Network Working Group D. Benjamin
Internet-Draft Google
Updates: draft-ietf-tls-tls13 (if approved) June 14, 2018
Intended status: Informational
Expires: December 16, 2018

Universal PSKs for TLS
draft-davidben-tls-universal-psk-00

Abstract

This document describes universal PSKs (Pre-Shared Keys) for TLS. Universal PSKs abstract the TLS 1.3 requirement that each PSK can only be used with a single hash function. This allows PSKs to be provisioned without depending on details of the TLS negotiation, which may change as TLS evolves. Additionally, this document describes a compatibility profile for using TLS 1.3 with PSKs provisioned for the TLS 1.2 PSK mechanism.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

TLS 1.3 [I-D.ietf-tls-tls13] provides a PSK mechanism to authenticate connections with symmetric keys provisioned externally to TLS. However, unlike the analogous mechanism in earlier versions of TLS [RFC4279], TLS 1.3 PSKs must be constrained to a single hash function.

While this constraint simplifies the analysis and does not hinder the resumption use case, it is cumbersome for external PSKs. It ties the PSK provisioning process to details of TLS. The application protocol configuring TLS is usually abstracted from TLS's details. In some cases, the underlying TLS implementation may even be updated without changes to the calling application.

Additionally, applications using TLS with PSKs typically require some PSK be negotiated, so parameter selection must follow the hash constraint. In contrast, applications using resumption typically allow the session to be declined in favor of a full handshake, so parameter selection may complete independently of this constraint. Switching the order of the selections for external PSKs adds implementation complexity and complicates analysis of the server's configuration.

This document resolves these issues by adding an extra key derivation step to reuse the same secret for all TLS 1.3 KDF hashes, including hashes to be defined in the future.

1.1. Requirements Language

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

2. Universal PSKs

A universal PSK consists of the following:

- An identity. This is a public opaque byte string.
- A secret. This is a secret opaque byte string.
- A KDF hash function for use with HKDF [RFC5869]. Unless otherwise specified, this is SHA-256 [SHS].

In this section's diagrams, "0" refers to a string of zero bytes with length matching the KDF hash. Derive-Secret refers to the corresponding function defined in TLS 1.3, using the KDF hash.

A universal PSK is advertised in TLS 1.3 by including the identity and index in the "pre_shared_key" extension of the ClientHello and ServerHello, respectively. The binder value is computed as in TLS 1.3, however, the binder key is derived with the universal PSK's secret and KDF hash as follows:

```
extracted = HKDF-Extract(universal_psk, 0)
binder_key = Derive-Secret(extracted, "univ binder", identity)
```

Unlike other PSKs, a universal PSK may be negotiated with any cipher suite, including those using a different KDF hash than the PSK. When negotiated, the universal PSK's secret is used to derive a hash-specific TLS 1.3 PSK as follows:

If the negotiated cipher suite uses a SHA-256 KDF hash, the PSK is derived as follows:

```
extracted = HKDF-Extract(universal_psk, 0)
psk = Derive-Secret(extracted, "sha256 psk", identity)
```

If the negotiated cipher suite uses a SHA-384 KDF hash, the PSK is derived as follows:

```
extracted = HKDF-Extract(universal_psk, 0)
psk = Derive-Secret(extracted, "sha384 psk", identity)
```

These PSKs are used in the key schedule as specified in TLS 1.3, except that they are not used to derive the "binder_key" value, already derived above.

Future KDF hash algorithms added to TLS 1.3 MUST specify how to compute the derived PSK from a universal PSK. Future versions of TLS MUST specify how to negotiate a universal PSK and how to use it when negotiated. Note, however, all versions of TLS using the "pre_shared_key" extension to negotiate PSKs MUST use the same binder derivation, while the derived PSKs SHOULD be version-specific.

Universal PSKs are not defined for use with 0-RTT. 0-RTT requires specifying many negotiated TLS parameters, which is not compatible with the goals of this specification. However, a client MAY choose to offer a universal PSK alongside a resumption-based or other 0-RTT-compatible PSK. The universal PSK is then analogous to the full handshake option when resumption is declined.

Note that whether a PSK is a universal PSK is not explicitly negotiated in TLS. It is provisioned alongside the secret itself when the PSK is pre-shared. This would typically be specified in the application protocol.

3. Compatibility with TLS 1.2 PSKs

Universal PSKs are only defined for use with TLS 1.3 and future versions of TLS. New protocols using TLS and PSKs SHOULD require TLS 1.3 or later. However, this may not be possible for existing protocols already using PSKs with TLS 1.2. This section describes a compatibility profile for upgrading to TLS 1.3.

A PSK provisioned for TLS 1.2 and earlier MUST NOT be used either as a universal PSK secret or directly as a TLS 1.3 PSK. This would invalidate security analysis of the two protocols individually. Instead, these PSKs MAY be used to derive a universal PSK. The identity is the TLS 1.2 PSK's identity. The secret is derived using the TLS 1.2 PRF function described in Section 5 of [RFC5246] with SHA-256 as the hash function, as follows:

```
universal_psk = PRF(pre_master_secret, "universal psk", "")[:32]
```

"pre_master_secret" is specified with the structure below, setting "psk" to TLS 1.2 PSK and "other_secret" to a string of all zeroes of the same length as the TLS 1.2 PSK.

```
struct {
  opaque other_secret<0..2^16-1>
  opaque psk<0..2^16-1>
}
```

Note this encoding and derivation aligns with the PSK's conversion to a premaster secret and then a master secret in [RFC5246].

Applications using this derivation are necessarily impacted by portions of TLS 1.2. New applications without a TLS 1.2 legacy SHOULD NOT use this derivation and instead SHOULD provision universal PSKs directly. Applications using it SHOULD migrate to this state after migrating to TLS 1.3.

4. Security Considerations

The security analysis for TLS 1.3 relies on each PSK having a single use. Using a TLS 1.3 PSK with two different hashes or with TLS 1.2 means the same secret is used with different KDF functions, invalidating that analysis. Universal PSKs instead derive independent PSKs using different KDF labels, so each derived PSK continues to have only a single use. The PSK identity is additionally included in each derivation to give a stronger connection between the identity and PSK.

TLS 1.3's analysis also depends on the KDF and MAC used to compute the PSK binder being collision-resistant. This document uses the same derivation as TLS 1.3, but with a different label and initial secret, so the collision-resistance properties carry over.

In [RFC5246], TLS 1.2 PSKs are used in premaster secret to master secret derivation. Section 3 aligns with that derivation, using a different label so the secret is derived independently. Note, however, that TLS 1.2 PSKs are not always associated with a single hash function, so they depend on stronger assumptions about hash functions than TLS 1.3 PSKs. The compatibility derivation is unavoidably dependent on this as well. It uses SHA-256, but some TLS 1.2 cipher suites use SHA-384, and earlier versions of TLS use an MD5 and SHA-1 concatenation.

Additionally, labels in the TLS 1.2 PRF function are not delimited from the seed parameter when concatenated. The labels in use thus must not only be distinct, but also prefix-free. This document registers its new TLS 1.2 label in the TLS Exporter Label registry. This registry is required by [RFC5705] to be prefix-free.

5. IANA Considerations

This document updates the note in the TLS Exporter Label registry <https://www.iana.org/assignments/tls-parameters> to read as follows:

Note: (1) These entries are reserved and MUST NOT be used for the purpose described in RFC 5705, in order to avoid confusion with similar, but distinct, use in the referenced document.

It additionally registers the label "universal psk". The "Note" column is marked with (1).

6. Acknowledgements

The author would like to thank Karthikeyan Bhargavan, Matt Caswell, Eric Rescorla, and Victor Vasiliev for discussions and feedback which led to this design.

7. Normative References

[I-D.ietf-tls-tls13]
Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", draft-ietf-tls-tls13-28 (work in progress), March 2018.

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

[RFC4279] Eronen, P., Ed. and H. Tschofenig, Ed., "Pre-Shared Key Ciphersuites for Transport Layer Security (TLS)", RFC 4279, DOI 10.17487/RFC4279, December 2005, <https://www.rfc-editor.org/info/rfc4279>.

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

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

[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>.

[SHS] Dang, Q., "Secure Hash Standard", National Institute of Standards and Technology report, DOI 10.6028/nist.fips.180-4, July 2015.

Author's Address

David Benjamin
Google
355 Main St
Cambridge, MA 02142
USA

Email: davidben@google.com

Network Working Group R. Housley
Internet-Draft Vigil Security
Intended status: Standards Track November 8, 2018
Expires: May 12, 2019

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

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

Abstract

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

Status of This Memo

Copyright Notice

1. Introduction

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

2. Terminology

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

3. Motivation and Design Rationale

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

The TLS 1.3 handshake protocol employs key agreement algorithms that could be broken by the invention of a large-scale quantum computer [I-D.hoffman-c2pq]. These algorithms include Diffie-Hellman (DH) [DH] and Elliptic Curve Diffie-Hellman (ECDH) [IEEE1363]. As a result, an adversary that stores a TLS 1.3 handshake protocol exchange today could decrypt the associated encrypted communications in the future when a large-scale quantum computer becomes available.

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

4. Extension Overview

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

The client includes the "tls_cert_with_extern_psk" extension in the ClientHello message. The "tls_cert_with_extern_psk" extension MUST accompanied by the "key_share", "psk_key_exchange_modes", and "pre_shared_key" extensions. The "pre_shared_key" extension MUST be the last extension in the ClientHello message, and it provides a list of external PSK identifiers that the client is willing to use with this server. Since tls_cert_with_extern_psk" extension is intended to be used only with initial handshakes, it MUST NOT be sent alongside the "early_data" extension. These extension are all described in Section 4.2 of [RFC8446].

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

The successful negotiation of the "tls_cert_with_extern_psk" extension requires the TLS 1.3 key schedule processing to include both the selected external PSK and the (EC)DHE shared secret value. As a result, the Early Secret, Handshake Secret, and Master Secret values all depend upon the value of the selected external PSK.

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

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

5. Certificate with External PSK Extension

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

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

Many server extensions are carried in the EncryptedExtensions message; however, the "tls_cert_with_extern_psk" extension is carried in the ServerHello message. It is only present in the ServerHello message if the server recognizes the "tls_cert_with_extern_psk" extension and the server possesses one of the external PSKs offered by the client in the "pre_shared_key" extension in the ClientHello message.

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

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

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

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

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

The "tls_cert_with_extern_psk" extension is relevant when the client and server possess an external PSK in common that can be used as an input to the TLS 1.3 key schedule.

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

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

The "psk_key_exchange_modes" extension is defined in Section 4.2.9 of [RFC8446]. The "psk_key_exchange_modes" extension restricts both the use of PSKs offered in this ClientHello and those which the server might supply via a subsequent NewSessionTicket. As a result, clients MUST include the psk_dhe_ke mode, and clients MAY also include the psk_ke mode to support a subsequent NewSessionTicket. Servers MUST select the psk_dhe_ke mode for the initial handshake. Servers MUST select a key exchange mode that is listed by the client for subsequent handshakes that include the resumption PSK from the initial handshake.

The "supported_groups" extension is defined in Section 4.2.7 of [RFC8446].

The "pre_shared_key" extension is defined in Section 4.2.11 of [RFC8446]. the syntax is repeated below for convenience. All of the listed PSKs MUST be external PSKs.

```
struct {
    opaque identity<1..2^16-1>;
    uint32 obfuscated_ticket_age;
} PskIdentity;

opaque PskBinderEntry<32..255>;

struct {
    PskIdentity identities<7..2^16-1>;
    PskBinderEntry binders<33..2^16-1>;
} OfferedPsks;

struct {
    select (Handshake.msg_type) {
        case client_hello: OfferedPsks;
        case server_hello: uint16 selected_identity;
    };
} PreSharedKeyExtension;
```

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

The identities are a list of external PSK identities that the client is willing to negotiate with the server. Each external PSK has an associated identity that is known to the client and the server. (The identity is also referred to as an identifier or a label.)

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

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

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

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

TLS 1.3 does not permit the server to send a CertificateRequest message when a PSK is being used. This restriction is removed when the "tls_cert_with_extern_psk" extension is negotiated, allowing the certificate-based authentication for both the client and the server. If certificate-based client authentication is desired, this is accomplished by the client sending the Certificate and CertificateVerify messages as described in Sections 4.4.2 and 4.4.3 of [RFC8446].

Section 7.1 of [RFC8446] specifies the TLS 1.3 Key Schedule. The successful negotiation of the "tls_cert_with_extern_psk" extension requires the key schedule processing to include both the external PSK and the (EC)DHE shared secret value.

If the client and the server have different values associated with the selected external PSK identifier, then the client and the server will compute different values for every entry in the key schedule, which will lead to the termination of the connection with a "decrypt_error" alert.

6. IANA Considerations

IANA is requested to update the TLS ExtensionType Registry to include "tls_cert_with_extern_psk" with a value of TBD and the list of messages "CH, SH" in which the "tls_cert_with_extern_psk" extension may appear.

7. Security Considerations

The Security Considerations in [RFC8446] remain relevant.

TLS 1.3 [RFC8446] does not permit the server to send a CertificateRequest message when a PSK is being used. This restriction is removed when the "tls_cert_with_extern_psk" extension is offered by the client and accepted by the server. However, TLS 1.3 does not permit an external PSK to be used in the same fashion as a resumption PSK, and this extension does not alter those restrictions. Thus, a certificate MUST NOT be used with a resumption PSK.

Implementations must protect the external pre-shared key (PSK). Compromise of the external PSK will make the encrypted session content vulnerable to the future invention of a large-scale quantum computer.

Implementers should not transmit the same content on a connection that is protected with an external PSK and a connection that is not. Doing so may allow an eavesdropper to correlate the connections, making the content vulnerable to the future invention of a large-scale quantum computer.

Implementations must choose external PSKs with a secure key management technique, such as pseudo-random generation of the key or derivation of the key from one or more other secure keys. The use of inadequate pseudo-random number generators (PRNGs) to generate external PSKs can result in little or no security. An attacker may find it much easier to reproduce the PRNG environment that produced the external PSKs and searching the resulting small set of possibilities, rather than brute force searching the whole key space. The generation of quality random numbers is difficult. [RFC4086] offers important guidance in this area.

TLS 1.3 [RFC8446] takes a conservative approach to PSKs; they are bound to a specific hash function and KDF. By contrast, TLS 1.2 [RFC5246] allows PSKs to be used with any hash function and the TLS 1.2 PRF. Thus, the safest approach is to use a PSK with either TLS 1.2 or TLS 1.3. However, any PSK that might be used with both TLS 1.2 and TLS 1.3 must be used with only one hash function, which is the one that is bound for use in TLS 1.3. This restriction is less than optimal when users want to provision a single PSK. While the constructions used in TLS 1.2 and TLS 1.3 are both based on HMAC [RFC2104], the constructions are different, and there is no known way in which reuse of the same PSK in TLS 1.2 and TLS 1.3 that would produce related outputs.

8. Acknowledgments

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

9. References

9.1. Normative References

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

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

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

9.2. Informative References

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

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

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

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

[RFC4086] Eastlake 3rd, D., Schiller, J., and S. Crocker, "Randomness Requirements for Security", BCP 106, RFC 4086, DOI 10.17487/RFC4086, June 2005, <https://www.rfc-editor.org/info/rfc4086>.

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

Author's Address

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

Email: housley@vigilsec.com

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

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

Abstract

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

Status of This Memo

Copyright Notice

Table of Contents

1. Requirements Notation

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

2. Introduction

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

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

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

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

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

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

3. DNSSEC Authentication Chain Extension

3.1. Protocol, TLS 1.2

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

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

3.2. Protocol, TLS 1.3

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

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

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

3.3. Raw Public Keys

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

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

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

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

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

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

3.4. DNSSEC Authentication Chain Data

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

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

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

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

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

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

where RR(i) denotes the ith RR.

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

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

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

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

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

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

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

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

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

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

4. Construction of Serialized Authentication Chains

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

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

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

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

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

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

5. Caching and Regeneration of the Authentication Chain

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

6. Verification

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

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

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

7. Trust Anchor Maintenance

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

8. Mandating use of this extension

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

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

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

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

9. DANE and Traditional PKIX Interoperation

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

10. Security Considerations

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

11. IANA Considerations

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

12. Acknowledgments

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

13. References

13.1. Normative References

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

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

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

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

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

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

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

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

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

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

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

[TLS13] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", March 2018, <https://tools.ietf.org/html/draft-ietf-tls-tls13>.

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

13.2. Informative References

[RFC3261] Rosenberg, J., Schulzrinne, H., Camarillo, G., Johnston, A., Peterson, J., Sparks, R., Handley, M., and E. Schooler, "SIP: Session Initiation Protocol", RFC 3261, DOI 10.17487/RFC3261, June 2002, <https://www.rfc-editor.org/info/rfc3261>.

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

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

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

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

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

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

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

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

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

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

Appendix A. Test vectors

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


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


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

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

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

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

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

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

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

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

```
_443._tcp.www.example.com.  3600  IN  TLSA  ( 3 1 1
        c66bef6a5c1a3e78b82016e13f314f3cc5fa25b1e52aab9adb9ec5989b165
        ada )
_443._tcp.www.example.com.  3600  IN  RRSIG  ( TLSA 13 5 3600
        20181128000000 20151103000000 1870 example.com.
        uml1DUjp5RfrXn9WtuMxEQV+ygzrONcuzsnyfOGSszwaDdkSOJ0Kndcfbb2Il
        LUV04Z+V488+Sd1jr7/21tsKA== )
example.com.  3600  IN  DNSKEY  ( 257 3 13
        JnA1XgyJTZz+psWvbrfUWLV6ULqIJyUS2CQdhUH9VK35bslWeJpRzrlxCUs7s
        /TsSfZMaGWVvlsuieh5nHcXzA== ) ; Key ID = 1870
example.com.  3600  IN  RRSIG  ( DNSKEY 13 2 3600
        20181128000000 20151103000000 1870 example.com.
        HujA9vQTbCxMeaYjDOCF0fYyHhajTl5xPztrp5u6P2vYV8naYQLG3zUF1gaer
        WBOagXXblaSSbYwB96LU3uSdg== )
example.com.  900  IN  DS  ( 1870 13 2 e9b533a049798e900b5c29c90cd25a
        986e8a44f319ac3cd302bafc08f5b81e16 )
example.com.  900  IN  RRSIG  ( DS 13 2 900 20181128000000
        20151103000000 34327 com.
        1tua9ntAqZvOnK5UztzIjN38Bqs6mJ8KAT7L4+AxevDL+z0Jft7RC1/g6Qrfa
        In1wqF4U7TvC8PYOD0U/HYtwQ== )
com.  900  IN  DNSKEY  ( 256 3 13
        7IIE5Dol8jSMUqHTvOOiZapdEbQ9wqRxFi/zQcSdufUKLhpByvLpzSAQTqCWj
        3URIZ8L3Fa2gBLMOZUzZ1GQCw== ) ; Key ID = 34327
com.  900  IN  DNSKEY  ( 257 3 13
        RbkcO+96XZmnp8jYIuM4lryAp3egQjSmBaSoiA7H76Tm0RLHPNPUxlVk+nQ0f
        Ic3I8xfZDNw8Wa0Pe3/g2QA/w== ) ; Key ID = 18931
com.  900  IN  DNSKEY  ( 257 3 13
        szc7biLo5J4OHlkan1vZrF4aD4YYf+NHA/GAqdNslY9xxK9Izg68XHkqck4Rt
        DiVk37lNAQmgSlHbrGu0yOTkA== ) ; Key ID = 28809
com.  900  IN  RRSIG  ( DNSKEY 13 1 900 20181128000000
```

```
         20151103000000 18931 com.
         lZmTBrfcRgVbqHJIfCVr6c3HUDgy3MlNSCSnrVV2S5/NmB3ZiFcvIDn0iqXPm
         7YQfvfWi6utyxBu/fSD6S1ARw== )
com.  900  IN  RRSIG  ( DNSKEY 13 1 900 20181128000000
         20151103000000 28809 com.
         8qZOVM4X8wGt5XPWhG2HO4FAD6Kvs5eIhZUz+7DVCrZ/XMEVrMIHcm1Q+sq0s
         hm4cSivK2BxOO24PHJXoZN2Lw== )
com.  86400  IN  DS  ( 18931 13 2 20f7a9db42d0e2042fbbb9f9ea015941202
         f9eabb94487e658c188e7bcb52115 )
com.  86400  IN  DS  ( 28809 13 2 ad66b3276f796223aa45eda773e92c6d98e
         70643bbde681db342a9e5cf2bb380 )
com.  86400  IN  RRSIG  ( DS 13 1 86400 20181128000000
         20151103000000 31918 .
         5KQVa0NP+6k7VEGMmeky2/Y3wIGM70Fkm0vp5NmQ6KPk8L1XMJPltcJDWGGjc
         EU3Uc4z2DUxzZyWgEDdrSOcdw== )
.  86400  IN  DNSKEY  ( 256 3 13
         zKz+DCWkNA/vuheiVPcGqsH40U84KZAlrMRIyozj9WHzf8PsFp/oR8j8vmjjW
         P98cbte4d8NvlGLxzbUzo3+FA== ) ; Key ID = 31918
.  86400  IN  DNSKEY  ( 256 3 13
         8wMZZ4lzHdyKZ4fv8kys/t3QMlgvEadbsbyqWrMhwddSXCZYGRrsAbPpireRW
         xbVcd1VtOrlFBcRDMTN0R0XEQ== ) ; Key ID = 2635
.  86400  IN  DNSKEY  ( 257 3 13
         yvX+VNTUjxZiGvtr060hVbrPV9H6rVusQtF9lIxCFzbZOJxMQBFmbqlc8Xclv
         Q+gDOXnFOTsgs/frMmxyGOtRg== ) ; Key ID = 47005
.  86400  IN  RRSIG  ( DNSKEY 13 0 86400 20181128000000
         20151103000000 47005 .
         ehAzuZD3yT0pShXkKavrMdz+DKvvFvbZ+sGRZ5iQTni+ulMzZxHQ5+kSha65B
         Y2AIUphjyWcGr6VwP3Ne74iZA== )
```

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


```
0000:  04 5f 34 34 33 04 5f 74  63 70 03 77 77 77 07 65
0010:  78 61 6d 70 6c 65 03 63  6f 6d 00 00 34 00 01 00
0020:  00 0e 10 00 23 03 01 01  c6 6b ef 6a 5c 1a 3e 78
0030:  b8 20 16 e1 3f 31 4f 3c  c5 fa 25 b1 e5 2a ab 9a
0040:  db 9e c5 98 9b 16 5a da  04 5f 34 34 33 04 5f 74
0050:  63 70 03 77 77 77 07 65  78 61 6d 70 6c 65 03 63
0060:  6f 6d 00 00 2e 00 01 00  00 0e 10 00 5f 00 34 0d
0070:  05 00 00 0e 10 5b fd da  80 56 37 f9 00 07 4e 07
0080:  65 78 61 6d 70 6c 65 03  63 6f 6d 00 ba 69 75 0d
0090:  48 e9 e5 17 eb 5e 7f 56  b6 e3 31 11 05 7e ca 0c
00a0:  eb 38 d7 2e ce c9 f2 7c  e1 92 b3 3c 1a 0d d9 12
00b0:  38 9d 0a 9d d7 1f 6d bd  88 94 b5 15 d3 86 7e 57
00c0:  8f 3c f9 27 75 8e be ff  db 5b 6c 28 07 65 78 61
00d0:  6d 70 6c 65 03 63 6f 6d  00 00 30 00 01 00 00 0e
00e0:  10 00 44 01 01 03 0d 26  70 35 5e 0c 89 4d 9c fe
00f0:  a6 c5 af 6e b7 d4 58 b5  7a 50 ba 88 27 25 12 d8
```

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

```
0400:  71 38 ed b8 3c 72 57 a1  93 76 2f 03 63 6f 6d 00
0410:  00 2b 00 01 00 01 51 80  00 24 49 f3 0d 02 20 f7
0420:  a9 db 42 d0 e2 04 2f bb  b9 f9 ea 01 59 41 20 2f
0430:  9e ab b9 44 87 e6 58 c1  88 e7 bc b5 21 15 03 63
0440:  6f 6d 00 00 2b 00 01 00  01 51 80 00 24 70 89 0d
0450:  02 ad 66 b3 27 6f 79 62  23 aa 45 ed a7 73 e9 2c
0460:  6d 98 e7 06 43 bb de 68  1d b3 42 a9 e5 cf 2b b3
0470:  80 03 63 6f 6d 00 00 2e  00 01 00 01 51 80 00 53
0480:  00 2b 0d 01 00 01 51 80  5b fd da 80 56 37 f9 00
0490:  7c ae 00 e4 a4 15 6b 43  4f fb a9 3b 54 41 8c 99
04a0:  e9 32 db f6 37 c0 81 8c  ef 41 64 9b 4b e9 e4 d9
04b0:  90 e8 a3 e4 f0 bd 57 30  93 e5 b5 c2 43 58 61 a3
04c0:  70 45 37 51 ce 33 d8 35  31 cd 9c 96 80 40 dd ad
04d0:  23 9c 77 00 00 30 00 01  00 01 51 80 00 44 01 00
04e0:  03 0d cc ac fe 0c 25 a4  34 0f ef ba 17 a2 54 f7
04f0:  06 aa c1 f8 d1 4f 38 29  90 25 ac c4 48 ca 8c e3
0500:  f5 61 f3 7f c3 ec 16 9f  e8 47 c8 fc be 68 e3 58
0510:  ff 7c 71 bb 5e e1 df 0d  be 51 8b c7 36 d4 ce 8d
0520:  fe 14 00 00 30 00 01 00  01 51 80 00 44 01 00 03
0530:  0d f3 03 19 67 89 73 1d  dc 8a 67 87 ef f2 4c ac
0540:  fe dd d0 32 58 2f 11 a7  5b b1 bc aa 5a b3 21 c1
0550:  d7 52 5c 26 58 19 1a ec  01 b3 e9 8a b7 91 5b 16
0560:  d5 71 dd 55 b4 ea e5 14  17 11 0c c4 cd d1 1d 17
0570:  11 00 00 30 00 01 00 01  51 80 00 44 01 01 03 0d
0580:  ca f5 fe 54 d4 d4 8f 16  62 1a fb 6b d3 ad 21 55
0590:  ba cf 57 d1 fa ad 5b ac  42 d1 7d 94 8c 42 17 36
05a0:  d9 38 9c 4c 40 11 66 6e  a9 5c f1 77 25 bd 0f a0
05b0:  0c e5 e7 14 e4 ec 82 cf  df ac c9 b1 c8 63 ad 46
05c0:  00 00 2e 00 01 00 01 51  80 00 53 00 30 0d 00 00
05d0:  01 51 80 5b fd da 80 56  37 f9 00 b7 9d 00 7a 10
05e0:  33 b9 90 f7 c9 3d 29 4a  15 e4 29 ab eb 31 dc fe
05f0:  0c ab ef 16 f6 d9 fa c1  91 67 98 90 4e 78 be ba
0600:  53 33 67 11 d0 e7 e9 12  85 ae b9 05 8d 80 21 4a
0610:  61 8f 25 9c 1a be 95 c0  fd cd 7b be 22 64
```

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

```
_25._tcp.example.com.  3600  IN  TLSA  ( 3 1 1
        c66bef6a5c1a3e78b82016e13f314f3cc5fa25b1e52aab9adb9ec5989b165
        ada )
_25._tcp.example.com.  3600  IN  RRSIG  ( TLSA 13 3 3600
        20181128000000 20151103000000 1870 example.com.
        e7Q5L2x7Ca3SkSY6pRjqgtRxkEN1uYUcgyMlPp6GQ4zxAZxoO1Y1vGqxN4eNA
        +yBnlUSIJQ46KKVS5PC79Qipg== )
*._tcp.example.com.  3600  IN  NSEC  (
        _443._tcp.www.example.com. RRSIG NSEC TLSA )
*._tcp.example.com.  3600  IN  RRSIG  ( NSEC 13 3 3600
```

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

```
          yvX+VNTUjxZiGvtr060hVbrPV9H6rVusQtF9lIxCFzbZOJxMQBFmbqlc8Xclv
          Q+gDOXnFOTsgs/frMmxyGOtRg== ) ; Key ID = 47005
   .  86400  IN  RRSIG  ( DNSKEY 13 0 86400 20181128000000
          20151103000000 47005 .
          ehAzuZD3yT0pShXkKavrMdz+DKvvFvbZ+sGRZ5iQTni+ulMzZxHQ5+kSha65B
          Y2AIUphjyWcGr6VwP3Ne74iZA== )
```

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

```
   _443._tcp.www.example.org.  3600  IN  CNAME  (
          dane311.example.org. )
   _443._tcp.www.example.org.  3600  IN  RRSIG  ( CNAME 13 5 3600
          20181128000000 20151103000000 56566 example.org.
          wLQYbRNMqrXCD65GZJqwwsD0TDF2VQTklBYdYCMo+JTjqvZw1UFYmcJXmwJsL
          KezLIzSdKW6jK0LMJ3YUw3Bmw== )
   dane311.example.org.  3600  IN  TLSA  ( 3 1 1
          c66bef6a5c1a3e78b82016e13f314f3cc5fa25b1e52aab9adb9ec5989b165
          ada )
   dane311.example.org.  3600  IN  RRSIG  ( TLSA 13 3 3600
          20181128000000 20151103000000 56566 example.org.
          AllKVcpLz/9vG/xJQFwWEK0cHbjO6lI65ELWSoWxPvYJ5o8QnSbRkzfCM4lTs
          g94s5VvzMLYIbSZ1TWo2hcCdg== )
   example.org.  3600  IN  DNSKEY  ( 256 3 13
          NrbL6utGqIW1wrhhjeexdA6bMdD1lC1hj0Fnpevaa1AMyY2uy83TmoGnR996N
          UR5TlG4Zh+YPbbmUIixe4nS3w== ) ; Key ID = 56566
   example.org.  3600  IN  DNSKEY  ( 257 3 13
          uspaqp17jsMTX6AWVgmbog/3Sttz+9ANFUWLn6qKUHr0BOqRuChQWj8jyYUUr
          Wy9txxesNQ9MkO4LUrFght1LQ== ) ; Key ID = 44384
   example.org.  3600  IN  RRSIG  ( DNSKEY 13 2 3600
          20181128000000 20151103000000 44384 example.org.
          ZsQ5wl2ZvofwDq7uYlvoqEeq9byHbl59Ap4EPXdB4PpnWy2dJkIElgXCfILrU
          EUCD1aKb2SoRZe18EJ8LMVJuw== )
   example.org.  900  IN  DS  ( 44384 13 2 ec307e2efc8f0117ed96ab48a513c
          8003e1d9121f1ff11a08b4cdd348d090aa6 )
   example.org.  900  IN  RRSIG  ( DS 13 2 900 20181128000000
          20151103000000 9523 org.
          15KUWAaNkJehAUdqm46TdeGg6mVm6bVKeaWLr34FTJlfMWWij+kmA6SM/bZbq
          kZBjtMWT55XersA+llFQNQI/Q== )
   org.  900  IN  DNSKEY  ( 256 3 13
          fuLp60znhSSEr9HowILpTpyLKQdM6ixcgkTE0gqVdsLx+DSNHSc69o6fLWC0e
          HfWx7kzlBBoJB0vLrvsJtXJ6g== ) ; Key ID = 47417
   org.  900  IN  DNSKEY  ( 256 3 13
          zTHbb7JM627Bjr8CGOySUarsic91xZU3vvLJ5RjVix9YH6+iwpBXb6qfHyQHy
          mlMiAAoaoXh7BUkEBVgDVN8sQ== ) ; Key ID = 9523
   org.  900  IN  DNSKEY  ( 257 3 13
          Uf24EyNt51DMcLV+dHPInhSpmjPnqAQNUTouU+SGLu+lFRRlBetgw1bJUZNI6
          Dlger0VJTm0QuX/JVXcyGVGoQ== ) ; Key ID = 49352
```

```
   org.  900  IN  DNSKEY  ( 257 3 13
           0SZfoe8Yx+eoaGgyAGEeJax/ZBV1AuG+/smcOgRm+F6doNlgc3lddcM1MbTvJ
           HTjK6Fvy8W6yZ+cAptn8sQheg== ) ; Key ID = 12651
   org.  900  IN  RRSIG  ( DNSKEY 13 1 900 20181128000000
           20151103000000 12651 org.
           G9I7dIh5Zn2hBu8jhgnLDTXZUpnPRkOMHjl1RcyHNbvJGLIiaPRVtcJXW0Vr+
           arygWmsHrDgWz0vw2IXZr3qKw== )
   org.  900  IN  RRSIG  ( DNSKEY 13 1 900 20181128000000
           20151103000000 49352 org.
           iQmYWqUdU07Syw1Fqwx+8+hSk0w06tCGmkwdppyxUSFESumEhkOXgOv6NuIEn
           eKjwMIaLj5HFB+9WnOkzgGE5Q== )
   org.  86400  IN  DS  ( 12651 13 2 3979a51f98bbf219fcaf4a4176e766dfa8f
           9db5c24a75743eb1e704b97a9fabc )
   org.  86400  IN  DS  ( 49352 13 2 03d11a1aa114abbb8f708c3c0ff0db765fe
           f4a2f18920db5f58710dd767c293b )
   org.  86400  IN  RRSIG  ( DS 13 1 86400 20181128000000
           20151103000000 31918 .
           JGPMvEbfLoWNUELn/5cjjdRZx2CmdikbHuH6N/1BrxACWrGy05NuPvBPTEVOr
           mPFfm5SIMLLTWgxf0K0FsNHoQ== )
   .  86400  IN  DNSKEY  ( 256 3 13
           zKz+DCWkNA/vuheiVPcGqsH40U84KZAlrMRIyozj9WHzf8PsFp/oR8j8vmjjW
           P98cbte4d8NvlGLxzbUzo3+FA== ) ; Key ID = 31918
   .  86400  IN  DNSKEY  ( 256 3 13
           8wMZZ4lzHdyKZ4fv8kys/t3QMlgvEadbsbyqWrMhwddSXCZYGRrsAbPpireRW
           xbVcd1VtOrlFBcRDMTN0R0XEQ== ) ; Key ID = 2635
   .  86400  IN  DNSKEY  ( 257 3 13
           yvX+VNTUjxZiGvtr060hVbrPV9H6rVusQtF9lIxCFzbZOJxMQBFmbqlc8Xclv
           Q+gDOXnFOTsgs/frMmxyGOtRg== ) ; Key ID = 47005
   .  86400  IN  RRSIG  ( DNSKEY 13 0 86400 20181128000000
           20151103000000 47005 .
           ehAzuZD3yT0pShXkKavrMdz+DKvvFvbZ+sGRZ5iQTni+ulMzZxHQ5+kSha65B
           Y2AIUphjyWcGr6VwP3Ne74iZA== )
```

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

```
   example.net.  3600  IN  DNAME  example.com.
   example.net.  3600  IN  RRSIG  ( DNAME 13 2 3600 20181128000000
           20151103000000 48085 example.net.
           +MJa5ZEmYh/kHYOhabF3ibfJ5xhJDJAA76Sugc/LFyTDJbmYW/nlYf3XLdcDh
           7lv6NfCkPuv6eCkSFGnVVvriA== )
   _443._tcp.www.example.net.  3600  IN  CNAME  (
           _443._tcp.www.example.com. )
   _443._tcp.www.example.com.  3600  IN  TLSA  ( 3 1 1
           c66bef6a5c1a3e78b82016e13f314f3cc5fa25b1e52aab9adb9ec5989b165
           ada )
   _443._tcp.www.example.com.  3600  IN  RRSIG  ( TLSA 13 5 3600
           20181128000000 20151103000000 1870 example.com.
```

```
         uml1DUjp5RfrXn9WtuMxEQV+ygzrONcuzsnyfOGSszwaDdkSOJ0Kndcfbb2Il
         LUV04Z+V488+Sd1jr7/21tsKA== )
example.net.  3600  IN  DNSKEY  ( 257 3 13
         X9GHpJcS7bqKVEsLiVAbddHUHTZqqBbVa3mzIQmdp+5cTJk7qDazwH68Kts8d
         9MvN55HddWgsmeRhgzePz6hMg== ) ; Key ID = 48085
example.net.  3600  IN  RRSIG  ( DNSKEY 13 2 3600
         20181128000000 20151103000000 48085 example.net.
         Qu7q2IheqxAKGnchYSvQeJuXdnBj/+wJoEmv67wemOUI6qvWWIo535w+hguUV
         mZm/W5rp3qWBGChLxxfqIK13g== )
example.net.  900  IN  DS  ( 48085 13 2 7c1998ce683df60e2fa41460c453f
         88f463dac8cd5d074277b4a7c04502921be )
example.net.  900  IN  RRSIG  ( DS 13 2 900 20181128000000
         20151103000000 10713 net.
         xxSlIJlpOSmrUgwR++os2SHTpRf53SO95G6FQyH5lEslnTnbZoq0p/AVrlB8q
         Qw3qmSXjRwGW3VFbkV60/tWCg== )
net.  900  IN  DNSKEY  ( 256 3 13
         061EoQs4sBcDsPiz17vt4nFSGLmXAGguqLStOesmKNCimi4/lw/vtyfqALuLF
         JiFjtCK3HMPi8HQ1jbGEwbGCA== ) ; Key ID = 10713
net.  900  IN  DNSKEY  ( 257 3 13
         LkNCPE+v3S4MVnsOqZFhn8n2NSwtLYOZLZjjgVsAKgu4XZncaDgq1R/7ZXRO5
         oVx2zthxuu2i+mGbRrycAaCvA== ) ; Key ID = 485
net.  900  IN  RRSIG  ( DNSKEY 13 1 900 20181128000000
         20151103000000 485 net.
         CC494bZrtBHXImEZpe6E3h6NL0R5fRR/MEuC1f2sfC6/dlCjRwFjCy9eOKnFL
         ar4Rxbpf7dvEwqGHNTawEo6jw== )
net.  86400  IN  DS  ( 485 13 2 ab25a2941aa7f1eb8688bb783b25587515a0c
         d8c247769b23adb13ca234d1c05 )
net.  86400  IN  RRSIG  ( DS 13 1 86400 20181128000000
         20151103000000 31918 .
         q+G4l97pYbFgAUhzzOW5+YoFiJc5omUbe20H28AwMHOrx19BdGp/2XhKDQ5F3
         tUTNerRmklzYm+7J/XtLpGXAw== )
.  86400  IN  DNSKEY  ( 256 3 13
         zKz+DCWkNA/vuheiVPcGqsH40U84KZAlrMRIyozj9WHzf8PsFp/oR8j8vmjjW
         P98cbte4d8NvlGLxzbUzo3+FA== ) ; Key ID = 31918
.  86400  IN  DNSKEY  ( 256 3 13
         8wMZZ4lzHdyKZ4fv8kys/t3QMlgvEadbsbyqWrMhwddSXCZYGRrsAbPpireRW
         xbVcd1VtOrlFBcRDMTN0R0XEQ== ) ; Key ID = 2635
.  86400  IN  DNSKEY  ( 257 3 13
         yvX+VNTUjxZiGvtr060hVbrPV9H6rVusQtF9lIxCFzbZOJxMQBFmbqlc8Xclv
         Q+gDOXnFOTsgs/frMmxyGOtRg== ) ; Key ID = 47005
.  86400  IN  RRSIG  ( DNSKEY 13 0 86400 20181128000000
         20151103000000 47005 .
         ehAzuZD3yT0pShXkKavrMdz+DKvvFvbZ+sGRZ5iQTni+ulMzZxHQ5+kSha65B
         Y2AIUphjyWcGr6VwP3Ne74iZA== )
example.com.  3600  IN  DNSKEY  ( 257 3 13
         JnA1XgyJTZz+psWvbrfUWLV6ULqIJyUS2CQdhUH9VK35bslWeJpRzrlxCUs7s
         /TsSfZMaGWVvlsuieh5nHcXzA== ) ; Key ID = 1870
example.com.  3600  IN  RRSIG  ( DNSKEY 13 2 3600
```

```
          20181128000000 20151103000000 1870 example.com.
          HujA9vQTbCxMeaYjDOCF0fYyHhajTl5xPztrp5u6P2vYV8naYQLG3zUF1gaer
          WBOagXXblaSSbYwB96LU3uSdg== )
   example.com.  900  IN  DS  ( 1870 13 2 e9b533a049798e900b5c29c90cd25a
          986e8a44f319ac3cd302bafc08f5b81e16 )
   example.com.  900  IN  RRSIG  ( DS 13 2 900 20181128000000
          20151103000000 34327 com.
          1tua9ntAqZvOnK5UztzIjN38Bqs6mJ8KAT7L4+AxevDL+z0Jft7RC1/g6Qrfa
          In1wqF4U7TvC8PYOD0U/HYtwQ== )
   com.  900  IN  DNSKEY  ( 256 3 13
          7IIE5Dol8jSMUqHTvOOiZapdEbQ9wqRxFi/zQcSdufUKLhpByvLpzSAQTqCWj
          3URIZ8L3Fa2gBLMOZUzZ1GQCw== ) ; Key ID = 34327
   com.  900  IN  DNSKEY  ( 257 3 13
          RbkcO+96XZmnp8jYIuM4lryAp3egQjSmBaSoiA7H76Tm0RLHPNPUxlVk+nQ0f
          Ic3I8xfZDNw8Wa0Pe3/g2QA/w== ) ; Key ID = 18931
   com.  900  IN  DNSKEY  ( 257 3 13
          szc7biLo5J4OHlkan1vZrF4aD4YYf+NHA/GAqdNslY9xxK9Izg68XHkqck4Rt
          DiVk37lNAQmgSlHbrGu0yOTkA== ) ; Key ID = 28809
   com.  900  IN  RRSIG  ( DNSKEY 13 1 900 20181128000000
          20151103000000 18931 com.
          lZmTBrfcRgVbqHJIfCVr6c3HUDgy3MlNSCSnrVV2S5/NmB3ZiFcvIDn0iqXPm
          7YQfvfWi6utyxBu/fSD6S1ARw== )
   com.  900  IN  RRSIG  ( DNSKEY 13 1 900 20181128000000
          20151103000000 28809 com.
          8qZOVM4X8wGt5XPWhG2HO4FAD6Kvs5eIhZUz+7DVCrZ/XMEVrMIHcm1Q+sq0s
          hm4cSivK2BxOO24PHJXoZN2Lw== )
   com.  86400  IN  DS  ( 18931 13 2 20f7a9db42d0e2042fbbb9f9ea015941202
          f9eabb94487e658c188e7bcb52115 )
   com.  86400  IN  DS  ( 28809 13 2 ad66b3276f796223aa45eda773e92c6d98e
          70643bbde681db342a9e5cf2bb380 )
   com.  86400  IN  RRSIG  ( DS 13 1 86400 20181128000000
          20151103000000 31918 .
          5KQVa0NP+6k7VEGMmeky2/Y3wIGM70Fkm0vp5NmQ6KPk8L1XMJPltcJDWGGjc
          EU3Uc4z2DUxzZyWgEDdrSOcdw== )
```

Authors’ Addresses

Melinda Shore
Fastly

EMail: mshore@fastly.com

Richard Barnes
Mozilla

EMail: rlb@ipv.sx

Shumon Huque
Salesforce

EMail: shuque@gmail.com

Willem Toorop
NLnet Labs

EMail: willem@nlnetlabs.nl

TLS E. Rescorla, Ed.
Internet-Draft RTFM, Inc.
Updates: 6347 (if approved) H. Tschofenig, Ed.
Intended status: Standards Track T. Fossati
Expires: May 6, 2021 Arm Limited
November 02, 2020

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

Abstract

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

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

Table of Contents

1. Introduction

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

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

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

2. Conventions and Terminology

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

This document assumes familiarity with DTLS 1.2 [RFC6347].

3. The "connection_id" Extension

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

The extension type is specified as follows.

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

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

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

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

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

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

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

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

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

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

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

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

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

4. Record Layer Extensions

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

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

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

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

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

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

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

Figure 1: DTLS 1.2 Plaintext Record Payload.

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

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

Figure 2: New DTLSInnerPlaintext Payload Structure.

content Corresponds to the fragment of a given length.

real_type The content type describing the payload.

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

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

Figure 3: DTLS 1.2 CID-enhanced Ciphertext Record.

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

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

enc_content The encrypted form of the serialized DTLSInnerPlaintext structure.

All other fields are as defined in RFC 6347.

5. Record Payload Protection

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

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

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

cid Value of the negotiated CID (variable length).

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

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

Note "+" denotes concatenation.

5.1. Block Ciphers

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

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

5.2. Block Ciphers with Encrypt-then-MAC processing

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

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

5.3. AEAD Ciphers

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

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

6. Peer Address Update

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

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

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

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

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

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

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

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

7. Examples

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

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

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


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

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

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

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

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


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

                               <========        Application Data

   Legend:

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

Figure 4: Example DTLS 1.2 Exchange with CID

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

8. Privacy Considerations

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

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

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

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

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

9. Security Considerations

An on-path adversary can create reflection attacks against third parties because a DTLS peer has no means to distinguish a genuine address update event (for example, due to a NAT rebinding) from one that is malicious. This attack is of concern when there is a large asymmetry of request/response message sizes.

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

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

10. IANA Considerations

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

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

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

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

11. References

11.1. Normative References

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

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

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

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

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

11.2. Informative References

[I-D.ietf-tls-dtls13]
Rescorla, E., Tschofenig, H., and N. Modadugu, "The Datagram Transport Layer Security (DTLS) Protocol Version 1.3", draft-ietf-tls-dtls13-38 (work in progress), May 2020.

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

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

11.3. URIs

[1] mailto:tls@ietf.org

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

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

Appendix A. History

RFC EDITOR: PLEASE REMOVE THE THIS SECTION

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

- RRC draft moved from normative to informative.

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

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

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

- Updated IANA considerations

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

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

- Restructed Section 5 "Record Payload Protection"

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

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

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

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

- Updated list of contributors

- Updated list of contributors and acknowledgements

- Updated example

- Changed record layer design

- Changed record payload protection

- Updated introduction and security consideration section

- Author- and affiliation changes

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

- Move to internal content types a la DTLS 1.3.

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

- Remove 1.3 based on the WG consensus at IETF 101

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

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

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

- Initial version

Appendix B. Working Group Information

RFC EDITOR: PLEASE REMOVE THE THIS SECTION

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

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

Appendix C. Contributors

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

* Yin Xinxing
  Huawei
  yinxinxing@huawei.com

* Nikos Mavrogiannopoulos
  RedHat
  nmav@redhat.com

* Tobias Gondrom
  tobias.gondrom@gondrom.org

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

- Martin Thomson (Mozilla)

- Christian Huitema (Private Octopus Inc.)

- Jana Iyengar (Google)

- Daniel Kahn Gillmor (ACLU)

- Patrick McManus (Mozilla)

- Ian Swett (Google)

- Mark Nottingham (Fastly)

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

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

Appendix D. Acknowledgements

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

Authors' Addresses

Eric Rescorla (editor)
RTFM, Inc.

EMail: ekr@rtfm.com

Hannes Tschofenig (editor)
Arm Limited

EMail: hannes.tschofenig@arm.com

Thomas Fossati
Arm Limited

EMail: thomas.fossati@arm.com

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

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

November 02, 2020

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

Abstract

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

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

Table of Contents

1. Introduction

RFC EDITOR: PLEASE REMOVE THE FOLLOWING PARAGRAPH

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

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

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

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

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

2. Conventions and Terminology

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

The following terms are used:

- client: The endpoint initiating the DTLS connection.

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

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

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

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

- receiver: An endpoint that is receiving records.

- sender: An endpoint that is transmitting records.

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

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

- CID: Connection ID

- MSL: Maximum Segment Lifetime

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

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

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

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

- '*' indicates optional or situation-dependent messages/extensions that are not always sent.

- '{}' indicates messages protected using keys derived from a [sender]_handshake_traffic_secret.

- '[]' indicates messages protected using keys derived from traffic_secret_N.

3. DTLS Design Rationale and Overview

The basic design philosophy of DTLS is to construct "TLS over datagram transport". Datagram transport does not require nor provide reliable or in-order delivery of data. The DTLS protocol preserves this property for application data. Applications such as media streaming, Internet telephony, and online gaming use datagram transport for communication due to the delay-sensitive nature of transported data. The behavior of such applications is unchanged when the DTLS protocol is used to secure communication, since the DTLS protocol does not compensate for lost or reordered data traffic.

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

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

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

3. During the handshake, messages are implicitly acknowledged by other handshake messages. Some handshake messages, such as the NewSessionTicket message, do not result in any direct response that would allow the sender to detect loss. DTLS adds an acknowledgment message to enable better loss recovery.

4. Handshake messages are potentially larger than can be contained in a single datagram. DTLS adds fields to handshake messages to support fragmentation and reassembly.

5. Datagram transport protocols, like UDP, are susceptible to abusive behavior effecting denial of service attacks against nonparticipants. DTLS adds a return-routability check that uses the TLS HelloRetryRequest message (see Section 5.1 for details).

3.1. Packet Loss

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

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

                        X<-- HelloRetryRequest
                                         (lost)

[Timer Expires]

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

Figure 1: DTLS retransmission example

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

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

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

3.2. Reordering

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

3.3. Message Size

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

3.4. Replay Detection

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

4. The DTLS Record Layer

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

1. The DTLSCiphertext structure omits the superfluous version number and type fields.

2. DTLS adds an epoch and sequence number to the TLS record header. This sequence number allows the recipient to correctly verify the DTLS MAC. However, the number of bits used for the epoch and sequence number fields in the DTLSCiphertext structure have been reduced from those in previous versions.

3. The DTLSCiphertext structure has a variable length header.

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

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

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

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

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

Figure 2: DTLS 1.3 Record Format

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

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

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

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

Figure 3: DTLS 1.3 CipherText Header

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

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

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

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

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

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

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

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

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

Figure 4 illustrates different record layer header types.

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

 DTLSPlaintext
   Structure
```

Figure 4: Header Examples

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

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

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

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

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

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

4.1. Determining the Header Format

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

- If the first byte is alert(21), handshake(22), or ack(proposed, 26), the record MUST be interpreted as a DTLSPlaintext record.

- If the first byte is any other value, then receivers MUST check to see if the leading bits of the first byte are 001. If so, the implementation MUST process the record as DTLSCiphertext; the true content type will be inside the protected portion.

- Otherwise, the record MUST be rejected as if it had failed deprotection, as described in Section 4.5.2.

4.2. Sequence Number and Epoch

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

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

4.2.1. Processing Guidelines

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

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

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

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

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

4.2.2. Reconstructing the Sequence Number and Epoch

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

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

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

4.2.3. Sequence Number Encryption

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

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

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

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

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

The sn_key is computed as follows:

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

[sender] denotes the sending side. The Secret value to be used is described in Section 7.3 of [TLS13].

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

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

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

4.3. Transport Layer Mapping

DTLS messages MAY be fragmented into multiple DTLS records. Each DTLS record MUST fit within a single datagram. In order to avoid IP fragmentation, clients of the DTLS record layer SHOULD attempt to size records so that they fit within any PMTU estimates obtained from the record layer.

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

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

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

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

4.4. PMTU Issues

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

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

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

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

- The DTLS handshake messages can exceed the PMTU.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

4.5. Record Payload Protection

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

4.5.1. Anti-Replay

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

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

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

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

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

4.5.2. Handling Invalid Records

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

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

4.5.3. AEAD Limits

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

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

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

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

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

The AEAD_AES_128_CCM_8 AEAD, as used in TLS_AES_128_CCM_SHA256, does not have a limit on the number of records that fail authentication that both limits the probability of forgery by the same amount and does not expose implementations to the risk of denial of service; see Appendix B.3. Therefore, TLS_AES_128_CCM_SHA256 MUST NOT used in DTLS without additional safeguards against forgery. Implementations MUST set usage limits for AEAD_AES_128_CCM_8 based on an understanding of any additional forgery protections that are used.

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

5. The DTLS Handshake Protocol

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

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

2. Retransmission timers are introduced to handle message loss.

3. A new ACK content type has been added for reliable message delivery of handshake messages.

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

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

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

5.1. Denial-of-Service Countermeasures

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

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

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

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

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

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

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

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

                      <----- HelloRetryRequest
                              + cookie

ClientHello           ------>
 + cookie

[Rest of handshake]
```

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

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

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

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

The handshake transcript is not reset with the second ClientHello and a stateless server-cookie implementation requires the transcript of the HelloRetryRequest to be stored in the cookie or the internal state of the hash algorithm, since only the hash of the transcript is required for the handshake to complete.

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

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

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

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

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

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

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

5.2. DTLS Handshake Message Format

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

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

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

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

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

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

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

5.3. ClientHello Message

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

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

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

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

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

random: Same as for TLS 1.3.

legacy_session_id: Same as for TLS 1.3.

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

cipher_suites: Same as for TLS 1.3.

legacy_compression_methods: Same as for TLS 1.3.

extensions: Same as for TLS 1.3.

5.4. Handshake Message Fragmentation and Reassembly

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

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

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

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

5.5. End Of Early Data

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

5.6. DTLS Handshake Flights

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

```
Client                                             Server

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

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


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

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


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

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

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

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

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


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

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

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

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

```
 Client                                            Server

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

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


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

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

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

Figure 8: Message flights for the Zero-RTT handshake

```
 Client                                            Server

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

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

Figure 9: Message flights for the new session ticket message

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

5.7. Timeout and Retransmission

5.7.1. State Machine

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

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

```
               |           |
               +-----------+
                   |  /|\
                   |   |
                   |   |
                   +---+

             Server read retransmit
                 Retransmit ACK
```

Figure 10: DTLS timeout and retransmission state machine

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

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

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

There are four ways to exit the WAITING state:

1. The retransmit timer expires: the implementation transitions to the SENDING state, where it retransmits the flight, resets the retransmit timer, and returns to the WAITING state.

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

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

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

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

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

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

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

5.7.2. Timer Values

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

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

5.7.3. State machine duplication for post-handshake messages

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

1. NewSessionTicket

2. KeyUpdate

3. NewConnectionId

4. RequestConnectionId

5. Post-handshake client authentication

Messages of each category can be sent independently, and reliability is established via independent state machines each of which behaves as described in Section 5.7.1. For example, if a server sends a NewSessionTicket and a CertificateRequest message, two independent state machines will be created.

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

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

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

5.8. CertificateVerify and Finished Messages

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

5.9. Cryptographic Label Prefix

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

5.10. Alert Messages

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

5.11. Establishing New Associations with Existing Parameters

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

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

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

6. Example of Handshake with Timeout and Retransmission

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

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

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

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


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

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


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

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

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

Figure 11: Example DTLS exchange illustrating message loss

6.1. Epoch Values and Rekeying

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

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

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

- epoch value (1) is used for messages protected using keys derived from client_early_traffic_secret. Note this epoch is skipped if the client does not offer early data.

- epoch value (2) is used for messages protected using keys derived from [sender]_handshake_traffic_secret. Messages transmitted during the initial handshake, such as EncryptedExtensions, CertificateRequest, Certificate, CertificateVerify, and Finished belong to this category. Note, however, post-handshake are protected under the appropriate application traffic key and are not included in this category.

- epoch value (3) is used for payloads protected using keys derived from the initial [sender]_application_traffic_secret_0. This may include handshake messages, such as post-handshake messages (e.g., a NewSessionTicket message).

- epoch value (4 to 2^16-1) is used for payloads protected using keys from the [sender]_application_traffic_secret_N (N>0).

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

                          Some time later ...
                            (Rekeying)

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

Figure 12: Example DTLS exchange with epoch information

7. ACK Message

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

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

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

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

During the handshake, ACKs only cover the current outstanding flight (this is possible because DTLS is generally a lockstep protocol). Thus, an ACK from the server would not cover both the ClientHello and the client's Certificate. Implementations can accomplish this by clearing their ACK list upon receiving the start of the next flight.

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

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

7.1. Sending ACKs

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

- When they receive a message or fragment which is out of order, either because it is not the next expected message or because it is not the next piece of the current message.

- When they have received part of a flight and do not immediately receive the rest of the flight (which may be in the same UDP datagram). A reasonable approach here is to set a timer for 1/4 the current retransmit timer value when the first record in the flight is received and then send an ACK when that timer expires.

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

1. Handshake flights other than the client's final flight

2. The server's post-handshake CertificateRequest.

ACKs SHOULD NOT be sent for these flights unless generating the responding flight takes significant time. In this case, implementations MAY send explicit ACKs for the complete received flight even though it will eventually also be implicitly acknowledged through the responding flight. A notable example for this is the case of post-handshake client authentication in constrained environments, where generating the CertificateVerify message can take considerable time on the client. All other flights MUST be ACKed. Implementations MAY acknowledge the records corresponding to each transmission of each flight or simply acknowledge the most recent one. In general, implementations SHOULD ACK as many received packets as can fit into the ACK record, as this provides the most complete information and thus reduces the chance of spurious retransmission; if space is limited, implementations SHOULD favor including records which have not yet been acknowledged.

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

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

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

7.2. Receiving ACKs

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

7.3. Design Rational

ACK messages are used in two circumstances, namely :

- on sign of disruption, or lack of progress, and

- to indicate complete receipt of the last flight in a handshake.

In the first case the use of the ACK message is optional because the peer will retransmit in any case and therefore the ACK just allows for selective retransmission, as opposed to the whole flight retransmission in previous versions of DTLS. For instance in the flow shown in Figure 11 if the client does not send the ACK message when it received and processed record 1 indicating loss of record 0, the entire flight would be retransmitted. When DTLS 1.3 is used in deployments with loss networks, such as low-power, long range radio networks as well as low-power mesh networks, the use of ACKs is recommended.

The use of the ACK for the second case is mandatory for the proper functioning of the protocol. For instance, the ACK message sent by the client in Figure 12, acknowledges receipt and processing of record 2 (containing the NewSessionTicket message) and if it is not sent the server will continue retransmission of the NewSessionTicket indefinitely.

8. Key Updates

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

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

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

9. Connection ID Updates

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

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

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

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

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

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

Endpoints SHOULD use receiver-provided CIDs in the order they were provided. Endpoints MUST NOT have more than one NewConnectionId message outstanding.

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

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

num_cids The number of CIDs desired.

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

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

9.1. Connection ID Example

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

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

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

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


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

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

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

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

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

Figure 13: Example DTLS 1.3 Exchange with CIDs

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

10. Application Data Protocol

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

11. Security Considerations

Security issues are discussed primarily in [TLS13].

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

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

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

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

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

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

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

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

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

- DTLS 1.3 encrypts handshake messages much earlier than in previous DTLS versions. Therefore, less information identifying the DTLS client, such as the client certificate, is available to an on-path adversary.

12. Changes to DTLS 1.2

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

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

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

- Removed support for weaker and older cryptographic algorithms

- HelloRetryRequest of TLS 1.3 used instead of HelloVerifyRequest

- More flexible ciphersuite negotiation

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

13. IANA Considerations

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

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

14. References

14.1. Normative References

[CHACHA] Nir, Y. and A. Langley, "ChaCha20 and Poly1305 for IETF Protocols", RFC 8439, DOI 10.17487/RFC8439, June 2018, <https://www.rfc-editor.org/info/rfc8439>.

[I-D.ietf-tls-dtls-connection-id] Rescorla, E., Tschofenig, H., and T. Fossati, "Connection Identifiers for DTLS 1.2", draft-ietf-tls-dtls-connection-id-07 (work in progress), October 2019.

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

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

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

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

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

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

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

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

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

14.2. Informative References

[AEBounds] Luykx, A. and K. Paterson, "Limits on Authenticated Encryption Use in TLS", March 2016, <http://www.isg.rhul.ac.uk/~kp/TLS-AEbounds.pdf>.

[CCM-ANALYSIS] Jonsson, J., "On the Security of CTR + CBC-MAC", Selected Areas in Cryptography pp. 76-93, DOI 10.1007/3-540-36492-7_7, 2003.

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

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

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

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

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

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

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

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

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

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

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

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

14.3. URIs

[1] mailto:tls@ietf.org

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

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

Appendix A. Protocol Data Structures and Constant Values

This section provides the normative protocol types and constants definitions.

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

Appendix B. Analysis of Limits on CCM Usage

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

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

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

t: The size of the authentication tag in bits. For this cipher, t is 128.

n: The size of the block function in bits. For this cipher, n is 128.

l: The number of blocks in each packet (see below).

q: The number of genuine packets created and protected by endpoints. This value is the bound on the number of packets that can be protected before updating keys.

v: The number of forged packets that endpoints will accept. This value is the bound on the number of forged packets that an endpoint can reject before updating keys.

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

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

B.1. Confidentiality Limits

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

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

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

q <= 2^23

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

B.2. Integrity Limits

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

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

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

v + q <= 2^24.5

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

B.3. Limits for AEAD_AES_128_CCM_8

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

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

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

$v <= 2^7$

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

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

Appendix C. History

RFC EDITOR: PLEASE REMOVE THE THIS SECTION

IETF Drafts

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

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

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

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

draft-35: - I-D.ietf-tls-dtls-connection-id became a normative reference - Removed duplicate reference to I-D.ietf-tls-dtls-connection-id. - Fix figure 11 to have the right numbers andno cookie in message 1. - Clarify when you can ACK. - Clarify additional data computation.

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

draft-32: - Editorial improvements and clarifications.

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

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

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

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

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

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

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

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

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

draft-00 - First IETF Draft

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

draft-00

- Initial version using TLS 1.3 as a baseline.

- Use of epoch values instead of KeyUpdate message

- Use of cookie extension instead of cookie field in ClientHello and HelloVerifyRequest messages

- Added ACK message

- Text about sequence number handling

Appendix D. Working Group Information

RFC EDITOR: PLEASE REMOVE THIS SECTION.

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

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

Appendix E. Contributors

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

In addition, we would like to thank:

* David Benjamin
  Google
  davidben@google.com

* Thomas Fossati
  Arm Limited
  Thomas.Fossati@arm.com

* Tobias Gondrom
  Huawei
  tobias.gondrom@gondrom.org

* Felix Guenther
  ETH Zurich
  mail@felixguenther.info

* Ilari Liusvaara
  Independent
  ilariliusvaara@welho.com

* Martin Thomson
  Mozilla
  martin.thomson@gmail.com

* Christopher A. Wood
  Apple Inc.
  cawood@apple.com

* Yin Xinxing
  Huawei
  yinxinxing@huawei.com

* Hanno Becker
  Arm Limited
  Hanno.Becker@arm.com

Appendix F. Acknowledgements

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

Authors' Addresses

Eric Rescorla
RTFM, Inc.

EMail: ekr@rtfm.com

Hannes Tschofenig
Arm Limited

EMail: hannes.tschofenig@arm.com

Nagendra Modadugu
Google, Inc.

EMail: nagendra@cs.stanford.edu

tls
Internet-Draft
Intended status: Standards Track
Expires: 19 April 2021

E. Rescorla
RTFM, Inc.
K. Oku
Fastly
N. Sullivan
C.A. Wood
Cloudflare
16 October 2020

TLS Encrypted Client Hello
draft-ietf-tls-esni-08

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

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

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

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

This Internet-Draft will expire on 19 April 2021.

Copyright Notice

Table of Contents

1. Introduction

DISCLAIMER: This is very early a work-in-progress design 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 information unencrypted 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 usual (we will refer to this as the ClientHelloInner message) and then encrypts this message using the public key of the ECH configuration. It then constructs a new ClientHello (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, the client-facing 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 supports ECH 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 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, 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.)

Informally, a primary goal of ECH is ensuring that connections to servers in the same anonymity set are indistinguishable from one another 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-05 of HPKE for public key encryption [I-D.irtf-cfrg-hpke]. The ECH configuration is defined by the following "ECHConfigs" 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;
} ECHCipherSuite;

struct {
    opaque public_name<1..2^16-1>;

    HpkePublicKey public_key;
    HpkeKemId kem_id;
    ECHCipherSuite cipher_suites<4..2^16-4>;

    uint16 maximum_name_length;
    Extension extensions<0..2^16-1>;
} ECHConfigContents;

struct {
    uint16 version;
    uint16 length;
    select (ECHConfig.version) {
      case 0xfe08: ECHConfigContents contents;
    }
} ECHConfig;

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

The "ECHConfigs" 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.

The "ECHConfig" 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:

public_name The non-empty name of client-facing server, i.e., the entity trusted to update these encryption keys. This is used to repair misconfigurations, as described in Section 6.3.

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

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.

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

maximum_name_length The largest name the server expects to support, 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 resource record value.

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

4.1. Configuration Extensions

ECH configuration extensions are used to 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(0xfe08), (65535)
} ExtensionType;
```

The extension request is carried by the ClientHelloOuter, i.e., the ClientHello transmitted to the client-facing server. The payload contains the following "ClientECH" structure:

```
struct {
   ECHCipherSuite cipher_suite;
   opaque config_id<0..255>;
   opaque enc<1..2^16-1>;
   opaque payload<1..2^16-1>;
} ClientECH;
```

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

config_id The configuration identifier, equal to "Expand(Extract("", config), "tls ech config id", Nh)", where "config" is the "ECHConfig" structure and "Extract", "Expand", and "Nh" are as specified by the cipher suite KDF. (Passing the literal """" as the salt is interpreted by "Extract" as no salt being provided.) The length of this value SHOULD NOT be less than 16 bytes unless it is optional for an application; see Section 10.4.

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 offering the "encrypted_client_hello" extension in its ClientHelloOuter, the client MUST also offer an empty "encrypted_client_hello" extension in its ClientHelloInner, wherever applicable. (This requirement is not applicable when the extension is generated as described in Section 6.4.)

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 {
   ECHConfigs retry_configs;
} ServerECH;
```

retry_configs An ECHConfigs structure containing one or more ECHConfig structures, in decreasing order of preference, to be used by the client in subsequent connection attempts.

This document also defines the "ech_required" alert, which clients 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 and having them both in ClientHelloInner and ClientHelloOuter will lead to a very large overall size. One particularly pathological example is "key_share" with post-quantum algorithms. In order to reduce the impact of duplicated extensions, the client may use the "outer_extensions" extension.

```
enum {
   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 "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 "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 "outer_extensions" extension is only used for compressing the ClientHelloInner. It MUST NOT be sent in either ClientHelloOuter or ClientHelloInner.

5.2. Authenticating the ClientHelloOuter

To prevent a network attacker from modifying the reconstructed ClientHelloInner (see Section 10.10.3), ECH authenticates ClientHelloOuter by deriving a ClientHelloOuterAAD value. This is computed by serializing ClientHelloOuter with the "encrypted_client_hello" extension removed. ClientHelloOuterAAD is then passed as the associated data parameter to the HPKE encryption.

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

6.1. Sending an Encrypted ClientHello

To offer ECH, the client first chooses a suitable ECH configuration. To determine if a given "ECHConfig" is suitable, it checks that it supports the KEM algorithm identified by "ECHConfig.kem_id" and at least one KDF/AEAD algorithm identified by "ECHConfig.cipher_suites". Once a suitable configuration is found, the client selects the cipher suite it will use for encryption. It MUST NOT choose a cipher suite not advertised by the configuration.

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. Note 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 6.2.

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

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 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.)

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

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

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

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. Finally, it computes the encapsulated key, context, HRR key (see Section 6.3.3), and payload as:

```
pkR = Deserialize(ECHConfig.public_key)
enc, context = SetupBaseS(pkR,
                          "tls ech" || 0x00 || ECHConfig)
ech_hrr_key = context.Export("tls ech hrr key", 32)
payload = context.Seal(ClientHelloOuterAAD,
                       EncodedClientHelloInner)
```

Note that the HPKE functions Deserialize and SetupBaseS are those which match "ECHConfig.kem_id" and the AEAD/KDF used with "context" are those which match the client's chosen preference from "ECHConfig.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:

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

* "config_id", the identifier of the chosen ECHConfig structure;

* "enc", as computed above; and

* "payload", as computed above.

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

6.2. 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.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.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.

6.3. 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 "accept_confirmation" as defined in Section 7.2. Otherwise, it presumes rejection.

6.3.1. Accepted ECH

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

6.3.2. Rejected ECH

If the server used ClientHelloOuter, the client proceeds with the handshake, authenticating for ECHConfig.public_name as described in Section 6.3.2.1. If authentication or the handshake fails, the client MUST return a failure to the calling application. It MUST NOT use the retry keys.

Otherwise, when the handshake completes successfully with the public name authenticated, 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 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 that value to encrypt the ClientHello. The value may only be applied to the retry connection. The client MUST continue to use the previously-advertised keys for subsequent connections. This avoids introducing pinning concerns or a tracking vector, should a malicious server present client-specific retry keys to identify clients.

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

ECHConfigContents.public_name as described in Section 6.3.2.1. 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.

6.3.2.1. 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 ECHConfigContents.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 6.3. This may be implemented, for instance, by reporting a failed connection with a dedicated error code.

6.3.3. HelloRetryRequest

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.

Clients bind encryption of the second ClientHelloInner to encryption of the first ClientHelloInner via the derived ech_hrr_key by modifying HPKE setup as follows:

```
pkR = Deserialize(ECHConfig.public_key)
enc, context = SetupPSKS(pkR, "tls ech" || 0x00 || ECHConfig,
                         ech_hrr_key, "hrr key")
```

The "info" parameter to SetupPSKS is the concatenation of "tls ech", a zero byte, and the serialized ECHConfig. Clients then encrypt the second ClientHelloInner using this new HPKE context. In doing so, the encrypted value is also authenticated by ech_hrr_key. The rationale for this is described in Section 10.10.2.

Client-facing servers perform the corresponding process when decrypting second ClientHelloInner messages. In particular, upon receipt of a second ClientHello message with a ClientECH value, servers set up their HPKE context and decrypt ClientECH as follows:

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

ClientHelloOuterAAD is computed from the second ClientHelloOuter as described in Section 5.2. The "info" parameter to SetupPSKR is computed as 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.

[[OPEN ISSUE: Should we be using the PSK input or the info input? On the one hand, the requirements on info seem weaker, but maybe actually this needs to be secret? Analysis needed.]]

6.4. GREASE Extensions

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 as follows:

* Set the "suite" field to a supported ECHCipherSuite. 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 "config_id" field to a randomly-generated string of "Nh" bytes, where "Nh" is the output length of the "Extract" function of the KDF associated with the chosen cipher suite. (The KDF API is specified in [I-D.irtf-cfrg-hpke].)

* 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 selected AEAD scheme and L is the size of the ClientHelloInner message the client would use given an ECHConfig structure, padded according to Section 6.2.

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

7.1. Client-Facing Server

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

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

1. Compare ClientECH.config_id against identifiers of known ECHConfigs and select the one that matches, if any, as a candidate.

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

Some uses of ECH, such as local discovery mode, may omit 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 all candidate ECHConfigs, 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)
ech_hrr_key = context.Export("tls ech hrr key", 32)
```

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 consider candidate ECHConfigs.

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 modified 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 ECHConfigs 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 6.4), 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. HelloRetryRequest

In case a HelloRetryRequest (HRR) is sent, the client-facing server MUST consistently accept or decline ECH between the two ClientHellos, using the same ECHConfig, and abort the handshake if this is not possible. This is achieved as follows. Let CH1 and CH2 denote, respectively, the first and second ClientHello transmitted on the wire by the client:

1. If CH1 contains the "encrypted_client_hello" extension but CH2 does not, or if CH2 contains the "encrypted_client_hello" extension but CH1 does not, then the server MUST abort the handshake with an "illegal_parameter" alert.

2. If the "encrypted_client_hello" extension is sent in CH2, the server follows the procedure in Section 7.1 to decrypt the extension, but it uses the previously-selected ECHConfig as the set of candidate ECHConfigs. If decryption fails, the server aborts the connection with a "decrypt_error" alert rather than continuing the handshake with the second ClientHelloOuter.

[[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 #333.]]

7.2. Backend Server Behavior

When the client-facing server accepts ECH, it forwards the ClientHelloInner to the backend server, who terminates the connection. If the ClientHelloInner contains an empty "encrypted_client_hello" extension, then the backend server MUST confirm ECH acceptance by setting ServerHello.random[24:32] to

```
accept_confirmation = HKDF-Expand-Label(
    HKDF-Extract(0, ClientHelloInner.random),
    "ech accept confirmation",
    ServerHello.random[0:24], 8)
```

where HKDF-Expand-Label and HKDF-Extract are as defined in [RFC8446]. The value of ServerHello.random[0:24] is generated as usual by invoking a secure random number generator (see [RFC8446], Section 4.1.2).

8. Compatibility Issues

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

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

8.1. Misconfiguration and Deployment Concerns

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

The retry mechanism repairs inconsistencies, provided the server is authoritative for the public name. If server and advertised keys mismatch, the server will 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 6.3.

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 6.3 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 6.4) 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 either an empty config_id or a randomly generated config_id in the ClientECH. (The precise implementation choice for this mechanism is out of scope for this document.) 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.

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). The backend server's signal of acceptance does not interfere with this mechanism because ECH is only supported 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 6.4, 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 ECHConfigs 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 6.2 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 binding the first and second ClientHello messages together. In particular, since the attacker does not possess the ech_hrr_key, it cannot generate a valid encryption of the second inner ClientHello. The server will attempt decryption using ech_hrr_key, detect failure, and fail the connection.

If the second ClientHello were not bound to the first, 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
         + outer_extensions(pre_shared_key)
       + pre_shared_key
                    -------->
                          (intercept)
                          ClientHello
                          + key_share
                          + 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 two entries in the existing registry for ExtensionType (defined in [RFC8446]):

1. encrypted_client_hello(0xfe08), with "TLS 1.3" column values being set to "CH, EE", and "Recommended" column being set to "Yes".

2. outer_extensions(0xfd00), with the "TLS 1.3" column values being set to "", and "Recommended" column being 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 being 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-01, 13 July 2020, <http://www.ietf.org/internet-drafts/draft-ietf-dnsop-svcb-https-01.txt>.

[I-D.ietf-tls-exported-authenticator]
Sullivan, N., "Exported Authenticators in TLS", Work in Progress, Internet-Draft, draft-ietf-tls-exported-authenticator-13, 26 June 2020, <http://www.ietf.org/internet-drafts/draft-ietf-tls-exported-authenticator-13.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-05, 30 July 2020, <http://www.ietf.org/internet-drafts/draft-irtf-cfrg-hpke-05.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.

Authors' Addresses

Eric Rescorla
RTFM, Inc.

Email: ekr@rtfm.com

Kazuho Oku
Fastly

Email: kazuhooku@gmail.com

Nick Sullivan
Cloudflare

Email: nick@cloudflare.com

Christopher A. Wood
Cloudflare

Email: caw@heapingbits.net

Internet Engineering Task Force
Internet-Draft
Obsoletes: 5469 7507 (if approved)
Updates: 8422 8261 7568 7562 7525 7465 7030 6750 6749 6739 6460 6614 6367 6347 6176 6084 6083 6042 6012 5878 5734 5456 5422 5415 5364 5281 5263 5238 5216 5158 5091 5054 5049 5024 5023 5019 5018 4992 4976 4975 4964 4851 4823 4791 4785 4744 4743 4732 4712 4681 4680 4642 4616 4582 4540 4531 4513 4497 4279 4261 4235 4217 4168 4162 4111 4097 3983 3943 3903 3887 3871 3856 3767 3749 3656 3568 3552 3501 3470 3436 3329 3261 (if approved)
Intended status: Best Current Practice
Expires: May 13, 2021

K. Moriarty
Dell EMC
S. Farrell
Trinity College Dublin
November 9, 2020

Deprecating TLSv1.0 and TLSv1.1
draft-ietf-tls-oldversions-deprecate-09

Abstract

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

This document also deprecates Datagram TLS (DTLS) version 1.0 (RFC6347), but not DTLS version 1.2, and there is no DTLS version 1.1.

This document updates many RFCs that normatively refer to TLSv1.0 or TLSv1.1 as described herein. This document also updates the best practices for TLS usage in RFC 7525 and hence is part of BCP195.

Status of This Memo

This Internet-Draft will expire on May 13, 2021.

Copyright Notice

Table of Contents

1. Introduction

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

Technical reasons for deprecating these versions include:

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

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

1.1. RFCs Updated

This document updates the following RFCs that normatively reference TLSv1.0 or TLSv1.1 or DTLS1.0. The update is to obsolete usage of these older versions. Fallback to these versions are prohibited through this update. Specific references to mandatory minimum protocol versions of TLSv1.0 or TLSv1.1 are replaced by TLSv1.2, and references to minimum protocol version DTLSv1.0 are replaced by

DTLSv1.2. Statements that "TLS 1.0 is the most widely deployed version and will provide the broadest interoperability" are removed without replacement.

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

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

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

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

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

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

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

The DES and IDEA cipher suites specified in [RFC5469] were specifically removed from TLSv1.2 by [RFC5246]; since the only versions of TLS for which their usage is defined are now Historic, RFC 5469 (will be|has been) moved to Historic as well.

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

1.2. Terminology

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

2. Support for Deprecation

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

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

> TLS 1.1, specified in [RFC4346], was developed to address weaknesses discovered in TLS 1.0, primarily in the areas of initialization vector selection and padding error processing. Initialization vectors were made explicit to prevent a certain class of attacks on the Cipher Block Chaining (CBC) mode of operation used by TLS. The handling of padding errors was altered to treat a padding error as a bad message authentication code, rather than a decryption failure. In addition, the TLS 1.1 RFC acknowledges attacks on CBC mode that rely on the time to compute the message authentication code (MAC). The TLS 1.1 specification states that to defend against such attacks, an implementation must process records in the same manner regardless of whether padding errors exist. Further implementation considerations for CBC modes

(which were not included in RFC4346 [RFC4346]) are discussed in Section 3.3.2.

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

TLS 1.3, specified in TLSv1.3 [RFC8446], represents a significant change to TLS that aims to address threats that have arisen over the years. Among the changes are a new handshake protocol, a new key derivation process that uses the HMAC-based Extract-and-Expand Key Derivation Function (HKDF), and the removal of cipher suites that use static RSA or DH key exchanges, the CBC mode of operation, or SHA-1. The list of extensions that can be used with TLS 1.3 has been reduced considerably.

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

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

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

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

See [Bhargavan2016] for additional detail.

4. Do Not Use TLSv1.0

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

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

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

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

5. Do Not Use TLSv1.1

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

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

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

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

6. Updates to RFC7525

RFC7525 is BCP195, "Recommendations for Secure Use of Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)", which is the most recent best practice document for implementing TLS and was based on TLSv1.2. At the time of publication, TLSv1.0 and TLSv1.1 had not yet been deprecated. As such, this document is called out specifically to update text implementing the deprecation recommendations of this document.

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

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

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

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

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

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

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

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

7. Security Considerations

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

8. Acknowledgements

Thanks to those that provided usage data, reviewed and/or improved this document, including: David Benjamin, David Black, Alan DeKok, Viktor Dukhovni, Julien Elie, Gary Gapinski, Alessandro Ghedini, Jeremy Harris, James Hodgkinson, Russ Housley, Hubert Kario, Benjamin Kaduk, John Mattsson, Eric Mill, Yoav Nir, Andrei Popov, Eric

Rescorla, Yaron Sheffer, Robert Sparks, Martin Thomson, Loganaden Velvindron, and Jakub Wilk.

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

9. IANA Considerations

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

10. References

10.1. Normative References

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

[RFC2246] Dierks, T. and C. Allen, "The TLS Protocol Version 1.0", RFC 2246, DOI 10.17487/RFC2246, January 1999, <https://www.rfc-editor.org/info/rfc2246>.

[RFC3261] Rosenberg, J., Schulzrinne, H., Camarillo, G., Johnston, A., Peterson, J., Sparks, R., Handley, M., and E. Schooler, "SIP: Session Initiation Protocol", RFC 3261, DOI 10.17487/RFC3261, June 2002, <https://www.rfc-editor.org/info/rfc3261>.

[RFC3329] Arkko, J., Torvinen, V., Camarillo, G., Niemi, A., and T. Haukka, "Security Mechanism Agreement for the Session Initiation Protocol (SIP)", RFC 3329, DOI 10.17487/RFC3329, January 2003, <https://www.rfc-editor.org/info/rfc3329>.

[RFC3436] Jungmaier, A., Rescorla, E., and M. Tuexen, "Transport Layer Security over Stream Control Transmission Protocol", RFC 3436, DOI 10.17487/RFC3436, December 2002, <https://www.rfc-editor.org/info/rfc3436>.

[RFC3470] Hollenbeck, S., Rose, M., and L. Masinter, "Guidelines for the Use of Extensible Markup Language (XML) within IETF Protocols", BCP 70, RFC 3470, DOI 10.17487/RFC3470, January 2003, <https://www.rfc-editor.org/info/rfc3470>.

[RFC3501] Crispin, M., "INTERNET MESSAGE ACCESS PROTOCOL - VERSION 4rev1", RFC 3501, DOI 10.17487/RFC3501, March 2003, <https://www.rfc-editor.org/info/rfc3501>.

[RFC3552] Rescorla, E. and B. Korver, "Guidelines for Writing RFC Text on Security Considerations", BCP 72, RFC 3552, DOI 10.17487/RFC3552, July 2003, <https://www.rfc-editor.org/info/rfc3552>.

[RFC3568] Barbir, A., Cain, B., Nair, R., and O. Spatscheck, "Known Content Network (CN) Request-Routing Mechanisms", RFC 3568, DOI 10.17487/RFC3568, July 2003, <https://www.rfc-editor.org/info/rfc3568>.

[RFC3656] Siemborski, R., "The Mailbox Update (MUPDATE) Distributed Mailbox Database Protocol", RFC 3656, DOI 10.17487/RFC3656, December 2003, <https://www.rfc-editor.org/info/rfc3656>.

[RFC3749] Hollenbeck, S., "Transport Layer Security Protocol Compression Methods", RFC 3749, DOI 10.17487/RFC3749, May 2004, <https://www.rfc-editor.org/info/rfc3749>.

[RFC3767] Farrell, S., Ed., "Securely Available Credentials Protocol", RFC 3767, DOI 10.17487/RFC3767, June 2004, <https://www.rfc-editor.org/info/rfc3767>.

[RFC3856] Rosenberg, J., "A Presence Event Package for the Session Initiation Protocol (SIP)", RFC 3856, DOI 10.17487/RFC3856, August 2004, <https://www.rfc-editor.org/info/rfc3856>.

[RFC3871] Jones, G., Ed., "Operational Security Requirements for Large Internet Service Provider (ISP) IP Network Infrastructure", RFC 3871, DOI 10.17487/RFC3871, September 2004, <https://www.rfc-editor.org/info/rfc3871>.

[RFC3887] Hansen, T., "Message Tracking Query Protocol", RFC 3887, DOI 10.17487/RFC3887, September 2004, <https://www.rfc-editor.org/info/rfc3887>.

[RFC3903] Niemi, A., Ed., "Session Initiation Protocol (SIP) Extension for Event State Publication", RFC 3903, DOI 10.17487/RFC3903, October 2004, <https://www.rfc-editor.org/info/rfc3903>.

[RFC3943] Friend, R., "Transport Layer Security (TLS) Protocol Compression Using Lempel-Ziv-Stac (LZS)", RFC 3943, DOI 10.17487/RFC3943, November 2004, <https://www.rfc-editor.org/info/rfc3943>.

[RFC3983] Newton, A. and M. Sanz, "Using the Internet Registry Information Service (IRIS) over the Blocks Extensible Exchange Protocol (BEEP)", RFC 3983, DOI 10.17487/RFC3983, January 2005, <https://www.rfc-editor.org/info/rfc3983>.

[RFC4097] Barnes, M., Ed., "Middlebox Communications (MIDCOM) Protocol Evaluation", RFC 4097, DOI 10.17487/RFC4097, June 2005, <https://www.rfc-editor.org/info/rfc4097>.

[RFC4111] Fang, L., Ed., "Security Framework for Provider-Provisioned Virtual Private Networks (PPVPNs)", RFC 4111, DOI 10.17487/RFC4111, July 2005, <https://www.rfc-editor.org/info/rfc4111>.

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

[RFC4168] Rosenberg, J., Schulzrinne, H., and G. Camarillo, "The Stream Control Transmission Protocol (SCTP) as a Transport for the Session Initiation Protocol (SIP)", RFC 4168, DOI 10.17487/RFC4168, October 2005, <https://www.rfc-editor.org/info/rfc4168>.

[RFC4217] Ford-Hutchinson, P., "Securing FTP with TLS", RFC 4217, DOI 10.17487/RFC4217, October 2005, <https://www.rfc-editor.org/info/rfc4217>.

[RFC4235] Rosenberg, J., Schulzrinne, H., and R. Mahy, Ed., "An INVITE-Initiated Dialog Event Package for the Session Initiation Protocol (SIP)", RFC 4235, DOI 10.17487/RFC4235, November 2005, <https://www.rfc-editor.org/info/rfc4235>.

[RFC4261] Walker, J. and A. Kulkarni, Ed., "Common Open Policy Service (COPS) Over Transport Layer Security (TLS)", RFC 4261, DOI 10.17487/RFC4261, December 2005, <https://www.rfc-editor.org/info/rfc4261>.

[RFC4279] Eronen, P., Ed. and H. Tschofenig, Ed., "Pre-Shared Key Ciphersuites for Transport Layer Security (TLS)", RFC 4279, DOI 10.17487/RFC4279, December 2005, <https://www.rfc-editor.org/info/rfc4279>.

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

[RFC4497] Elwell, J., Derks, F., Mourot, P., and O. Rousseau, "Interworking between the Session Initiation Protocol (SIP) and QSIG", BCP 117, RFC 4497, DOI 10.17487/RFC4497, May 2006, <https://www.rfc-editor.org/info/rfc4497>.

[RFC4513] Harrison, R., Ed., "Lightweight Directory Access Protocol (LDAP): Authentication Methods and Security Mechanisms", RFC 4513, DOI 10.17487/RFC4513, June 2006, <https://www.rfc-editor.org/info/rfc4513>.

[RFC4531] Zeilenga, K., "Lightweight Directory Access Protocol (LDAP) Turn Operation", RFC 4531, DOI 10.17487/RFC4531, June 2006, <https://www.rfc-editor.org/info/rfc4531>.

[RFC4540] Stiemerling, M., Quittek, J., and C. Cadar, "NEC's Simple Middlebox Configuration (SIMCO) Protocol Version 3.0", RFC 4540, DOI 10.17487/RFC4540, May 2006, <https://www.rfc-editor.org/info/rfc4540>.

[RFC4582] Camarillo, G., Ott, J., and K. Drage, "The Binary Floor Control Protocol (BFCP)", RFC 4582, DOI 10.17487/RFC4582, November 2006, <https://www.rfc-editor.org/info/rfc4582>.

[RFC4616] Zeilenga, K., Ed., "The PLAIN Simple Authentication and Security Layer (SASL) Mechanism", RFC 4616, DOI 10.17487/RFC4616, August 2006, <https://www.rfc-editor.org/info/rfc4616>.

[RFC4642] Murchison, K., Vinocur, J., and C. Newman, "Using Transport Layer Security (TLS) with Network News Transfer Protocol (NNTP)", RFC 4642, DOI 10.17487/RFC4642, October 2006, <https://www.rfc-editor.org/info/rfc4642>.

[RFC4680] Santesson, S., "TLS Handshake Message for Supplemental Data", RFC 4680, DOI 10.17487/RFC4680, October 2006, <https://www.rfc-editor.org/info/rfc4680>.

[RFC4681] Santesson, S., Medvinsky, A., and J. Ball, "TLS User Mapping Extension", RFC 4681, DOI 10.17487/RFC4681, October 2006, <https://www.rfc-editor.org/info/rfc4681>.

[RFC4712] Siddiqui, A., Romascanu, D., Golovinsky, E., Rahman, M., and Y. Kim, "Transport Mappings for Real-time Application Quality-of-Service Monitoring (RAQMON) Protocol Data Unit (PDU)", RFC 4712, DOI 10.17487/RFC4712, October 2006, <https://www.rfc-editor.org/info/rfc4712>.

[RFC4732] Handley, M., Ed., Rescorla, E., Ed., and IAB, "Internet Denial-of-Service Considerations", RFC 4732, DOI 10.17487/RFC4732, December 2006, <https://www.rfc-editor.org/info/rfc4732>.

[RFC4743] Goddard, T., "Using NETCONF over the Simple Object Access Protocol (SOAP)", RFC 4743, DOI 10.17487/RFC4743, December 2006, <https://www.rfc-editor.org/info/rfc4743>.

[RFC4744] Lear, E. and K. Crozier, "Using the NETCONF Protocol over the Blocks Extensible Exchange Protocol (BEEP)", RFC 4744, DOI 10.17487/RFC4744, December 2006, <https://www.rfc-editor.org/info/rfc4744>.

[RFC4785] Blumenthal, U. and P. Goel, "Pre-Shared Key (PSK) Ciphersuites with NULL Encryption for Transport Layer Security (TLS)", RFC 4785, DOI 10.17487/RFC4785, January 2007, <https://www.rfc-editor.org/info/rfc4785>.

[RFC4791] Daboo, C., Desruisseaux, B., and L. Dusseault, "Calendaring Extensions to WebDAV (CalDAV)", RFC 4791, DOI 10.17487/RFC4791, March 2007, <https://www.rfc-editor.org/info/rfc4791>.

[RFC4823] Harding, T. and R. Scott, "FTP Transport for Secure Peer-to-Peer Business Data Interchange over the Internet", RFC 4823, DOI 10.17487/RFC4823, April 2007, <https://www.rfc-editor.org/info/rfc4823>.

[RFC4851] Cam-Winget, N., McGrew, D., Salowey, J., and H. Zhou, "The Flexible Authentication via Secure Tunneling Extensible Authentication Protocol Method (EAP-FAST)", RFC 4851, DOI 10.17487/RFC4851, May 2007, <https://www.rfc-editor.org/info/rfc4851>.

[RFC4964] Allen, A., Ed., Holm, J., and T. Hallin, "The P-Answer-State Header Extension to the Session Initiation Protocol for the Open Mobile Alliance Push to Talk over Cellular", RFC 4964, DOI 10.17487/RFC4964, September 2007, <https://www.rfc-editor.org/info/rfc4964>.

[RFC4975] Campbell, B., Ed., Mahy, R., Ed., and C. Jennings, Ed., "The Message Session Relay Protocol (MSRP)", RFC 4975, DOI 10.17487/RFC4975, September 2007, <https://www.rfc-editor.org/info/rfc4975>.

[RFC4976] Jennings, C., Mahy, R., and A. Roach, "Relay Extensions for the Message Sessions Relay Protocol (MSRP)", RFC 4976, DOI 10.17487/RFC4976, September 2007, <https://www.rfc-editor.org/info/rfc4976>.

[RFC4992] Newton, A., "XML Pipelining with Chunks for the Internet Registry Information Service", RFC 4992, DOI 10.17487/RFC4992, August 2007, <https://www.rfc-editor.org/info/rfc4992>.

[RFC5018] Camarillo, G., "Connection Establishment in the Binary Floor Control Protocol (BFCP)", RFC 5018, DOI 10.17487/RFC5018, September 2007, <https://www.rfc-editor.org/info/rfc5018>.

[RFC5019] Deacon, A. and R. Hurst, "The Lightweight Online Certificate Status Protocol (OCSP) Profile for High-Volume Environments", RFC 5019, DOI 10.17487/RFC5019, September 2007, <https://www.rfc-editor.org/info/rfc5019>.

[RFC5023] Gregorio, J., Ed. and B. de hOra, Ed., "The Atom Publishing Protocol", RFC 5023, DOI 10.17487/RFC5023, October 2007, <https://www.rfc-editor.org/info/rfc5023>.

[RFC5024] Friend, I., "ODETTE File Transfer Protocol 2.0", RFC 5024, DOI 10.17487/RFC5024, November 2007, <https://www.rfc-editor.org/info/rfc5024>.

[RFC5049] Bormann, C., Liu, Z., Price, R., and G. Camarillo, Ed., "Applying Signaling Compression (SigComp) to the Session Initiation Protocol (SIP)", RFC 5049, DOI 10.17487/RFC5049, December 2007, <https://www.rfc-editor.org/info/rfc5049>.

[RFC5054] Taylor, D., Wu, T., Mavrogiannopoulos, N., and T. Perrin, "Using the Secure Remote Password (SRP) Protocol for TLS Authentication", RFC 5054, DOI 10.17487/RFC5054, November 2007, <https://www.rfc-editor.org/info/rfc5054>.

[RFC5091] Boyen, X. and L. Martin, "Identity-Based Cryptography Standard (IBCS) #1: Supersingular Curve Implementations of the BF and BB1 Cryptosystems", RFC 5091, DOI 10.17487/RFC5091, December 2007, <https://www.rfc-editor.org/info/rfc5091>.

[RFC5158] Huston, G., "6to4 Reverse DNS Delegation Specification", RFC 5158, DOI 10.17487/RFC5158, March 2008, <https://www.rfc-editor.org/info/rfc5158>.

[RFC5216] Simon, D., Aboba, B., and R. Hurst, "The EAP-TLS Authentication Protocol", RFC 5216, DOI 10.17487/RFC5216, March 2008, <https://www.rfc-editor.org/info/rfc5216>.

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

[RFC5263] Lonnfors, M., Costa-Requena, J., Leppanen, E., and H. Khartabil, "Session Initiation Protocol (SIP) Extension for Partial Notification of Presence Information", RFC 5263, DOI 10.17487/RFC5263, September 2008, <https://www.rfc-editor.org/info/rfc5263>.

[RFC5281] Funk, P. and S. Blake-Wilson, "Extensible Authentication Protocol Tunneled Transport Layer Security Authenticated Protocol Version 0 (EAP-TTLSv0)", RFC 5281, DOI 10.17487/RFC5281, August 2008, <https://www.rfc-editor.org/info/rfc5281>.

[RFC5364] Garcia-Martin, M. and G. Camarillo, "Extensible Markup Language (XML) Format Extension for Representing Copy Control Attributes in Resource Lists", RFC 5364, DOI 10.17487/RFC5364, October 2008, <https://www.rfc-editor.org/info/rfc5364>.

[RFC5422] Cam-Winget, N., McGrew, D., Salowey, J., and H. Zhou, "Dynamic Provisioning Using Flexible Authentication via Secure Tunneling Extensible Authentication Protocol (EAP-FAST)", RFC 5422, DOI 10.17487/RFC5422, March 2009, <https://www.rfc-editor.org/info/rfc5422>.

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

[RFC5734] Hollenbeck, S., "Extensible Provisioning Protocol (EPP) Transport over TCP", STD 69, RFC 5734, DOI 10.17487/RFC5734, August 2009, <https://www.rfc-editor.org/info/rfc5734>.

[RFC5878] Brown, M. and R. Housley, "Transport Layer Security (TLS) Authorization Extensions", RFC 5878, DOI 10.17487/RFC5878, May 2010, <https://www.rfc-editor.org/info/rfc5878>.

[RFC6042] Keromytis, A., "Transport Layer Security (TLS) Authorization Using KeyNote", RFC 6042, DOI 10.17487/RFC6042, October 2010, <https://www.rfc-editor.org/info/rfc6042>.

[RFC6176] Turner, S. and T. Polk, "Prohibiting Secure Sockets Layer (SSL) Version 2.0", RFC 6176, DOI 10.17487/RFC6176, March 2011, <https://www.rfc-editor.org/info/rfc6176>.

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

[RFC6739] Schulzrinne, H. and H. Tschofenig, "Synchronizing Service Boundaries and <mapping> Elements Based on the Location-to-Service Translation (LoST) Protocol", RFC 6739, DOI 10.17487/RFC6739, October 2012, <https://www.rfc-editor.org/info/rfc6739>.

[RFC6749] Hardt, D., Ed., "The OAuth 2.0 Authorization Framework", RFC 6749, DOI 10.17487/RFC6749, October 2012, <https://www.rfc-editor.org/info/rfc6749>.

[RFC6750] Jones, M. and D. Hardt, "The OAuth 2.0 Authorization Framework: Bearer Token Usage", RFC 6750, DOI 10.17487/RFC6750, October 2012, <https://www.rfc-editor.org/info/rfc6750>.

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

[RFC7465] Popov, A., "Prohibiting RC4 Cipher Suites", RFC 7465, DOI 10.17487/RFC7465, February 2015, <https://www.rfc-editor.org/info/rfc7465>.

[RFC7507] Moeller, B. and A. Langley, "TLS Fallback Signaling Cipher Suite Value (SCSV) for Preventing Protocol Downgrade Attacks", RFC 7507, DOI 10.17487/RFC7507, April 2015, <https://www.rfc-editor.org/info/rfc7507>.

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

[RFC7562] Thakore, D., "Transport Layer Security (TLS) Authorization Using Digital Transmission Content Protection (DTCP) Certificates", RFC 7562, DOI 10.17487/RFC7562, July 2015, <https://www.rfc-editor.org/info/rfc7562>.

[RFC7568] Barnes, R., Thomson, M., Pironti, A., and A. Langley, "Deprecating Secure Sockets Layer Version 3.0", RFC 7568, DOI 10.17487/RFC7568, June 2015, <https://www.rfc-editor.org/info/rfc7568>.

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

[RFC8422] Nir, Y., Josefsson, S., and M. Pegourie-Gonnard, "Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS) Versions 1.2 and Earlier", RFC 8422, DOI 10.17487/RFC8422, August 2018, <https://www.rfc-editor.org/info/rfc8422>.

10.2. Informative References

[Bhargavan2016]
Bhargavan, K. and G. Leuren, "Transcript Collision Attacks: Breaking Authentication in TLS, IKE, and SSH https://www.mitls.org/downloads/transcript-collisions.pdf", 2016.

[NIST800-52r2]
National Institute of Standards and Technology, "NIST SP800-52r2 https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-52r2.pdf", August 2019.

[RFC3316] Arkko, J., Kuijpers, G., Soliman, H., Loughney, J., and J. Wiljakka, "Internet Protocol Version 6 (IPv6) for Some Second and Third Generation Cellular Hosts", RFC 3316, DOI 10.17487/RFC3316, April 2003, <https://www.rfc-editor.org/info/rfc3316>.

[RFC3489] Rosenberg, J., Weinberger, J., Huitema, C., and R. Mahy, "STUN - Simple Traversal of User Datagram Protocol (UDP) Through Network Address Translators (NATs)", RFC 3489, DOI 10.17487/RFC3489, March 2003, <https://www.rfc-editor.org/info/rfc3489>.

[RFC3546] Blake-Wilson, S., Nystrom, M., Hopwood, D., Mikkelsen, J., and T. Wright, "Transport Layer Security (TLS) Extensions", RFC 3546, DOI 10.17487/RFC3546, June 2003, <https://www.rfc-editor.org/info/rfc3546>.

[RFC3588] Calhoun, P., Loughney, J., Guttman, E., Zorn, G., and J. Arkko, "Diameter Base Protocol", RFC 3588, DOI 10.17487/RFC3588, September 2003, <https://www.rfc-editor.org/info/rfc3588>.

[RFC3734] Hollenbeck, S., "Extensible Provisioning Protocol (EPP) Transport Over TCP", RFC 3734, DOI 10.17487/RFC3734, March 2004, <https://www.rfc-editor.org/info/rfc3734>.

[RFC3920] Saint-Andre, P., Ed., "Extensible Messaging and Presence Protocol (XMPP): Core", RFC 3920, DOI 10.17487/RFC3920, October 2004, <https://www.rfc-editor.org/info/rfc3920>.

[RFC4132] Moriai, S., Kato, A., and M. Kanda, "Addition of Camellia Cipher Suites to Transport Layer Security (TLS)", RFC 4132, DOI 10.17487/RFC4132, July 2005, <https://www.rfc-editor.org/info/rfc4132>.

[RFC4244] Barnes, M., Ed., "An Extension to the Session Initiation Protocol (SIP) for Request History Information", RFC 4244, DOI 10.17487/RFC4244, November 2005, <https://www.rfc-editor.org/info/rfc4244>.

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

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

[RFC4492] Blake-Wilson, S., Bolyard, N., Gupta, V., Hawk, C., and B. Moeller, "Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS)", RFC 4492, DOI 10.17487/RFC4492, May 2006, <https://www.rfc-editor.org/info/rfc4492>.

[RFC4507] Salowey, J., Zhou, H., Eronen, P., and H. Tschofenig, "Transport Layer Security (TLS) Session Resumption without Server-Side State", RFC 4507, DOI 10.17487/RFC4507, May 2006, <https://www.rfc-editor.org/info/rfc4507>.

[RFC4572] Lennox, J., "Connection-Oriented Media Transport over the Transport Layer Security (TLS) Protocol in the Session Description Protocol (SDP)", RFC 4572, DOI 10.17487/RFC4572, July 2006, <https://www.rfc-editor.org/info/rfc4572>.

[RFC4934] Hollenbeck, S., "Extensible Provisioning Protocol (EPP) Transport Over TCP", RFC 4934, DOI 10.17487/RFC4934, May 2007, <https://www.rfc-editor.org/info/rfc4934>.

[RFC5077] Salowey, J., Zhou, H., Eronen, P., and H. Tschofenig, "Transport Layer Security (TLS) Session Resumption without Server-Side State", RFC 5077, DOI 10.17487/RFC5077, January 2008, <https://www.rfc-editor.org/info/rfc5077>.

[RFC5081] Mavrogiannopoulos, N., "Using OpenPGP Keys for Transport Layer Security (TLS) Authentication", RFC 5081, DOI 10.17487/RFC5081, November 2007, <https://www.rfc-editor.org/info/rfc5081>.

[RFC5101] Claise, B., Ed., "Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of IP Traffic Flow Information", RFC 5101, DOI 10.17487/RFC5101, January 2008, <https://www.rfc-editor.org/info/rfc5101>.

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

[RFC5415] Calhoun, P., Ed., Montemurro, M., Ed., and D. Stanley, Ed., "Control And Provisioning of Wireless Access Points (CAPWAP) Protocol Specification", RFC 5415, DOI 10.17487/RFC5415, March 2009, <https://www.rfc-editor.org/info/rfc5415>.

[RFC5456] Spencer, M., Capouch, B., Guy, E., Ed., Miller, F., and K. Shumard, "IAX: Inter-Asterisk eXchange Version 2", RFC 5456, DOI 10.17487/RFC5456, February 2010, <https://www.rfc-editor.org/info/rfc5456>.

[RFC6012] Salowey, J., Petch, T., Gerhards, R., and H. Feng, "Datagram Transport Layer Security (DTLS) Transport Mapping for Syslog", RFC 6012, DOI 10.17487/RFC6012, October 2010, <https://www.rfc-editor.org/info/rfc6012>.

[RFC6083] Tuexen, M., Seggelmann, R., and E. Rescorla, "Datagram Transport Layer Security (DTLS) for Stream Control Transmission Protocol (SCTP)", RFC 6083, DOI 10.17487/RFC6083, January 2011, <https://www.rfc-editor.org/info/rfc6083>.

[RFC6084] Fu, X., Dickmann, C., and J. Crowcroft, "General Internet Signaling Transport (GIST) over Stream Control Transmission Protocol (SCTP) and Datagram Transport Layer Security (DTLS)", RFC 6084, DOI 10.17487/RFC6084, January 2011, <https://www.rfc-editor.org/info/rfc6084>.

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

[RFC6460] Salter, M. and R. Housley, "Suite B Profile for Transport Layer Security (TLS)", RFC 6460, DOI 10.17487/RFC6460, January 2012, <https://www.rfc-editor.org/info/rfc6460>.

[RFC6614] Winter, S., McCauley, M., Venaas, S., and K. Wierenga, "Transport Layer Security (TLS) Encryption for RADIUS", RFC 6614, DOI 10.17487/RFC6614, May 2012, <https://www.rfc-editor.org/info/rfc6614>.

[RFC7457] Sheffer, Y., Holz, R., and P. Saint-Andre, "Summarizing Known Attacks on Transport Layer Security (TLS) and Datagram TLS (DTLS)", RFC 7457, DOI 10.17487/RFC7457, February 2015, <https://www.rfc-editor.org/info/rfc7457>.

[RFC8143] Elie, J., "Using Transport Layer Security (TLS) with Network News Transfer Protocol (NNTP)", RFC 8143, DOI 10.17487/RFC8143, April 2017, <https://www.rfc-editor.org/info/rfc8143>.

[RFC8261] Tuexen, M., Stewart, R., Jesup, R., and S. Loreto, "Datagram Transport Layer Security (DTLS) Encapsulation of SCTP Packets", RFC 8261, DOI 10.17487/RFC8261, November 2017, <https://www.rfc-editor.org/info/rfc8261>.

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

[RFC8447] Salowey, J. and S. Turner, "IANA Registry Updates for TLS and DTLS", RFC 8447, DOI 10.17487/RFC8447, August 2018, <https://www.rfc-editor.org/info/rfc8447>.

Appendix A. Change Log

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

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

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

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

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

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

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

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

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

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

- PRs with typos and similar: so far just #1
- PR#2 noting msft browser announced deprecation (but this was OBE as per...)
- Implemented actions as per IETF-103 meeting:

  * Details about which RFC's, BCP's are affected were generated using a script in the git repo: https://github.com/tlswg/oldversions-deprecate/blob/master/nonobsnorms.sh
  * Removed the 'measurements' part
  * Removed SHA-1 deprecation (section 8 of -00)

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

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

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

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

Authors' Addresses

Kathleen Moriarty
Dell EMC
176 South Street
Hopkinton
United States

EMail: Kathleen.Moriarty.ietf@gmail.com

Stephen Farrell
Trinity College Dublin
Dublin 2
Ireland

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

TLS Working Group   P. Kampanakis, Ed.
Internet-Draft   Cisco
Intended status: Informational   M. Msahli, Ed.
Expires: April 25, 2019   Telecom ParisTech
                                                           October 22, 2018

TLS Authentication using ETSI TS 103 097 and IEEE 1609.2 certificates
draft-tls-certieee1609-02.txt

Abstract

This document specifies the use of a new certificate type to authenticate TLS entities. The first type enables the use of a certificate specified by the Institute of Electrical and Electronics Engineers (IEEE) and the European Telecommunications Standards Institute (ETSI).

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

The TLS protocol [RFC8446] [RFC5246] uses X509 and Raw Public Key in order to authenticate servers and clients. This document describes the use of certificates specified either by the Institute of Electrical and Electronics Engineers (IEEE) [IEEE1609.2] or the European Telecommunications Standards Institute (ETSI) [TS103097]. It is worth mentioning that the ETSI TS 103097 certificate is a profile of IEEE 1609.2 certificate and uses the same data structure. These standards are defined in order to secure communications in vehicular environments. Existing authentication methods, such as X509 and Raw Public Key, are designed for Internet use, particularly for flexibility and extensibility, and are not optimized for bandwidth and processing time to support delay-sensitive applications. That is why size-optimized certificates were standardized by ETSI and IEEE to secure data exchange in highly dynamic vehicular environment in Intelligent Transportation System (ITS).

2. Requirements Terminology

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

3. Extension Overview

This specification extends the Client Hello and Server Hello messages, by using the "extension_data" field of the ClientCertType Extension and the ServerCertType Extension structures defined in RFC7250. In order to negotiate the support of IEEE 1609.2 or ETSI TS 103097 certificate-based authentication, the clients and the servers MAY include the extension of type "client_certificate_type" and "server_certificate_type" in the extended Client Hello and "EncryptedExtensions". The "extension_data" field of this extension SHALL contain a list of supported certificate types proposed by the client as provided in the figure below:

```
  /* Managed by IANA */
   enum {
       X509(0),
       RawPublicKey(2),
       1609Dot2(?), /* Number 3 will be requested for 1609.2 */
       (255)
   } CertificateType;

   struct {
       select (certificate_type) {

           /* certificate type defined in this document.*/
            case 1609Dot2:
            opaque cert_data<1..2^24-1>;

            /* RawPublicKey defined in RFC 7250*/
           case RawPublicKey:
           opaque ASN.1_subjectPublicKeyInfo<1..2^24-1>;

           /* X.509 certificate defined in RFC 5246*/
           case X.509:
           opaque cert_data<1..2^24-1>;

            };

          Extension extensions<0..2^16-1>;
      } CertificateEntry;
```

In case where the TLS server accepts the described extension, it selects one of the certificate types in the extension described above. Note that a server MAY authenticate the client using other authentication methods. The end-entity certificate's public key has to be compatible with one of the certificate types listed in the extension described above.

4. TLS Client and Server Handshake

The "client_certificate_type" and "server_certificate_type" extensions MUST be sent in handshake phase as illustrated in Figure 1 below. The same extension shall be sent in Server Hello for TLS 1.2.

```
   Client                                           Server

 Key  ^ ClientHello
 Exch | + server_certificate_type*
      | + client_certificate_type*
      | + key_share*
      v + signature_algorithms*       -------->
                                                   ServerHello  ^ Key
                                                  + key_share*  v Exch
                                         {EncryptedExtensions}  ^ Server
                                    {+ server_certificate_type*}| Params
                                    {+ client_certificate_type*}|
                                         {CertificateRequest*}  v
                                                {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 with certificate type extension for Full TLS 1.3 Handshake

4.1. Client Hello

In order to indicate the support of IEEE 1609.2 or ETSI TS 103097 certificates, client MUST include an extension of type "client_certificate_type" and "server_certificate_type" in the extended Client Hello message. The Hello extension is described in Section 4.1.2 of TLS 1.3 [RFC8446].

The extension 'client_certificate_type' sent in the client hello MAY carry a list of supported certificate types, sorted by client preference. It is a list in the case where the client supports multiple certificate types.

Client MAY respond along with supported certificates by sending a "Certificate" message immediately followed by the "CetificateVerify" message. These specifications are valid for TLS 1.2 and TLS 1.3.

All implementations SHOULD be prepared to handle extraneous certificates and arbitrary orderings from any TLS version, with the exception of the end-entity certificate which MUST be first.

4.2. Server Hello

When the server receives the Client Hello containing the client_certificate_type extension and/or the server_certificate_type extension, the following options are possible:

- The server supports the extension described in this document. It selects a certificate type from the client_certificate_type field in the extended Client Hello and must take into account the client authentication list priority.

- The server does not support the proposed certificate type and terminates the session with a fatal alert of type "unsupported_certificate".

- The server does not support the extension defined in this document. In this case, the server returns the server hello without the extensions defined in this document in case of TLS 1.2.

- The server supports the extension defined in this document, but it does not have any certificate type in common with the client. Then, the server terminates the session with a fatal alert of type "unsupported_certificate".

- The server supports the extensions defined in this document and has at least one certificate type in common with the client. In this case, the server MUST include the client_certificate_type extension in the Server Hello for TLS 1.2 or in Encrypted Extension for TLS 1.3. Then, the server requests a certificate from the client (via the certificate_request message)

It is worth to mention that the TLS client or server public keys are obtained from a certificate chain from a web page.

5. Certificate Verification

Verification of an IEEE 1609.2/ ETSI TS 103097 certificates or certificate chain is described in section 5.5.2 of [IEEE1609.2].

6. Examples

Some of exchanged messages examples are illustrated in Figures 2 and 3.

6.1. TLS Server and TLS Client use the 1609Dot2 Certificate

This section shows an example where the TLS client as well as the TLS server use the IEEE 1609.2 certificate. In consequence, both the server and the client populate the client_certificate_type and server_certificate_type with extension IEEE 1609.2 certificates as mentioned in figure 2.

```
   Client                                                Server

ClientHello,
client_certificate_type*=1609Dot2,
server_certificate_type*=1609Dot2,   -------->     ServerHello,
                                            {EncryptedExtensions}
                              {client_certificate_type*=1609Dot2}
                              {server_certificate_type*=1609Dot2}
                                            {CertificateRequest*}
                                                   {Certificate*}
                                             {CertificateVerify*}
                                                       {Finished}
  {Certificate*}            <-------         [Application Data*]
  {CertificateVerify*}
  {Finished}               -------->
  [Application Data]       <------->          [Application Data]
```

Figure 2: TLS Client and TLS Server use the IEEE 1609.2 certificate

6.2. TLS Client uses the IEEE 1609.2 certificate and TLS Server uses the X 509 certificate

This example shows the TLS authentication, where the TLS Client populates the server_certificate_type extension with the X509 certificate and Raw Public Key type as presented in figure 3. the client indicates its ability to receive and to validate an X509 certificate from the server. The server chooses the X509 certificateto make its authentication with the Client.

```
  Client                                              Server

ClientHello,
client_certificate_type*=(1609Dot2),
server_certificate_type*=(1609.9Dot, X509,RawPublicKey), ------->    ServerHello
,
                                        {EncryptedExtensions}
                             {client_certificate_type*=1609Dot2}
                          {server_certificate_type*=X509}
                                               {Certificate*}
                                         {CertificateVerify*}
                                                   {Finished}
                            <---------    [Application Data*]
{Finished}                  --------->
[Application Data]          <-------->     [Application Data]
```

Figure 3: TLS Client uses the IEEE 1609.2 certificate and TLS Server uses the X 509 certificate

7. Security Considerations

This section provides an overview of the basic security considerations which need to be taken into account before implementing the necessary security mechanisms. The security considerations described throughout [RFC8446] and [RFC5246] apply here as well.

For security considerations in a vehicular environment, the minimal use of any TLS extensions is recommended such as :

The "client_certificate_type" [IANA value 19] extension who's purpose was previously described in [RFC7250].

The "server_certificate_type" [IANA value 20] extension who's purpose was previously described in [RFC7250].

The "SessionTicket" [IANA value 35] extension for session resumption.

In addition, servers SHOULD not support renegotiation [RFC5746] which presented Man-In-The-Middle (MITM) type attacks over the past years for TLS 1.2.

8. Privacy Considerations

For privacy considerations in a vehicular environment the use of EEE 1609.2/ETSI TS 103097 certificate is recommended for many reasons:

In order to address the risk of a personal data leakage, messages exchanged for V2V communications are signed using IEEE 1609.2/ETSI TS 103097 pseudonym certificates

The purpose of these certificates is to provide privacy relying on geographical and/or temporal validity criteria, and minimizing the exchange of private data

9. 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:

- o Value: TBD Description: 1609Dot2 Reference: [THIS RFC]

10. Acknowledgements

The authors wish to thank Eric Rescola and Ilari Liusvaara for their feedback and suggestions on improving this document. Thanks are due to Sean Turner for his valuable and detailed comments. Special thanks to William Whyte and Maik Seewald for their guidance and support in the early stages of the draft.

11. References

11.1. Normative References

[IEEE1609.2]
IEEE, "IEEE Standard for Wireless Access in Vehicular Environments - Security Services for Applications and Management Messages", 2016.

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

[RFC4492] Blake-Wilson, S., Bolyard, N., Gupta, V., Hawk, C., and B. Moeller, "Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS)", May 2006.

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

[RFC5746] Rescorla, E., Ray, M., Dispensa, S., and N. Oskov, "Transport Layer Security (TLS) Renegotiation Indication Extension"", February 2010.

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

[RFC8446] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", August 2018.

[TS103097] ETSI, "ETSI TS 103 097 v1.3.1 (2017-10): Intelligent Transport Systems (ITS); Security; Security header and certificate formats", October 2017.

11.2. Informative References

[draft-serhrouchni-tls-certieee1609-00] KAISER, A., LABIOD, H., LONC, B., MSAHLI, M., and A. SERHROUCHNI, "Transport Layer Security (TLS) Authentication using ITS ETSI and IEEE certificates", august 2017.

Appendix A. Co-Authors

- Nancy Cam-Winget
  CISCO, USA
  ncamwing@cisco.com

- Houda Labiod
  Telecom Paristech, France
  houda.labiod@telecom-paristech.fr

- Ahmed Serhrouchni
  Telecom ParisTech
  ahmed.serhrouchni@telecom-paristech.fr

Authors' Addresses

Panos Kampanakis (editor)
Cisco
USA

EMail: EMail: pkampana@cisco.com

Mounira Msahli (editor)
Telecom ParisTech
France

EMail: mounira.msahli@telecom-paristech.fr

tls D. Benjamin
Internet-Draft Google, LLC.
Intended status: Experimental C. Wood
Expires: September 12, 2019 Apple, Inc.
March 11, 2019

Importing External PSKs for TLS
draft-wood-tls-external-psk-importer-01

Abstract

This document describes an interface for importing external PSK (Pre-Shared Key) into TLS.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

TLS 1.3 [RFC8446] supports pre-shared key (PSK) resumption, wherein PSKs can be established via session tickets from prior connections or externally via some out-of-band mechanism. The protocol mandates that each PSK only be used with a single hash function. This was done to simplify protocol analysis. TLS 1.2, in contrast, has no such requirement, as a PSK may be used with any hash algorithm and the TLS 1.2 PRF. This means that external PSKs could possibly be re-used in two different contexts with the same hash functions during key derivation. Moreover, it requires external PSKs to be provisioned for specific hash functions.

To mitigate these problems, external PSKs can be bound to a specific hash function when used in TLS 1.3, even if they are associated with a different KDF (and hash function) when provisioned. This document specifies an interface by which external PSKs may be imported for use in a TLS 1.3 connection to achieve this goal. In particular, it describes how KDF-bound PSKs can be differentiated by different hash algorithms to produce a set of candidate PSKs, each of which are bound to a specific hash function. This expands what would normally have been a single PSK identity into a set of PSK identities. However, it requires no change to the TLS 1.3 key schedule.

2. Conventions and Definitions

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

14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

3. Overview

Intuitively, key importers mirror the concept of key exporters in TLS in that they diversify a key based on some contextual information before use in a connection. In contrast to key exporters, wherein differentiation is done via an explicit label and context string, the key importer defined herein uses a label and set of hash algorithms to differentiate an external PSK into one or more PSKs for use.

Imported keys do not require negotiation for use, as a client and server will not agree upon identities if not imported correctly. Thus, importers induce no protocol changes with the exception of expanding the set of PSK identities sent on the wire.

3.1. Terminology

- o External PSK (EPSK): A PSK established or provisioned out-of-band, i.e., not from a TLS connection, which is a tuple of (Base Key, External Identity, KDF). The associated KDF (and hash function) may be undefined.

- o Base Key: The secret value of an EPSK.

- o External Identity: The identity of an EPSK.

- o Imported Identity: The identity of a PSK as sent on the wire.

4. Key Import

A key importer takes as input an EPSK with external identity 'external_identity' and base key 'epsk', as defined in Section 3.1, along with an optional label, and transforms it into a set of PSKs and imported identities for use in a connection based on supported HashAlgorithms. In particular, for each supported HashAlgorithm 'hash', the importer constructs an ImportedIdentity structure as follows:

```
struct {
    opaque external_identity<1...2^16-1>;
    opaque label<0..2^8-1>;
    HashAlgorithm hash;
} ImportedIdentity;
```

[[TODO: An alternative design might combine label and hash into the same field so that future protocols which don't have a notion of HashAlgorithm don't need this field.]]

ImportedIdentity.label MUST be bound to the protocol for which the key is imported. Thus, TLS 1.3 and QUICv1 [I-D.ietf-quic-transport] MUST use "tls13" as the label. Similarly, TLS 1.2 and all prior TLS versions should use "tls12" as ImportedIdentity.label, as well as SHA256 as ImportedIdentity.hash. Note that this means future versions of TLS will increase the number of PSKs derived from an external PSK.

A unique and imported PSK (IPSK) with base key 'ipskx' bound to this identity is then computed as follows:

```
epskx = HKDF-Extract(0, epsk)
ipskx = HKDF-Expand-Label(epskx, "derived psk",
                          Hash(ImportedIdentity), Hash.length)
```

[[TODO: The length of ipskx MUST match that of the corresponding and supported ciphersuites.]]

The hash function used for HKDF [RFC5869] is that which is associated with the external PSK. It is not bound to ImportedIdentity.hash. If no hash function is specified, SHA-256 MUST be used. Differentiating epsk by ImportedIdentity.hash ensures that each imported PSK is only used with at most one hash function, thus satisfying the requirements in [RFC8446]. Endpoints MUST import and derive an ipsk for each hash function used by each ciphersuite they support. For example, importing a key for TLS_AES_128_GCM_SHA256 and TLS_AES_256_GCM_SHA384 would yield two PSKs, one for SHA256 and another for SHA384. In contrast, if TLS_AES_128_GCM_SHA256 and TLS_CHACHA20_POLY1305_SHA256 are supported, only one derived key is necessary.

The resulting IPSK base key 'ipskx' is then used as the binder key in TLS 1.3 with identity ImportedIdentity. With knowledge of the supported hash functions, one may import PSKs before the start of a connection.

EPSKs may be imported for early data use if they are bound to protocol settings and configurations that would otherwise be required for early data with normal (ticket-based PSK) resumption. Minimally, that means ALPN, QUIC transport settings, etc., must be provisioned alongside these EPSKs.

5. Deprecating Hash Functions

If a client or server wish to deprecate a hash function and no longer use it for TLS 1.3, they may remove this hash function from the set of hashes used during while importing keys. This does not affect the KDF operation used to derive concrete PSKs.

6. Backwards Compatibility

Recall that TLS 1.2 permits computing the TLS PRF with any hash algorithm and PSK. Thus, an external PSK may be used with the same KDF (and underlying HMAC hash algorithm) as TLS 1.3 with importers. However, critically, the derived PSK will not be the same since the importer differentiates the PSK via the identity and hash function. Thus, PSKs imported for TLS 1.3 are distinct from those used in TLS 1.2, and thereby avoid cross-protocol collisions.

7. Security Considerations

This is a WIP draft and has not yet seen significant security analysis.

8. Privacy Considerations

DISCLAIMER: This section contains a sketch of a design for protecting external PSK identities. It is not meant to be implementable as written.

External PSK identities are typically static by design so that endpoints may use them to lookup keying material. For some systems and use cases, this identity may become a persistent tracking identifier. One mitigation to this problem is encryption. Future drafts may specify a way for encrypting PSK identities using a mechanism similar to that of the Encrypted SNI proposal [I-D.ietf-tls-esni]. Another approach is to replace the identity with an unpredictable or "obfuscated" value derived from the corresponding PSK. One such proposal, derived from a design outlined in [I-D.ietf-dnssd-privacy], is as follows. Let ipskx be the imported PSK with identity ImportedIdentity, and N be a unique nonce of length equal to that of ImportedIdentity.hash. With these values, construct the following "obfuscated" identity:

```
struct {
    opaque nonce[hash.length];
    opaque obfuscated_identity<1..2^16-1>;
    HashAlgorithm hash;
} ObfuscatedIdentity;
```

ObfuscatedIdentity.nonce carries N, ObfuscatedIdentity.obfuscated_identity carries HMAC(ipskx, N), where HMAC is computed with ImportedIdentity.hash, and ObfuscatedIdentity.hash is ImportedIdentity.hash.

Upon receipt of such an obfuscated identity, a peer must lookup the corresponding PSK by exhaustively trying to compute ObfuscatedIdentity.obfuscated_identity using ObfuscatedIdentity.nonce and each of its known imported PSKs. If N is chosen in a predictable fashion, e.g., as a timestamp, it may be possible for peers to precompute these obfuscated identities to ease the burden of trial decryption.

9. IANA Considerations

This document makes no IANA requests.

10. References

10.1. Normative References

[I-D.ietf-quic-transport]
Iyengar, J. and M. Thomson, "QUIC: A UDP-Based Multiplexed and Secure Transport", draft-ietf-quic-transport-18 (work in progress), January 2019.

[RFC1035] Mockapetris, P., "Domain names - implementation and specification", STD 13, RFC 1035, DOI 10.17487/RFC1035, November 1987, <https://www.rfc-editor.org/info/rfc1035>.

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

[RFC5869] Krawczyk, H. and P. Eronen, "HMAC-based Extract-and-Expand Key Derivation Function (HKDF)", RFC 5869, DOI 10.17487/RFC5869, May 2010, <https://www.rfc-editor.org/info/rfc5869>.

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

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

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

10.2. Informative References

[I-D.ietf-dnssd-privacy]
Huitema, C. and D. Kaiser, "Privacy Extensions for DNS-SD", draft-ietf-dnssd-privacy-05 (work in progress), October 2018.

[I-D.ietf-tls-esni]
Rescorla, E., Oku, K., Sullivan, N., and C. Wood, "Encrypted Server Name Indication for TLS 1.3", draft-ietf-tls-esni-03 (work in progress), March 2019.

Appendix A. Acknowledgements

The authors thank Eric Rescorla and Martin Thomson for discussions that led to the production of this document, as well as Christian Huitema for input regarding privacy considerations of external PSKs.

Authors' Addresses

David Benjamin
Google, LLC.

Email: davidben@google.com

Christopher A. Wood
Apple, Inc.

Email: cawood@apple.com

Network Working Group
Internet-Draft
Intended status: Informational
Expires: April 16, 2019

T. Pauly
D. Schinazi
C. Wood
Apple Inc.

October 13, 2018

TLS Ticket Requests
draft-wood-tls-ticketrequests-01

Abstract

TLS session tickets enable stateless connection resumption for clients without server-side per-client state. Servers vend session tickets to clients, at their discretion, upon connection establishment. Clients store and use tickets when resuming future connections. Moreover, clients should use tickets at most once for session resumption, especially if such keying material protects early application data. Single-use tickets bound the number of parallel connections a client may initiate by the number of tickets received from a given server. To address this limitation, this document describes a mechanism by which clients may specify the desired number of tickets needed for future connections.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

As per [RFC5077], and as described in [RFC8446], TLS servers send clients session tickets at their own discretion in NewSessionTicket messages. Clients are in complete control of how many tickets they may use when establishing future and subsequent connections. For example, clients may open multiple TLS connections to the same server for HTTP, or may race TLS connections across different network interfaces. The latter is especially useful in transport systems that implement Happy Eyeballs [RFC8305]. Since connection concurrency and resumption is controlled by clients, a standard mechanism to request more than one ticket is desirable.

This document specifies a new TLS extension - ticket_request - that may be used by clients to express their desired number of session tickets. Servers may use this extension as a hint of the number of NewSessionTicket messages to vend. This extension is only applicable to TLS 1.3 [RFC8446] and future versions of TLS.

1.1. Requirements Language

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

2. Use Cases

The ability to request one or more tickets is useful for a variety of purposes:

- Parallel HTTP connections: To minimize ticket reuse while still improving performance, it may be useful to use multiple, distinct tickets when opening parallel connections. Clients must therefore bound the number of parallel connections they initiate by the number of tickets in their possession, or risk ticket re-use.

- Connection racing: Happy Eyeballs V2 [RFC8305] describes techniques for performing connection racing. The Transport Services Architecture implementation from [I-D.ietf-taps-impl] also describes how connections may race across interfaces and address families. In cases where clients have early data to send and want to minimize or avoid ticket re-use, unique tickets for each unique connection attempt are useful. Moreover, as some servers may implement single-use tickets (and even session ticket encryption keys), distinct tickets will be needed to prevent premature ticket invalidation by racing.

- Connection priming: In some systems, connections may be primed or bootstrapped by a centralized service or daemon for faster connection establishment. Requesting tickets on demand allows such services to vend tickets to clients to use for accelerated handshakes with early data. (Note that if early data is not needed by these connections, this method SHOULD NOT be used. Fresh handshakes SHOULD be performed instead.)

- Less ticket waste: Currently, TLS servers use application-specific, and often implementation-specific, logic to determine how many tickets to issue. By moving the burden of ticket count to clients, servers do not generate wasteful tickets for clients. Moreover, as ticket generation may involve expensive computation, e.g., public key cryptographic operations, avoiding waste is desirable.

3. Ticket Requests

Clients may indicate to servers their desired number of tickets via the following "ticket_request" extension:

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

Clients may send this extension in ClientHello. It contains the following structure:

```
struct {
    uint8 count;
} TicketRequestContents;
```

count The number of tickets desired by the client.

A supporting server MAY vend TicketRequestContents.count NewSessionTicket messages to a requesting client, and SHOULD NOT send more than TicketRequestContents.count NewSessionTicket messages to a requesting client. Servers SHOULD place a limit on the number of tickets they are willing to vend to clients. Thus, the number of NewSessionTicket messages sent should be the minimum of the server's self-imposed limit and TicketRequestContents.count. Servers MUST NOT send more than 255 tickets to clients.

Servers that support ticket requests MUST NOT echo "ticket_request" in the EncryptedExtensions.

4. IANA Considerations

IANA is requested to Create an entry, ticket_requests(TBD), in the existing registry for ExtensionType (defined in [RFC8446]), with "TLS 1.3" column values being set to "CH", and "Recommended" column being set to "Yes".

5. Security Considerations

Ticket re-use is a security and privacy concern. Moreover, ticket pooling as a means of avoiding or amortizing handshake costs must be used carefully. If servers do not rotate session ticket encryption keys frequently, clients may be encouraged to obtain and use tickets beyond common lifetime windows of, e.g., 24 hours. Despite ticket lifetime hints provided by servers, clients SHOULD dispose of pooled tickets after some reasonable amount of time that mimics the ticket rotation period.

6. Acknowledgments

The authors would like to thank David Benjamin, Eric Rescorla, Nick Sullivan, and Martin Thomson for discussions on earlier versions of this draft.

7. Normative References

[I-D.ietf-taps-impl]
Brunstrom, A., Pauly, T., Enghardt, T., Grinnemo, K., Jones, T., Tiesel, P., Perkins, C., and M. Welzl, "Implementing Interfaces to Transport Services", draft-ietf-taps-impl-01 (work in progress), July 2018.

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

[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>.

[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>.

[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>.

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

Authors' Addresses

Tommy Pauly
Apple Inc.
One Apple Park Way
Cupertino, California 95014
United States of America

Email: tpauly@apple.com

David Schinazi
Apple Inc.
One Apple Park Way
Cupertino, California 95014
United States of America

Email: dschinazi@apple.com

Christopher A. Wood
Apple Inc.
One Apple Park Way
Cupertino, California 95014
United States of America

Email: cawood@apple.com