

Network Working Group
Internet-Draft
Intended status: Informational
Expires: January 17, 2019

R. Barnes
O. Friel
Cisco
July 16, 2018

Usage of PAKE with TLS 1.3
draft-barnes-tls-pake-04

Abstract

The pre-shared key mechanism available in TLS 1.3 is not suitable for usage with low-entropy keys, such as passwords entered by users. This document describes an extension that enables the use of password-authenticated key exchange protocols with TLS 1.3.

Status of This Memo

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

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

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

This Internet-Draft will expire on January 17, 2019.

Copyright Notice

Copyright (c) 2018 IETF Trust and the persons identified as the document authors. All rights reserved.

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

Table of Contents

1. Introduction	2
2. Terminology	3
3. Setup	3
4. TLS Extensions	3
5. Compatible PAKE Protocols	5
6. SPAKE2+ Implementation	6
7. Pre-provisioned Parameters	6
8. Content of the TLS Extensions	7
9. Security Considerations	8
9.1. Security when using SPAKE2+	8
10. Open Items	9
10.1. PAKE Algorithm Negotiation	9
11. IANA Considerations	9
12. References	10
12.1. Normative References	10
12.2. Informative References	10
Authors' Addresses	11

1. Introduction

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

In some applications, it is desirable to enable a client and server to authenticate to one another using a low-entropy pre-shared value, such as a user-entered password.

In prior versions of TLS, this functionality has been provided by the integration of the Secure Remote Password PAKE protocol (SRP) [RFC5054]. The specific SRP integration described in RFC 5054 does not immediately extend to TLS 1.3 because it relies on the Client Key Exchange and Server Key Exchange messages, which no longer exist in 1.3.

TLS 1.3 itself provides a mechanism for authentication with pre-shared keys (PSKs). However, PSKs used with this protocol need to be "full-entropy", because the binder values used for authentication can be used to mount a dictionary attack on the PSK. So while the TLS 1.3 PSK mechanism is suitable for the session resumption cases for which it is specified, it cannot be used when the client and server share only a low-entropy secret.

Enabling TLS to address this use case effectively requires the TLS handshake to execute a password-authenticated key establishment

(PAKE) protocol. This document describes a TLS extension "pake" that can carry data necessary to execute a PAKE.

This extension is generic, in that it can be used to carry key exchange information for multiple different PAKEs. We assume that the client and server have pre-negotiated a choice of PAKE (and any required parameters) in addition to the password itself. As a first case, this document defines a concrete protocol for executing the SPAKE2+ PAKE protocol [I-D.irtf-cfrg-spake2].

2. 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].

The mechanisms described in this document also apply to DTLS 1.3 [I-D.ietf-tls-dtls13], but for brevity, we will refer only to TLS throughout.

3. Setup

In order to use this protocol, a TLS client and server need to have pre-provisioned the values required to execute the protocol:

- o A choice of PAKE protocol
- o Any parameters required by the PAKE protocol
- o A password (or a derived value as described by the PAKE protocol)

Servers will of course have multiple instances of this configuration information for different clients. Clients may also have multiple identities, even within a given server. We assume that in either case, a single opaque "identity" value is sufficient to identify the required parameters.

4. TLS Extensions

A client offers to authenticate with PAKE by including a "pake" extension in its ClientHello. The content of this extension is a "PAKEClientHello" value, providing a list of identities under which the client can authenticate, and for each identity, the client's first message from the underlying PAKE protocol.

If a client sends the "pake" extension, then it MAY also send the "key_share" and "pre_shared_key" extensions, to allow the server to choose an authentication mode. Unlike PSK-based authentication,

however, authentication with PAKE cannot be combined with the normal TLS ECDH mechanism. Forward secrecy is provided by the PAKE itself.

```
struct {  
    opaque identity<0..2^16-1>;  
    opaque pake_message<1..2^16-1>;  
} PAKEShare;  
  
struct {  
    PAKEShare client_shares<0..2^16-1>;  
} PAKEClientHello;
```

A server that receives a "pake" extension examines the list of client shares to see if there is one with an identity the server recognizes. If so, the server may indicate its choice of PAKE authentication by including a "pake" extension in its ServerHello. The content of this extension is a "PAKEServerHello" value, specifying the identity value for the password the server has selected, and the server's first message in the PAKE protocol.

Use of PAKE authentication is compatible with standard certificate-based authentication of both clients and servers. If a server includes an "pake" extension in its ServerHello, it may still send the Certificate and CertificateVerify messages, and/or send a CertificateRequest message to the client.

If a server uses PAKE authentication, then it MUST NOT send an extension of type "key_share", "pre_shared_key", or "early_data".

```
struct {  
    PAKEShare server_share;  
} PAKEServerHello;
```

Based on the messages exchanged in the ClientHello and ServerHello, the client and server execute the specified PAKE protocol to derive a shared key. This key is used as the "ECHD(E)" input to the TLS 1.3 key schedule.

As with client authentication via certificates, the server has not authenticated the client until after it has received the client's Finished message. When a server negotiates the use of this mechanism for authentication, it MUST NOT send application data before it has received the client's Finished message.

5. Compatible PAKE Protocols

In order to be usable with the "pake" extension, a PAKE protocol must specify some syntax for its messages, and the protocol itself must be compatible with the message flow described above. A specification describing the use of a particular PAKE protocol with TLS must provide the following details:

- o Parameters that must be pre-provisioned
- o Content of the "pake_message" field in a ClientHello
- o Content of the "pake_message" field in a ServerHello
- o How the PAKE protocol is executed based on those messages
- o How the outputs of the PAKE protocol are used to populate the "PSK" and "ECDH(E)" inputs to the TLS key schedule.

The underlying cryptographic protocol must be compatible with the message flow described above:

- o It must be possible to execute in one round-trip, with the client speaking first
- o The Finished MAC must provide sufficient key confirmation for the protocol, taking into account the contents of the handshake messages

In addition, to be compatible with the security requirements of TLS 1.3, PAKE protocols defined for use with TLS 1.3 MUST provide forward secrecy.

Several current PAKE protocols satisfy these requirements, for example:

- o SPAKE2+ (described below) [I-D.irtf-cfrg-spake2]
- o SPEKE and derivatives such as Dragonfly [speke] [I-D.harkins-tls-dragonfly]
- o OPAQUE [opaque]
- o SRP [RFC2945]

6. SPAKE2+ Implementation

7. Pre-provisioned Parameters

In order to use SPAKE2+, a TLS client and server need to have pre-provisioned the values required to execute the SPAKE2+ protocol (see Section 3.1 of [I-D.irtf-cfrg-spake2]):

- o A DH group of order "p*h", with "p" a large prime, and generator "G"
- o Fixed elements "M" and "N" for the group
- o A hash function "H"
- o A password "pw"

Note that the hash function "H" might be different from the hash function associated with the ciphersuite negotiated by the two parties. The hash function "H" MUST be a hash function suitable for hashing passwords, e.g., Argon2 or scrypt [I-D.irtf-cfrg-argon2] [RFC7914].

The TLS client and server roles map to the "A" and "B" roles in the SPAKE2+ specification, respectively. The identity of the server is the domain name sent in the "server_name" extension of the ClientHello message. The identity of the client is an opaque octet string, specified in the "spake2" ClientHello extension, defined below.

From the shared password, each party computes two shared integers "w0" and "w1" by running the following algorithm twice (changing the "context" value each time):

```
struct {  
    uint16 context;  
    opaque client_identity<0..255>;  
    opaque server_name<0..255>;  
    opaque password<0..255>;  
} PasswordInput;
```

- o Encode the following values into a "PasswordInput" structure:
 - * "client_identity": The client's identity, as described above.
 - * "server_name": The server's identity, as described above.
 - * "password": The password "pw"

- * "context": One of the following values:
 - + 0x7730, when generating "w0"
 - + 0x7731, when generating "w1"
- o Use the hash function "H" with the encoded "PasswordInput" structure as input to derive an "n"-byte string, where "n" is the byte-length of "p".
- o Interpret the "n"-bit string as an integer "w" in network byte order. Return the result $(w \% p) * h$ of reducing "w" mod p and multiplying it by "h".

Servers MUST store only the value "w0" and the product $L = w1 * G$, where "G" is the fixed generator of the group. Clients will need to have access to the values "w0" and "w1" directly, but SHOULD generate these values dynamically, rather than caching them.

8. Content of the TLS Extensions

The content of a "pake_message" in a ClientHello is the client's key share "T". The value "T" is computed as specified in [I-D.irtf-cfrg-spake2], as $T = w * M + X$, where "M" is a fixed value for the DH group and "X" is the public key of a fresh DH key pair. The format of the key share "T" is the same as for a "KeyShareEntry.key_exchange" value from the same group.

The content of a "pake_message" in a ServerHello is the server's key share "S". The value "S" is computed as specified in [I-D.irtf-cfrg-spake2], as $S = w * N + Y$, where "N" is a fixed value for the DH group and "Y" is the public key of a fresh DH key pair. The format of the key share "S" is the same as for a "KeyShareEntry.key_exchange" value from the same group.

Based on these messages, both the client and server can compute the two shared values as specified in [I-D.irtf-cfrg-spake2].

Name	Value	Client	Server
Z	$x * y * G$	$x * (S - w0 * N)$	$x * (T - w0 * M)$
V	$w1 * y * G$	$w1 * (S - w0 * N)$	$y * L$

The following value is used as the "(EC)DHE" input to the TLS 1.3 key schedule:

$$K = H(Z \parallel V)$$

Here "H" is the hash function corresponding to the TLS cipher suite in use and " \parallel " represents concatenation of octet strings.

9. Security Considerations

Many of the security properties of this protocol will derive from the PAKE protocol being used. Security considerations for PAKE protocols are noted in Section 5.

The mechanism defined in this document does not provide protection for the client's identity, in contrast to TLS client authentication with certificates.

TLS servers that offer this mechanism can be used by third party attackers as an oracle for two questions:

1. Whether the server knows about a given identity
2. Whether the server recognizes a given (identity, password) pair

The former is signaled by whether the server returns a "pake" extension.

[[TODO: Similar to <https://tools.ietf.org/html/rfc5054#section-2.5.1.3>, the server could run through a complete handshake calculation and fail at the end so that the attacker only knows that the identity/password pair is incorrect, but does not know if the identity is recognized or not. This requires that the server can interpret the pake_message and ascertain the associated PAKE algorithm, group parameters, etc., which requires a reworking of some text in this draft as the identity is currently defined as providing a map to said group parameters. This is related to the discussion in the Open Items section.]]

The latter is signaled by whether the connection succeeds. These oracles are all-or-nothing: If the attacker does not have the correct identity or password, he does not learn anything about the correct value.

9.1. Security when using SPAKE2+

For the most part, the security properties of the password-based authentication described in this document are the same as those described in the Security Considerations of [I-D.irtf-cfrg-spake2]. The TLS Finished MAC provides the key confirmation required for the security of the protocol. Note that all of the elements covered by

the example confirmation hash listed in that document are also covered by the Finished MAC:

- o "A", "B", and "T" are included via the ClientHello
- o "S" via the ServerHello
- o "K", and "w" via the TLS key schedule

The "x" and "y" values used in the SPAKE2 protocol MUST have the same ephemerality properties as the key shares sent in the "key_shares" extension. In particular, "x" and "y" MUST NOT be equal to zero. This ensures that TLS sessions using SPAKE2 have the same forward secrecy properties as sessions using the normal TLS (EC)DH mechanism.

10. Open Items

10.1. PAKE Algorithm Negotiation

It is possible that a client may know the password to use, but may not know in advance which PAKE protocols(s) a particular server supports. A potential solution to this is similar to TLS1.3 ClientHello "key_share" operation: the client may send an empty "client_shares" vector in its PAKEClientHello extension. The server can then send an HelloRetryRequest indicating which PAKE protocol, and associated group parameters, the client should use. The client then sends another ClientHello that includes "pake_message" in the PAKEClientHello extension calculated using the correct algorithm. This requires definition of a suitable field for transporting PAKE algorithm and group parameters.

As an optimisation, similar to TLS1.3 key_share operation, the client could guess the PAKE protocol and include a "pake_message" derived from its guess in the initial ClientHello. If the server does not support the selected PAKE protocol (or protocol group parameter, etc.), the server can send an HelloRetryRequest indicating the supported PAKE protocol and group parameters. Note: it is TBD if sending two different "pake_messages" derived from two different protocol and/or group parameters in two different ClientHello messages constitutes a significant attack vector. This needs cryptographic review.

11. IANA Considerations

This document requests that IANA add a value to the TLS ExtensionType Registry with the following contents:

Value	Extension Name	TLS 1.3	Reference
TBD	pake	CH, SH	RFC XXXX

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

12. References

12.1. Normative References

- [I-D.ietf-tls-dtls13]
Rescorla, E., Tschofenig, H., and N. Modadugu, "The Datagram Transport Layer Security (DTLS) Protocol Version 1.3", draft-ietf-tls-dtls13-28 (work in progress), July 2018.
- [I-D.irtf-cfrg-spake2]
Ladd, W. and B. Kaduk, "SPAKE2, a PAKE", draft-irtf-cfrg-spake2-05 (work in progress), February 2018.
- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <<https://www.rfc-editor.org/info/rfc2119>>.

12.2. Informative References

- [I-D.harkins-tls-dragonfly]
Harkins, D., "Secure Password Ciphersuites for Transport Layer Security (TLS)", draft-harkins-tls-dragonfly-03 (work in progress), July 2018.
- [I-D.irtf-cfrg-argon2]
Biryukov, A., Dinu, D., Khovratovich, D., and S. Josefsson, "The memory-hard Argon2 password hash and proof-of-work function", draft-irtf-cfrg-argon2-03 (work in progress), August 2017.
- [opaque] Xu, J., "OPAQUE: An Asymmetric PAKE Protocol Secure Against Pre-Computation Attacks", 2018.
- [RFC2945] Wu, T., "The SRP Authentication and Key Exchange System", RFC 2945, DOI 10.17487/RFC2945, September 2000, <<https://www.rfc-editor.org/info/rfc2945>>.

- [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>>.
- [RFC7914] Percival, C. and S. Josefsson, "The scrypt Password-Based Key Derivation Function", RFC 7914, DOI 10.17487/RFC7914, August 2016, <<https://www.rfc-editor.org/info/rfc7914>>.
- [speke] Jablon, D., "Extended Password Key Exchange Protocols Immune to Dictionary Attacks", 1997.

Authors' Addresses

Richard Barnes
Cisco

Email: rlb@ipv.sx

Owen Friel
Cisco

Email: ofriel@cisco.com

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

R. Housley
Vigil Security
November 8, 2018

TLS 1.3 Extension for Certificate-based Authentication with an External
Pre-Shared Key
draft-housley-tls-tls13-cert-with-extern-psk-03

Abstract

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

Status of This Memo

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

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

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

This Internet-Draft will expire on May 12, 2019.

Copyright Notice

Copyright (c) 2018 IETF Trust and the persons identified as the document authors. All rights reserved.

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

1. Introduction

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

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 be 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 extensions 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 be accompanied by the "key_share" and "pre_shared_key" extensions. If none of the external PSKs in the list provided by the client is acceptable to the server, then the "tls_cert_with_extern_psk" extension is omitted from the ServerHello message.

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 "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>;
} OfferedPsk;

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

The OfferedPsk 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

Internet Engineering Task Force
Internet-Draft
Intended status: Standards Track
Expires: December 27, 2018

J. Hoyland, Ed.
Royal Holloway, University of London
June 25, 2018

Layered Exported Authenticators in TLS
draft-hoyland-tls-layered-exported-authenticator-00

Abstract

This document describes an extension that allows for Exported Authenticators (EAs) to authenticate each other. The extension includes a reference to a previous EA. An EA containing this extension constitutes an attestation of the authenticity of the referenced EA.

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 December 27, 2018.

Copyright Notice

Copyright (c) 2018 IETF Trust and the persons identified as the document authors. All rights reserved.

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

Table of Contents

1. Introduction	2
1.1. Requirements Language	3
2. Extension Format	3
3. Acknowledgements	4
4. IANA Considerations	4
5. Security Considerations	4
6. References	5
6.1. Normative References	5
6.2. Informative References	5
Author's Address	5

1. Introduction

Exported Authenticators (EAs)[EA] provide a method for authenticating one party of a Transport Layer Security (TLS) communication to the other after the session has been established. EAs are defined for TLS 1.3[TLS13] and TLS 1.2 with extended master secret, RFC 7627 [RFC7627]. Multiple EAs sent on the same channel do not prove joint authentication. They prove that the sender is individually authoritative over each certificate, but not jointly authoritative over all certificates. By including this extension a sender can prove joint authentication. This extension can be included in CertificateRequest messages and Certificate messages.

Joint authentication could be used, for example, to securely update pinned certificates. When a client connects to a server for which it has a pinned certificate, the server could send the new certificate to be pinned, and then bind the previously pinned certificate to it. This proves to the client that the server is jointly authoritative over both certificates. To defeat this mechanism an attacker is required to both compromise the key of the old certificate and improperly obtain a certificate from the PKI.

Another potential use is to provide proof that a certificate has been accepted. Because EAs do not have a response mechanism, the sender of an EA does not know the receiver's view of its authentication status. By using this extension to reference EAs sent by its peer, a party can prove to its peer that it has accepted a particular certificate.

By constructing a chain of referenced EAs complex joint authentication properties can be achieved.

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 RFC 2119 [RFC2119].

2. Extension Format

The "extension_data" field of this extension SHALL contain:

```
struct {  
    opaque prev_certificate_request_context<0..2^8-1>;  
    opaque binding[Hash.length];  
} LayeredEA;
```

where "prev_certificate_request_context" is the certificate request context of the EA you are referencing, and "binding" is the Finished message of that same EA. The hash used is that used in the exported authenticator, which is the hash function used by the TLS connection.

A party who wishes its peer to prove it is jointly authoritative over multiple certificates can request a sequence of certificates, each bound to its predecessor. Receipt of a series of EAs binding these certificates into a chain proves the sender is jointly authoritative over all those certificates.

A party who receives a CertificateRequest with this extension MUST verify that it previously received or sent an EA with the appropriate certificate request context and Finished message. If so then the party MAY respond with a Certificate fulfilling the request, or it MAY choose to not fulfil the request.

A party who receives a request from its peer for which it does not recognise the referenced certificate or does not want to link to the referenced certificate for some other reason, but still wishes to respond with an EA MAY send an EA omitting the extension, or it MAY choose to not fulfil the request. If the peer receives an EA with the extension omitted it proves the sender is authoritative over the certificate in the EA, but makes no claims about the previous EA referenced in the request.

For spontaneous certificates The server MUST include a unique (within the context of the connection) certificate_request_context for any EA it may wish to bind to. To be able to verify bindings both parties must keep a list of accepted EAs they are willing to bind to, including certificate_request_contexts and Finished messages. A client that receives a spontaneous EA with a

certificate_request_context that it has already seen and for which it is willing to receive a binding MUST ignore it.

3. Acknowledgements

4. IANA Considerations

This document requests IANA to update the TLS ExtensionsType registry, defined in [TLS13], to include the layered_exported_authenticator extension.

5. Security Considerations

For the authentication guarantees to apply, requests, and thus responses, must unambiguously identify previous EAs. Because EAs do not place a restriction on both parties to a connection using the same certificate_request_context, the certificate_request_context is not sufficient to unambiguously identify previous EAs. Because EAs are unidirectional, and the Finished message is dependent on the labels used to enforce this, the Finished message is sufficient to identify previous EAs unambiguously. In the case of spontaneous EAs a malicious server or an attacker who had compromised the TLS channel could send two identical spontaneous EAs. To militate against this a client receiving such an EA MUST check that it has not already accepted an EA with the same certificate_request_context that it is willing to bind to. If it previously accepted such a certificate but did not add it to the list of certificates which it was willing to bind to, adding it to the list is still secure. The certificate_request_context is included in the request to ease identification of the previous EA, but is not sufficient alone.

Both parties can be sure the Finished messages that are used to reference previous EAs are unique. For requested EAs the inclusion of the certificate_request_context, which is generated by the requestor, guarantees this is the case. For spontaneous certificates the client may only accept EAs after checking it does not have any EAs it is willing to bind to with the same certificate_request_context.

The Finished messages amount to channel bindings as defined in RFC5056 [RFC5056], and thus publication of them should not weaken the security of either the referenced EA or the TLS channel.

This extension only authenticates prior EAs. Thus, an attacker who is able to compromise a TLS connection could append authentications to the connection. Any attempt to bind to these certificates by an honest agent would not be accepted by the peer.

6. References

6.1. Normative References

- [EA] Sullivan, N., "Exported Authenticators in TLS", draft-ietf-tls-exported-authenticator-07 (work in progress), June 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>>.
- [RFC7627] Bhargavan, K., Ed., Delignat-Lavaud, A., Pironti, A., Langley, A., and M. Ray, "Transport Layer Security (TLS) Session Hash and Extended Master Secret Extension", RFC 7627, DOI 10.17487/RFC7627, September 2015, <<https://www.rfc-editor.org/info/rfc7627>>.
- [TLS13] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", draft-ietf-tls-tls13-28 (work in progress), March 2018.

6.2. Informative References

- [RFC5056] Williams, N., "On the Use of Channel Bindings to Secure Channels", RFC 5056, DOI 10.17487/RFC5056, November 2007, <<https://www.rfc-editor.org/info/rfc5056>>.

Author's Address

Jonathan Hoyland (editor)
Royal Holloway, University of London
Egham
UK

Email: jonathan.hoyland@gmail.com

TLS
Internet-Draft
Intended status: Standards Track
Expires: September 22, 2018

M. Shore
Fastly
R. Barnes
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

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 <http://datatracker.ietf.org/drafts/current/>.

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

This Internet-Draft will expire on September 22, 2018.

Copyright Notice

Copyright (c) 2018 IETF Trust and the persons identified as the document authors. All rights reserved.

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

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

Table of Contents

1. Requirements Notation	2
2. Introduction	2
3. DNSSEC Authentication Chain Extension	3
3.1. Protocol, TLS 1.2	3
3.2. Protocol, TLS 1.3	4
3.3. Raw Public Keys	4
3.4. DNSSEC Authentication Chain Data	5
4. Construction of Serialized Authentication Chains	7
5. Caching and Regeneration of the Authentication Chain	8
6. Verification	9
7. Trust Anchor Maintenance	9
8. Mandating use of this extension	9
9. DANE and Traditional PKIX Interoperation	10
10. Security Considerations	11
11. IANA Considerations	11
12. Acknowledgments	11
13. References	11
13.1. Normative References	11
13.2. Informative References	12
Appendix A. Test vectors	14
A.1. _443._tcp.www.example.com	15
A.2. _25._tcp.example.com wildcard	18
A.3. _443._tcp.www.example.org CNAME	20
A.4. _443._tcp.www.example.net DNAME	21

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:

- o 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,
- o 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 EEkey.

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..2¹⁶-1>

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.

$RR(i) = \text{owner} \mid \text{type} \mid \text{class} \mid \text{TTL} \mid \text{RDATA length} \mid \text{RDATA}$

where $RR(i)$ denotes the i th 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 RRsigs 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 RRsigs 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:

$\text{signature} = \text{sign}(\text{RRSIG_RDATA} \mid \text{RR}(1) \mid \text{RR}(2) \dots)$

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
MQswCQYDVQQGEwJVUzEVMBMGA1UEChMMRGlnaUNlcnQgSW5jMRkwFwYDVQQLEExB
d3cuZGlnaWNlcnQuY29tMS8wLQYDVQQDEyZEaWdpQ2VydCBTSEEyIEhpZ2ggQXNz
dXJhbmNlIFNlcnZlciBDQTAeFw0xNTEyMDMwMDAwMDBaFw0xODEyMjg0MDAwMDA
MIGlMQswCQYDVQQGEwJVUzEVMBMGA1UECBMhQ2FsaWZvcm5pYTEUMBIGA1UEBxML
TG9zIEFuZ2VsZXN0PDA6BgNVBAoTM0ludGVybmV0IENvcnBvcnF0aW9uIGZvciBB
c3NpZ25lZCB0YXN0cyBhbmQgTnVtYmVyczETMBEGA1UECzMkVGVjaG5vbG9neTEY
MBYGA1UEAxMPd3d3LmV4YW1wbGUub3JnMIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8A
MIIBCgKCAQEAs0CWL2FjPiXB1611rfvve0KzLJmG9LWAC3bcBjgsH6NiVVo2dt6u
Xfzi5bTm7F3K7srfUBYkLO78mraM9qizrHoIeyofrV/n+pZZJauQsPjCPxMEJnRo
D8Z4KpWKX0LyDu1SputoI4nlQ/htEhtiQnuoBfNZxF7WxcxGwEsZuS1KcXIkH15V
RJ0reKFHTaXcBlqcZ/QRaBIv0yhxvKlyBTwWddT4cli6GfHcCe3xGMSL328Fgs3
jYrvG29PueB6VJi/tbbPu6qTfwp/Hlbrqjdjh29U52Bhb0fJkM9DWxCP/Cattcc7a
z8EXnCO+LK8vkhw/kAiJWPkx4RBvgy73nwIDAQABo4ICUDCCAKwwHwYDVR0jBBgw
FoAUUWj/kK8CB3U8zn1lZGKiErhZcjsWHQYDVR0OBByEFKZPYB4fLdHn8SogKpUW
5Oia6m5IMIGBBGnVHREeEjB4gg93d3cuZXhhbXBsZS5vcmeCC2V4YW1wbGUuY29t
ggtleGFtcGx1LmVkdYILZXhhbXBsZS5uZXSCC2V4YW1wbGUub3Jngg93d3cuZXhh
bXBsZS5jb22CD3d3dy5leGFtcGx1LmVkdYIPd3d3LmV4YW1wbGUubmV0MA4GA1Ud
DwEB/wQEAwIFoDAdBgNVHSUEFjAUBggrBgEFBQcDAQYIKwYBBQUHAWIwdQYDVR0f
BG4wbDA0oDKgMIYuaHR0cDovL2NybdMuZGlnaWNlcnQuY29tL3NoYTItaGEtc2Vy
dmVYLWc0LmNybdA0oDKgMIYuaHR0cDovL2NybdQuZGlnaWNlcnQuY29tL3NoYTIt
aGEtc2VydMvYLWc0LmNybdBMBGnVHSAERTBDMDCGCWCGSAGG/WwBATAqMCgGCCsG
AQUFBwIBFhxodHRwczovL3d3dy5kaWdpY2VydC5jb20vQ1BTMAgGBmeBDAECAjCB
gwYIKwYBBQUHAQEEdzBlMCQGCCsGAQUFBzABhhhodHRwOi8vb2NzcC5kaWdpY2VydC5jb20wTQYIKwYBBQUHMAKGQWh0dHA6Ly9jYWNlcnRzLmRwZ21jZXJ0LmNvbS9E
aWdpQ2VydFNIQTJiawdoQXNzdXJhbmNlU2VydMvYQ0EuY3J0MAwGA1UdEwEB/wQC
MAAwDQYJKoZIhvcNAQELBQADggEBAISomhGn2L0LJn5SJHuyVZ3qMlRCIdvqe0Q
6ls+C8ctRwRO3UU3x8q8OH+2ahxlQmpzdc5al4XQzJLiLjiJ2Q1p+hub8MFIMmVP
PZjb2tZm2ipWVuMRM+zgpRVM6nVJ9F3vFFUSHOb4/JseIUvPY+d8/Krc+kPQwLvy
ieqRbcuFjmgyPmUv1U9QoI4TQikpw7TZU0zYZANP4C/gj4Ry48/znmUarvy2kvI
l7gRQ21qJTK5suoiYoYNo3J9T+pXPGU7Lydz/HwW+w0DpArtAaukI8aNX4ohFUKS
wDSiIIWIWJiJGbeEIO0TIFwEVWTONbnl/faPXpk5IRXicapqiII=
-----END CERTIFICATE-----
```

For brevity and reproducibility 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
        c66bef6a5cla3e78b82016e13f314f3cc5fa25ble52aab9adb9ec5989b165
        ada )
_443._tcp.www.example.com. 3600 IN RRSIG ( TLSA 13 5 3600
        20181128000000 20151103000000 1870 example.com.
        uml1DUjp5RfrXn9WtuMxEQV+ygzrONcuzsnfyOGSszwaDdkSOJ0Kndcfbb2I1
        LUV04Z+V488+Sdljr7/2ltsKA== )
example.com. 3600 IN DNSKEY ( 257 3 13
        JnAlXgyJTZz+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.
        ltua9ntAqZvOnK5UztzIjN38Bqs6mJ8KAT7L4+AxevDL+z0Jft7RC1/g6Qrfa
        InlwqF4U7TvC8PYOD0U/HYtwQ== )
com. 900 IN DNSKEY ( 256 3 13
        7IIE5Dol8jSMUqHTvOOiZapdEbQ9wqRxFi/zQcSdufUKLhpByvLpzSAQTqCWj
        3URIZ8L3Fa2gBLMOZUzZ1GQCw== ) ; Key ID = 34327
com. 900 IN DNSKEY ( 257 3 13
        Rbkco+96XZmnp8jYIuM4lryAp3egQjSmBaSoiA7H76Tm0RLHPNPuxlVn+nQ0f
        Ic3I8xfZDNw8Wa0Pe3/g2QA/w== ) ; Key ID = 18931
com. 900 IN DNSKEY ( 257 3 13
        szc7biLo5J4OHlkanlvZrF4aD4YYf+NHA/GAqdNslY9xxK9Izg68XHkqck4Rt
        DiVk37lNAQmgSlHbrGu0yOTkA== ) ; Key ID = 28809
com. 900 IN RRSIG ( DNSKEY 13 1 900 20181128000000
```

```

20151103000000 18931 com.
1ZmTBrfcRgVbqHJIfCVr6c3HUDgy3MlNSCSnrVV2S5/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/frMmxyG0tRg== ) ; 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.
    e7Q5L2x7Ca3SkSY6pRjqgtRxxENluYUcgyMlPp6GQ4zxAZxo01Y1vGqxN4eNA
    +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+jK94Fxa2JsaZChHln/DsYb5KS7J
G5GyubhMFTLeIqwTngx6JCktg== )
example.com. 3600 IN DNSKEY ( 257 3 13
JnAlXgyJTZz+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
InlwqF4U7TvC8PYOD0U/HYtwQ== )
com. 900 IN DNSKEY ( 256 3 13
7IIE5Dol8jSMUqHTvOOiZapdEbQ9wqRxFi/zQcSdufUKLhpByvLpzSAQTqCWj
3URIZ8L3Fa2gBLMOZUzZlGQCw== ) ; Key ID = 34327
com. 900 IN DNSKEY ( 257 3 13
RbkcO+96XZmnp8jYIuM4lryAp3egQjSmBaSoiA7H76Tm0RLHPNPUxlVn+nQ0f
Ic3I8xfZDNw8Wa0Pe3/g2QA/w== ) ; Key ID = 18931
com. 900 IN DNSKEY ( 257 3 13
szc7biLo5J4OHlkanlvZrF4aD4YYf+NHA/GAqdNslY9xxK9Izg68XHkqck4Rt
DiVn37lNAQmgSlHbrGu0yOTkA== ) ; Key ID = 28809
com. 900 IN RRSIG ( DNSKEY 13 1 900 20181128000000
20151103000000 18931 com.
lZmTBrfcRgVbqHJIfCVr6c3HUDgy3MlNSCSnrVV2S5/NmB3ZiFcvIDn0iqXPm
7YQfvfWi6utyxBu/fSD6SlARw== )
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/xJQFwWEK0cHbj061I65ELWSoWxPvYJ5o8QnSbRkzfCM4lTs
g94s5VvzMLYIbSZ1TWo2hcCdG== )
example.org. 3600 IN DNSKEY ( 256 3 13
NrbL6utGqIWlwrhhjeexdA6bMdD1lC1hj0Fnpevaa1AMyY2uy83TmoGnr996N
UR5TlG4Zh+YPbbmUIixe4nS3w== ) ; Key ID = 56566
example.org. 3600 IN DNSKEY ( 257 3 13
uspaqp17jsMTX6AWVgmbog/3Sttz+9ANFUWLn6qKUhr0BOqRuChQWj8jyYUUr
Wy9ttxxesNQ9MkO4LUrFght1LQ== ) ; Key ID = 44384
example.org. 3600 IN RRSIG ( DNSKEY 13 2 3600
20181128000000 20151103000000 44384 example.org.
ZsQ5wl2ZvofwDq7uYlvoqEeq9byHb159Ap4EPXdB4PpnWy2dJkIElgXCfILrU
EUCD1aKb2SoRZe18EJ8LMVJuw== )
example.org. 900 IN DS ( 44384 13 2 ec307e2efc8f0117ed96ab48a513c
8003eld9121f1ff11a08b4cdd348d090aa6 )
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
HfWx7kzlBBoJB0vLrvsJtXJG6== ) ; 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.
    G9I7dIh5Zn2hBu8jhgnLDTXZUpnPRkOMHj1lRcyHNbvJGLIiaPRVtcJXW0Vr+
    arygWmsHrDgWz0vw2IXZr3qKw== )
org. 900 IN RRSIG ( DNSKEY 13 1 900 20181128000000
    20151103000000 49352 org.
    iQmYWqUdU07Syw1Fqwx+8+hSk0w06tCGmkwdppyxUSFESumEhkOXgOv6NuIE
    eKjwMIaLj5HFB+9WnOkzgGE5Q== )
org. 86400 IN DS ( 12651 13 2 3979a51f98bbf219fc4f4a4176e766dfa8f
    9db5c24a75743ebl704b97a9fab )
org. 86400 IN DS ( 49352 13 2 03d11a1aa114abbb8f708c3c0ff0db765fe
    f4a2f18920db5f58710dd767c293b )
org. 86400 IN RRSIG ( DS 13 1 86400 20181128000000
    20151103000000 31918 .
    JGPMvEbfLoWNUELn/5cjjdRZx2CmdikbHuH6N/1BrxACWrGy05NuPvBPTEVOr
    mPFfm5SIMLLTWgxf0K0FsNH0Q== )
. 86400 IN DNSKEY ( 256 3 13
    zKz+DCWkNA/vuheivPcGqSH40U84KZAlrMRIyozj9WHzf8PsFp/or8j8vmjjW
    P98cbte4d8NvlGLxzbUzo3+FA== ) ; Key ID = 31918
. 86400 IN DNSKEY ( 256 3 13
    8wMZZ4lzHdyKZ4fv8kys/t3QmlgvEadbsbyqWrMhwddSXCZYGRrsAbPpireRW
    xbVcdlVtOrlFBcRDMTN0R0XEQ== ) ; Key ID = 2635
. 86400 IN DNSKEY ( 257 3 13
    yvX+VNTUjxZiGvtr060hVbrPV9H6rVusQtF9lIxCFzbZOJxMQBFmbqlc8Xclv
    Q+gDOXnFOTsgs/frMmxyG0tRg== ) ; 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+ygzrONcuzsnyfOGSszwaDdkSOJ0Kndcfbb2I1
        LUV04Z+V488+Sd1jr7/2ltsKA== )
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/frMmxyG0tRg== ) ; 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
        JnAlXgyJTz+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.
ltua9ntAqZvOnK5UztzIjN38Bqs6mJ8KAT7L4+AxevDL+z0Jft7RC1/g6Qrfa
InlwqF4U7TvC8PYOD0U/HYtwQ== )
com. 900 IN DNSKEY ( 256 3 13
7IIE5Dol8jSMUqHTvOOiZapdEbQ9wqRxFi/zQcSdufUKLhpByvLpzSAQTqCWj
3URIZ8L3Fa2gBLMOZUzZ1GQCw== ) ; Key ID = 34327
com. 900 IN DNSKEY ( 257 3 13
RbkcO+96XZmnp8jYIuM4lryAp3egQjSmBaSoiA7H76Tm0RLHPNPuXlVknQ0f
Ic3I8xfZDNw8Wa0Pe3/g2QA/w== ) ; Key ID = 18931
com. 900 IN DNSKEY ( 257 3 13
szc7biLo5J4OHlkanlvZrF4aD4YYf+NHA/GAqdNslY9xxK9Izg68XHkqck4Rt
DiV37lNAQmgSlHbrGu0yOTkA== ) ; Key ID = 28809
com. 900 IN RRSIG ( DNSKEY 13 1 900 20181128000000
20151103000000 18931 com.
lZmTBrfcRgVbqHJIfCVr6c3HUDgy3MlNSCSnrVV2S5/NmB3ZiFcvIDn0iqXPm
7YQfvfWi6utyxBu/fSD6S1ARw== )
com. 900 IN RRSIG ( DNSKEY 13 1 900 20181128000000
20151103000000 28809 com.
8qZOVm4X8wGt5XPWhG2HO4FAD6Kvs5eIhZUz+7DVCrZ/XMEVrMIHcm1Q+sq0s
hm4cSivK2BxOO24PHJXoZN2Lw== )
com. 86400 IN DS ( 18931 13 2 20f7a9db42d0e2042fbbb9f9ea015941202
f9eabb94487e658c188e7bcb52115 )
com. 86400 IN DS ( 28809 13 2 ad66b3276f796223aa45eda773e92c6d98e
70643bbde681db342a9e5cf2bb380 )
com. 86400 IN RRSIG ( DS 13 1 86400 20181128000000
20151103000000 31918 .
5KQVa0NP+6k7VEGMmeky2/Y3wIGM70Fkm0vp5NmQ6KPk8L1XMJPltcJDWGGjc
EU3Uc4z2DUxzZyWgEDdrSOcdw== )

```

Authors' Addresses

Melinda Shore
Fastly

EMail: mshore@fastly.com

Richard Barnes
Mozilla

EMail: rlb@ipv.sx

Shumon Huque
Salesforce

EMail: shuque@gmail.com

Willem Toorop
NLnet Labs

EMail: willem@nlnetlabs.nl

TLS
Internet-Draft
Updates: 6347 (if approved)
Intended status: Standards Track
Expires: 24 December 2021

E. Rescorla, Ed.
RTFM, Inc.
H. Tschofenig, Ed.
T. Fossati
Arm Limited
A. Kraus
Bosch.IO GmbH
22 June 2021

Connection Identifiers for DTLS 1.2
draft-ietf-tls-dtls-connection-id-13

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.

The new ciphertext record format with CID also provides content type encryption and record-layer padding.

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 24 December 2021.

Copyright Notice

Copyright (c) 2021 IETF Trust and the persons identified as the document authors. All rights reserved.

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

This document may contain material from IETF Documents or IETF Contributions published or made publicly available before November 10, 2008. The person(s) controlling the copyright in some of this material may not have granted the IETF Trust the right to allow modifications of such material outside the IETF Standards Process. Without obtaining an adequate license from the person(s) controlling the copyright in such materials, this document may not be modified outside the IETF Standards Process, and derivative works of it may not be created outside the IETF Standards Process, except to format it for publication as an RFC or to translate it into languages other than English.

Table of Contents

1. Introduction	3
2. Conventions and Terminology	3
3. The "connection_id" Extension	4
4. Record Layer Extensions	5
5. Record Payload Protection	7
5.1. Block Ciphers	8
5.2. Block Ciphers with Encrypt-then-MAC processing	8
5.3. AEAD Ciphers	9
6. Peer Address Update	9
7. Examples	10
8. Privacy Considerations	12
9. Security Considerations	12
10. IANA Considerations	13
10.1. Extra Column to TLS ExtensionType Values Registry	13
10.2. Entry to the TLS ExtensionType Values Registry	13
10.3. Entry to the TLS ContentType Registry	13
11. References	13
11.1. Normative References	14
11.2. Informative References	14
Appendix A. History	15

Appendix B. Working Group Information	17
Appendix C. Contributors	17
Appendix D. Acknowledgements	18
Authors' Addresses	18

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 Connection ID (CID) to the DTLS record layer. The presence of the CID is negotiated via a DTLS extension.

Adding a CID to the ciphertext record format presents an opportunity to make other changes to the record format. In keeping with the best practices established by TLS 1.3, the type of the record is encrypted, and a mechanism provided for adding padding to obfuscate the plaintext length.

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.

This document assumes familiarity with DTLS 1.2 [RFC6347]. The presentation language used in this document is described in Section 3 of [RFC8446].

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 using a CID but does not wish the server to use one when sending.

```
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 using the client's CID but does not wish the client to include a CID when sending.

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 deployment-specific 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. Since the CID length information is not included in the record itself, 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.

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, or a zero-length CID has been advertised for a given direction, then the RFC 6347-defined record format and content type MUST be used to send in the indicated direction(s).

If DTLS peers have negotiated the use of a non-zero-length CID for a given direction, then once encryption is enabled they MUST send with the record format defined in Figure 3 with the new MAC computation defined in Section 5 and the content type `tls12_cid`. Plaintext payloads never use the new record format or the CID content type.

When 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 a non-zero length CID is expected. If a non-zero-length CID is expected for the retrieved association, then the datagram MUST be treated as invalid, as described in Section 4.1.2.1 of [RFC6347].

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 real 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.

```

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 cleartext 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.)

The DTLSInnerPlaintext byte sequence is then encrypted. To create the DTLSCiphertext structure shown in Figure 3 the CID is added.

```

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 receive 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 serialized `DTLSInnerPlaintext` (two-byte integer). The length **MUST NOT** exceed 2^{14} .

`seq_num_placeholder` 8 bytes of 0xff

Note "+" denotes concatenation.

5.1. Block Ciphers

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

```
MAC(MAC_write_key,
    seq_num_placeholder +
    tls12_cid +
    cid_length +
    tls12_cid +
    DTLSCiphertext.version +
    epoch +
    sequence_number +
    cid +
    length_of_DTLSInnerPlaintext +
    DTLSInnerPlaintext.content +
    DTLSInnerPlaintext.real_type +
    DTLSInnerPlaintext.zeros
);
```

The rationale behind this construction is to separate the MAC input for DTLS without the connection ID from the MAC input with the connection ID. The former always consists of a sequence number followed by some other content type than `tls12_cid`; the latter always consists of the `seq_num_placeholder` followed by `tls12_cid`. Although $2^{64}-1$ is potentially a valid sequence number, `tls12_cid` will never be a valid content type when the connection ID is not in use. In addition, the `epoch` and `sequence_number` are now fed into the MAC in the same order as they appear on the wire.

5.2. Block Ciphers with Encrypt-then-MAC processing

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

```
MAC(MAC_write_key,
    seq_num_placeholder +
    tls12_cid +
    cid_length +
    tls12_cid +
    DTLSCiphertext.version +
    epoch +
    sequence_number +
    cid +
    DTLSCiphertext.length +
    IV +
    ENC(content + padding + padding_length));
```

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_placeholder +
                  tls12_cid +
                  cid_length +
                  tls12_cid +
                  DTLS_Ciphertext.version +
                  epoch +
                  sequence_number +
                  cid +
                  length_of_DTLSInnerPlaintext;
```

6. Peer Address Update

When a record with a CID is received that has a source address different from 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 strategy is mandated by this specification but see note (*) below.

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.

(*) Note: Application protocols that implement protection against spoofed addresses 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.ietf-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 unidirectionally 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
-----

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. 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 these 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 particular concern when the request is small and the response large. See Section 6 for more on address updates.

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

This document requests three actions from IANA.

10.1. Extra Column to TLS ExtensionType Values Registry

IANA is requested to add an extra column named "DTLS-Only" 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.

10.2. Entry to the TLS ExtensionType Values Registry

IANA is requested to allocate an entry to the existing "TLS ExtensionType Values" registry, for connection_id(TBD1) as described in the table below. Although the value 53 has been allocated by early allocation for a previous version of this document, it is incompatible with this document. Once this document is approved for publication, the early allocation will be deprecated in favor of this assignment.

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

A new column "DTLS-Only" is added to the registry. The valid entries are "Y" if the extension is only applicable to DTLS, "N" otherwise. All the pre-existing entries are given the value "N".

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].

10.3. Entry to the TLS ContentType Registry

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>>.
- [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>>.
- [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>>.

11.2. Informative References

- [I-D.ietf-tls-dtls-rrc]
Tschofenig, H. and T. Fossati, "Return Routability Check for DTLS 1.2 and DTLS 1.3", Work in Progress, Internet-Draft, draft-ietf-tls-dtls-rrc-00, 9 June 2021, <<https://www.ietf.org/archive/id/draft-ietf-tls-dtls-rrc-00.txt>>.
- [I-D.ietf-tls-dtls13]
Rescorla, E., Tschofenig, H., and N. Modadugu, "The Datagram Transport Layer Security (DTLS) Protocol Version 1.3", Work in Progress, Internet-Draft, draft-ietf-tls-dtls13-43, 30 April 2021, <<https://www.ietf.org/archive/id/draft-ietf-tls-dtls13-43.txt>>.
- [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>>.

Appendix A. History

RFC EDITOR: PLEASE REMOVE THE THIS SECTION

draft-ietf-tls-dtls-connection-id-12

- * Improved peer address update text
- * Editorial improvements
- * Clarification regarding the use of the TLS ExtensionType Values Registry

draft-ietf-tls-dtls-connection-id-11

- * Enhanced IANA considerations section
- * Clarifications regarding CID negotiation and zero-length CIDs

draft-ietf-tls-dtls-connection-id-10

- * Clarify privacy impact.
- * Have security considerations point to Section 6.

draft-ietf-tls-dtls-connection-id-09

- * Changed MAC/additional data calculation.
- * Disallow sending MAC failure fatal alerts to non-validated peers.
- * Incorporated editorial review comments by Ben Kaduk.

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

- * Restructured 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 ala 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 (<mailto:tls@ietf.org>). Information on the group and information on how to subscribe to the list is at <https://www1.ietf.org/mailman/listinfo/tls> (<https://www1.ietf.org/mailman/listinfo/tls>)

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

Appendix 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.

Appendix D. Acknowledgements

We would like to thank Hanno Becker, Martin Duke, Lars Eggert, Ben Kaduk, Warren Kumari, Francesca Palombini, Tom Petch, John Scudder, Sean Turner, Eric Vyncke, and Robert Wilton for their review comments.

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

Achim Kraus
Bosch.IO GmbH

Email: achim.kraus@bosch.io

TLS
Internet-Draft
Obsoletes: 6347 (if approved)
Intended status: Standards Track
Expires: 1 November 2021

E. Rescorla
RTFM, Inc.
H. Tschofenig
Arm Limited
N. Modadugu
Google, Inc.
30 April 2021

The Datagram Transport Layer Security (DTLS) Protocol Version 1.3
draft-ietf-tls-dtls13-43

Abstract

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

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

This document obsoletes RFC 6347.

Status of This Memo

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

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

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

This Internet-Draft will expire on 1 November 2021.

Copyright Notice

Copyright (c) 2021 IETF Trust and the persons identified as the document authors. All rights reserved.

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

This document may contain material from IETF Documents or IETF Contributions published or made publicly available before November 10, 2008. The person(s) controlling the copyright in some of this material may not have granted the IETF Trust the right to allow modifications of such material outside the IETF Standards Process. Without obtaining an adequate license from the person(s) controlling the copyright in such materials, this document may not be modified outside the IETF Standards Process, and derivative works of it may not be created outside the IETF Standards Process, except to format it for publication as an RFC or to translate it into languages other than English.

Table of Contents

1. Introduction	4
2. Conventions and Terminology	4
3. DTLS Design Rationale and Overview	6
3.1. Packet Loss	7
3.2. Reordering	8
3.3. Fragmentation	8
3.4. Replay Detection	8
4. The DTLS Record Layer	8
4.1. Demultiplexing DTLS Records	12
4.2. Sequence Number and Epoch	14
4.2.1. Processing Guidelines	14
4.2.2. Reconstructing the Sequence Number and Epoch	15
4.2.3. Record Number Encryption	15
4.3. Transport Layer Mapping	16
4.4. PMTU Issues	17
4.5. Record Payload Protection	19
4.5.1. Anti-Replay	19
4.5.2. Handling Invalid Records	20
4.5.3. AEAD Limits	20
5. The DTLS Handshake Protocol	22
5.1. Denial-of-Service Countermeasures	22
5.2. DTLS Handshake Message Format	25
5.3. ClientHello Message	27
5.4. ServerHello Message	28
5.5. Handshake Message Fragmentation and Reassembly	28

5.6.	End Of Early Data	29
5.7.	DTLS Handshake Flights	30
5.8.	Timeout and Retransmission	34
5.8.1.	State Machine	34
5.8.2.	Timer Values	37
5.8.3.	Large Flight Sizes	38
5.8.4.	State machine duplication for post-handshake messages	38
5.9.	CertificateVerify and Finished Messages	40
5.10.	Cryptographic Label Prefix	40
5.11.	Alert Messages	40
5.12.	Establishing New Associations with Existing Parameters	40
6.	Example of Handshake with Timeout and Retransmission	41
6.1.	Epoch Values and Rekeying	42
7.	ACK Message	45
7.1.	Sending ACKs	46
7.2.	Receiving ACKs	47
7.3.	Design Rationale	48
8.	Key Updates	48
9.	Connection ID Updates	50
9.1.	Connection ID Example	51
10.	Application Data Protocol	53
11.	Security Considerations	53
12.	Changes since DTLS 1.2	55
13.	Updates affecting DTLS 1.2	56
14.	IANA Considerations	56
15.	References	57
15.1.	Normative References	57
15.2.	Informative References	58
Appendix A.	Protocol Data Structures and Constant Values	61
A.1.	Record Layer	61
A.2.	Handshake Protocol	62
A.3.	ACKs	64
A.4.	Connection ID Management	64
Appendix B.	Analysis of Limits on CCM Usage	64
B.1.	Confidentiality Limits	65
B.2.	Integrity Limits	66
B.3.	Limits for AEAD_AES_128_CCM_8	66
Appendix C.	Implementation Pitfalls	67
Appendix D.	History	67
Appendix E.	Working Group Information	70
Appendix F.	Contributors	70
Appendix G.	Acknowledgements	71
Authors' Addresses	71

1. Introduction

RFC EDITOR: PLEASE REMOVE THE FOLLOWING PARAGRAPH

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

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

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

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

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

2. Conventions and Terminology

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

The following terms are used:

- * **client**: The endpoint initiating the DTLS connection.
- * **association**: Shared state between two endpoints established with a DTLS handshake.
- * **connection**: Synonym for association.
- * **endpoint**: Either the client or server of the connection.
- * **epoch**: one set of cryptographic keys used for encryption and decryption.
- * **handshake**: An initial negotiation between client and server that establishes the parameters of the connection.
- * **peer**: An endpoint. When discussing a particular endpoint, "peer" refers to the endpoint that is remote to the primary subject of discussion.
- * **receiver**: An endpoint that is receiving records.
- * **sender**: An endpoint that is transmitting records.
- * **server**: The endpoint which did not initiate the DTLS connection.
- * **CID**: Connection ID
- * **MSL**: Maximum Segment Lifetime

The reader is assumed to be familiar with [TLS13]. As in TLS 1.3, the HelloRetryRequest has the same format as a ServerHello message, but for convenience we use the term HelloRetryRequest throughout this document as if it were a distinct message.

DTLS 1.3 uses network byte order (big-endian) format for encoding messages based on the encoding format defined in [TLS13] and earlier (D)TLS specifications.

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

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

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

- * `'**'` indicates optional or situation-dependent messages/extensions that are not always sent.
- * `'{}'` indicates messages protected using keys derived from a `[sender]_handshake_traffic_secret`.
- * `'[]'` indicates messages protected using keys derived from `traffic_secret_N`.

3. DTLS Design Rationale and Overview

The basic design philosophy of DTLS is to construct "TLS over datagram transport". Datagram transport does not require nor provide reliable or in-order delivery of data. The DTLS protocol preserves this property for application data. Applications, such as media streaming, Internet telephony, and online gaming use datagram transport for communication due to the delay-sensitive nature of transported data. The behavior of such applications is unchanged when the DTLS protocol is used to secure communication, since the DTLS protocol does not compensate for lost or reordered data traffic. Note that while low-latency streaming and gaming use DTLS to protect data (e.g. for protection of a WebRTC data channel), telephony utilizes DTLS for key establishment, and Secure Real-time Transport Protocol (SRTP) for protection of data [RFC5763].

TLS cannot be used directly over datagram transports the following five reasons:

1. TLS relies on an implicit sequence number on records. If a record is not received, then the recipient will use the wrong sequence number when attempting to remove record protection from subsequent records. DTLS solves this problem by adding sequence numbers to records.
2. The TLS handshake is a lock-step cryptographic protocol. Messages must be transmitted and received in a defined order; any other order is an error. The DTLS handshake includes message sequence numbers to enable fragmented message reassembly and in-order delivery in case datagrams are lost or reordered.
3. During the handshake, messages are implicitly acknowledged by other handshake messages. Some handshake messages, such as the `NewSessionTicket` message, do not result in any direct response that would allow the sender to detect loss. DTLS adds an acknowledgment message to enable better loss recovery.

4. Handshake messages are potentially larger than can be contained in a single datagram. DTLS adds fields to handshake messages to support fragmentation and reassembly.
5. Datagram transport protocols, like UDP, are susceptible to abusive behavior effecting denial of service attacks against nonparticipants. DTLS adds a return-routability check and DTLS 1.3 uses the TLS 1.3 HelloRetryRequest message (see Section 5.1 for details).

3.1. Packet Loss

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

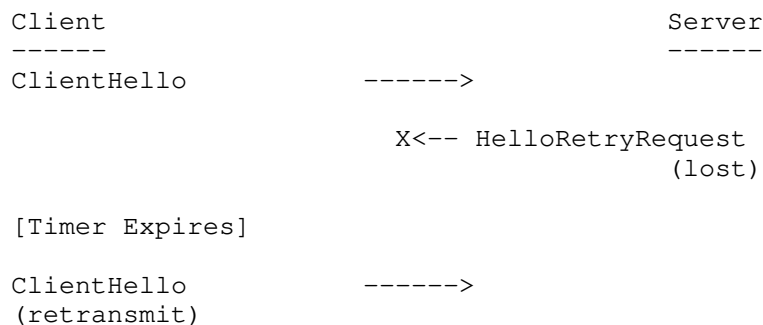


Figure 1: DTLS retransmission example

Once the client has transmitted the ClientHello message, it expects to see a HelloRetryRequest or a ServerHello from the server. However, if the timer expires, the client knows that either the ClientHello or the response from the server has been lost, which causes the the client to retransmit the ClientHello. When the server receives the retransmission, it knows to retransmit its HelloRetryRequest or ServerHello.

The server also maintains a retransmission timer for messages it sends (other than HelloRetryRequest) and retransmits when that timer expires. Not applying retransmissions to the HelloRetryRequest avoids the need to create state on the server. The HelloRetryRequest is designed to be small enough that it will not itself be fragmented, thus avoiding concerns about interleaving multiple HelloRetryRequests.

For more detail on timeouts and retransmission, see Section 5.8.

3.2. Reordering

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

3.3. Fragmentation

TLS and DTLS handshake messages can be quite large (in theory up to $2^{24}-1$ bytes, in practice many kilobytes). By contrast, UDP datagrams are often limited to less than 1500 bytes if IP fragmentation is not desired. In order to compensate for this limitation, each DTLS handshake message may be fragmented over several DTLS records, each of which is intended to fit in a single UDP datagram (see Section 4.4 for guidance). Each DTLS handshake message contains both a fragment offset and a fragment length. Thus, a recipient in possession of all bytes of a handshake message can reassemble the original unfragmented message.

3.4. Replay Detection

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

4. The DTLS Record Layer

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

1. The DTLSCiphertext structure omits the superfluous version number and type fields.
2. DTLS adds an epoch and sequence number to the TLS record header. This sequence number allows the recipient to correctly verify the DTLS MAC. However, the number of bits used for the epoch and sequence number fields in the DTLSCiphertext structure have been reduced from those in previous versions.
3. The DTLSCiphertext structure has a variable length header.

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

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

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

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

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

Figure 2: DTLS 1.3 Record Formats

legacy_record_version This value MUST be set to {254, 253} for all records other than the initial ClientHello (i.e., one not generated after a HelloRetryRequest), where it may also be {254, 255} for compatibility purposes. It MUST be ignored for all purposes. See [TLS13]; Appendix D.1 for the rationale for this.

unified_hdr: The unified header (unified_hdr) is a structure of variable length, as shown in Figure 3.

encrypted_record: The AEAD-encrypted form of the serialized DTLSInnerPlaintext structure.

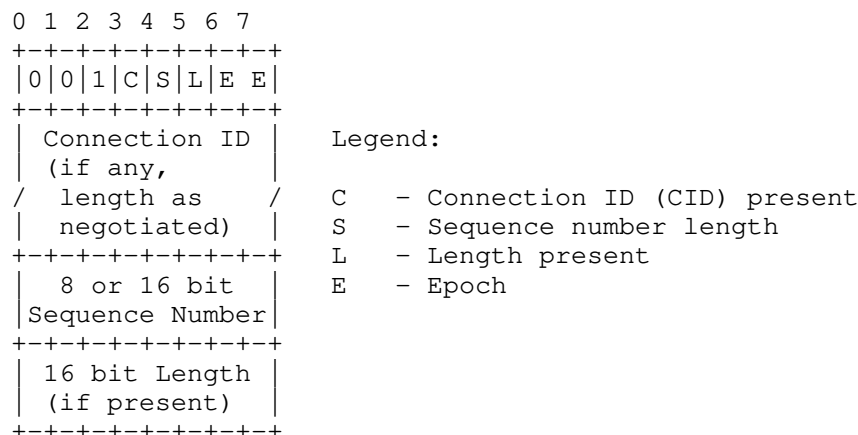


Figure 3: DTLS 1.3 Unified Header

Fixed Bits: The three high bits of the first byte of the unified header are set to 001. This ensures that the value will fit within the DTLS region when multiplexing is performed as described in [RFC7983]. It also ensures that distinguishing encrypted DTLS 1.3 records from encrypted DTLS 1.2 records is possible when they are carried on the same host/port quartet; such multiplexing is only possible when CIDs [I-D.ietf-tls-dtls-connection-id] are in use, in which case DTLS 1.2 records will have the content type `tls12_cid` (25).

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

S: The S bit (0x08) indicates the size of the sequence number. 0 means an 8-bit sequence number, 1 means 16-bit. Implementations MAY mix sequence numbers of different lengths on the same connection.

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

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

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

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

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

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

Figure 4 illustrates different record headers.

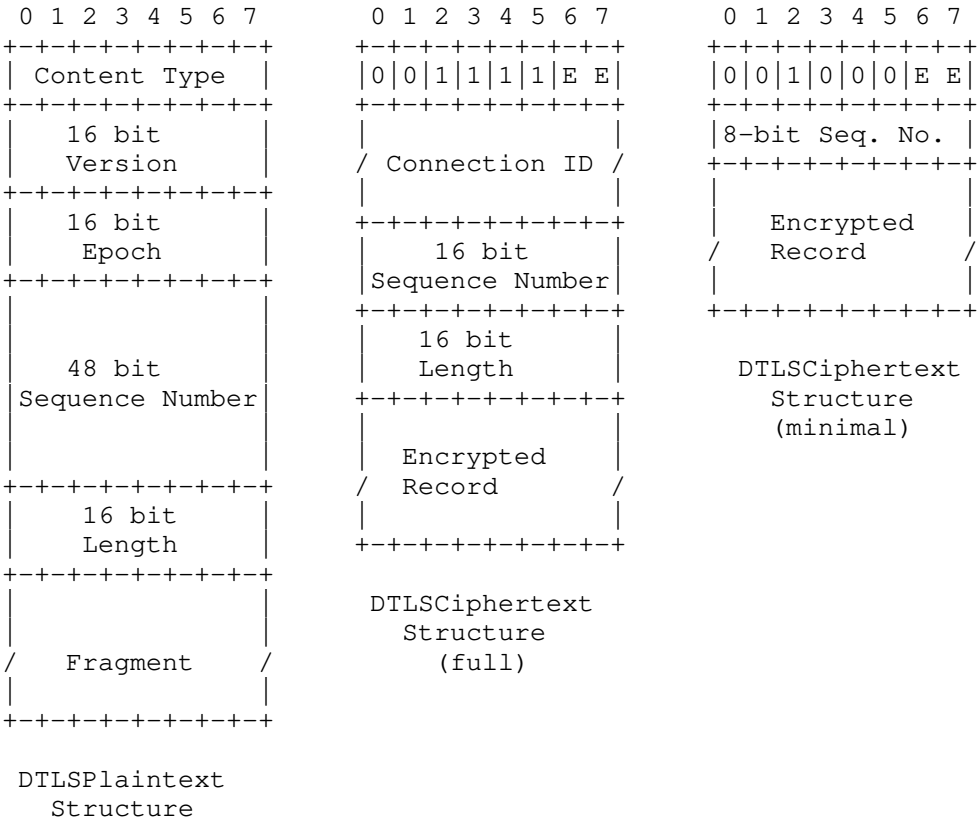


Figure 4: DTLS 1.3 Header Examples

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

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

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

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

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

The entire header value shown in Figure 4 (but prior to record number encryption, see Section 4.2.3) is used as the additional data value for the AEAD function. For instance, if the minimal variant is used, the AAD is 2 octets long. Note that this design is different from the additional data calculation for DTLS 1.2 and for DTLS 1.2 with Connection ID.

4.1. Demultiplexing DTLS Records

DTLS 1.3 uses a variable length record format and hence the demultiplexing process is more complex since more header formats need to be distinguished. Implementations can demultiplex DTLS 1.3 records by examining the first byte as follows:

- * If the first byte is alert(21), handshake(22), or ack(proposed, 26), the record MUST be interpreted as a DTLSPlaintext record.
- * If the first byte is any other value, then receivers MUST check to see if the leading bits of the first byte are 001. If so, the implementation MUST process the record as DTLSCiphertext; the true content type will be inside the protected portion.
- * Otherwise, the record MUST be rejected as if it had failed deprotection, as described in Section 4.5.2.

Figure 5 shows this demultiplexing procedure graphically taking DTLS 1.3 and earlier versions of DTLS into account.

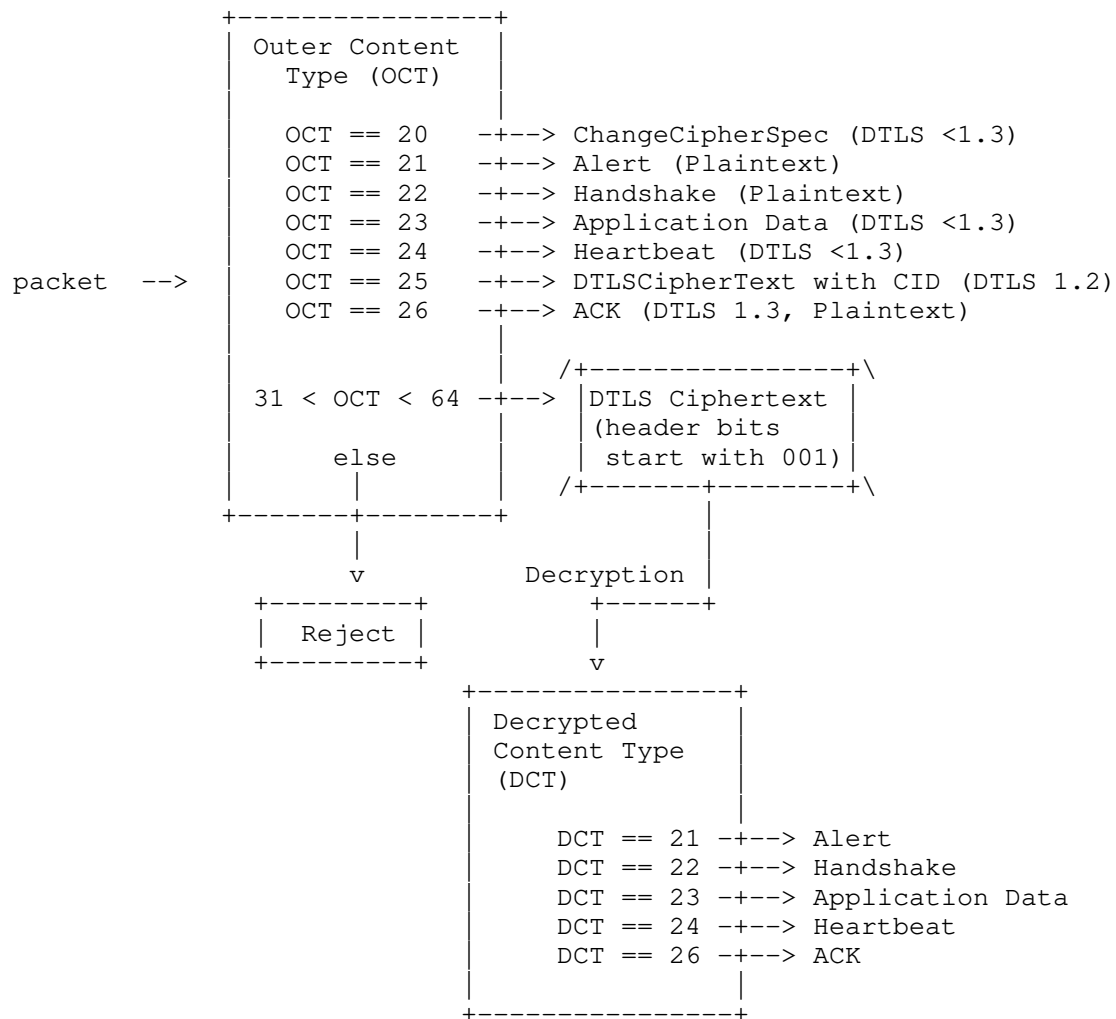


Figure 5: Demultiplexing DTLS 1.2 and DTLS 1.3 Records

Note: The optimized DTLS header format shown in Figure 3, which does not carry the Content Type in the Unified Header format, requires a different demultiplexing strategy compared to what was used in previous DTLS versions where the Content Type was conveyed in every record. As described in Figure 5, the first byte determines how an incoming DTLS record is demultiplexed. The first 3 bits of the first byte distinguish a DTLS 1.3 encrypted record from record types used in previous DTLS versions and plaintext DTLS 1.3 record types. Hence, the range 32 (0b0010 0000) to 63 (0b0011 1111) needs to be excluded from future allocations by IANA to avoid problems while demultiplexing; see Section 14.

4.2. Sequence Number and Epoch

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

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

4.2.1. Processing Guidelines

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

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

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

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

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

4.2.2. Reconstructing the Sequence Number and Epoch

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

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

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

4.2.3. Record Number Encryption

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

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

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

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

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

The `sn_key` is computed as follows:

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

[sender] denotes the sending side. The Secret value to be used is described in Section 7.3 of [TLS13]. Note that a new key is used for each epoch: because the epoch is sent in the clear, this does not result in ambiguity.

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

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

Future cipher suites, which are not based on AES or ChaCha20, MUST define their own record sequence number encryption in order to be used with DTLS.

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

4.3. Transport Layer Mapping

DTLS messages MAY be fragmented into multiple DTLS records. Each DTLS record MUST fit within a single datagram. In order to avoid IP fragmentation, clients of the DTLS record layer SHOULD attempt to size records so that they fit within any Path MTU (PMTU) estimates obtained from the record layer. For more information about PMTU issues see Section 4.4.

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

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

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

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

4.4. PMTU Issues

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

- * The DTLS record framing expands the datagram size, thus lowering the effective PMTU from the application's perspective.
- * In some implementations, the application may not directly talk to the network, in which case the DTLS stack may absorb ICMP [RFC1191] "Datagram Too Big" indications or ICMPv6 [RFC4443] "Packet Too Big" indications.
- * The DTLS handshake messages can exceed the PMTU.

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

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

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

- * For DTLS over DCCP, the upper layer protocol SHOULD be allowed to obtain the current estimate of the PMTU.
- * For DTLS over TCP or SCTP, which automatically fragment and reassemble datagrams, there is no PMTU limitation. However, the upper layer protocol MUST NOT write any record that exceeds the maximum record size of 2^{14} bytes.

The DTLS record layer SHOULD also allow the upper layer protocol to discover the amount of record expansion expected by the DTLS processing; alternately it MAY report PMTU estimates minus the estimated expansion from the transport layer and DTLS record framing.

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

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

The DTLS record layer SHOULD NOT interfere with upper layer protocols performing PMTU discovery, whether via [RFC1191] and [RFC4821] for IPv4 or via [RFC8201] for IPv6. In particular:

- * Where allowed by the underlying transport protocol, the upper layer protocol SHOULD be allowed to set the state of the DF bit (in IPv4) or prohibit local fragmentation (in IPv6).
- * If the underlying transport protocol allows the application to request PMTU probing (e.g., DCCP), the DTLS record layer SHOULD honor this request.

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

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

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

4.5. Record Payload Protection

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

4.5.1. Anti-Replay

Each DTLS record contains a sequence number to provide replay protection. Sequence number verification SHOULD be performed using the following sliding window procedure, borrowed from Section 3.4.3 of [RFC4303]. Because each epoch resets the sequence number space, a separate sliding window is needed for each epoch.

The received record counter for an epoch MUST be initialized to zero when that epoch is first used. For each received record, the receiver MUST verify that the record contains a sequence number that does not duplicate the sequence number of any other record received in that epoch during the lifetime of the association. This check SHOULD happen after deprotecting the record; otherwise the record discard might itself serve as a timing channel for the record number. Note that computing the full record number from the partial is still a potential timing channel for the record number, though a less powerful one than whether the record was deprotected.

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

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

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

4.5.2. Handling Invalid Records

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

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

Note that because invalid records are rejected at a layer lower than the handshake state machine, they do not affect pending retransmission timers.

4.5.3. AEAD Limits

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

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

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

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

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

The AEAD_AES_128_CCM_8 AEAD, as used in TLS_AES_128_CCM_8_SHA256, does not have a limit on the number of records that fail authentication that both limits the probability of forgery by the same amount and does not expose implementations to the risk of denial of service; see Appendix B.3. Therefore, TLS_AES_128_CCM_8_SHA256 MUST NOT be used in DTLS without additional safeguards against forgery. Implementations MUST set usage limits for AEAD_AES_128_CCM_8 based on an understanding of any additional forgery protections that are used.

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

5. The DTLS Handshake Protocol

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

1. To handle message loss, reordering, and fragmentation modifications to the handshake header are necessary.
2. Retransmission timers are introduced to handle message loss.
3. A new ACK content type has been added for reliable message delivery of handshake messages.

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

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

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

5.1. Denial-of-Service Countermeasures

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

1. An attacker can consume excessive resources on the server by transmitting a series of handshake initiation requests, causing the server to allocate state and potentially to perform expensive cryptographic operations.
2. An attacker can use the server as an amplifier by sending connection initiation messages with a forged source address that belongs to a victim. The server then sends its response to the victim machine, thus flooding it. Depending on the selected parameters this response message can be quite large, as is the case for a Certificate message.

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

MAY respond with a HelloRetryRequest message. The HelloRetryRequest message, as well as the cookie extension, is defined in TLS 1.3. The HelloRetryRequest message contains a stateless cookie (see [TLS13]; Section 4.2.2). The client MUST send a new ClientHello with the cookie added as an extension. The server then verifies the cookie and proceeds with the handshake only if it is valid. This mechanism forces the attacker/client to be able to receive the cookie, which makes DoS attacks with spoofed IP addresses difficult. This mechanism does not provide any defense against DoS attacks mounted from valid IP addresses.

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

The exchange is shown in Figure 6. Note that the figure focuses on the cookie exchange; all other extensions are omitted.

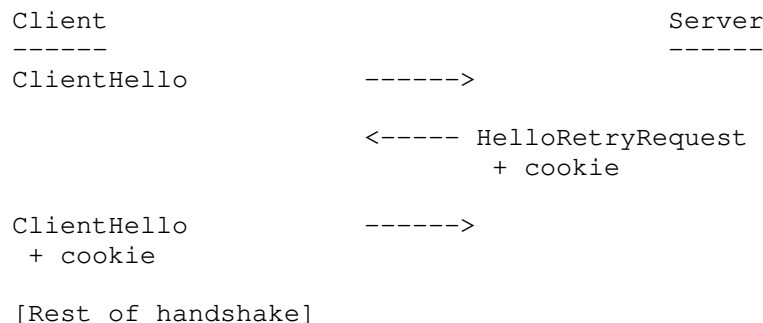


Figure 6: DTLS exchange with HelloRetryRequest containing the "cookie" extension

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

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

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

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

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

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

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

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

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

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

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

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

5.2. DTLS Handshake Message Format

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

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

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

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

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

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

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

5.3. ClientHello Message

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

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

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

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

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

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

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

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

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

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

legacy_compression_methods: Same as for TLS 1.3.

extensions: Same as for TLS 1.3.

5.4. ServerHello Message

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

5.5. Handshake Message Fragmentation and Reassembly

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

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

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

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

5.6. End Of Early Data

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

5.7. DTLS Handshake Flights

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

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

Table 1: Flight Handshake Message Combinations.

Remarks:

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

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

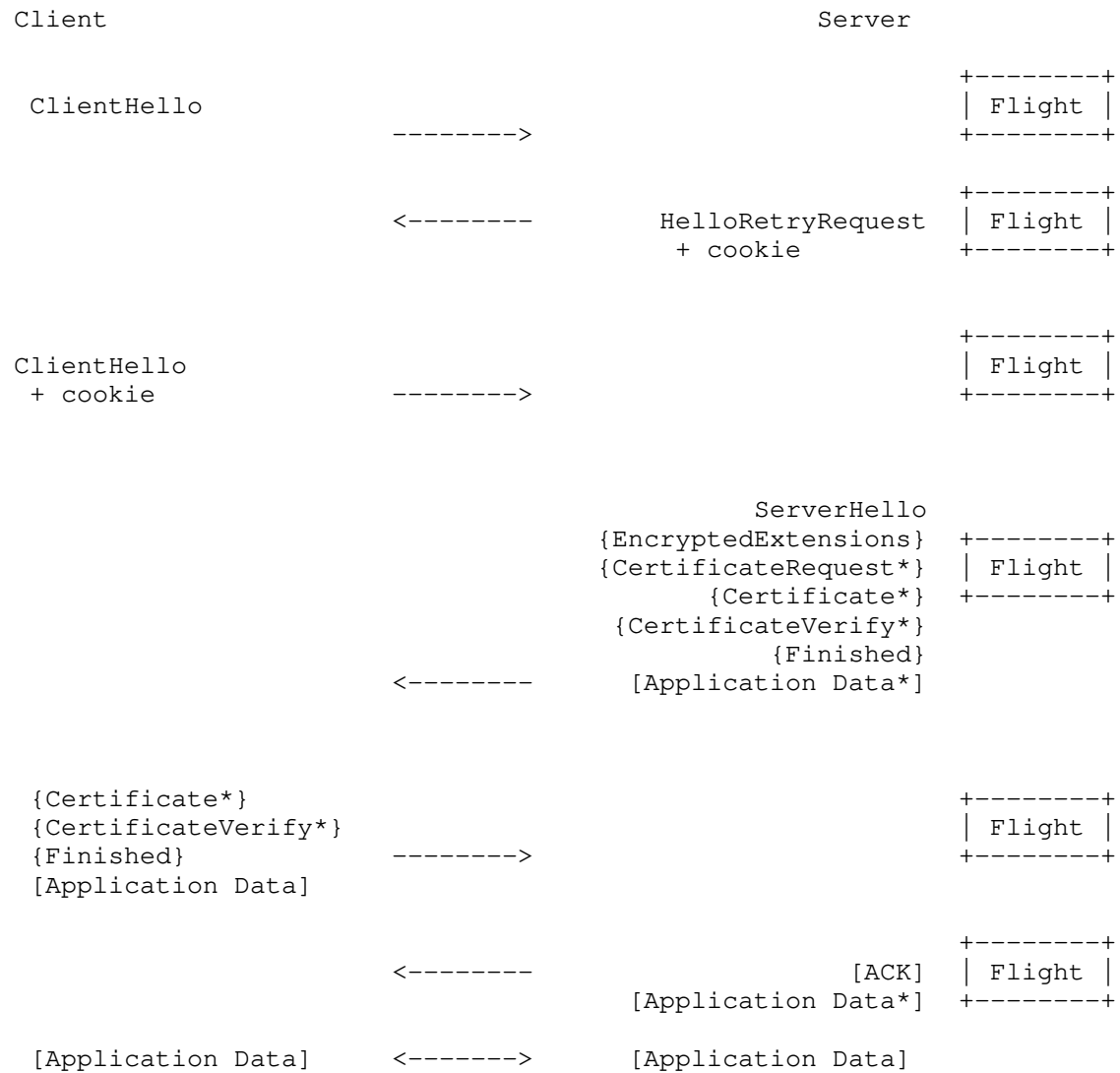


Figure 7: Message flights for a full DTLS Handshake (with cookie exchange)

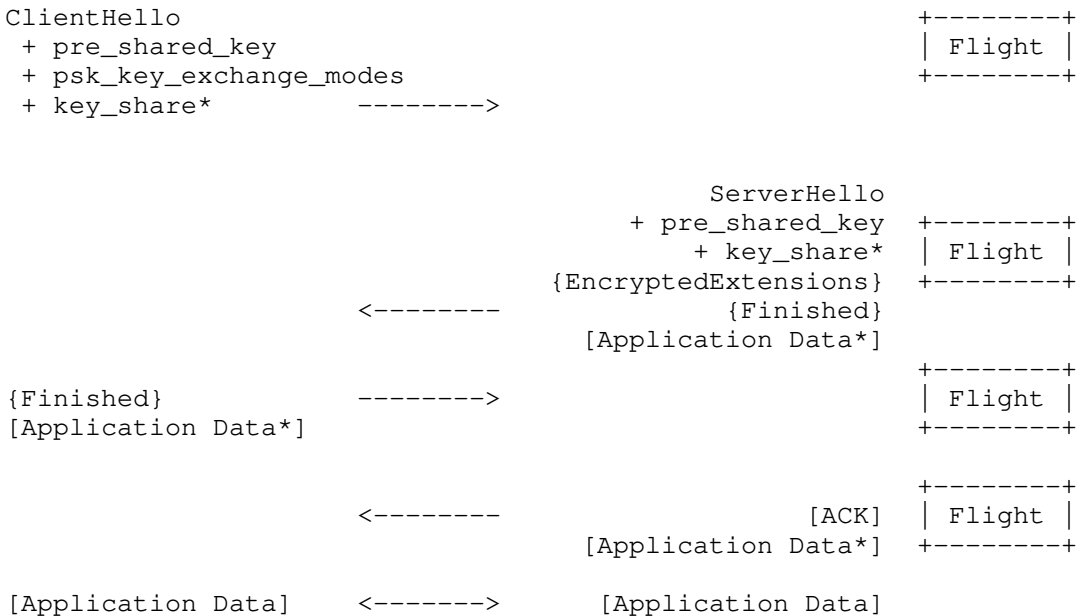


Figure 8: Message flights for resumption and PSK handshake (without cookie exchange)

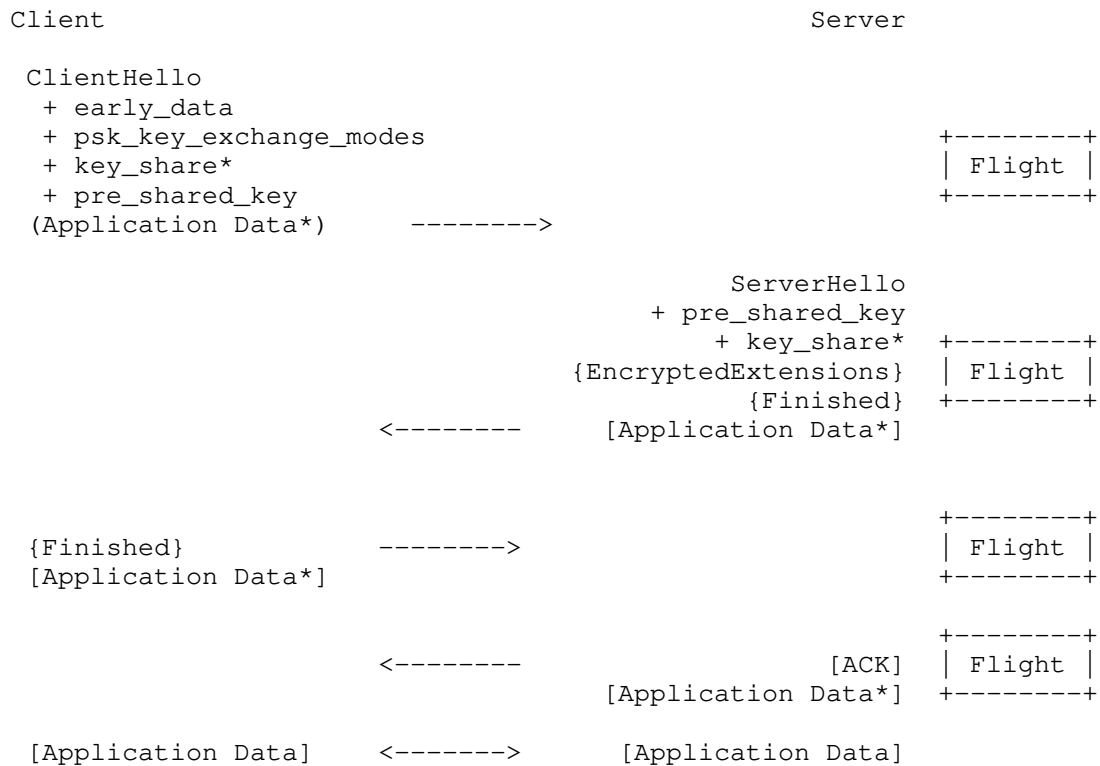
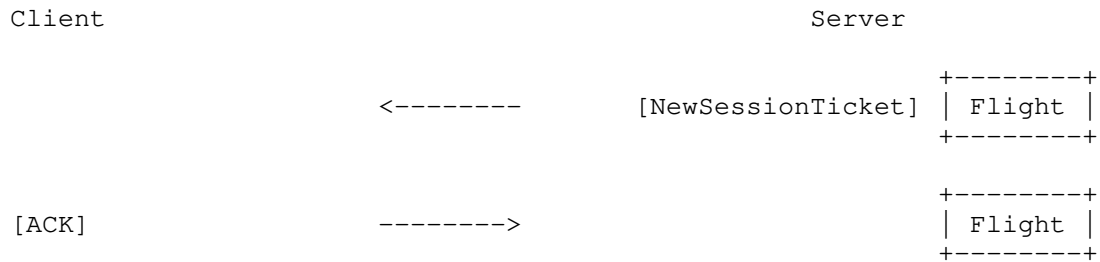


Figure 9: Message flights for the Zero-RTT handshake

Figure 10: Message flights for the `NewSessionTicket` message

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

5.8. Timeout and Retransmission

5.8.1. State Machine

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

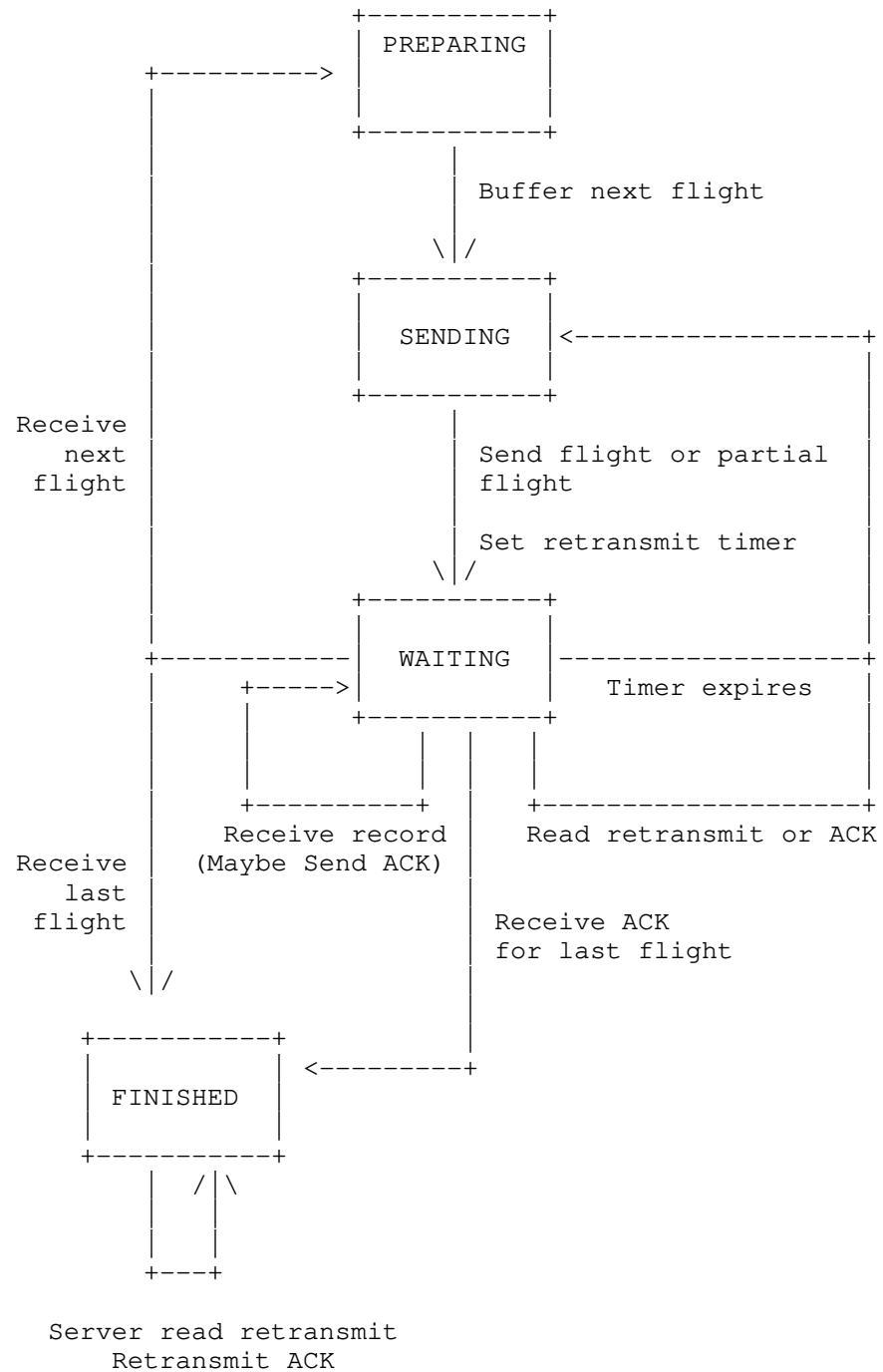


Figure 11: DTLS timeout and retransmission state machine

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

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

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

There are four ways to exit the WAITING state:

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

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

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

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

5.8.2. Timer Values

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

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

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

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

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

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

5.8.3. Large Flight Sizes

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

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

5.8.4. State machine duplication for post-handshake messages

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

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

5. Post-handshake client authentication

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

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

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

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

5.9. CertificateVerify and Finished Messages

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

5.10. Cryptographic Label Prefix

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

5.11. Alert Messages

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

5.12. Establishing New Associations with Existing Parameters

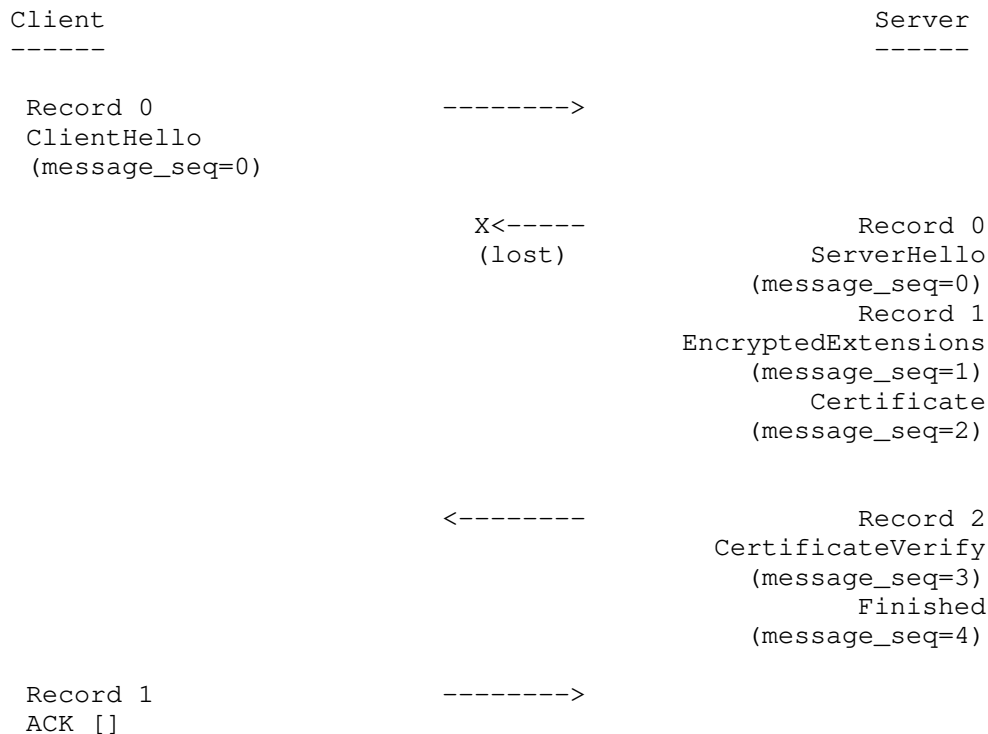
If a DTLS client-server pair is configured in such a way that repeated connections happen on the same host/port quartet, then it is possible that a client will silently abandon one connection and then initiate another with the same parameters (e.g., after a reboot). This will appear to the server as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

blind attackers from destroying associations merely by sending forged ClientHellos.

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

6. Example of Handshake with Timeout and Retransmission

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



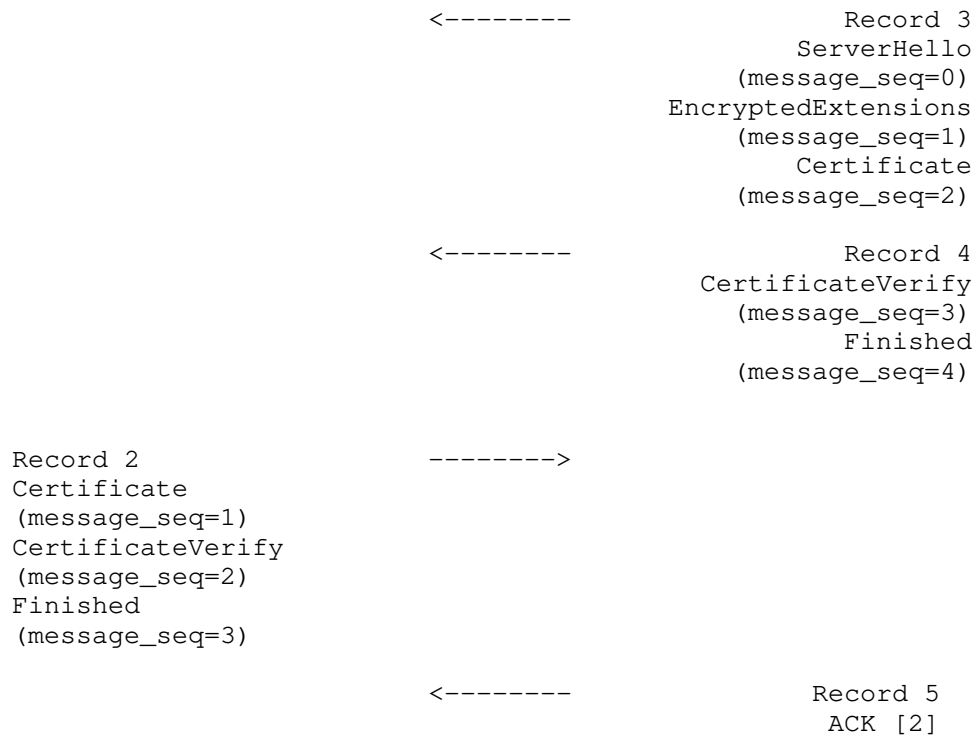


Figure 12: Example DTLS exchange illustrating message loss

6.1. Epoch Values and Rekeying

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

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

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

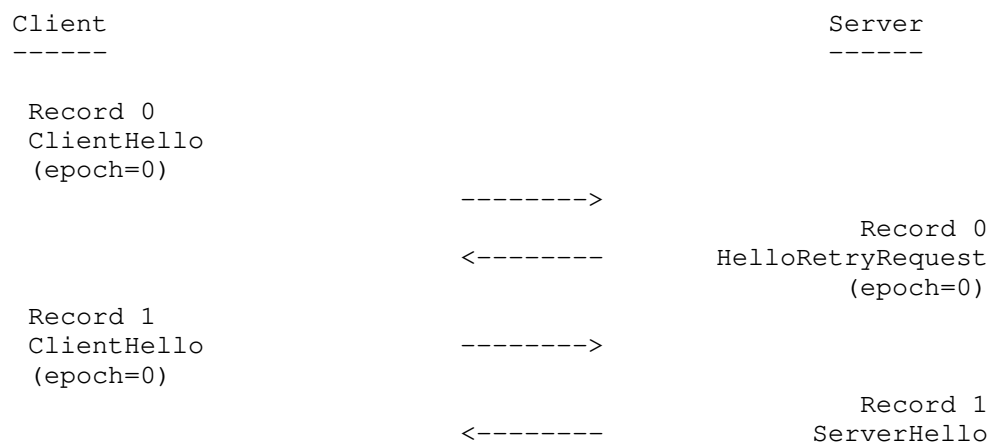
- * epoch value (1) is used for messages protected using keys derived from `client_early_traffic_secret`. Note this epoch is skipped if the client does not offer early data.
- * epoch value (2) is used for messages protected using keys derived from `[sender]_handshake_traffic_secret`. Messages transmitted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

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

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

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

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



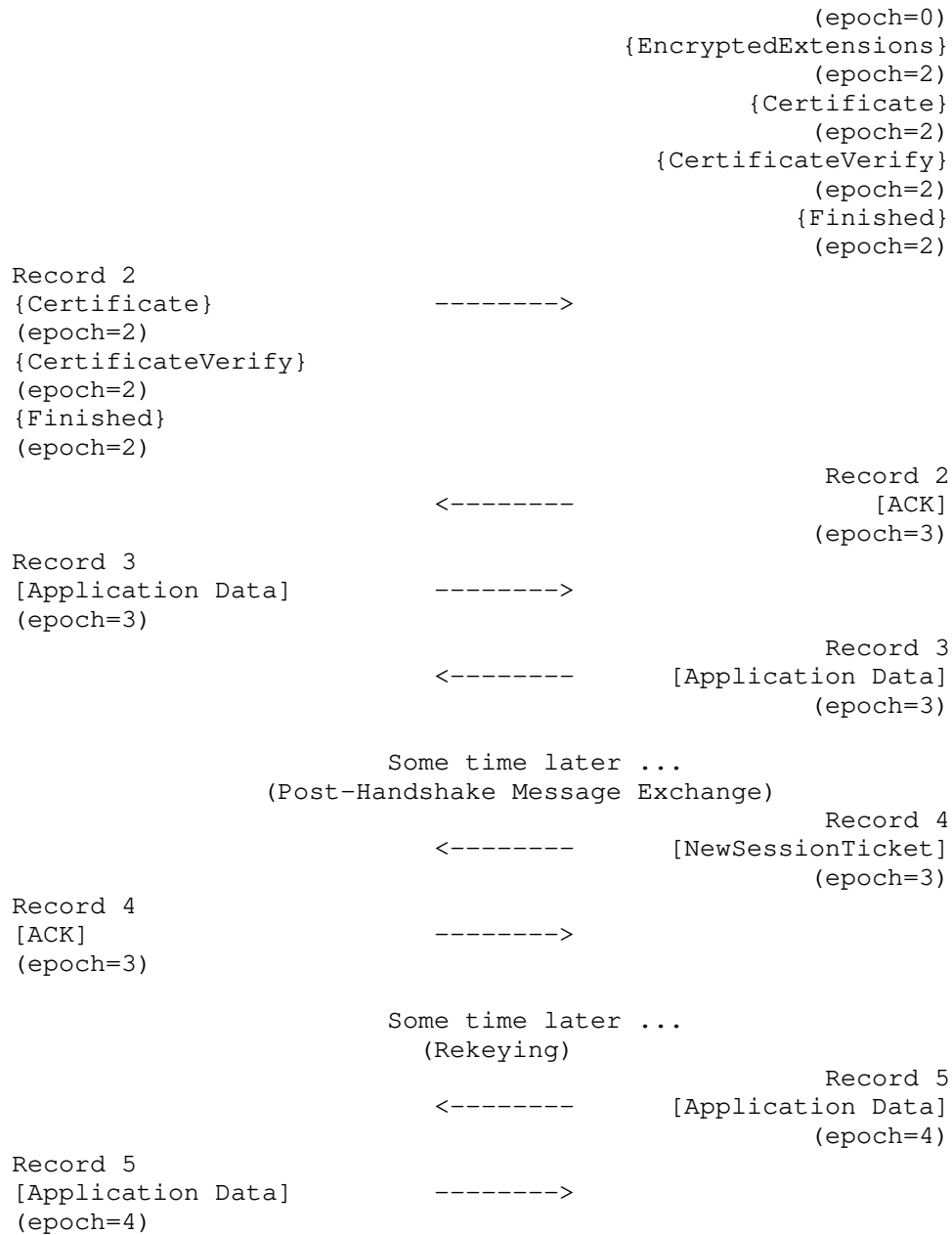


Figure 13: Example DTLS exchange with epoch information

7. ACK Message

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

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

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

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

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

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

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

7.1. Sending ACKs

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

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

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

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

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

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

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

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

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

7.2. Receiving ACKs

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

7.3. Design Rationale

ACK messages are used in two circumstances, namely :

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

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

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

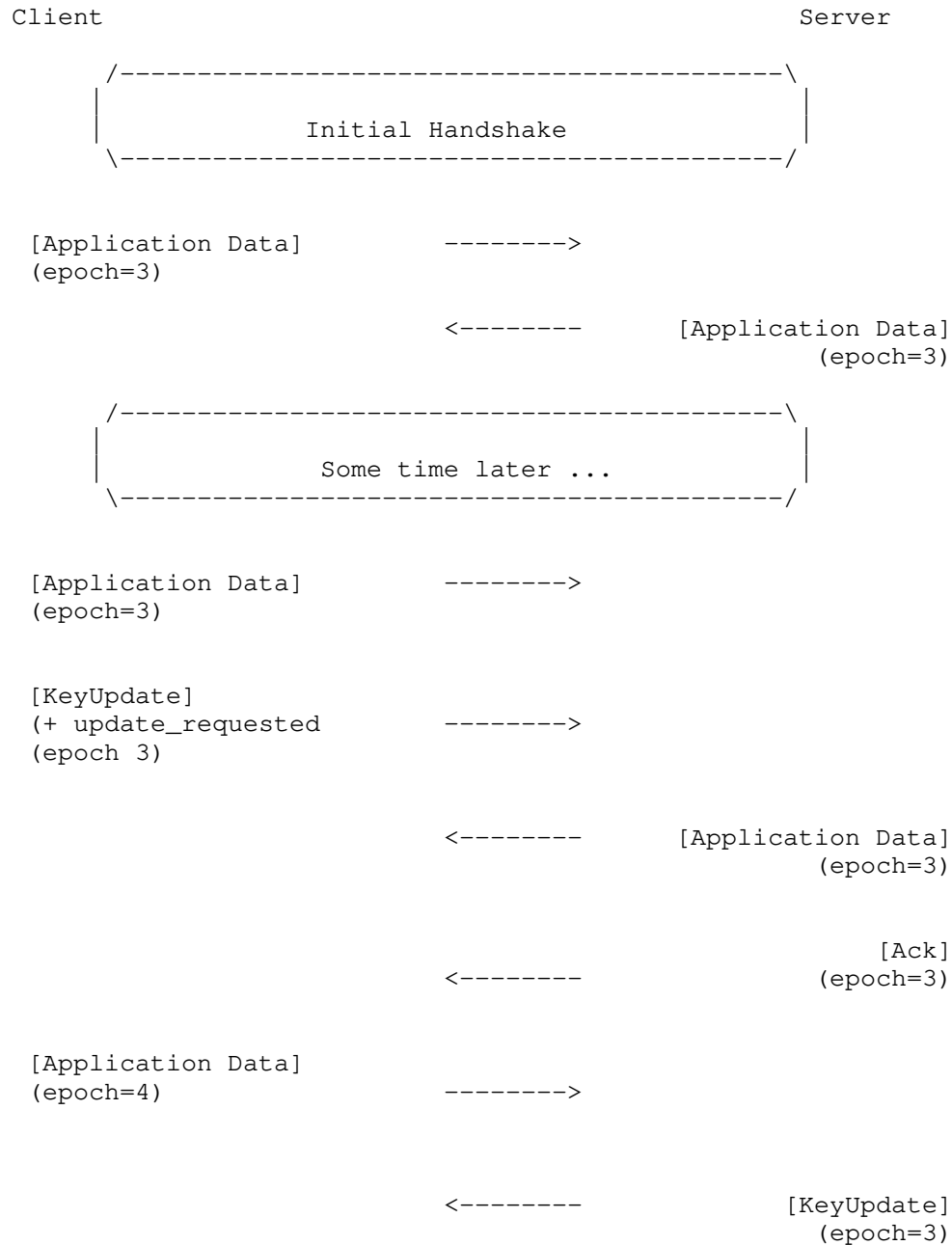
8. Key Updates

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

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

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

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



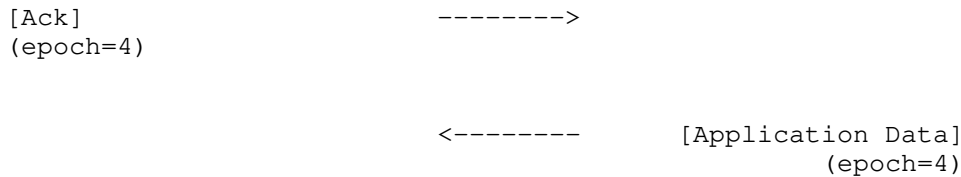


Figure 14: Example DTLS Key Update

9. Connection ID Updates

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

```

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

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

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

```

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

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

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

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

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

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

num_cids The number of CIDs desired.

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

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

9.1. Connection ID Example

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

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

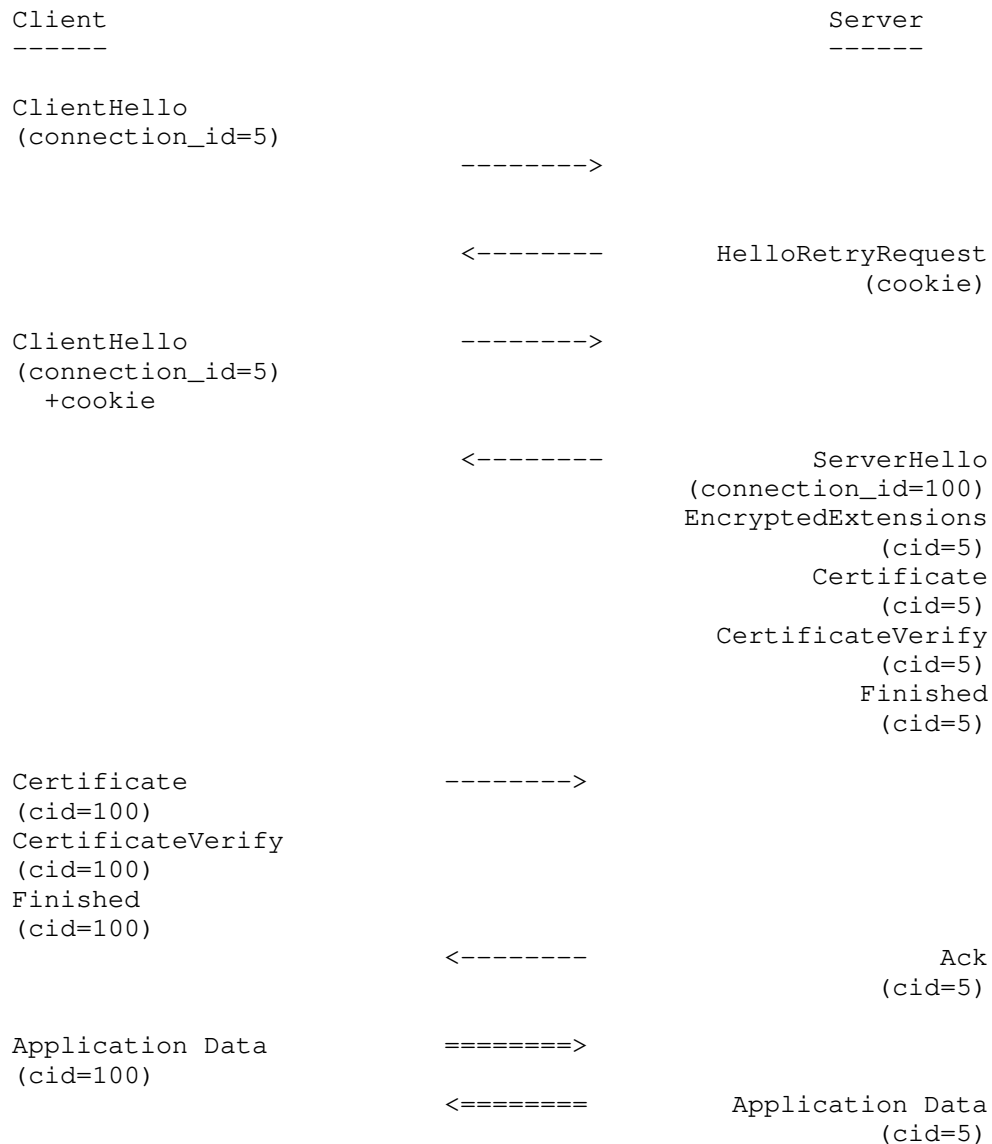


Figure 15: Example DTLS 1.3 Exchange with CIDs

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

10. Application Data Protocol

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

11. Security Considerations

Security issues are discussed primarily in [TLS13].

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

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

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

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

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

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

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

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

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

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

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

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

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

12. Changes since DTLS 1.2

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

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

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

13. Updates affecting DTLS 1.2

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

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

14. IANA Considerations

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

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

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

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

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

15. References

15.1. Normative References

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

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

15.2. Informative References

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

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

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

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

Appendix A. Protocol Data Structures and Constant Values

This section provides the normative protocol types and constants definitions.

A.1. Record Layer

```

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

```

```

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

```

```

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

```

```

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

```

Legend:

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

```

struct {
    uint16 epoch;
    uint48 sequence_number;
} RecordNumber;

```

A.2. Handshake Protocol

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

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

uint16 ProtocolVersion;
opaque Random[32];

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

```

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

```

A.3. ACKs

```

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

```

A.4. Connection ID Management

```

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

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

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

struct {
    uint8 num_cids;
} RequestConnectionId;

```

Appendix B. Analysis of Limits on CCM Usage

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

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

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

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

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

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

B.1. Confidentiality Limits

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

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

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

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

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

B.2. Integrity Limits

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

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

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

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

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

B.3. Limits for AEAD_AES_128_CCM_8

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

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

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

$$v \leq 2^7$$

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

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

Appendix C. Implementation Pitfalls

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

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

Appendix D. History

RFC EDITOR: PLEASE REMOVE THE THIS SECTION

(*) indicates a change that may affect interoperability.

IETF Drafts draft-42

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

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

draft-40

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

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

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

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

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

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

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

draft-32: - Editorial improvements and clarifications.

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

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

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

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

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

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

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

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

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

draft-00 - First IETF Draft

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

draft-00

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

Appendix E. Working Group Information

RFC EDITOR: PLEASE REMOVE THIS SECTION.

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

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

Appendix F. Contributors

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

In addition, we would like to thank:

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

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

Appendix G. Acknowledgements

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

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

Authors' Addresses

Eric Rescorla
RTFM, Inc.

Email: ekr@rtfm.com

Hannes Tschofenig
Arm Limited

Email: hannes.tschofenig@arm.com

Nagendra Modadugu
Google, Inc.

Email: nagendra@cs.stanford.edu

TLS
Internet-Draft
Intended status: Standards Track
Expires: 5 September 2022

N. Sullivan
Cloudflare Inc.
4 March 2022

Exported Authenticators in TLS
draft-ietf-tls-exported-authenticator-15

Abstract

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

Status of This Memo

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 5 September 2022.

Copyright Notice

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

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

Table of Contents

1. Introduction	2
2. Conventions and Terminology	3
3. Message Sequences	4
4. Authenticator Request	4
5. Authenticator	6
5.1. Authenticator Keys	6
5.2. Authenticator Construction	7
5.2.1. Certificate	8
5.2.2. CertificateVerify	8
5.2.3. Finished	10
5.2.4. Authenticator Creation	10
6. Empty Authenticator	10
7. API considerations	11
7.1. The "request" API	11
7.2. The "get context" API	11
7.3. The "authenticate" API	11
7.4. The "validate" API	12
8. IANA Considerations	13
8.1. Update of the TLS ExtensionType Registry	13
8.2. Update of the TLS Exporter Labels Registry	13
8.3. Update of the TLS HandshakeType Registry	13
9. Security Considerations	13
10. Acknowledgements	14
11. References	14
11.1. Normative References	14
11.2. Informative References	15
Author's Address	16

1. Introduction

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

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

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

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

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

Post-handshake authentication is defined in section 4.6.3 of TLS 1.3 [RFC8446], but it has the disadvantage of requiring additional state to be stored as part of the TLS state machine. Furthermore, the authentication boundaries of TLS 1.3 post-handshake authentication align with TLS record boundaries, which are often not aligned with the authentication boundaries of the higher-layer protocol. For example, multiplexed connection protocols like HTTP/2 [RFC7540] do not have a notion of which TLS record a given message is a part of.

Exported Authenticators are meant to be used as a building block for application protocols. Mechanisms such as those required to advertise support and handle authentication errors are not handled by TLS (or DTLS).

The minimum version of TLS and DTLS required to implement the mechanisms described in this document are TLS 1.2 [RFC6347] and DTLS 1.2 [RFC5246].

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.

This document uses terminology such as client, server, connection, handshake, endpoint, peer that are defined in section 1.1 of [RFC8446]. The term "initial connection" refers to the (D)TLS connection from which the exported authenticator messages are derived.

3. Message Sequences

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

Client Authentication

- * Server generates Authenticator Request
- * Client generates Authenticator from Server's Authenticator Request
- * Server validates Client's Authenticator

Server Authentication

- * Client generates Authenticator Request
- * Server generates Authenticator from Client's Authenticator Request
- * Client validates Server's Authenticator

Spontaneous Server Authentication

- * Server generates Authenticator
- * Client validates Server's Authenticator

4. Authenticator Request

The authenticator request is a structured message that can be created by either party of a (D)TLS connection using data exported from that connection. It can be transmitted to the other party of the (D)TLS connection at the application layer. The application layer protocol used to send the authenticator request SHOULD use a secure transport channel with equivalent security to TLS, such as QUIC [RFC9001], as its underlying transport to keep the request confidential. The application MAY use the existing (D)TLS connection to transport the authenticator.

An authenticator request message can be constructed by either the client or the server. Server-generated authenticator requests use the `CertificateRequest` message from Section 4.3.2 of [RFC8446]. Client-generated authenticator requests use a new message, called the `ClientCertificateRequest`, which uses the same structure as `CertificateRequest`. (Note that the latter is not a request for a client certificate, but rather a certificate request generated by the client.) These message structures are used even if the connection protocol is TLS 1.2 or DTLS 1.2.

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

The structures are defined to be:

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

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

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

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

The uniqueness requirements of the `certificate_request_context` apply only to `CertificateRequest` and `ClientCertificateRequest` messages that are used as part of authenticator requests, but do apply across `CertificateRequest` and `ClientCertificateRequest` messages. A `certificate_request_context` value used in a `ClientCertificateRequest` cannot be used in an authenticator `CertificateRequest` on the same connection, and vice versa. There is no impact if the value of a `certificate_request_context` used in an authenticator request matches the value of a `certificate_request_context` in the handshake or in a post-handshake message.

5. Authenticator

The authenticator is a structured message that can be exported from either party of a (D)TLS connection. It can be transmitted to the other party of the (D)TLS connection at the application layer. The application layer protocol used to send the authenticator SHOULD use a secure transport channel with equivalent security to TLS, such as QUIC [RFC9001], as its underlying transport to keep the authenticator confidential. The application MAY use the existing (D)TLS connection to transport the authenticator.

An authenticator message can be constructed by either the client or the server given an established (D)TLS connection, an identity, such as an X.509 certificate, and a corresponding private key. Clients MUST NOT send an authenticator without a preceding authenticator request; for servers an authenticator request is optional. For authenticators that do not correspond to authenticator requests, the `certificate_request_context` is chosen by the server.

5.1. Authenticator Keys

Each authenticator is computed using a Handshake Context and Finished MAC Key derived from the (D)TLS connection. These values are derived using an exporter as described in Section 4 of [RFC5705] (for (D)TLS 1.2) or Section 7.5 of [RFC8446] (for (D)TLS 1.3). For (D)TLS 1.3, the `exporter_master_secret` MUST be used, not the `early_exporter_master_secret`. These values use different labels depending on the role of the sender:

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

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

The context_value used for the exporter is empty (zero length) for all four values. There is no need to include additional context information at this stage since the application-supplied context is included in the authenticator itself. The length of the exported value is equal to the length of the output of the hash function associated with the selected cipher suite (for TLS 1.3) or the hash function used for the pseudorandom function (PRF) (for (D)TLS 1.2). Exported authenticators cannot be used with (D)TLS 1.2 cipher suites that do not use the TLS PRF and with TLS 1.3 cipher suites that do not have an associated hash function. This hash is referred to as the authenticator hash.

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

5.2. Authenticator Construction

An authenticator is formed from the concatenation of TLS 1.3 [RFC8446] Certificate, CertificateVerify, and Finished messages. These messages are encoded as TLS handshake messages, including length and type fields. They do not include any TLS record layer framing and are not encrypted with a handshake or application-data key.

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

ClientHello extensions are used to determine permissible extensions in the server's unsolicited Certificate message in order to follow the general model for extensions in (D)TLS in which extensions can only be included as part of a Certificate message if they were previously sent as part of a CertificateRequest message or ClientHello message. This ensures that the recipient will be able to process such extensions.

5.2.1. Certificate

The Certificate message contains the identity to be used for authentication, such as the end-entity certificate and any supporting certificates in the chain. This structure is defined in [RFC8446], Section 4.4.2.

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

Certificates chosen in the Certificate message MUST conform to the requirements of a Certificate message in the negotiated version of (D)TLS. In particular, the entries of `certificate_list` MUST be valid for the signature algorithms indicated by the peer in the `"signature_algorithms"` and `"signature_algorithms_cert"` extension, as described in Section 4.2.3 of [RFC8446] for (D)TLS 1.3 or from Sections 7.4.2 and 7.4.6 of [RFC5246] for (D)TLS 1.2.

In addition to `"signature_algorithms"` and `"signature_algorithms_cert"`, the `"server_name"` [RFC6066], `"certificate_authorities"` (Section 4.2.4. of [RFC8446]), and `"oid_filters"` (Section 4.2.5. of [RFC8446]) extensions are used to guide certificate selection.

Only the X.509 certificate type defined in [RFC8446] is supported. Alternative certificate formats such as [RFC7250] Raw Public Keys are not supported in this version of the specification and their use in this context has not yet been analysed.

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

5.2.2. CertificateVerify

This message is used to provide explicit proof that an endpoint possesses the private key corresponding to its identity. The format of this message is taken from TLS 1.3:

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

The algorithm field specifies the signature algorithm used (see Section 4.2.3 of [RFC8446] for the definition of this field). The signature is a digital signature using that algorithm.

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

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

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

- * A string that consists of octet 32 (0x20) repeated 64 times
- * The context string "Exported Authenticator" (which is not NUL-terminated)
- * A single 0 octet which serves as the separator
- * The hashed authenticator transcript

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

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

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

If the party that generates the exported authenticator does so with a different connection than the party that is validating it, then the Handshake Context will not match, resulting in a CertificateVerify message that does not validate. This includes situations in which

the application data is sent via TLS-terminating proxy. Given a failed CertificateVerify validation, it may be helpful for the application to confirm that both peers share the same connection using a value derived from the connection secrets (such as the Handshake Context) before taking a user-visible action.

5.2.3. Finished

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

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

5.2.4. Authenticator Creation

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

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

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

6. Empty Authenticator

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

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

7. API considerations

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

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

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

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

7.1. The "request" API

The "request" API takes as input:

- * `certificate_request_context` (from 0 to 255 octets)
- * set of extensions to include (this MUST include `signature_algorithms`) and the contents thereof

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

7.2. The "get context" API

The "get context" API takes as input:

- * authenticator or authenticator request

It returns the `certificate_request_context`.

7.3. The "authenticate" API

The "authenticate" API takes as input:

- * a reference to the initial connection

- * an identity, such as a set of certificate chains and associated extensions (OCSP [RFC6960], SCT [RFC6962], etc.)
- * a signer (either the private key associated with the identity, or interface to perform private key operations) for each chain
- * an authenticator request or `certificate_request_context` (from 0 to 255 octets)

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

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

7.4. The "validate" API

The "validate" API takes as input:

- * a reference to the initial connection
- * an optional authenticator request
- * an authenticator
- * a function for validating a certificate chain

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

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

When validating an authenticator, constant-time comparison should be used.

8. IANA Considerations

8.1. Update of the TLS ExtensionType Registry

IANA is requested to update the entry for `server_name(0)` in the registry for ExtensionType (defined in [RFC8446]) by replacing the value in the "TLS 1.3" column with the value "CH, EE, CR" and adding this document in the "Reference" column.

IANA is also requested to add the following note to the registry:

The addition of the "CR" to the "TLS 1.3" column for the `server_name(0)` extension only marks the extension as valid in a ClientCertificateRequest created as part of client-generated authenticator requests.

8.2. Update of the TLS Exporter Labels Registry

IANA is requested to add the following entries to the registry for Exporter Labels (defined in [RFC5705]): "EXPORTER-client authenticator handshake context", "EXPORTER-server authenticator handshake context", "EXPORTER-client authenticator handshake context", "EXPORTER-client authenticator finished key" and "EXPORTER-server authenticator finished key" with "DTLS-OK" and "Recommended" set to "Y" and this document added to the "Reference" column.

8.3. Update of the TLS HandshakeType Registry

IANA is requested to add the following entry to the registry for HandshakeType (defined in [RFC8446]): "client_certificate_request" with "DTLS-OK" and "Recommended" set to "Y" and this document added to the "Reference" column with the following in the "Note" column: "Used in TLS versions prior to 1.3."

9. Security Considerations

The Certificate/Verify/Finished pattern intentionally looks like the TLS 1.3 pattern which now has been analyzed several times. For example, [SIGMAC] presents a relevant framework for analysis, and section 10. of [RFC8446] contains a comprehensive set of references.

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

- * This property makes it difficult to formally prove that a server is jointly authoritative over multiple identities, rather than individually authoritative over each.
- * There is no indication in (D)TLS about which point in time an authenticator was computed. Any feedback about the time of creation or validation of the authenticator should be tracked as part of the application layer semantics if required.

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

In TLS 1.3 the client can not explicitly learn from the TLS layer whether its Finished message was accepted. Because the application traffic keys are not dependent on the client's final flight, receiving messages from the server does not prove that the server received the client's Finished. To avoid disagreement between the client and server on the authentication status of EAs, servers MUST verify the client Finished before sending an EA or processing a received EA.

10. Acknowledgements

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

11. References

11.1. Normative References

- [HMAC] Krawczyk, H., Bellare, M., and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication", RFC 2104, DOI 10.17487/RFC2104, February 1997, <<https://www.rfc-editor.org/info/rfc2104>>.
- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <<https://www.rfc-editor.org/info/rfc2119>>.
- [RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, DOI 10.17487/RFC5246, August 2008, <<https://www.rfc-editor.org/info/rfc5246>>.

- [RFC5705] Rescorla, E., "Keying Material Exporters for Transport Layer Security (TLS)", RFC 5705, DOI 10.17487/RFC5705, March 2010, <<https://www.rfc-editor.org/info/rfc5705>>.
- [RFC6066] Eastlake 3rd, D., "Transport Layer Security (TLS) Extensions: Extension Definitions", RFC 6066, DOI 10.17487/RFC6066, January 2011, <<https://www.rfc-editor.org/info/rfc6066>>.
- [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>>.
- [RFC7627] Bhargavan, K., Ed., Delignat-Lavaud, A., Pironti, A., Langley, A., and M. Ray, "Transport Layer Security (TLS) Session Hash and Extended Master Secret Extension", RFC 7627, DOI 10.17487/RFC7627, September 2015, <<https://www.rfc-editor.org/info/rfc7627>>.
- [RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <<https://www.rfc-editor.org/info/rfc8174>>.
- [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>>.

11.2. Informative References

- [RFC6960] Santesson, S., Myers, M., Ankney, R., Malpani, A., Galperin, S., and C. Adams, "X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP", RFC 6960, DOI 10.17487/RFC6960, June 2013, <<https://www.rfc-editor.org/info/rfc6960>>.
- [RFC6962] Laurie, B., Langley, A., and E. Kasper, "Certificate Transparency", RFC 6962, DOI 10.17487/RFC6962, June 2013, <<https://www.rfc-editor.org/info/rfc6962>>.
- [RFC7250] Wouters, P., Ed., Tschofenig, H., Ed., Gilmore, J., Weiler, S., and T. Kivinen, "Using Raw Public Keys in Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)", RFC 7250, DOI 10.17487/RFC7250, June 2014, <<https://www.rfc-editor.org/info/rfc7250>>.

- [RFC7540] Belshé, M., Peon, R., and M. Thomson, Ed., "Hypertext Transfer Protocol Version 2 (HTTP/2)", RFC 7540, DOI 10.17487/RFC7540, May 2015, <<https://www.rfc-editor.org/info/rfc7540>>.
- [RFC9001] Thomson, M., Ed. and S. Turner, Ed., "Using TLS to Secure QUIC", RFC 9001, DOI 10.17487/RFC9001, May 2021, <<https://www.rfc-editor.org/info/rfc9001>>.
- [SIGMAC] Krawczyk, H., "A Unilateral-to-Mutual Authentication Compiler for Key Exchange (with Applications to Client Authentication in TLS 1.3)", 2016, <<https://eprint.iacr.org/2016/711.pdf>>.

Author's Address

Nick Sullivan
Cloudflare Inc.
Email: nick@cloudflare.com

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

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

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

Abstract

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

Discussion Venues

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

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

Status of This Memo

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

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

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

This Internet-Draft will expire on 10 November 2022.

Copyright Notice

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

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

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

Table of Contents

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

1. Introduction

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

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

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

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

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

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

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

2. Conventions and Terminology

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

2.1. Change Log

RFC EDITOR PLEASE DELETE THIS SECTION.

(*) indicates changes to the wire protocol.

draft-11

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

draft-10

- * Address superficial comments
- * Add example certificate

draft-09

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

draft-08

- * Include details about the impact of signature forgery attacks

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

draft-07

- * Minor text improvements

draft-06

- * Modified IANA section, fixed nits

draft-05

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

draft-04

- * Add support for client certificates.

draft-03

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

draft-02

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

3. Solution Overview

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

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

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

CertificateRequest that indicates support for this mechanism.

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

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

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

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

3.1. Rationale

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

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

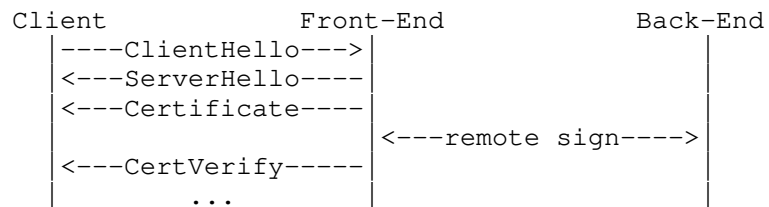
is not cryptographically protected, it is possible that a proxy certificate could be bound to another certificate with the same public key, with different X.509 parameters. Delegated credentials, which rely on a cryptographic binding between the entire certificate and the delegated credential, cannot.

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

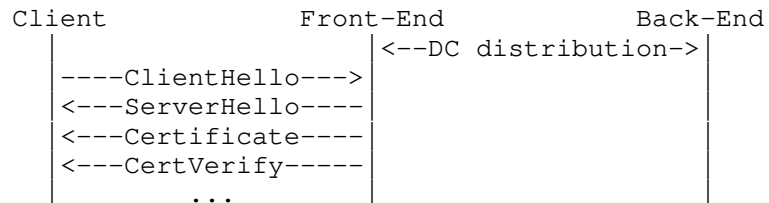
3.2. Related Work

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

Remote key signing:



Delegated Credential:



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

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

credentials.

4. Delegated Credentials

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

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

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

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

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

The DelegatedCredential has the following structure:

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

cred: The Credential structure as previously defined.

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

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

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

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

authentication.

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

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

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

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

4.1. Client and Server Behavior

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

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

4.1.1. Server Authentication

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

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

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

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

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

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

4.1.2. Client Authentication

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

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

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

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

4.1.3. Validating a Delegated Credential

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

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

in Section 4.2.

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

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

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

4.2. Certificate Requirements

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

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

```
DelegationUsage ::= NULL
```

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

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

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

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

5. Operational Considerations

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

5.1. Client Clock Skew

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

6. IANA Considerations

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

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

7. Security Considerations

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

7.1. Security of Delegated Credential's Private Key

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

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

7.2. Re-use of Delegated Credentials in Multiple Contexts

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

7.3. Revocation of Delegated Credentials

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

7.4. Interactions with Session Resumption

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

7.5. Privacy Considerations

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

7.6. The Impact of Signature Forgery Attacks

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

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

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

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

8. Acknowledgements

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

9. References

9.1. Normative References

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

9.2. Informative References

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

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

Appendix A. ASN.1 Module

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

```

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

DEFINITIONS IMPLICIT TAGS ::=
BEGIN

-- EXPORT ALL

IMPORTS

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

-- OID

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

-- EXTENSION

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

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

DelegationUsage ::= NULL

END

```

Appendix B. Example Certificate

The following certificate has the Delegated Credentials OID.

```
-----BEGIN CERTIFICATE-----
MIIFRjCCBMugAwIBAgIQDGeVB+1Y0o/OecHFSJ6YnTAKBggqhkjOPQQDAzBMMQsw
CQYDVQQGEwJVUzEVMBMGA1UEChMMRGlnaUNlcnQgSW5jMSYwJAYDVQQDEx1EaWdp
Q2VydCBFQ0MgU2VjdXJlIFN1cnZlciBDQTAEFw0xOTAzMjYwMDAwMDBaFw0yMTAz
MzAxMjAwMDBaMGoxCzAJBgNVBAYTA1VTMRMwEQYDVQIEwPDYXpZm9ybmlhMRYw
FAYDVQQHEw1TYW4gRnJhbmNpc2NvMRkwFwYDVQQKEwBDbG91ZGZsYXJlLCBjbmu
MRMwEQYDVQQDEwprYzJrZG0uY29tMFkwEwYHKoZIzj0CAQYIKoZIzj0DAQcDQgAE
d4azI83Bw0fcPgfoeiZpZznwGuxjBjv++wzE0zAj8vNiUkKxOWSQiGNLn+xlWUpL
lw9djRN1rLmVmn2gb9GgdK0CA28wggNrMB8GA1UdIwQYMBaAFK0d5h/52j1PwG7o
kcuVpd0x4gqfMB0GA1UdDgQWBBSfcb7fS3fUFAYB91fRcwoDPtgtJjAjBgNVHREE
HDAaggrYzJrZG0uY29tggwqLmtjMmtkbS5jb20wDgYDVROPAQH/BAQDAgeAMB0G
A1UdJQQWMBQGCCsGAQUFBwMBBggrBgEFBQcDAjBpBgNVHR8EYjBqMC6gLKAqhiho
dHRwOi8vY3JsMy5kaWdpY2VydC5jb20vc3NjYS1lY2MtZzEuY3JsMC6gLKAqhiho
dHRwOi8vY3JsNC5kaWdpY2VydC5jb20vc3NjYS1lY2MtZzEuY3JsMEwGA1UdIARF
MEMwNwYJYIZIAyB9bAEBMcowKAYIKwYBBQUHAQEWHGh0dHBzOi8vd3d3LmRpZ21j
ZXJ0LmNvbS9DUFMwCAYGZ4EMAQICMHSGCCsGAQUFBwEBBG8wbTAKBggrBgEFBQcw
AYYYaHR0cDovL29jc3AuZGlnaWN1cnQuY29tMEUGCCsGAQUFBzAChjlodHRwOi8v
Y2FjZXJ0cy5kaWdpY2VydC5jb20vRGlnaUNlcnRFQ0NTZWV1cmVTZXJ2ZXJ0Q5j
cnQwDAYDVROTAQH/BAIwADAPBgkrBgEEAYLaSywEAgUAMIIBfgYKKwYBBAHWeQIE
AgSCAW4EggFqAWgAdgC72d+8H4pxtZOUi5eqkntHOFVCqtS6BqQ1mQ2jh7RhQAA
AWm5hYJ5AAAEAwBHMEUCICiGfq+hSThRL2m8H0awoDR8OpnEHNkF0nI6nL5yYL/j
AiEAXwebGs/T6Es0YarPzoQJrVZqk+sHH/t+jrSrKd5TDjCAdgCHdb/nWXz4jEOZ
X73zbv9WjUdWnV9KtWDBtOr/XqCDDwAAAWm5hYNgAAAEAwBHMEUCIQD9OWA8KGL6
bxDKfGileHJWB0iWieRs88VgJyFag/aFDgIgQ/OsdSF9XOy1foqge0D2DM2FExuw
0JR0AGZWXoNtJzMAAgBE1GUusO7Or8RAB9io/iJA2uaCvtjLmBU/0zOWtbaBqAAA
AWm5hYHgAAAEAwBHMEUCIQC4vua1n3BqthEqpA/VBTcsNwMtAwpcuac2IhJ9wx6X
/AIgb+o00k28JQo9TMpP4vzJ3BD3HXWSNc2Zizbq7mkUQYMwCgYIKoZIzj0EAwMD
aQAwwZgIXAJsX7d0SuA8ddf/m7IWfNfs3MQfJyGkEezMJX1t6sRso5z50SS12LpXe
muGalFE2ZgIXAL+CDUF5pz7mhrAEIjQ1MqlpF9tH40dJGvYZZQ3W23cMzSkDfv1t
y5S4RfWHIIPjBw==
-----END CERTIFICATE-----
```

Authors' Addresses

Richard Barnes
Cisco
Email: rlb@ipv.sx

Subodh Iyengar
Facebook
Email: subodh@fb.com

Nick Sullivan
Cloudflare
Email: nick@cloudflare.com

Eric Rescorla
Mozilla
Email: ekr@rtfm.com

Internet Engineering Task Force
Internet-Draft
Updates: [[List TBD]] (if approved)
Intended status: Standards Track
Expires: January 26, 2019

K. Moriarty
Dell EMC
S. Farrell
Trinity College Dublin
July 25, 2018

Deprecating TLSv1.0 and TLSv1.1
draft-moriarty-tls-oldversions-diediedie-01

Abstract

This document [if approved] formally deprecates Transport Layer Security (TLS) versions 1.0 [RFC2246] and 1.1 [RFC4346] and moves these documents to the historic state. These versions lack support for current and recommended cipher suites, 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. Products having to support older versions increase the attack surface unnecessarily and increase opportunities for misconfigurations. Supporting these older versions also requires additional effort for library and product maintenance.

This document updates the backward compatibility sections of TLS RFCs [[list TBD]] to prohibit fallback to TLSv1.0 and TLSv1.1. This document also updates RFC 7525.

Status of This Memo

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

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

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

This Internet-Draft will expire on January 26, 2019.

Copyright Notice

Copyright (c) 2018 IETF Trust and the persons identified as the document authors. All rights reserved.

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

Table of Contents

1. Introduction	2
1.1. Terminology	4
2. Support for Deprecation	4
3. Removing Support	5
4. Usage	5
4.1. Web	6
4.2. Mail	6
4.3. Operating Systems	7
4.4. Enterprise Networks	7
5. SHA-1	8
6. Do Not Use TLSv1.0	8
7. Do Not Use TLSv1.1	9
8. Do Not Use SHA-1 in TLSv1.2	9
9. Updates to RFC7525	9
10. Security Considerations	10
11. Acknowledgements	10
12. IANA Considerations	10
13. References	11
13.1. Normative References	11
13.2. Informative References	11
Appendix A. Change Log	14
Authors' Addresses	14

1. Introduction

[[Text in double-square brackets is intended to be fixed as the draft evolves. You've seen that we need to figure out the list of RFCs that this'd update in the abstract. There is a repo for this at: <https://github.com/sftcd/tls-oldversions-diediedie> - PRs (on the xml file) are welcome there.]]

Transport Layer Security (TLS) versions 1.0 [RFC2246] and 1.1 [RFC4346] were superseded by TLSv1.2 [RFC5246] in 2008, which has now itself been superseded by TLSv1.3 [I-D.ietf-tls-tls13]. It is therefore timely to further deprecate these old versions. The expectation is that TLSv1.2 will continue to be used for many years alongside TLSv1.3.

TLSv1.1 and TLSv1.0 are also actively being deprecated in accordance with guidance from government agencies (e.g. NIST SP 80052r2 [NIST800-52r2]) and industry consortia such as the Payment Card Industry Association (PCI) [PCI-TLS1].

The primary technical reasons for deprecating these versions include:

- o They require implementation of older cipher suites that are no longer desirable for cryptographic reasons, e.g. TLSv1.0 makes TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA mandatory to implement
- o Lack of support for current recommended cipher suites, especially using AEAD ciphers which are not supported prior to TLS 1.2.
Note: registry entries for no-longer-desirable ciphersuites remain in the registries, but many TLS registries are being updated through [I-D.ietf-tls-iana-registry-updates] which denotes such entries as "not recommended."
- o Integrity of the handshake depends on SHA-1 hash
- o Authentication of the peers depends on SHA-1 signatures
- o Support for four protocol versions increases the likelihood of misconfiguration
- o At least one widely-used library has plans to drop TLSv1.1 and TLSv1.0 support in upcoming releases; products using such libraries would need to use older versions of the libraries to support TLSv1.0 and TLSv1.1, which is clearly undesirable

Deprecation of these versions is intended to assist developers as additional justification to no longer support older TLS versions and to migrate to a minimum of 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.

[[This draft is being written now so that the TLS WG chairs can just hit the "publication requested" button as soon as there is WG consensus to deprecate these ancient versions of TLS. The authors however think that deprecation now is timely.]]

1.1. 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

Industry has actively followed guidance provided by NIST and the PCI Council to deprecate TLSv1.0 and TLSv1.1 by June 30, 2018. TLSv1.2 should remain a minimum baseline for TLS support at this time.

Specific details on attacks against TLSv1.0 and TLSv1.1 as well as their mitigations are provided in NIST SP800-52r2 [NIST800-52r2], RFC 7457 [RFC7457] and other referenced RFCs. Although the attacks have been mitigated, if support is dropped for future library releases for these versions, it is unlikely attacks found going forward will be mitigated in older library releases.

NIST for example have provided the following rationale, copied with permission from NIST SP800-52r2 [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.

TLS 1.2, 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. TLS 1.2 also adds authenticated encryption with associated data (AEAD) cipher suites.

TLS 1.3, specified in TLSv1.3 [I-D.ietf-tls-tls13], 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.

The Canadian government treasury board have mandated that these old versions of TLS not be used. [Canada]

3. Removing Support

[[This section can be removed upon publication - or maybe keep it?]]

Support for TLSv1.0 has been removed by the July 2018 PCI deadline from the following standards, products, and services:

- o 3GPP 5G
- o Amazon Elastic Load Balancing [Amazon]
- o CloudFare [CloudFlare]
- o Digicert [Digicert]
- o GitHub [GIT]
- o KeyCDN [KeyCDN]
- o PayPal [paypal]
- o Stripe [stripe]
- o [[Numerous web sites...]]

Many web sites have taken the action of including the deprecation of TLSv1.1 into their plans for deprecating TLSv1.0 for the PCI council deadline. Support for TLSv1.1 has been removed by the July 2018 PCI deadline from the following standards, products, and services:

- o 3GPP 5G Release 16
- o Amazon Elastic Load Balancing [Amazon]
- o CloudFare [CloudFlare]
- o GitHub [GIT]
- o PayPal [paypal]
- o Stripe [stripe]
- o [[Numerous web sites...]]

4. Usage

[[This section can be removed upon publication - or maybe keep it?]]

4.1. Web

Usage statistics for TLSv1.0 and TLSv1.1 on the public web vary, but have been in general very low and declined further with the impending PCI deadline to migrate off of TLSv1.0 by June 30, 2018. As of January 2018, [StackExchange] quoted 4 percent of browsers using TLSv1.0.

The number of websites supporting TLS 1.2 is still growing (+0.4%), and has reached 92% according to sslpulse as of June 19, 2018. [SSLPulse] Deprecating TLS 1.0 and TLS 1.1 will thus not have a major impact on browser or web server implementations.

Figure 1 presents statistics for use of TLS versions in the web.

Name/Ref	Date	SSLv3	TLSv1.0	TLSv1.1	TLSv1.2	TLSv1.3
Alexa [1]	20180226	-	2.0	<0.1	97.9	-
Cloudflare [2]	20180518	0.0	9.3	0.2	84.9	5.5
Firefox [3]	20180709	-	1.0	-	94.0	5.0
Chrome [4]	20180711	-	0.4	<0.1	-	-

[1] <https://scotthelme.co.uk/alexa-top-1-million-analysis-february-2018/>
 [2] <https://www.ietf.org/mail-archive/web/tls/current/msg26578.html>
 [3] <https://www.ietf.org/mail-archive/web/tls/current/msg26575.html>
 [4] <https://www.ietf.org/mail-archive/web/tls/current/msg26620.html>

Figure 1: Web Statistics

4.2. Mail

E-Mail uses TLS for SMTP, submission (port 587), POP/POP3 and IMAP. Typically email deployments lag public web deployments in terms of the rate of adoption of new TLS versions. Figure 2 presents statistics for use of TLS versions in the email applications.

Name/Ref	Date	SSLv3	TLSv1.0	TLSv1.1	TLSv1.2	TLSv1.3
Clusters [1]	20180316	<0.1	10.6	<0.1	89.3	-
TLSA [2]	20180710	-	1.4	0.1	98.5	-
UK-ESP [3]	20180710	-	19.9	<0.1	-	-

[1] <https://eprint.iacr.org/2018/299>
[2] <https://www.ietf.org/mail-archive/web/tls/current/msg26603.html>
[3] <https://www.ietf.org/mail-archive/web/tls/current/msg26603.html>

Figure 2: Mail Statistics

4.3. Operating Systems

Figure 3 presents statistics for use of TLS versions in operating systems.

Name/Ref	Date	SSLv3	TLSv1.0	TLSv1.1	TLSv1.2	TLSv1.3
Windows cli [1]	20180709	-	>10.0	~0.3	-	-
Windows svr [1]	20180709	-	~1.5	~0.0	-	-
Apple [2]	20180709	-	0.4	-	99.6	-

[1] <https://www.ietf.org/mail-archive/web/tls/current/msg26577.html>
[2] <https://www.ietf.org/mail-archive/web/tls/current/msg26634.html>

Figure 3: Operating System Statistics

4.4. Enterprise Networks

Figure 4 presents statistics for use of TLS versions in the enterprise networks. The tcd.ie numbers below were the result of a student project and need further validation.

Name/Ref	Date	SSLv3	TLSv1.0	TLSv1.1	TLSv1.2	TLSv1.3
tcd.ie [1]	20180713	18.0	35.0	0	45.0	0

[1] <https://www.ietf.org/mail-archive/web/tls/current/msg26633.html>

Figure 4: Enterprise Network Statistics

5. SHA-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 SHA-1 hash or a not 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.

6. 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. TLSv1.0 can be configured to prevent interception, though using the highest version available is preferable.

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.

[[Text here is derived (or stolen:-) from [RFC7568]]]

7. 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. TLSv1.1 can be configured to prevent interception, though using the highest version available is preferable. 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.

8. Do Not Use SHA-1 in TLSv1.2

[[This section was suggested in PR#2 by Hubert Kario. We're not clear if the WG would like this draft to include this or not, so will ask the TLS WG at the appropriate time.]]

SHA-1 as a signature hash MUST NOT be used. That means that clients MUST send signature_algorithms extension and that extension MUST NOT include pairs that include SHA-1 hash. In particular, values {2, 1}, {2, 2} and {2, 3} MUST NOT be present in the extension.

Note: this does not affect cipher suites that use SHA-1 HMAC for data integrity as the HMAC construction is still considered secure and when they are used in TLSv1.2 SHA-256 is used for handshake integrity.

9. Updates to RFC7525

[[Since RFC7525 is BCP195, there'll probably be some process-fun to do an update of that. Formally, it may be that this document becomes

a new part of BCP195 I guess, but we can figure that out with chairs and ADs.]]

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

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

Rationale: TLS 1.0 (published in 1999) does not support many modern, strong cipher suites. In addition, TLS 1.0 lacks a per-record Initialization Vector (IV) for CBC-based cipher suites and does not warn against common padding errors.

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

Rationale: TLS 1.1 (published in 2006) is a security improvement over TLS 1.0 but still does not support certain stronger cipher suites.

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

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

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

10. Security Considerations

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

11. Acknowledgements

Thanks to those that provided usage data, reviewed and/or improved this document, including: David Benjamin, David Black, Viktor Dukhovni, Alessandro Ghedini, Jeremy Harris, Russ Housley, Hubert Kario, Loganaden Velvindron, Eric Mill, Yoav Nir, Andrei Popov, Eric Rescorla, and Yaron Sheffer.

12. IANA Considerations

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

13. References

13.1. 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>>.
- [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>>.
- [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>>.
- [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>>.

13.2. Informative References

- [Amazon] Amazon, "Amazon Elastic Load Balancing Support Deprecated TLSv1.0 and TLSv1.1 <https://aws.amazon.com/about-aws/whats-new/2017/02/elastic-load-balancing-support-for-tls-1-1-and-tls-1-2-pre-defined-security-policies/>", 2017.
- [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.
- [Canada] Treasury Board of Canada Secretariat, "Implementing HTTPS for Secure Web Connections: Information Technology Policy Implementation Notice (ITPIN)", June 2018, <<https://www.canada.ca/en/treasury-board-secretariat/services/information-technology/policy-implementation-notice/implementing-https-secure-web-connections-itspin.html>>.

- [CloudFlare] CloudFlare, "CloudFlare Deprecated TLSv1.0 and TLSv1.1 <https://blog.cloudflare.com/deprecating-old-tls-versions-on-cloudflare-dashboard-and-api/>", 2018.
- [Digicert] Digicert, "Deprecating TLS 1.0 and 1.1 <https://www.digicert.com/blog/deprecating-tls-1-0-and-1-1/>", 2018.
- [GIT] GitHub, "GitHub Deprecates TLSv1.0 and TLSv1.1 <https://githubengineering.com/crypto-removal-notice/>", 2018.
- [I-D.ietf-tls-iana-registry-updates] Salowey, J. and S. Turner, "IANA Registry Updates for Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)", draft-ietf-tls-iana-registry-updates-05 (work in progress), May 2018.
- [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.
- [KeyCDN] KeyCDN, "Deprecating TLS 1.0 and 1.1 Enhancing Security for Everyone <https://www.keycdn.com/blog/deprecating-tls-1-0-and-1-1/>", 2018.
- [NIST800-52r2] National Institute of Standards and Technology, "NIST SP800-52r2 <https://csrc.nist.gov/CSRC/media/Publications/sp/800-52/rev-2/draft/documents/sp800-52r2-draft.pdf>", 2018.
- [paypal] Paypal, "'TLS1.2 and HTTP/1.1 Upgrade' <https://www.paypal-notice.com/en/TLS-1.2-and-HTTP1.1-Upgrade/>", 2018.
- [PCI-TLS1] PCI Security Standards Council, "Migrating from SSL and Early TLS https://www.pcisecuritystandards.org/documents/Migrating-from-SSL-Early-TLS-Info-Supp-v1_1.pdf", 2016.
- [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>>.

- [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>>.
- [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>>.
- [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>>.
- [SSLpulse] SSLpulse - will be deleted before publication, "SSLpulse <https://www.ssllabs.com/ssl-pulse/>", 2018.
- [StackExchange] StackExchange - will be deleted before publication, "Stackexchange <https://security.stackexchange.com/questions/177182/is-there-a-list-of-old-browsers-that-only-support-tls-1-0>", 2018.
- [stripe] Stripe, "Upgrading to SHA-2 and TLS 1.2" <https://stripe.com/blog/upgrading-tls>", 2018.

Appendix A. Change Log

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

From -00 to -01:

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

Authors' Addresses

Kathleen Moriarty
Dell EMC
176 South Street
Hopkinton
United States

EMail: Kathleen.Moriarty.ietf@gmail.com

Stephen Farrell
Trinity College Dublin
Dublin 2
Ireland

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

tls
Internet-Draft
Intended status: Experimental
Expires: January 3, 2019

E. Rescorla
RTFM, Inc.
K. Oku
Fastly
N. Sullivan
Cloudflare
C. Wood
Apple, Inc.
July 02, 2018

Encrypted Server Name Indication for TLS 1.3
draft-rescorla-tls-esni-00

Abstract

This document defines a simple mechanism for encrypting the Server Name Indication for TLS 1.3.

Status of This Memo

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

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

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

This Internet-Draft will expire on January 3, 2019.

Copyright Notice

Copyright (c) 2018 IETF Trust and the persons identified as the document authors. All rights reserved.

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

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

Table of Contents

1. Introduction	3
2. Conventions and Definitions	4
3. Overview	4
3.1. Topologies	4
3.2. SNI Encryption	5
4. Publishing the SNI Encryption Key	5
5. The "encrypted_server_name" extension	7
5.1. Client Behavior	8
5.2. Client-Facing Server Behavior	9
5.3. Shared Mode Server Behavior	10
5.4. Split Mode Server Behavior	10
6. Compatibility Issues	10
6.1. Misconfiguration	11
6.2. Middleboxes	11
7. Security Considerations	11
7.1. Why is cleartext DNS OK?	12
7.2. Comparison Against Criteria	12
7.2.1. Mitigate against replay attacks	12
7.2.2. Avoid widely-deployed shared secrets	12
7.2.3. Prevent SNI-based DoS attacks	13
7.2.4. Do not stick out	13
7.2.5. Forward secrecy	13
7.2.6. Proper security context	13
7.2.7. Split server spoofing	13
7.2.8. Supporting multiple protocols	13
7.3. Misrouting	14
8. IANA Considerations	14
8.1. Update of the TLS ExtensionType Registry	14
9. References	14
9.1. Normative References	14
9.2. Informative References	15
Appendix A. Communicating SNI to Backend Server	16
Appendix B. Alternative SNI Protection Designs	16
B.1. TLS-layer	16
B.1.1. TLS in Early Data	16
B.1.2. Combined Tickets	17
B.2. Application-layer	17
B.2.1. HTTP/2 CERTIFICATE Frames	17
Appendix C. Total Client Hello Encryption	17
Appendix D. Acknowledgments	18
Authors' Addresses	18

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 [I-D.ietf-tls-tls13] encrypts most of the handshake, including the server certificate, there are several other channels that allow an on-path attacker to determine the domain name the client is trying to connect to, including:

- o Cleartext client DNS queries.
- o Visible server IP addresses, assuming the the server is not doing domain-based virtual hosting.
- o Cleartext Server Name Indication (SNI) [RFC6066] in ClientHello messages.

DoH [I-D.ietf-doh-dns-over-https] 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, SNI is an explicit signal used to determine the server's identity. Indirect mechanisms such as traffic analysis also exist.

The TLS WG has extensively studied the problem of protecting SNI, but has been unable to develop a completely generic solution. [I-D.ietf-tls-sni-encryption] provides a description of the problem space and some of the proposed techniques. One of the more difficult problems is "Do not stick out" ([I-D.ietf-tls-sni-encryption]; Section 3.4): if only sensitive/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 SNI is being protected. Unfortunately, the result often has undesirable performance consequences, incomplete coverage, or both.

The design in this document takes a different approach: it assumes that private origins will co-locate with or hide behind a provider (CDN, app server, etc.) which is able to activate encrypted SNI (ESNI) for all of the domains it hosts. Thus, the use of encrypted SNI does not indicate that the client is attempting to reach a private origin, but only that it is going to a particular service provider, which the observer could already tell from the IP address.

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

This document is designed to operate in one of two primary topologies shown below, which we call "Shared Mode" and "Split Mode"

3.1. Topologies



Figure 1: Shared Mode Topology

In Shared Mode, the provider is the origin server for all the domains whose DNS records point to it and clients form a TLS connection directly to that provider, which has access to the plaintext of the connection.

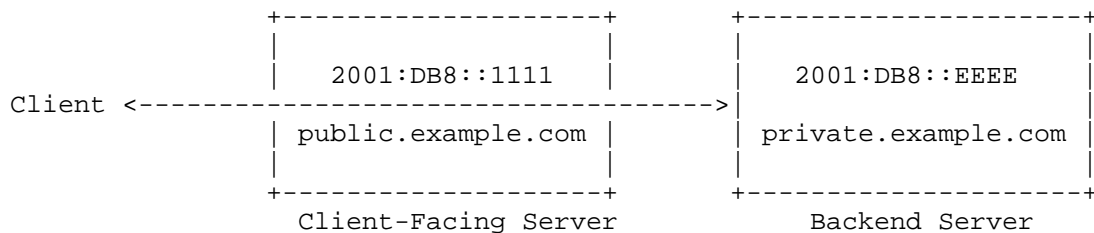


Figure 2: Split Mode Topology

In Split Mode, the provider is not the origin server for private domains. Rather the DNS records for private domains point to the provider, but the provider's server just relays the connection back to the backend server, which is the true origin server. The provider

does not have access to the plaintext of the connection. In principle, the provider might not be the origin for any domains, but as a practical matter, it is probably the origin for a large set of innocuous domains, but is also providing protection for some private domains. Note that the backend server can be an unmodified TLS 1.3 server.

3.2. SNI Encryption

The protocol designed in this document is quite straightforward.

First, the provider publishes a public key which is used for SNI encryption for all the domains for which it serves directly or indirectly (via Split mode). This document defines a publication mechanism using DNS, but other mechanisms are also 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 preconfigured.

When a client wants to form a TLS connection to any of the domains served by an ESNI-supporting provider, it replaces the "server_name" extension in the ClientHello with an "encrypted_server_name" extension, which contains the true extension encrypted under the provider's public key. The provider can then decrypt the extension and either terminate the connection (in Shared Mode) or forward it to the backend server (in Split Mode).

4. Publishing the SNI Encryption Key

SNI Encryption keys can be published in the DNS using the ESNIKeys structure, defined below.

```
// Copied from TLS 1.3
struct {
    NamedGroup group;
    opaque key_exchange<1..2^16-1>;
} KeyShareEntry;

struct {
    uint8 checksum[4];
    KeyShareEntry keys<4..2^16-1>;
    CipherSuite cipher_suites<2..2^16-2>;
    uint16 padded_length;
    uint64 not_before;
    uint64 not_after;
    Extension extensions<0..2^16-1>;
} ESNIKeys;
```

checksum The first four (4) octets of the SHA-256 message digest [RFC6234] of the ESNIKeys structure starting from the first octet of "keys" to the end of the structure.

keys The list of keys which can be used by the client to encrypt the SNI. Every key being listed MUST belong to a different group.

padded_length : The length to pad the ServerNameList value to prior to encryption. This value SHOULD be set to the largest ServerNameList the server expects to support rounded up the nearest multiple of 16. If the server supports wildcard names, it SHOULD set this value to 260.

not_before The moment when the keys become valid for use. The value is represented as seconds from 00:00:00 UTC on Jan 1 1970, not including leap seconds.

not_after The moment when the keys become invalid. Uses the same unit as not_before.

extensions A list of extensions that the client can take into consideration when generating a Client Hello message. The format is defined in [I-D.ietf-tls-tls13]; Section 4.2. The purpose of the field is to provide room for additional features in the future; this document does not define any extension.

The semantics of this structure are simple: any of the listed keys may be used to encrypt the SNI for the associated domain name. The cipher suite list is orthogonal to the list of keys, so each key may be used with any cipher suite.

This structure is placed in the RRData section of a TXT record as a base64-encoded string. If this encoding exceeds the 255 octet limit of TXT strings, it must be split across multiple concatenated strings as per Section 3.1.3 of [RFC4408].

The name of each TXT record MUST match the name composed of _esni and the query domain name. That is, if a client queries example.com, the ESNI TXT Resource Record might be:

_esni.example.com. 60S IN TXT "... "..."

Servers MUST ensure that if multiple A or AAAA records are returned for a domain with ESNI support, all the servers pointed to by those records are able to handle the keys returned as part of a ESNI TXT record for that domain.

Clients obtain these records by querying DNS for ESNI-enabled server domains. Thus, servers operating in Split Mode SHOULD have DNS configured to return the same A (or AAAA) record for all ESNI-enabled servers they service. This yields an anonymity set of cardinality equal to the number of ESNI-enabled server domains supported by a given client-facing server. Thus, even with SNI encryption, an attacker which can enumerate the set of ESNI-enabled domains supported by a client-facing server can guess the correct SNI with probability at least $1/K$, where K is the size of this ESNI-enabled server anonymity set. This probability may be increased via traffic analysis or other mechanisms.

The "checksum" field provides protection against transmission errors, including those caused by intermediaries such as a DNS proxy running on a home router.

"not_before" and "not_after" fields represent the validity period of the published ESNI keys. Clients MUST NOT use ESNI keys that was covered by an invalid checksum or beyond the published period. Servers SHOULD set the Resource Record TTL small enough so that the record gets discarded by the cache before the ESNI keys reach the end of their validity period. Note that servers MAY need to retain the decryption key for some time after "not_after", and will need to consider clock skew, internal caches and the like, when selecting the "not_before" and "not_after" values.

Client MAY cache the ESNIKeys for a particular domain based on the TTL of the Resource Record, but SHOULD NOT cache it based on the not_after value, to allow servers to rotate the keys often and improve forward secrecy.

Note that the length of this structure MUST NOT exceed $2^{16} - 1$, as the RDLENGTH is only 16 bits [RFC1035].

5. The "encrypted_server_name" extension

The encrypted SNI is carried in an "encrypted_server_name" extension, which contains an EncryptedSNI structure:

```
struct {
    CipherSuite suite;
    opaque record_digest<0..2^16-1>;
    opaque encrypted_sni<0..2^16-1>;
} EncryptedSNI;
```

record_digest A cryptographic hash of the ESNIKeys structure from which the ESNI key was obtained, i.e., from the first byte of

"checksum" to the end of the structure. This hash is computed using the hash function associated with "suite".

suite The cipher suite used to encrypt the SNI.

encrypted_sni The original ServerNameList from the "server_name" extension, padded and AEAD-encrypted using cipher suite "suite" and with the key generated as described below.

5.1. Client Behavior

In order to send an encrypted SNI, the client MUST first select one of the server ESNIKeyShareEntry values and generate an (EC)DHE share in the matching group. This share is then used for the client's "key_share" extension and will be used to derive both the SNI encryption key and the (EC)DHE shared secret which is used in the TLS key schedule. This has two important implications:

- o The client MUST only provide one KeyShareEntry
- o The server is committing to support every group in the ESNIKeys list (see below for server behavior).

The SNI encryption key is computed from the DH shared secret Z as follows:

```
Zx = HKDF-Extract(0, Z)
key = HKDF-Expand-Label(Zx, "esni key", Hash(ClientHello.Random), key_length)
iv = HKDF-Expand-Label(Zx, "esni iv", Hash(ClientHello.Random), iv_length)
```

The client then creates a PaddedServerNameList:

```
struct {
    ServerNameList sni;
    opaque zeros[ESNIKeys.padded_length - length(sni)];
} PaddedServerNameList;
```

This value consists of the serialized ServerNameList padded with enough zeroes to make the total structure ESNIKeys.padded_length bytes long. The purpose of the padding is to prevent attackers from using the length of the "encrypted_server_name" extension to determine the true SNI. If the serialized ServerNameList is longer than ESNIKeys.padded_length, the client MUST NOT use the "encrypted_server_name" extension.

The EncryptedSNI.encrypted_sni value is then computed using the usual TLS 1.3 AEAD:


```
encrypted_sni = AEAD-Encrypt(key, iv, "", PaddedServerNameList)
```

Note: future extensions may end up reusing the server's ESNIKeyShareEntry for other purposes within the same message (e.g., encrypting other values). Those usages MUST have their own HKDF labels to avoid reuse.

[[OPEN ISSUE: If in future you were to reuse these keys for 0-RTT priming, then you would have to worry about potentially expanding twice of Z_extracted. We should think about how to harmonize these to make sure that we maintain key separation. Similarly, if the server uses the same key for ESNI as it does in ServerKeyShare, this is going to involve re-use of Z in some hard to analyze ways. Of course, this would also involve abandoning PFS.]]

This value is placed in an "encrypted_server_name" extension.

The client MAY either omit the "server_name" extension or provide an innocuous dummy one (this is required for technical conformance with [RFC7540]; Section 9.2.)

5.2. Client-Facing Server Behavior

Upon receiving an "encrypted_server_name" extension, the client-facing server MUST first perform the following checks:

- o If it is unable to negotiate TLS 1.3 or greater, it MUST abort the connection with a "handshake_failure" alert.
- o If the EncryptedSNI.record_digest value does not match the cryptographic hash of any known ENSIKeys structure, it MUST abort the connection with an "illegal_parameter" alert. This is necessary to prevent downgrade attacks. [[OPEN ISSUE: We looked at ignoring the extension but concluded this was better.]]
- o If more than one KeyShareEntry has been provided, or if that share's group does not match that for the SNI encryption key, it MUST abort the connection with an "illegal_parameter" alert.
- o If the length of the "encrypted_server_name" extension is inconsistent with the advertised padding length (plus AEAD expansion) the server MAY abort the connection with an "illegal_parameter" alert without attempting to decrypt.

Assuming these checks succeed, the server then computes K_sni and decrypts the ServerName value. If decryption fails, the server MUST abort the connection with a "decrypt_error" alert.

If the decrypted value's length is different from the advertised `ESNIKeys.padded_length` or the padding consists of any value other than 0, then the server MUST abort the connection with an `illegal_parameter` alert. Otherwise, the server uses the `PaddedServerNameList.sni` value as if it were the "server_name" extension. Any actual "server_name" extension is ignored.

Upon determining the true SNI, the client-facing server then either serves the connection directly (if in Shared Mode), in which case it executes the steps in the following section, or forwards the TLS connection to the backend server (if in Split Mode). In the latter case, it does not make any changes to the TLS messages, but just blindly forwards them.

5.3. Shared Mode Server Behavior

A server operating in Shared Mode uses `PaddedServerNameList.sni` as if it were the "server_name" extension to finish the handshake. It SHOULD pad the Certificate message, via padding at the record layer, such that its length equals the size of the largest possible Certificate (message) covered by the same ESNI key.

5.4. Split Mode Server Behavior

The backend Server ignores both the "encrypted_server_name" and the "server_name" (if any) and completes the handshake as usual. If in Shared Mode, the server will still know the true SNI, and can use it for certificate selection. In Split Mode, it may not know the true SNI and so will generally be configured to use a single certificate. Appendix A describes a mechanism for communicating the true SNI to the backend server.

Similar to the Shared Mode behavior, the backend server in Split Mode SHOULD pad the Certificate message, via padding at the record layer such that its length equals the size of the largest possible Certificate (message) covered by the same ESNI key.

[[OPEN ISSUE: Do we want "encrypted_server_name" in EE? It's clearer communication, but would make it so you could not operate a current TLS 1.3 server as a backend server.]]

6. Compatibility Issues

In general, this mechanism is designed only to be used with servers which have opted in, thus minimizing compatibility issues. However, there are two scenarios where that does not apply, as detailed below.

6.1. Misconfiguration

If DNS is misconfigured so that a client receives ESNI keys for a server which is not prepared to receive ESNI, then the server will ignore the "encrypted_server_name" extension, as required by [I-D.ietf-tls-tls13]; Section 4.1.2. If the server does not require SNI, it will complete the handshake with its default certificate. Most likely, this will cause a certificate name mismatch and thus handshake failure. Clients SHOULD not fall back to cleartext SNI, because that allows a network attacker to disclose the SNI. They MAY attempt to use another server from the DNS results, if one is provided.

6.2. Middleboxes

A more serious problem is MITM proxies which do not support this extension. [I-D.ietf-tls-tls13]; Section 9.3 requires that such proxies remove any extensions they do not understand. This will have one of two results when connecting to the client-facing server:

1. The handshake will fail if the client-facing server requires SNI.
2. The handshake will succeed with the client-facing server's default certificate.

A Web client can securely detect case (2) because it will result in a connection which has an invalid identity (most likely) but which is signed by a certificate which does not chain to a publicly known trust anchor. The client can detect this case and disable ESNI while in that network configuration.

In order to enable this mechanism, client-facing servers SHOULD NOT require SNI, but rather respond with some default certificate.

A non-conformant MITM proxy will forward the ESNI extension, substituting its own KeyShare value, with the result that the client-facing server will not be able to decrypt the SNI. This causes a hard failure. Detecting this case is difficult, but clients might opt to attempt captive portal detection to see if they are in the presence of a MITM proxy, and if so disable ESNI. Hopefully, the TLS 1.3 deployment experience has cleaned out most such proxies.

7. Security Considerations

7.1. Why is cleartext DNS OK?

In comparison to [I-D.kazuho-protected-sni], wherein DNS Resource Records are signed via a server private key, ESNIKeys 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 ESNIKeys (so that the client encrypts SNI to them) or strip the ESNIKeys from the response. However, in the face of an attacker that controls DNS, no SNI 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 ESNIKeys 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 SNI. Moreover, as noted in the introduction, SNI encryption is less useful without encryption of DNS queries in transit via DoH or DPRIVE mechanisms.

7.2. Comparison Against Criteria

[I-D.ietf-tls-sni-encryption] lists several requirements for SNI encryption. In this section, we re-iterate these requirements and assess the ESNI design against them.

7.2.1. Mitigate against replay attacks

Since the SNI encryption key is derived from a (EC)DH operation between the client's ephemeral and server's semi-static ESNI key, the ESNI encryption is bound to the Client Hello. It is not possible for an attacker to "cut and paste" the ESNI value in a different Client Hello, with a different ephemeral key share, as the terminating server will fail to decrypt and verify the ESNI value.

7.2.2. Avoid widely-deployed 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 ESNI key is provided, sharing is optimally bound by the number of hosts that share an IP address. Server operators may further limit sharing by sending different Resource Records containing ESNIKeys with different keys using a short TTL.

7.2.3. Prevent SNI-based DoS attacks

This design requires servers to decrypt ClientHello messages with EncryptedSNI 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.

7.2.4. Do not stick out

By sending SNI and ESNI values (with illegitimate digests), or by sending legitimate ESNI values for and "fake" SNI values, clients do not display clear signals of ESNI intent to passive eavesdroppers. As more clients enable ESNI support, e.g., as normal part of Web browser functionality, with keys supplied by shared hosting providers, the presence of ESNI extensions becomes less suspicious and part of common or predictable client behavior. In other words, if all Web browsers start using ESNI, the presence of this value does not signal suspicious behavior to passive eavesdroppers.

7.2.5. Forward secrecy

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

7.2.6. Proper security context

This design permits servers operating in Split Mode to forward connections directly to backend origin servers, thereby avoiding unnecessary MiTM attacks.

7.2.7. Split server spoofing

Assuming ESNIKeys retrieved from DNS are validated, e.g., via DNSSEC or fetched from a trusted Recursive Resolver, spoofing a server operating in Split Mode is not possible. See Section 7.1 for more details regarding cleartext DNS.

7.2.8. Supporting multiple protocols

This design has no impact on application layer protocol negotiation. It only affects connection routing, server certificate selection, and client certificate verification. Thus, it is compatible with multiple protocols.

7.3. Misrouting

Note that the backend server has no way of knowing what the SNI was, but that does not lead to additional privacy exposure because the backend server also only has one identity. This does, however, change the situation slightly in that the backend server might previously have checked SNI and now cannot (and an attacker can route a connection with an encrypted SNI to any backend server and the TLS connection will still complete). However, the client is still responsible for verifying the server's identity in its certificate.

[[TODO: Some more analysis needed in this case, as it is a little odd, and probably some precise rules about handling ESNI and no SNI uniformly?]]

8. IANA Considerations

8.1. Update of the TLS ExtensionType Registry

IANA is requested to Create an entry, `encrypted_server_name(0xffce)`, in the existing registry for ExtensionType (defined in [I-D.ietf-tls-tls13]), with "TLS 1.3" column values being set to "CH", and "Recommended" column being set to "Yes".

9. References

9.1. Normative References

- [I-D.ietf-tls-exported-authenticator]
Sullivan, N., "Exported Authenticators in TLS", draft-ietf-tls-exported-authenticator-07 (work in progress), June 2018.
- [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.
- [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>>.

- [RFC4408] Wong, M. and W. Schlitt, "Sender Policy Framework (SPF) for Authorizing Use of Domains in E-Mail, Version 1", RFC 4408, DOI 10.17487/RFC4408, April 2006, <<https://www.rfc-editor.org/info/rfc4408>>.
- [RFC6066] Eastlake 3rd, D., "Transport Layer Security (TLS) Extensions: Extension Definitions", RFC 6066, DOI 10.17487/RFC6066, January 2011, <<https://www.rfc-editor.org/info/rfc6066>>.
- [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>>.
- [RFC7540] Belshe, M., Peon, R., and M. Thomson, Ed., "Hypertext Transfer Protocol Version 2 (HTTP/2)", RFC 7540, DOI 10.17487/RFC7540, May 2015, <<https://www.rfc-editor.org/info/rfc7540>>.
- [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>>.

9.2. Informative References

- [I-D.ietf-doh-dns-over-https]
Hoffman, P. and P. McManus, "DNS Queries over HTTPS (DoH)", draft-ietf-doh-dns-over-https-12 (work in progress), June 2018.
- [I-D.ietf-tls-sni-encryption]
Huitema, C. and E. Rescorla, "Issues and Requirements for SNI Encryption in TLS", draft-ietf-tls-sni-encryption-03 (work in progress), May 2018.
- [I-D.kazuho-protected-sni]
Oku, K., "TLS Extensions for Protecting SNI", draft-kazuho-protected-sni-00 (work in progress), July 2017.
- [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>>.

[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>>.

Appendix A. Communicating SNI to Backend Server

As noted in Section 5.4, the backend server will generally not know the true SNI in Split Mode. It is possible for the client-facing server to communicate the true SNI to the backend server, but at the cost of having that communication not be unmodified TLS 1.3. The basic idea is to have a shared key between the client-facing server and the backend server (this can be a symmetric key) and use it to AEAD-encrypt Z and send the encrypted blob at the beginning of the connection before the ClientHello. The backend server can then decrypt ESNI to recover the true SNI.

An obvious alternative here would be to have the client-facing server forward the true SNI, but that would allow the client-facing server to lie. In this design, the attacker would need to be able to find a Z which would expand into a key that would validly AEAD-encrypt a message of his choice, which should be intractable (Hand-waving alert!).

Appendix B. 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.

B.1. TLS-layer

B.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.

B.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.

B.2. Application-layer

B.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 C. Total Client Hello Encryption

The design described here only provides encryption for the SNI, but not for other extensions, such as ALPN. Another potential design would be to encrypt all of the extensions using the same basic structure as we use here for ESNI. That design has the following advantages:

- o It protects all the extensions from ordinary eavesdroppers
- o If the encrypted block has its own KeyShare, it does not necessarily require the client to use a single KeyShare, because

the client's share is bound to the SNI by the AEAD (analysis needed).

It also has the following disadvantages:

- o The client-facing server can still see the other extensions. By contrast we could introduce another EncryptedExtensions block that was encrypted to the backend server and not the client-facing server.
- o It requires a mechanism for the client-facing server to provide the extension-encryption key to the backend server (as in Appendix A and thus cannot be used with an unmodified backend server.
- o A conformant middlebox will strip every extension, which might result in a ClientHello which is just unacceptable to the server (more analysis needed).

Appendix D. Acknowledgments

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 ESNI key. Richard Barnes, Christian Huitema, Patrick McManus, Matthew Prince, Nick Sullivan, Martin Thomson, and Chris Wood also provided important ideas.

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
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

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 <http://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 April 16, 2019.

Copyright Notice

Copyright (c) 2018 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents

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

Table of Contents

1. Introduction	2
1.1. Requirements Language	2
2. Use Cases	3
3. Ticket Requests	3
4. IANA Considerations	4
5. Security Considerations	4
6. Acknowledgments	4
7. Normative References	5
Authors' Addresses	5

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:

- o 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.
- o 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.
- o 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.)
- o 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