section 4.4.2)?

* When processing the plaintext fragment produced by AEAD-Decrypt and scanning from the end for the ContentType, do you avoid scanning past the start of the cleartext in the event that the peer has sent a malformed plaintext of all zeros?

* Do you properly ignore unrecognized cipher suites (Section 4.1.2), hello extensions (Section 4.2), named groups (Section 4.2.7), key shares (Section 4.2.8), supported versions (Section 4.2.1), and signature algorithms (Section 4.2.3) in the ClientHello?

* As a server, do you send a HelloRetryRequest to clients which support a compatible (EC)DHE group but do not predict it in the "key_share" extension? As a client, do you correctly handle a HelloRetryRequest from the server?

Cryptographic details:

* What countermeasures do you use to prevent timing attacks [TIMING]?

* When using Diffie-Hellman key exchange, do you correctly preserve leading zero bytes in the negotiated key (see Section 7.4.1)?

* Does your TLS client check that the Diffie-Hellman parameters sent by the server are acceptable (see Section 4.2.8.1)?

* Do you use a strong and, most importantly, properly seeded random number generator (see Appendix C.1) when generating Diffie-Hellman private values, the ECDSA "k" parameter, and other security-critical values? It is RECOMMENDED that implementations implement "deterministic ECDSA" as specified in [RFC6979].

* Do you zero-pad Diffie-Hellman public key values and shared secrets to the group size (see Section 4.2.8.1 and Section 7.4.1)?

* Do you verify signatures after making them, to protect against RSA-CRT key leaks [FW15]?

C.4. Client Tracking Prevention

Clients SHOULD NOT reuse a ticket for multiple connections. Reuse of a ticket allows passive observers to correlate different connections. Servers that issue tickets SHOULD offer at least as many tickets as the number of connections that a client might use; for example, a web browser using HTTP/1.1 [RFC7230] might open six connections to a server. Servers SHOULD issue new tickets with every connection. This ensures that clients are always able to use a new ticket when creating a new connection.

Offering a ticket to a server additionally allows the server to correlate different connections. This is possible independent of ticket reuse. Client applications SHOULD NOT offer tickets across connections that are meant to be uncorrelated. For example, [FETCH] defines network partition keys to separate cache lookups in web browsers.

C.5. Unauthenticated Operation

Previous versions of TLS offered explicitly unauthenticated cipher suites based on anonymous Diffie-Hellman. These modes have been deprecated in TLS 1.3. However, it is still possible to negotiate parameters that do not provide verifiable server authentication by several methods, including:

* Raw public keys [RFC7250].

* Using a public key contained in a certificate but without validation of the certificate chain or any of its contents.

Either technique used alone is vulnerable to man-in-the-middle attacks and therefore unsafe for general use. However, it is also possible to bind such connections to an external authentication mechanism via out-of-band validation of the server's public key, trust on first use, or a mechanism such as channel bindings (though the channel bindings described in [RFC5929] are not defined for TLS 1.3). If no such mechanism is used, then the connection has no protection against active man-in-the-middle attack; applications MUST NOT use TLS in such a way absent explicit configuration or a specific application profile.

Appendix D. Updates to TLS 1.2

To align with the names used this document, the following terms from [RFC5246] are renamed:

* The master secret, computed in Section 8.1 of [RFC5246], is renamed to the main secret. It is referred to as main_secret in formulas and structures, instead of master_secret. However, the label parameter to the PRF function is left unchanged for compatibility.

* The premaster secret is renamed to the preliminary secret. It is referred to as preliminary_secret in formulas and structures, instead of pre_master_secret.

* The PreMasterSecret and EncryptedPreMasterSecret structures, defined in Section 7.4.7.1 of [RFC5246], are renamed to PreliminarySecret and EncryptedPreliminarySecret, respectively.

Correspondingly, the extension defined in [RFC7627] is renamed to the "Extended Main Secret" extension. The extension code point is renamed to "extended_main_secret". The label parameter to the PRF function in Section 4 of [RFC7627] is left unchanged for compatibility.

Appendix E. Backward Compatibility

The TLS protocol provides a built-in mechanism for version negotiation between endpoints potentially supporting different versions of TLS.

TLS 1.x and SSL 3.0 use compatible ClientHello messages. Servers can also handle clients trying to use future versions of TLS as long as the ClientHello format remains compatible and there is at least one protocol version supported by both the client and the server.

Prior versions of TLS used the record layer version number (TLSPlaintext.legacy_record_version and TLSCiphertext.legacy_record_version) for various purposes. As of TLS 1.3, this field is deprecated. The value of TLSPlaintext.legacy_record_version MUST be ignored by all implementations. The value of TLSCiphertext.legacy_record_version is included in the additional data for deprotection but MAY otherwise be ignored or MAY be validated to match the fixed constant value. Version negotiation is performed using only the handshake versions (ClientHello.legacy_version and ServerHello.legacy_version, as well as the ClientHello, HelloRetryRequest, and ServerHello "supported_versions" extensions). In order to maximize interoperability with older endpoints, implementations that negotiate the use of TLS 1.0-1.2 SHOULD set the record layer version number to the negotiated version for the ServerHello and all records thereafter.

For maximum compatibility with previously non-standard behavior and misconfigured deployments, all implementations SHOULD support validation of certification paths based on the expectations in this document, even when handling prior TLS versions' handshakes (see Section 4.4.2.2).

TLS 1.2 and prior supported an "Extended Main Secret" [RFC7627] extension which digested large parts of the handshake transcript into the secret and derived keys. Note this extension was renamed in Appendix D. Because TLS 1.3 always hashes in the transcript up to the server Finished, implementations which support both TLS 1.3 and earlier versions SHOULD indicate the use of the Extended Main Secret extension in their APIs whenever TLS 1.3 is used.

E.1. Negotiating with an Older Server

A TLS 1.3 client who wishes to negotiate with servers that do not support TLS 1.3 will send a normal TLS 1.3 ClientHello containing 0x0303 (TLS 1.2) in ClientHello.legacy_version but with the correct version(s) in the "supported_versions" extension. If the server does not support TLS 1.3, it will respond with a ServerHello containing an older version number. If the client agrees to use this version, the negotiation will proceed as appropriate for the negotiated protocol. A client using a ticket for resumption SHOULD initiate the connection using the version that was previously negotiated.

Note that 0-RTT data is not compatible with older servers and SHOULD NOT be sent absent knowledge that the server supports TLS 1.3. See Appendix E.3.

If the version chosen by the server is not supported by the client (or is not acceptable), the client MUST abort the handshake with a "protocol_version" alert.

Some legacy server implementations are known to not implement the TLS specification properly and might abort connections upon encountering TLS extensions or versions which they are not aware of. Interoperability with buggy servers is a complex topic beyond the scope of this document. Multiple connection attempts may be required in order to negotiate a backward-compatible connection; however, this practice is vulnerable to downgrade attacks and is NOT RECOMMENDED.

E.2. Negotiating with an Older Client

A TLS server can also receive a ClientHello indicating a version number smaller than its highest supported version. If the "supported_versions" extension is present, the server MUST negotiate using that extension as described in Section 4.2.1. If the "supported_versions" extension is not present, the server MUST negotiate the minimum of ClientHello.legacy_version and TLS 1.2. For example, if the server supports TLS 1.0, 1.1, and 1.2, and legacy_version is TLS 1.0, the server will proceed with a TLS 1.0 ServerHello. If the "supported_versions" extension is absent and the server only supports versions greater than ClientHello.legacy_version, the server MUST abort the handshake with a "protocol_version" alert.

Note that earlier versions of TLS did not clearly specify the record layer version number value in all cases (TLSPlaintext.legacy_record_version). Servers will receive various TLS 1.x versions in this field, but its value MUST always be ignored.

E.3. 0-RTT Backward Compatibility

0-RTT data is not compatible with older servers. An older server will respond to the ClientHello with an older ServerHello, but it will not correctly skip the 0-RTT data and will fail to complete the handshake. This can cause issues when a client attempts to use 0-RTT, particularly against multi-server deployments. For example, a deployment could deploy TLS 1.3 gradually with some servers implementing TLS 1.3 and some implementing TLS 1.2, or a TLS 1.3 deployment could be downgraded to TLS 1.2.

A client that attempts to send 0-RTT data MUST fail a connection if it receives a ServerHello with TLS 1.2 or older. It can then retry the connection with 0-RTT disabled. To avoid a downgrade attack, the client SHOULD NOT disable TLS 1.3, only 0-RTT.

To avoid this error condition, multi-server deployments SHOULD ensure a uniform and stable deployment of TLS 1.3 without 0-RTT prior to enabling 0-RTT.

E.4. Middlebox Compatibility Mode

Field measurements [Ben17a] [Ben17b] [Res17a] [Res17b] have found that a significant number of middleboxes misbehave when a TLS client/server pair negotiates TLS 1.3. Implementations can increase the chance of making connections through those middleboxes by making the TLS 1.3 handshake look more like a TLS 1.2 handshake:

* The client always provides a non-empty session ID in the ClientHello, as described in the legacy_session_id section of Section 4.1.2.

* If not offering early data, the client sends a dummy change_cipher_spec record (see the third paragraph of Section 5) immediately before its second flight. This may either be before its second ClientHello or before its encrypted handshake flight. If offering early data, the record is placed immediately after the first ClientHello.

* The server sends a dummy change_cipher_spec record immediately after its first handshake message. This may either be after a ServerHello or a HelloRetryRequest.

When put together, these changes make the TLS 1.3 handshake resemble TLS 1.2 session resumption, which improves the chance of successfully connecting through middleboxes. This "compatibility mode" is partially negotiated: the client can opt to provide a session ID or not, and the server has to echo it. Either side can send change_cipher_spec at any time during the handshake, as they must be ignored by the peer, but if the client sends a non-empty session ID, the server MUST send the change_cipher_spec as described in this appendix.

E.5. Security Restrictions Related to Backward Compatibility

Implementations negotiating the use of older versions of TLS SHOULD prefer forward secret and AEAD cipher suites, when available.

The security of RC4 cipher suites is considered insufficient for the reasons cited in [RFC7465]. Implementations MUST NOT offer or negotiate RC4 cipher suites for any version of TLS for any reason.

Old versions of TLS permitted the use of very low strength ciphers. Ciphers with a strength less than 112 bits MUST NOT be offered or negotiated for any version of TLS for any reason.

The security of SSL 3.0 [RFC6101] 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 F. Overview of Security Properties

A complete security analysis of TLS is outside the scope of this document. In this appendix, we provide an informal description of the desired properties as well as references to more detailed work in the research literature which provides more formal definitions.

We cover properties of the handshake separately from those of the record layer.

F.1. Handshake

The TLS handshake is an Authenticated Key Exchange (AKE) protocol which is intended to provide both one-way authenticated (server-only) and mutually authenticated (client and server) functionality. At the completion of the handshake, each side outputs its view of the following values:

* A set of "session keys" (the various secrets derived from the main secret) from which can be derived a set of working keys.

* A set of cryptographic parameters (algorithms, etc.).

* The identities of the communicating parties.

We assume the attacker to be an active network attacker, which means it has complete control over the network used to communicate between the parties [RFC3552]. Even under these conditions, the handshake should provide the properties listed below. Note that these properties are not necessarily independent, but reflect the protocol consumers' needs.

Establishing the same session keys: The handshake needs to output the same set of session keys on both sides of the handshake, provided that it completes successfully on each endpoint (see [CK01]; Definition 1, part 1).

Secrecy of the session keys: The shared session keys should be known only to the communicating parties and not to the attacker (see [CK01]; Definition 1, part 2). Note that in a unilaterally authenticated connection, the attacker can establish its own session keys with the server, but those session keys are distinct from those established by the client.

Peer Authentication: The client's view of the peer identity should reflect the server's identity. If the client is authenticated, the server's view of the peer identity should match the client's identity.

Uniqueness of the session keys: Any two distinct handshakes should produce distinct, unrelated session keys. Individual session keys produced by a handshake should also be distinct and independent.

Downgrade Protection: The cryptographic parameters should be the same on both sides and should be the same as if the peers had been communicating in the absence of an attack (see [BBFGKZ16]; Definitions 8 and 9).

Forward secret with respect to long-term keys: If the long-term keying material (in this case the signature keys in certificate-based authentication modes or the external/resumption PSK in PSK with (EC)DHE modes) is compromised after the handshake is complete, this does not compromise the security of the session key (see [DOW92]), as long as the session key itself has been erased. The forward secrecy property is not satisfied when PSK is used in the "psk_ke" PskKeyExchangeMode.

Key Compromise Impersonation (KCI) resistance: In a mutually authenticated connection with certificates, compromising the long-term secret of one actor should not break that actor's authentication of their peer in the given connection (see [HGFS15]). For example, if a client's signature key is compromised, it should not be possible to impersonate arbitrary servers to that client in subsequent handshakes.

Protection of endpoint identities: The server's identity (certificate) should be protected against passive attackers. The client's identity (certificate) should be protected against both passive and active attackers. This property does not hold for cipher suites without confidentiality; while this specification does not define any such cipher suites, other documents may do so.

Informally, the signature-based modes of TLS 1.3 provide for the establishment of a unique, secret, shared key established by an (EC)DHE key exchange and authenticated by the server's signature over the handshake transcript, as well as tied to the server's identity by a MAC. If the client is authenticated by a certificate, it also signs over the handshake transcript and provides a MAC tied to both identities. [SIGMA] describes the design and analysis of this type of key exchange protocol. If fresh (EC)DHE keys are used for each connection, then the output keys are forward secret.

The external PSK and resumption PSK bootstrap from a long-term shared secret into a unique per-connection set of short-term session keys. This secret may have been established in a previous handshake. If PSK with (EC)DHE key establishment is used, these session keys will also be forward secret. The resumption PSK has been designed so that the resumption secret computed by connection N and needed to form connection N+1 is separate from the traffic keys used by connection N, thus providing forward secrecy between the connections. In addition, if multiple tickets are established on the same connection, they are associated with different keys, so compromise of the PSK associated with one ticket does not lead to the compromise of connections established with PSKs associated with other tickets. This property is most interesting if tickets are stored in a database (and so can be deleted) rather than if they are self-encrypted.

The PSK binder value forms a binding between a PSK and the current handshake, as well as between the session where the PSK was established and the current session. This binding transitively includes the original handshake transcript, because that transcript is digested into the values which produce the resumption secret. This requires that both the KDF used to produce the resumption secret and the MAC used to compute the binder be collision resistant. See Appendix F.1.1 for more on this. Note: The binder does not cover the binder values from other PSKs, though they are included in the Finished MAC.

Note: TLS does not currently permit the server to send a certificate_request message in non-certificate-based handshakes (e.g., PSK). If this restriction were to be relaxed in future, the client's signature would not cover the server's certificate directly. However, if the PSK was established through a NewSessionTicket, the client's signature would transitively cover the server's certificate through the PSK binder. [PSK-FINISHED] describes a concrete attack on constructions that do not bind to the server's certificate (see also [Kraw16]). It is unsafe to use certificate-based client authentication when the client might potentially share the same PSK/key-id pair with two different endpoints. Implementations MUST NOT combine external PSKs with certificate-based authentication of either the client or server. Future specifications MAY provide an extension to permit this.

If an exporter is used, then it produces values which are unique and secret (because they are generated from a unique session key). Exporters computed with different labels and contexts are computationally independent, so it is not feasible to compute one from another or the session secret from the exported value. Note: Exporters can produce arbitrary-length values; if exporters are to be used as channel bindings, the exported value MUST be large enough to provide collision resistance. The exporters provided in TLS 1.3 are derived from the same Handshake Contexts as the early traffic keys and the application traffic keys, respectively, and thus have similar security properties. Note that they do not include the client's certificate; future applications which wish to bind to the client's certificate may need to define a new exporter that includes the full handshake transcript.

For all handshake modes, the Finished MAC (and, where present, the signature) prevents downgrade attacks. In addition, the use of certain bytes in the random nonces as described in Section 4.1.3 allows the detection of downgrade to previous TLS versions. See [BBFGKZ16] for more details on TLS 1.3 and downgrade.

As soon as the client and the server have exchanged enough information to establish shared keys, the remainder of the handshake is encrypted, thus providing protection against passive attackers, even if the computed shared key is not authenticated. Because the server authenticates before the client, the client can ensure that if it authenticates to the server, it only reveals its identity to an authenticated server. Note that implementations must use the provided record-padding mechanism during the handshake to avoid leaking information about the identities due to length. The client's proposed PSK identities are not encrypted, nor is the one that the server selects.

F.1.1. Key Derivation and HKDF

Key derivation in TLS 1.3 uses HKDF as defined in [RFC5869] and its two components, HKDF-Extract and HKDF-Expand. The full rationale for the HKDF construction can be found in [Kraw10] and the rationale for the way it is used in TLS 1.3 in [KW16]. Throughout this document, each application of HKDF-Extract is followed by one or more invocations of HKDF-Expand. This ordering should always be followed (including in future revisions of this document); in particular, one SHOULD NOT use an output of HKDF-Extract as an input to another application of HKDF-Extract without an HKDF-Expand in between. Multiple applications of HKDF-Expand to some of the same inputs are allowed as long as these are differentiated via the key and/or the labels.

Note that HKDF-Expand implements a pseudorandom function (PRF) with both inputs and outputs of variable length. In some of the uses of HKDF in this document (e.g., for generating exporters and the resumption_secret), it is necessary that the application of HKDF-Expand be collision resistant; namely, it should be infeasible to find two different inputs to HKDF-Expand that output the same value. This requires the underlying hash function to be collision resistant and the output length from HKDF-Expand to be of size at least 256 bits (or as much as needed for the hash function to prevent finding collisions).

F.1.2. Certificate-Based Client Authentication

A client that has sent certificate-based authentication data to a server, either during the handshake or in post-handshake authentication, cannot be sure whether the server afterwards considers the client to be authenticated or not. If the client needs to determine if the server considers the connection to be unilaterally or mutually authenticated, this has to be provisioned by the application layer. See [CHHSV17] for details. In addition, the analysis of post-handshake authentication from [Kraw16] shows that the client identified by the certificate sent in the post-handshake phase possesses the traffic key. This party is therefore the client that participated in the original handshake or one to whom the original client delegated the traffic key (assuming that the traffic key has not been compromised).

F.1.3. 0-RTT

The 0-RTT mode of operation generally provides security properties similar to those of 1-RTT data, with the two exceptions that the 0-RTT encryption keys do not provide full forward secrecy and that the server is not able to guarantee uniqueness of the handshake (non-replayability) without keeping potentially undue amounts of state. See Section 8 for mechanisms to limit the exposure to replay.

F.1.4. Exporter Independence

The exporter_secret and early_exporter_secret are derived to be independent of the traffic keys and therefore do not represent a threat to the security of traffic encrypted with those keys. However, because these secrets can be used to compute any exporter value, they SHOULD be erased as soon as possible. If the total set of exporter labels is known, then implementations SHOULD pre-compute the inner Derive-Secret stage of the exporter computation for all those labels, then erase the [early_]exporter_secret, followed by each inner values as soon as it is known that it will not be needed again.

F.1.5. Post-Compromise Security

TLS does not provide security for handshakes which take place after the peer's long-term secret (signature key or external PSK) is compromised. It therefore does not provide post-compromise security [CCG16], sometimes also referred to as backwards or future secrecy. This is in contrast to KCI resistance, which describes the security guarantees that a party has after its own long-term secret has been compromised.

F.1.6. External References

The reader should refer to the following references for analysis of the TLS handshake: [DFGS15], [CHSV16], [DFGS16], [KW16], [Kraw16], [FGSW16], [LXZFH16], [FG17], and [BBK17].

F.2. Record Layer

The record layer depends on the handshake producing strong traffic secrets which can be used to derive bidirectional encryption keys and nonces. Assuming that is true, and the keys are used for no more data than indicated in Section 5.5, then the record layer should provide the following guarantees:

Confidentiality: An attacker should not be able to determine the plaintext contents of a given record.

Integrity: An attacker should not be able to craft a new record which is different from an existing record which will be accepted by the receiver.

Order protection/non-replayability: An attacker should not be able to cause the receiver to accept a record which it has already accepted or cause the receiver to accept record N+1 without having first processed record N.

Length concealment: Given a record with a given external length, the attacker should not be able to determine the amount of the record that is content versus padding.

Forward secrecy after key change: If the traffic key update mechanism described in Section 4.6.3 has been used and the previous generation key is deleted, an attacker who compromises the endpoint should not be able to decrypt traffic encrypted with the old key.

Informally, TLS 1.3 provides these properties by AEAD-protecting the plaintext with a strong key. AEAD encryption [RFC5116] provides confidentiality and integrity for the data. Non-replayability is provided by using a separate nonce for each record, with the nonce being derived from the record sequence number (Section 5.3), with the sequence number being maintained independently at both sides; thus records which are delivered out of order result in AEAD deprotection failures. In order to prevent mass cryptanalysis when the same plaintext is repeatedly encrypted by different users under the same key (as is commonly the case for HTTP), the nonce is formed by mixing the sequence number with a secret per-connection initialization vector derived along with the traffic keys. See [BT16] for analysis of this construction.

The rekeying technique in TLS 1.3 (see Section 7.2) follows the construction of the serial generator as discussed in [REKEY], which shows that rekeying can allow keys to be used for a larger number of encryptions than without rekeying. This relies on the security of the HKDF-Expand-Label function as a pseudorandom function (PRF). In addition, as long as this function is truly one way, it is not possible to compute traffic keys from prior to a key change (forward secrecy).

TLS does not provide security for data which is communicated on a connection after a traffic secret of that connection is compromised. That is, TLS does not provide post-compromise security/future secrecy/backward secrecy with respect to the traffic secret. Indeed, an attacker who learns a traffic secret can compute all future traffic secrets on that connection. Systems which want such guarantees need to do a fresh handshake and establish a new connection with an (EC)DHE exchange.

F.2.1. External References

The reader should refer to the following references for analysis of the TLS record layer: [BMMRT15], [BT16], [BDFKPPRSZZ16], [BBK17], and [PS18].

F.3. Traffic Analysis

TLS is susceptible to a variety of traffic analysis attacks based on observing the length and timing of encrypted packets [CLINIC] [HCJC16]. This is particularly easy when there is a small set of possible messages to be distinguished, such as for a video server hosting a fixed corpus of content, but still provides usable information even in more complicated scenarios.

TLS does not provide any specific defenses against this form of attack but does include a padding mechanism for use by applications: The plaintext protected by the AEAD function consists of content plus variable-length padding, which allows the application to produce arbitrary-length encrypted records as well as padding-only cover traffic to conceal the difference between periods of transmission and periods of silence. Because the padding is encrypted alongside the actual content, an attacker cannot directly determine the length of the padding, but may be able to measure it indirectly by the use of timing channels exposed during record processing (i.e., seeing how long it takes to process a record or trickling in records to see which ones elicit a response from the server). In general, it is not known how to remove all of these channels because even a constant-time padding removal function will likely feed the content into data-dependent functions. At minimum, a fully constant-time server or client would require close cooperation with the application-layer protocol implementation, including making that higher-level protocol constant time.

Note: Robust traffic analysis defenses will likely lead to inferior performance due to delays in transmitting packets and increased traffic volume.

F.4. Side Channel Attacks

In general, TLS does not have specific defenses against side-channel attacks (i.e., those which attack the communications via secondary channels such as timing), leaving those to the implementation of the relevant cryptographic primitives. However, certain features of TLS are designed to make it easier to write side-channel resistant code:

* Unlike previous versions of TLS which used a composite MAC-then-encrypt structure, TLS 1.3 only uses AEAD algorithms, allowing implementations to use self-contained constant-time implementations of those primitives.

* TLS uses a uniform "bad_record_mac" alert for all decryption errors, which is intended to prevent an attacker from gaining piecewise insight into portions of the message. Additional resistance is provided by terminating the connection on such errors; a new connection will have different cryptographic material, preventing attacks against the cryptographic primitives that require multiple trials.

Information leakage through side channels can occur at layers above TLS, in application protocols and the applications that use them. Resistance to side-channel attacks depends on applications and application protocols separately ensuring that confidential information is not inadvertently leaked.

F.5. Replay Attacks on 0-RTT

Replayable 0-RTT data presents a number of security threats to TLS-using applications, unless those applications are specifically engineered to be safe under replay (minimally, this means idempotent, but in many cases may also require other stronger conditions, such as constant-time response). Potential attacks include:

* Duplication of actions which cause side effects (e.g., purchasing an item or transferring money) to be duplicated, thus harming the site or the user.

* Attackers can store and replay 0-RTT messages in order to reorder them with respect to other messages (e.g., moving a delete to after a create).

* Exploiting cache timing behavior to discover the content of 0-RTT messages by replaying a 0-RTT message to a different cache node and then using a separate connection to measure request latency, to see if the two requests address the same resource.

If data can be replayed a large number of times, additional attacks become possible, such as making repeated measurements of the speed of cryptographic operations. In addition, they may be able to overload rate-limiting systems. For a further description of these attacks, see [Mac17].

Ultimately, servers have the responsibility to protect themselves against attacks employing 0-RTT data replication. The mechanisms described in Section 8 are intended to prevent replay at the TLS layer but do not provide complete protection against receiving multiple copies of client data. TLS 1.3 falls back to the 1-RTT handshake when the server does not have any information about the client, e.g., because it is in a different cluster which does not share state or because the ticket has been deleted as described in Section 8.1. If the application-layer protocol retransmits data in this setting, then it is possible for an attacker to induce message duplication by sending the ClientHello to both the original cluster (which processes the data immediately) and another cluster which will fall back to 1-RTT and process the data upon application-layer replay. The scale of this attack is limited by the client's willingness to retry transactions and therefore only allows a limited amount of duplication, with each copy appearing as a new connection at the server.

If implemented correctly, the mechanisms described in Section 8.1 and Section 8.2 prevent a replayed ClientHello and its associated 0-RTT data from being accepted multiple times by any cluster with consistent state; for servers which limit the use of 0-RTT to one cluster for a single ticket, then a given ClientHello and its associated 0-RTT data will only be accepted once. However, if state is not completely consistent, then an attacker might be able to have multiple copies of the data be accepted during the replication window. Because clients do not know the exact details of server behavior, they MUST NOT send messages in early data which are not safe to have replayed and which they would not be willing to retry across multiple 1-RTT connections.

Application protocols MUST NOT use 0-RTT data without a profile that defines its use. That profile needs to identify which messages or interactions are safe to use with 0-RTT and how to handle the situation when the server rejects 0-RTT and falls back to 1-RTT.

In addition, to avoid accidental misuse, TLS implementations MUST NOT enable 0-RTT (either sending or accepting) unless specifically requested by the application and MUST NOT automatically resend 0-RTT data if it is rejected by the server unless instructed by the application. Server-side applications may wish to implement special processing for 0-RTT data for some kinds of application traffic (e.g., abort the connection, request that data be resent at the application layer, or delay processing until the handshake completes). In order to allow applications to implement this kind of processing, TLS implementations MUST provide a way for the application to determine if the handshake has completed.

F.5.1. Replay and Exporters

Replays of the ClientHello produce the same early exporter, thus requiring additional care by applications which use these exporters. In particular, if these exporters are used as an authentication channel binding (e.g., by signing the output of the exporter) an attacker who compromises the PSK can transplant authenticators between connections without compromising the authentication key.

In addition, the early exporter SHOULD NOT be used to generate server-to-client encryption keys because that would entail the reuse of those keys. This parallels the use of the early application traffic keys only in the client-to-server direction.

F.6. PSK Identity Exposure

Because implementations respond to an invalid PSK binder by aborting the handshake, it may be possible for an attacker to verify whether a given PSK identity is valid. Specifically, if a server accepts both external-PSK and certificate-based handshakes, a valid PSK identity will result in a failed handshake, whereas an invalid identity will just be skipped and result in a successful certificate handshake. Servers which solely support PSK handshakes may be able to resist this form of attack by treating the cases where there is no valid PSK identity and where there is an identity but it has an invalid binder identically.

F.7. Sharing PSKs

TLS 1.3 takes a conservative approach to PSKs by binding them to a specific KDF. By contrast, TLS 1.2 allows PSKs to be used with any hash function and the TLS 1.2 PRF. Thus, any PSK which is used with both TLS 1.2 and TLS 1.3 must be used with only one hash in TLS 1.3, which is less than optimal if users want to provision a single PSK. The constructions in TLS 1.2 and TLS 1.3 are different, although they are both based on HMAC. While there is no known way in which the same PSK might produce related output in both versions, only limited analysis has been done. Implementations can ensure safety from cross-protocol related output by not reusing PSKs between TLS 1.3 and TLS 1.2.

F.8. Attacks on Static RSA

Although TLS 1.3 does not use RSA key transport and so is not directly susceptible to Bleichenbacher-type attacks [Blei98]if TLS 1.3 servers also support static RSA in the context of previous versions of TLS, then it may be possible to impersonate the server for TLS 1.3 connections [JSS15]. TLS 1.3 implementations can prevent this attack by disabling support for static RSA across all versions of TLS. In principle, implementations might also be able to separate certificates with different keyUsage bits for static RSA decryption and RSA signature, but this technique relies on clients refusing to accept signatures using keys in certificates that do not have the digitalSignature bit set, and many clients do not enforce this restriction.

Appendix G. Changes Since -00

[[RFC EDITOR: Please remove in final RFC.]]

* Update TLS 1.2 terminology

* Specify "certificate-based" client authentication

* Clarify that privacy guarantees don't apply when you have null encryption

* Shorten some names

* Address tracking implications of resumption

Contributors

Martin Abadi
University of California, Santa Cruz
abadi@cs.ucsc.edu

Christopher Allen
(co-editor of TLS 1.0)
Alacrity Ventures
ChristopherA@AlacrityManagement.com

Richard Barnes
Cisco

rlb@ipv.sx

Steven M. Bellovin
Columbia University
smb@cs.columbia.edu

David Benjamin
Google
davidben@google.com

Benjamin Beurdouche
INRIA & Microsoft Research
benjamin.beurdouche@ens.fr

Karthikeyan Bhargavan
(editor of [RFC7627])
INRIA
karthikeyan.bhargavan@inria.fr

Simon Blake-Wilson
(co-author of [RFC4492])
BCI
sblakewilson@bcisse.com

Nelson Bolyard
(co-author of [RFC4492])
Sun Microsystems, Inc.
nelson@bolyard.com

Ran Canetti
IBM
canetti@watson.ibm.com

Matt Caswell
OpenSSL
matt@openssl.org

Stephen Checkoway
University of Illinois at Chicago
sfc@uic.edu

Pete Chown
Skygate Technology Ltd
pc@skygate.co.uk

Katriel Cohn-Gordon
University of Oxford
me@katriel.co.uk

Cas Cremers
University of Oxford
cas.cremers@cs.ox.ac.uk

Antoine Delignat-Lavaud
(co-author of [RFC7627])
INRIA
antdl@microsoft.com

Tim Dierks
(co-author of TLS 1.0, co-editor of TLS 1.1 and 1.2)
Independent
tim@dierks.org

Roelof DuToit
Symantec Corporation
roelof_dutoit@symantec.com

Taher Elgamal
Securify
taher@securify.com

Pasi Eronen
Nokia
pasi.eronen@nokia.com

Cedric Fournet
Microsoft
fournet@microsoft.com

Anil Gangolli
anil@busybuddha.org

David M. Garrett
dave@nulldereference.com

Illya Gerasymchuk
Independent
illya@iluxonchik.me

Alessandro Ghedini
Cloudflare Inc.
alessandro@cloudflare.com

Daniel Kahn Gillmor
ACLU
dkg@fifthhorseman.net

Matthew Green
Johns Hopkins University
mgreen@cs.jhu.edu

Jens Guballa
ETAS
jens.guballa@etas.com

Felix Guenther
TU Darmstadt
mail@felixguenther.info

Vipul Gupta
(co-author of [RFC4492])
Sun Microsystems Laboratories
vipul.gupta@sun.com

Chris Hawk
(co-author of [RFC4492])
Corriente Networks LLC
chris@corriente.net

Kipp Hickman

Alfred Hoenes

David Hopwood
Independent Consultant
david.hopwood@blueyonder.co.uk

Marko Horvat
MPI-SWS
mhorvat@mpi-sws.org

Jonathan Hoyland
Royal Holloway, University of London
jonathan.hoyland@gmail.com

Subodh Iyengar
Facebook
subodh@fb.com

Benjamin Kaduk
Akamai Technologies
kaduk@mit.edu

Hubert Kario
Red Hat Inc.

hkario@redhat.com

Phil Karlton
(co-author of SSL 3.0)

Leon Klingele
Independent
mail@leonklingele.de

Paul Kocher
(co-author of SSL 3.0)
Cryptography Research
paul@cryptography.com

Hugo Krawczyk
IBM
hugokraw@us.ibm.com

Adam Langley
(co-author of [RFC7627])
Google
agl@google.com

Olivier Levillain
ANSSI
olivier.levillain@ssi.gouv.fr

Xiaoyin Liu
University of North Carolina at Chapel Hill
xiaoyin.l@outlook.com

Ilari Liusvaara
Independent
ilariliusvaara@welho.com

Atul Luykx
K.U. Leuven
atul.luykx@kuleuven.be

Colm MacCarthaigh
Amazon Web Services
colm@allcosts.net

Carl Mehner
USAA
carl.mehner@usaa.com

Jan Mikkelsen

Transactionware
janm@transactionware.com

Bodo Moeller
(co-author of [RFC4492])
Google
bodo@acm.org

Kyle Nekritz
Facebook
knekritz@fb.com

Erik Nygren
Akamai Technologies
erik+ietf@nygren.org

Magnus Nystrom
Microsoft
mnystrom@microsoft.com

Kazuho Oku
DeNA Co., Ltd.
kazuhooku@gmail.com

Kenny Paterson
Royal Holloway, University of London
kenny.paterson@rhul.ac.uk

Christopher Patton
University of Florida
cjpatton@ufl.edu

Alfredo Pironti
(co-author of [RFC7627])
INRIA
alfredo.pironti@inria.fr

Andrei Popov
Microsoft
andrei.popov@microsoft.com

Marsh Ray
(co-author of [RFC7627])
Microsoft
maray@microsoft.com

Robert Relyea
Netscape Communications

relyea@netscape.com

Kyle Rose
Akamai Technologies
krose@krose.org

Jim Roskind
Amazon
jroskind@amazon.com

Michael Sabin

Joe Salowey
Tableau Software
joe@salowey.net

Rich Salz
Akamai
rsalz@akamai.com

David Schinazi
Apple Inc.
dschinazi@apple.com

Sam Scott
Royal Holloway, University of London
me@samjs.co.uk

Thomas Shrimpton
University of Florida
teshrim@ufl.edu

Dan Simon
Microsoft, Inc.
dansimon@microsoft.com

Brian Smith
Independent
brian@briansmith.org

Brian Sniffen
Akamai Technologies
ietf@bts.evenmere.org

Nick Sullivan
Cloudflare Inc.
nick@cloudflare.com

Bjoern Tackmann
University of California, San Diego
btackmann@eng.ucsd.edu

Tim Taubert
Mozilla
ttaubert@mozilla.com

Martin Thomson
Mozilla
mt@mozilla.com

Hannes Tschofenig
Arm Limited
Hannes.Tschofenig@arm.com

Sean Turner
sn3rd
sean@sn3rd.com

Steven Valdez
Google
svaldez@google.com

Filippo Valsorda
Cloudflare Inc.
filippo@cloudflare.com

Thyla van der Merwe
Royal Holloway, University of London
tjvdmerwe@gmail.com

Victor Vasiliev
Google
vasilvv@google.com

Hoeteck Wee
Ecole Normale Superieure, Paris
hoeteck@alum.mit.edu

Tom Weinstein

David Wong
NCC Group
david.wong@nccgroup.trust

Christopher A. Wood
Apple Inc.

cawood@apple.com

Tim Wright
Vodafone
timothy.wright@vodafone.com

Peter Wu
Independent
peter@lekensteyn.nl

Kazu Yamamoto
Internet Initiative Japan Inc.
kazu@iij.ad.jp

Author's Address

Eric Rescorla
Mozilla

Email: ekr@rtfm.com

Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: 26 August 2021

N. Sullivan
Cloudflare
H. Krawczyk
IBM Research
O. Friel
R. Barnes
Cisco

22 February 2021

OPAQUE with TLS 1.3
draft-sullivan-tls-opaque-01

Abstract

This document describes two mechanisms for enabling the use of the OPAQUE password-authenticated key exchange in TLS 1.3.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

Note that this draft has not received significant security review and should not be the basis for production systems.

OPAQUE [opaque-paper] is a mutual authentication method that enables the establishment of an authenticated cryptographic key between a client and server based on a user's password, without ever exposing the password to servers or other entities other than the client machine and without relying on a Public Key Infrastructure (PKI). OPAQUE leverages a primitive called a Strong symmetric Password Authenticated Key Exchange (Strong aPAKE) to provide desirable properties including resistance to pre-computation attacks in the event of a server compromise.

In some cases, it is desirable to combine password-based authentication with traditional PKI-based authentication as a defense-in-depth measure. For example, in the case of IoT devices, it may be useful to validate that both parties were issued a certificate from a certain manufacturer. Another desirable property for password-based authentication systems is the ability to hide the client's identity from the network. This document describes the use of OPAQUE in TLS 1.3 [TLS13] both as part of the TLS handshake and post-handshake facilitated by Exported Authenticators [I-D.ietf-tls-exported-authenticator], how the different approaches satisfy the above properties and the trade-offs associated with each design.

The in-handshake instantiations of OPAQUE can be used to authenticate a TLS handshake with a password alone, or in conjunction with certificate-based (mutual) authentication but does not provide identity hiding for the client. The Exported Authenticators instantiation of OPAQUE provides client identity hiding by default and allows the application to do password authentication at any time during the connection, but requires PKI authentication for the initial handshake and application-layer semantics to be defined for transporting authentication messages.

2. Conventions and Definitions

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

3. OPAQUE

OPAQUE [opaque-paper] is a Strong Asymmetric Password-Authenticated Key Exchange (Strong aPAKE) built on an oblivious pseudo-random function (OPRF) and authenticated key exchange protocol that is secure against key-compromise impersonation (KCI) attacks. Unlike previous PAKE methods such as SRP [RFC2945] and SPAKE-2 [I-D.irtf-cfrg-spake2], which require a public salt value, a Strong aPAKE leverages the OPRF private key as salt, making it resistant to pre-computation attacks on the password database stored on the server.

TLS 1.3 provides a KCI-secure key agreement algorithm suitable for use with OPAQUE. This document describes two instantiations of OPAQUE in TLS 1.3: one based on digital signatures, called OPAQUE-Sign, and one on Diffie-Hellman key agreement, called OPAQUE-KEX.

OPAQUE consists of two distinct phases: password registration and authentication. We will describe the mechanisms for password registration in this document but it is assumed to have been done outside of a TLS connection. During password registration, the client and server establish a shared set of parameters for future authentication and two private-public key pairs are generated, one for the client and one for the server. The server keeps its private key and stores an encapsulated copy of the client’s key pair along with its own public key in an "envelope" that is encrypted with the result of the OPRF operation. Note that it is possible for the server to use the same key for multiple clients. It may be necessary to permit multiple simultaneous server keys in the even of a key rollover. The client does not store any state nor any PKI information.

In OPAQUE-Sign, the key pairs generated at password registration time are digital signature keys. These signature keys are used in place of certificate keys for both server and client authentication in a TLS handshake. Client authentication is technically optional, though in practice is almost universally required. OPAQUE-Sign cannot be used alongside certificate-based handshake authentication. This instantiation can also be leveraged to do part of a post-handshake authentication using Exported Authenticators [I-D.ietf-tls-exported-authenticator] given an established TLS connection protected with certificate-based authentication.

In OPAQUE-KEX, the key pairs are Diffie-Hellman keys and are used to establish a shared secret that is fed into the key schedule for the handshake. The handshake continues to use Certificate-based authentication and establishes the shared key using Diffie-Hellman. This instantiations is best suited to use cases in which both password and certificate-based authentication are needed during the initial handshake, which is useful in some scenarios. There is no unilateral authentication in this context, mutual authentication is demonstrated explicitly through the finished messages.

4. Password Registration

Password registration is run between a client U and a server S. It is assumed that U can authenticate S during this registration phase (this is the only part in OPAQUE that requires some form of authenticated channel, either physical, out-of-band, PKI-based, etc.) During this phase, clients run the registration flow in [I-D.irtf-cfrg-opaque] using a specific OPAQUE configuration consisting of a tuple (OPRF, Hash, MHF, AKE). The specific AKE is not used during registration. It is only used during login.

During this phase, a specific OPAQUE configuration is chosen, which consists of a tuple (OPRF, Hash, MHF, AKE). See [I-D.irtf-cfrg-opaque] for details about configuration parameters. In this context, AKE is either OPAQUE-Sign or OPAQUE-KEX.

5. Password Authentication

Password authentication integrates TLS into OPAQUE in such a way that clients prove knowledge of a password to servers. In this section, we describe TLS extensions that support this integration for both OPAQUE-KEX and OPAQUE-Sign.

5.1. TLS Extensions

We define several TLS extensions to signal support for OPAQUE and transport the parameters. The extensions used here have a similar structure to those described in Usage of PAKE with TLS 1.3 [I-D.barnes-tls-pake]. The permitted messages that these extensions are allowed and the expected protocol flows are described below.

First, this document specifies extensions used to convey OPAQUE client and server messages, called "opaque_client_auth" and "opaque_server_auth" respectively.

```
enum {
  ...
  opaque_client_auth(TBD),
  opaque_server_auth(TBD),
  (65535)
} ExtensionType;
```

The "opaque_client_auth" extension contains a "PAKEClientAuthExtension" struct and can only be included in the "CertificateRequest" and "Certificate" messages.

```
struct {
  opaque identity<0..2^16-1>;
} PAKEClientAuthExtension;
```

The "opaque_server_auth" extension contains a "PAKEServerAuthExtension" struct and can only be included in the "ClientHello", "EncryptedExtensions", "CertificateRequest" and "Certificate" messages, depending on the type.

```
struct {
  opaque idU<0..2^16-1>;
  CredentialRequest request;
} PAKEShareClient;

struct {
  opaque idS<0..2^16-1>;
  CredentialResponse response;
} PAKEShareServer;

struct {
  select (Handshake.msg_type) {
    ClientHello:
      PAKEShareClient client_shares<0..2^16-1>;
      OPAQUEType types<0..2^16-1>;
    EncryptedExtensions, Certificate:
      PAKEShareServer server_share;
      OPAQUEType type;
  }
} PAKEServerAuthExtension;
```

This document also defines the following set of types;

```
enum {
  OPAQUE-Sign(1),
  OPAQUE-KEX(2),
} OPAQUEType;
```

Servers use PAKEShareClient.idU to index the user's record on the server and create the PAKEShareServer.response. The types field indicates the set of supported auth types by the client. PAKEShareClient.request and PAKEShareServer.response, of type CredentialRequest and CredentialResponse, respectively, are defined in [I-D.irtf-cfrg-opaque].

This document also describes a new CertificateEntry structure that corresponds to an authentication via a signature derived using OPAQUE. This structure serves as a placeholder for the PAKEServerAuthExtension extension.

```
struct {
  select (certificate_type) {
    case OPAQUESign:
      /* Defined in this document */
      opaque null<0>

    case RawPublicKey:
      /* From RFC 7250 ASN.1_subjectPublicKeyInfo */
      opaque ASN1_subjectPublicKeyInfo<1..2^24-1>;

    case X509:
      opaque cert_data<1..2^24-1>;
  };
  Extension extensions<0..2^16-1>;
} CertificateEntry;
```

We request that IANA add an additional type to the "TLS Certificate Types" registry for this OPAQUESign.

Support for the OPAQUESign Certificate type for server authentication can be negotiated using the server_certificate_type [RFC7250] and the Certificate type for client authentication can be negotiated using the client_certificate_type extension [RFC7250].

Note that there needs to be a change to the client_certificate_type row in the IANA "TLS ExtensionType Values" table to allow client_certificate_type extension to be used as an extension to the CertificateRequest message.

6. Use of extensions in TLS handshake flows

6.1. OPAQUE-KEX

In this mode, OPAQUE private keys are used for key agreement algorithm and the result is fed into the TLS key schedule. Password validation is confirmed by the validation of the finished message. These modes can be used in conjunction with optional Certificate-based authentication.

It should be noted that since the identity of the client it is not encrypted as it is sent as an extension to the ClientHello. This may present a privacy problem unless a mechanism like Encrypted Client Hello [ECH] is created to protect it.

Upon receiving a PAKEServerAuth extension, the server checks to see if it has a matching record for this identity. If the record does not exist, the handshake is aborted with a "illegal_parameter" alert. If the record does exist, but the key type of the record does not match any of the supported_groups sent in the key_share extension of the ClientHello, an HRR is sent containing the set of valid key types that it found records for.

Given a matching key_share and an identity with a matching supported_group, the server returns its PAKEServerAuth as an extension to its EncryptedExtensions. Both parties then derive a shared OPAQUE key as follows:

```
 U computes
   K = H(g^y ^ PrivU || PubU ^ x || PubS ^ PrivU || IdU || IdS )
 S computes
   K = H(g^x ^ PrivS || PubS ^ y || PubU ^ PrivS || IdU || IdS )
```

IdU, IdS represent the identities of user (sent as identity in PAKEShareClient) and server (Certificate message). H is the HKDF function agreed upon in the TLS handshake.

The result, K, is then added as an input to the Master Secret in place of the 0 value defined in TLS 1.3. Specifically,

```
0 -> HKDF-Extract = Master Secret
```

becomes

```
K -> HKDF-Extract = Master Secret
```

In this construction, the finished messages cannot be validated unless the OPAQUE computation was done correctly on both sides, authenticating both client and server.

6.2. OPAQUE-Sign

In this modes of operation, the OPAQUE private keys are used for digital signatures and are used to define a new Certificate type and CertificateVerify algorithm. Like the OPAQUE-KEX instantiations above, the identity of the client is sent in the clear in the client’s first flight unless a mechanism like Encrypted Client Hello [ECH] is created to protect it.

Upon receiving a PAKEServerAuth extension, the server checks to see if it has a matching record for this identity. If the record does not exist, the handshake is aborted with a TBD error message. If the record does exist, but the key type of the record does not match any of the supported_signatures sent in the the ClientHello, the handshake must be aborted with a "illegal_parameter" error.

We define a new Certificate message type for an OPAQUE-Sign authenticated handshake.

```
enum {
  X509(0),
  RawPublicKey(2),
  OPAQUE-Sign(3),
  (255)
} CertificateType;
```

Certificates of this type have CertificateEntry structs of the form:

```
struct {
  Extension extensions<0..2^16-1>;
} CertificateEntry;
```

Given a matching signature_scheme and an identity with a matching key type, the server returns a certificate message with type OPAQUE-Sign with PAKEServerAuth as an extension. The private key used in the CertificateVerify message is set to the private key used during account registration, and the client verifies it using the server public key contained in the client's envelope.

It is RECOMMENDED that the server includes a CertificateRequest message with a PAKEClientAuth and the identity originally sent in the PAKEServerAuth extension from the client hello. On receiving a CertificateRequest message with a PAKEClientAuth extension, the client returns a CertificateVerify message signed by PrivC which is validated by the server using PubC.

7. Integration into Exported Authenticators

Neither of the above mechanisms provides privacy for the user during the authentication phase, as the user id is sent in the clear. Additionally, OPAQUE-Sign has the drawback that it cannot be used in conjunction with certificate-based authentication.

It is possible to address both the privacy concerns and the requirement for certificate-based authentication by using OPAQUE-Sign in an Exported Authenticator [I-D.ietf-tls-exported-authenticator] flow, since exported authenticators are sent over a secure channel that is typically established with certificate-based authentication. Using Exported Authenticators for OPAQUE has the additional benefit that it can be triggered at any time after a TLS session has been established, which better fits modern web-based authentication mechanism.

The ClientHello contains PAKEServerAuth, PAKEClientAuth with empty identity values to indicate support for these mechanisms.

1. Client creates Authenticator Request with CR extension PAKEServerAuth.

2. Server creates Exported Authenticator with OPAQUE-Sign (PAKEServerAuth) and CertificateVerify (signed with the OPAQUE private key).

If the server would like to then establish mutual authentication, it can do the following:

1. Server creates Authenticator Request with CH extension PAKEClientAuth (identity)

2. Client creates Exported Authenticator with OPAQUE-Sign Certificate and CertificateVerify (signed with user private key derived from the envelope).

Support for Exported Authenticators is negotiated at the application layer.

8. Summary of properties

Variant \ Property	Identity hiding	Certificate auth	Server-only auth	Post-handshake auth	Minimum round trips
OPAQUE-Sign with EA	yes	yes	yes	yes	2-RTT
OPAQUE-Sign	no	no	yes	no	1-RTT
OPAQUE-KEX	no	no	no	no	1-RTT

Table 1

9. Privacy considerations

TBD: cleartext identity, etc

10. Security Considerations

TODO: protecting against user enumeration

11. IANA Considerations

* Existing IANA references have not been updated yet to point to this document.

  IANA is asked to register a new value in the "TLS Certificate Types" registry of Transport Layer Security (TLS) Extensions (TLS-Certificate-Types-Registry), as follows:

* Value: 4 Description: OPAQUE Authentication Reference: This RFC

Correction request: The client_certificate_type row in the IANA TLS ExtensionType Values table to allow client_certificate_type extension to be used as an extension to the CertificateRequest message.

12. References

12.1. Normative References

[I-D.ietf-tls-exported-authenticator]
Sullivan, N., "Exported Authenticators in TLS", Work in Progress, Internet-Draft, draft-ietf-tls-exported-authenticator-14, 25 January 2021, <https://www.ietf.org/archive/id/draft-ietf-tls-exported-authenticator-14.txt>.

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

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

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

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

12.2. Informative References

[ECH] Rescorla, E., Oku, K., Sullivan, N., and C. A. Wood, "TLS Encrypted Client Hello", Work in Progress, Internet-Draft, draft-ietf-tls-esni-09, 16 December 2020, <https://www.ietf.org/archive/id/draft-ietf-tls-esni-09.txt>.

[I-D.barnes-tls-pake]
Barnes, R. and O. Friel, "Usage of PAKE with TLS 1.3", Work in Progress, Internet-Draft, draft-barnes-tls-pake-04, 16 July 2018, <https://www.ietf.org/archive/id/draft-barnes-tls-pake-04.txt>.

[I-D.irtf-cfrg-opaque]
Krawczyk, H., Lewi, K., and C. A. Wood, "The OPAQUE Asymmetric PAKE Protocol", Work in Progress, Internet-Draft, draft-irtf-cfrg-opaque-03, 21 February 2021, <https://www.ietf.org/archive/id/draft-irtf-cfrg-opaque-03.txt>.

[I-D.irtf-cfrg-spake2]
Ladd, W. and B. Kaduk, "SPAKE2, a PAKE", Work in Progress, Internet-Draft, draft-irtf-cfrg-spake2-18, 17 January 2021, <https://www.ietf.org/archive/id/draft-irtf-cfrg-spake2-18.txt>.

[opaque-paper]
Xu, J., "OPAQUE: An Asymmetric PAKE Protocol Secure Against Pre-Computation Attacks", 2018.

[RFC2945] Wu, T., "The SRP Authentication and Key Exchange System", RFC 2945, DOI 10.17487/RFC2945, September 2000, <https://www.rfc-editor.org/info/rfc2945>.

[RFC5869] Krawczyk, H. and P. Eronen, "HMAC-based Extract-and-Expand Key Derivation Function (HKDF)", RFC 5869, DOI 10.17487/RFC5869, May 2010, <https://www.rfc-editor.org/info/rfc5869>.

Appendix A. Acknowledgments

Authors' Addresses

Nick Sullivan
Cloudflare

Email: nick@cloudflare.com

Hugo Krawczyk
IBM Research

Email: hugo@ee.technion.ac.il

Owen Friel
Cisco

Email: ofriel@cisco.com

Richard Barnes
Cisco

Email: rlb@ipv.sx