Network Working Group R. Housley
Internet-Draft Vigil Security
Intended status: Standards Track March 1, 2018
Expires: September 2, 2018

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

draft-housley-tls-tls13-cert-with-extern-psk-00

Abstract

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

Status of This Memo

Copyright Notice

1. Introduction

The TLS 1.3 [I-D.ietf-tls-tls13] handshake protocol provides two mutually exclusive forms of server authentication. First, the server can be authenticated by providing a signature certificate and demonstrating that it possesses the corresponding private key by creating a valid digital signature. 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.

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.

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. Extension Overview

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

The client includes the "tls_cert_with_extern_psk" extension in the ClientHello message. The "tls_cert_with_extern_psk" extension MUST accompanied by the "key_share", "psk_key_exchange_modes", and "pre_shared_key" extensions. The "pre_shared_key" extension MUST be the last extension in the ClientHello message, and it provides a list of external PSK identifiers that the client is willing to use with this server. Since tls_cert_with_extern_psk" extension is intended to be used only with initial handshakes, it MUST NOT be sent alongside the "early_data" extension. These extension are all described in Section 4.2 of [I-D.ietf-tls-tls13].

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

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

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

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

4. 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 [I-D.ietf-tls-tls13], 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 [I-D.ietf-tls-tls13]; it is repeated here for convenience.

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

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

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

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

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

To use an external PSK with certificates, clients MUST provide the "tls_cert_with_extern_psk" extension, and it MUST be accompanied by the "key_share", "psk_key_exchange_modes", and "pre_shared_key" extensions in the ClientHello. If clients offer a "tls_cert_with_extern_psk" extension without all of these other extensions, servers MUST abort the handshake. The client MAY also find it useful to include the the "supported_groups" extension. Note that Section 4.2 of [I-D.ietf-tls-tls13] 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 [I-D.ietf-tls-tls13].

The "psk_key_exchange_modes" extension is defined in Section 4.2.9 of [I-D.ietf-tls-tls13]. 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 [I-D.ietf-tls-tls13].

The "pre_shared_key" extension is defined in Section 4.2.11 of [I-D.ietf-tls-tls13]. the syntax is repeated below for convenience. All of the listed PSKs MUST be external PSKs.

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

opaque PskBinderEntry<32..255>;

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

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

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

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

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

The binders are a series of HMAC values, one for each external PSK offered by the client, in the same order as the identities list. The HMAC value is computed using the binder_key, which is derived from the external PSK, and a partial transcript of the current handshake. Generation of the binder_key from the external PSK is described in Section 7.1 of [I-D.ietf-tls-tls13]. 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 [I-D.ietf-tls-tls13].

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

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

Section 7.1 of [I-D.ietf-tls-tls13] 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.

5. 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.

6. Security Considerations

The Security Considerations in [I-D.ietf-tls-tls13] remain relevant.

TLS 1.3 [I-D.ietf-tls-tls13] 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.

7. Acknowledgments

Many thanks to Peter Yee for his review and comments on an early draft of this document.

8. References

8.1. Normative References

[I-D.ietf-tls-tls13]
Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", draft-ietf-tls-tls13-24 (work in progress), February 2018.

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

[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>.

8.2. 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-03 (work in progress), February 2018.

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

[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>.

Author’s Address

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

Email: housley@vigilsec.com

Internet Engineering Task Force J. Hoyland, Ed.
Internet-Draft Royal Holloway, University of London
Intended status: Standards Track June 25, 2018
Expires: December 27, 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 constitues an attestation of the authenticity of the referenced EA.

Status of This Memo

Copyright Notice

Table of Contents

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

Copyright Notice

Table of Contents

1. Requirements Notation

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

2. Introduction

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

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

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

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

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

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

3. DNSSEC Authentication Chain Extension

3.1. Protocol, TLS 1.2

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

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

3.2. Protocol, TLS 1.3

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

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

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

3.3. Raw Public Keys

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

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

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

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

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

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

3.4. DNSSEC Authentication Chain Data

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

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

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

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

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

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

where RR(i) denotes the ith RR.
```

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

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

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

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

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

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

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

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

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

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

4. Construction of Serialized Authentication Chains

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

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

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

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

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

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

5. Caching and Regeneration of the Authentication Chain

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

6. Verification

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

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

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

7. Trust Anchor Maintenance

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

8. Mandating use of this extension

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

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

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

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

9. DANE and Traditional PKIX Interoperation

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

10. Security Considerations

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

11. IANA Considerations

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

12. Acknowledgments

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

13. References

13.1. Normative References

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

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

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

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

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

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

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

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

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

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

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

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

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

13.2. Informative References

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

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

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

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

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

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

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

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

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

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

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

Appendix A. Test vectors

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


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


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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Authors' Addresses

Melinda Shore
Fastly

EMail: mshore@fastly.com

Richard Barnes
Mozilla

EMail: rlb@ipv.sx

Shumon Huque
Salesforce

EMail: shuque@gmail.com

Willem Toorop
NLnet Labs

EMail: willem@nlnetlabs.nl

TLS
Internet-Draft
Updates: 6347 (if approved)
Intended status: Standards Track
Expires: January 3, 2019
E. Rescorla, Ed.
RTFM, Inc.
H. Tschofenig, Ed.
Arm Limited
T. Fossati
Nokia
T. Gondrom
Huawei
July 02, 2018

The Datagram Transport Layer Security (DTLS) Connection Identifier
draft-ietf-tls-dtls-connection-id-01

Abstract

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

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

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

The Datagram Transport Layer Security (DTLS) 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 longer period of time since the established keys will only need to be updated once the key lifetime expires.

In the current version of DTLS, 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 to add a connection ID to the DTLS record layer. The presence of the connection ID is negotiated via a DTLS extension.

2. Conventions and Terminology

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

The reader is assumed to be familiar with DTLS [RFC6347].

3. The "connection_id" Extension

This document defines a new extension type (connection_id(TBD)), which is used in ClientHello and ServerHello messages.

The extension type is specified as follows.

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

The extension_data field of this extension, when included in the ClientHello, MUST contain the CID structure, which carries the CID which the client wishes the server to use when sending messages towards it. A zero-length value indicates that the client is prepared to send with a connection ID but does not wish the server to use one when sending (alternately, this can be interpreted as the client wishes the server to use a zero-length CID; the result is the same).

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

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

When a session is resumed, the "connection_id" extension is negotiated afresh, not retained from previous connections in the session.

This is effectively the simplest possible design that will work. Previous design ideas for using cryptographically generated session ids, either using hash chains or public key encryption, were dismissed due to their inefficient designs. Note that a client always has the chance to fall back to a full handshake or more precisely to a handshake that uses session resumption.

Because each party sends in the extension_data the value that it will receive as a connection identifier in encrypted records, it is possible for an endpoint to use a globally constant length for such connection identifiers. This can in turn ease parsing and connection lookup, for example by having the length in question be a compile-time constant. Implementations which want to use variable-length CIDs are responsible for constructing the CID in such a way that its length can be determined on reception. Note that such implementations must still be able to send other length connection identifiers to other parties.

In DTLS, connection ids are exchanged at the beginning of the DTLS session only. There is no dedicated "connection id update" message that allows new connection ids to be established mid-session, because DTLS in general does not allow TLS 1.3-style post-handshake messages that do not themselves begin other handshakes. DTLS peers switch to the new record layer format when encryption is enabled.

4. Record Layer Extensions

This extension is applicable for use with DTLS 1.2 and below. Figure 1 illustrates the record format. [I-D.ietf-tls-dtls13] specifies how to carry the CID in a DTLS 1.3 record.

```
struct {
     ContentType type;
     ProtocolVersion version;
     uint16 epoch;
     uint48 sequence_number;
     opaque cid[cid_length];               // New field
     uint16 length;
     select (CipherSpec.cipher_type) {
         case block:  GenericBlockCipher;
         case aead:   GenericAEADCipher;
     } fragment;
} DTLSCiphertext;
```

Figure 1: DTLS 1.2 Record Format with Connection ID

Note that for both record formats, it is not possible to parse the records without knowing how long the Connection ID is.

In order to allow a receiver to determine whether a record has CID or not, connections which have negotiated this extension use new record types for all protected records. Table 1 shows the record types to use:

New ContentType	Value
alert_with_cid	25
handshake_with_cid	26
application_data_with_cid	27
heartbeat_with_cid	28

Table 1

5. Record Payload Protection

The CID value, when present, is included in the MAC calculation for the DTLS record. The MAC algorithm described in Section 4.1.2.1 of [RFC6347] and Section 6.2.3.1 of [RFC5246] is extended as follows:

```
      MAC(MAC_write_key, DTLSCompressed.epoch +
                            DTLSCompressed.sequence_number +
                            DTLSCompressed.type +
                            DTLSCompressed.version +
                            connection_id + // New field
                            cid_length +        // New input
                            cid +               // New input
                            DTLSCompressed.length +
                            DTLSCompressed.fragment);
   where "+" denotes concatenation.
```

6. Examples

Figure 2 shows an example exchange where a connection id is used uni-directionally from the client to the server.

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

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


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

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

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

   Certificate                 -------->
   ClientKeyExchange
   CertificateVerify
   [ChangeCipherSpec]
   Finished
   (cid=100)
                               <--------      [ChangeCipherSpec]
                                                        Finished

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

Figure 2: Example DTLS 1.2 Exchange with Connection IDs

7. Security and Privacy Considerations

The connection id 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].

In addition, endpoints can use the connection ID to attach arbitrary metadata to each record they receive. This may be used as a mechanism to communicate per-connection to on-path observers. There is no straightforward way to address this with connection IDs that contain arbitrary values; implementations concerned about this SHOULD refuse to use connection ids.

An on-path adversary, who is able to observe 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 connection id pair (for bi-directional communication). Without multi-homing or mobility, the use of the connection id is not different to the use of the 5-tuple.

With multi-homing, an adversary is able to correlate the communication interaction over the two paths, which adds further privacy concerns. In order to prevent this, implementations SHOULD attempt to use fresh connection IDs whenever they change local addresses or ports (though this is not always possible to detect).

Importantly, the sequence number makes it possible for a passive attacker to correlate packets across CID changes. Thus, even if a client/server pair do a rehandshake to change CID, that does not provide much privacy benefit.

This document does not change the security properties of DTLS [RFC6347]. It merely provides a more robust mechanism for associating an incoming packet with a stored security context.

8. IANA Considerations

IANA is requested to allocate an entry to the existing TLS "ExtensionType Values" registry, defined in [RFC5246], for connection_id(TBD) defined in this document.

IANA is requested to allocate the following new values in the "TLS ContentType Registry":

- alert_with_cid(25)

- handshake_with_cid(26)

- application_data_with_cid(27)

- heartbeat_with_cid(28)

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>.

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

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

9.2. Informative References

[I-D.ietf-tls-dtls13] Rescorla, E., Tschofenig, H., and N. Modadugu, "The Datagram Transport Layer Security (DTLS) Protocol Version 1.3", draft-ietf-tls-dtls13-26 (work in progress), March 2018.

[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>.

9.3. URIs

[1] mailto:tls@ietf.org

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

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

Appendix A. History

RFC EDITOR: PLEASE REMOVE THE THIS SECTION

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

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

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

Appendix C. Contributors

Many people have contributed to this specification since the functionality has been highly desired by the IoT community. We would like to thank the following individuals for their contributions in earlier specifications:

* Nikos Mavrogiannopoulos
  RedHat
  nmav@redhat.com

Additionally, we would like to thank Yin Xinxing (Huawei), Tobias Gondrom (Huawei), and 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)

Finally, we want to thank the IETF TLS working group chairs, 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
Nokia

EMail: thomas.fossati@nokia.com

Tobias Gondrom
Huawei

EMail: tobias.gondrom@gondrom.org

TLS   E. Rescorla
Internet-Draft   RTFM, Inc.
Obsoletes: 6347 (if approved)   H. Tschofenig
Intended status: Standards Track   Arm Limited
Expires: January 3, 2019   N. Modadugu
                                                            Google, Inc.
                                                           July 02, 2018

The Datagram Transport Layer Security (DTLS) Protocol Version 1.3

draft-ietf-tls-dtls13-28

Abstract

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

The DTLS 1.3 protocol is intentionally based on the Transport Layer Security (TLS) 1.3 protocol and provides equivalent security guarantees. Datagram semantics of the underlying transport are preserved by the DTLS protocol.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

RFC EDITOR: PLEASE REMOVE THE FOLLOWING PARAGRAPH

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

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

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

DTLS 1.0 [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 aligning with the efforts around TLS 1.3 [I-D.ietf-tls-tls13].

Implementations that speak both DTLS 1.2 and DTLS 1.3 can interoperate with those that speak only DTLS 1.2 (using DTLS 1.2 of course), just as TLS 1.3 implementations can interoperate with TLS 1.2 (see Appendix D of [I-D.ietf-tls-tls13] for details). While backwards compatibility with DTLS 1.0 is possible the use of DTLS 1.0 is not recommended as explained in Section 3.1.2 of RFC 7525 [RFC7525].

2. Conventions and Terminology

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

The following terms are used:

- client: The endpoint initiating the DTLS connection.

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

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

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

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

- receiver: An endpoint that is receiving records.

- sender: An endpoint that is transmitting records.

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

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

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

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 re-ordered data traffic.

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

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

2. The TLS handshake is a lock-step cryptographic handshake. Messages must be transmitted and received in a defined order; any other order is an error. This is incompatible with reordering and message loss.

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

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

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

3.1. Packet Loss

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

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

                        X<-- HelloRetryRequest
                                            (lost)

[Timer Expires]

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

Figure 1: DTLS retransmission example

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

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

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

3.1.1. Reordering

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

3.1.2. Message Size

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

3.2. Replay Detection

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

4. The DTLS Record Layer

The DTLS record layer is different from the TLS 1.3 record layer.

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

2. DTLS adds an explicit 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.

3. The DTLSCiphertext structure has a variable length header.

Note that the DTLS 1.3 record layer is different from the DTLS 1.2 record layer.

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                                 // DTLS field
    uint48 sequence_number;                          // DTLS field
    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;
[[OPEN ISSUE: Should we try to find some way to render this?]]
```

Figure 2: DTLS 1.3 Record Format

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

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

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

```
 0 1 2 3 4 5 6 7
+-+-+-+-+-+-+-+-+
|0|0|1|C|L|X|X|X|
+-+-+-+-+-+-+-+-+
|Ep.| 14 bit    |   Legend:
+-+-+           |
|Sequence Number|   Ep. - Epoch
+-+-+-+-+-+-+-+-+   C   - CID present
| Connection ID |   L   - Length present
| (if any,      |   X   - Reserved
/  length as    /
|  negotiated)  |
+-+-+-+-+-+-+-+-+
| 16 bit Length |
| (if present)  |
+-+-+-+-+-+-+-+-+
```

Figure 3: DTLS 1.3 CipherText Header

Ep. The low order two bits of the epoch.

sequence number: The low order 14 bits of the record sequence number.

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

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

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

Figure 4 illustrates different record layer header types.

```
 0 1 2 3 4 5 6 7        0 1 2 3 4 5 6 7         0 1 2 3 4 5 6 7
+-+-+-+-+-+-+-+-+      +-+-+-+-+-+-+-+-+      +-+-+-+-+-+-+-+-+
| Content Type  |      |0|0|1|C|L|X|X|X|      |0|0|1|0|0|X|X|X|
+-+-+-+-+-+-+-+-+      +-+-+-+-+-+-+-+-+      +-+-+-+-+-+-+-+-+
|   16 bit      |      |E|E| 14 bit    |      |E|E| 14 bit    |
|   Version     |      +-+-+           |      +-+-+           |
+-+-+-+-+-+-+-+-+      |Sequence Number|      |Sequence Number|
|   16 bit      |      +-+-+-+-+-+-+-+-+      +-+-+-+-+-+-+-+-+
|    Epoch      |      |               |      |               |
+-+-+-+-+-+-+-+-+      / Connection ID /      |   Encrypted   |
|               |      |               |      /   Record      /
|               |      +-+-+-+-+-+-+-+-+      |               |
|   48 bit      |      |   16 bit      |      +-+-+-+-+-+-+-+-+
|Sequence Number|      |   Length      |
|               |      +-+-+-+-+-+-+-+-+       DTLSCiphertext
|               |      |               |         Structure
+-+-+-+-+-+-+-+-+      |  Encrypted    |         (minimal)
|    16 bit     |      /  Record       /
|    Length     |      |               |
+-+-+-+-+-+-+-+-+      +-+-+-+-+-+-+-+-+
|               |
|               |       DTLSCiphertext
/   Fragment    /         Structure
|               |           (full)
+-+-+-+-+-+-+-+-+

 DTLSPlaintext
   Structure
```

Figure 4: Header Examples

The length field may be omitted and therefore 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 data which is protected with one of the application_traffic_secret values, and not for messages protected with either [sender]_handshake_traffic_sercret or [sender]_early_traffic_secret values. When using an [sender]_application_traffic_secret for message protection, Implementations MAY include the length field at their discretion.

4.1. Determining the Header Format

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

- If the first byte is alert(21), handshake(22), or ack(proposed, 25), the record MUST be interpreted as a DTLSPlaintext record.

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

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

4.2. Sequence Number and Epoch

4.2.1. Processing Guidlines

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

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.

Because DTLS records may be reordered, a record from epoch 1 may be received after epoch 2 has begun. In general, implementations SHOULD discard packets from earlier epochs, but if packet loss causes noticeable problems implementations MAY choose to retain keying material from previous epochs for up to the default MSL specified for TCP [RFC0793] to allow for packet reordering. (Note that the intention here is that implementers use the current guidance from the IETF for MSL, as specified in [RFC0793] or successors not that they attempt to interrogate the MSL that the system TCP stack is using.) Until the handshake has completed, implementations MUST accept packets from the old epoch.

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 packets, 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 packets may be sent still apply, and the receiver treats the packets as if they were sent in the right order. In particular, it is still impermissible to send data prior to completion of the first handshake.

Note that some care needs to be taken during the handshake to ensure that retransmitted messages use the right epoch and keying material.

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

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

4.2.2. Reconstructing the Sequence Number and Epoch

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

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

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

4.3. Transport Layer Mapping

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

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

DTLS records, as defined in this document, 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. However, the Connection ID extension defined in [I-D.ietf-tls-dtls-connection-id] adds an association identifier to DTLS records.

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

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

4.4. PMTU Issues

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

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

- In some implementations, the application may not directly talk to the network, in which case the DTLS stack may absorb ICMP [RFC1191] "Datagram Too Big" indications or ICMPv6 [RFC4443] "Packet Too Big" indications.

- The DTLS handshake messages can exceed the PMTU.

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

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

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

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

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

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

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

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

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

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

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

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

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

4.5. Record Payload Protection

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

4.5.1. Anti-Replay

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

The received packet counter for a session MUST be initialized to zero when that session is established. For each received record, the receiver MUST verify that the record contains a sequence number that does not duplicate the sequence number of any other record received during the lifetime of the session. This SHOULD be the first check applied to a packet after it has been matched to a session, to speed up the rejection of duplicate records.

Duplicates are rejected through the use of a sliding receive window. (How the window is implemented is a local matter, but the following text describes the functionality that the implementation must exhibit.) A minimum window size of 32 MUST be supported, but a window size of 64 is preferred and SHOULD be employed as the default. Another window size (larger than the minimum) MAY be chosen by the receiver. (The receiver does not notify the sender of the window size.)

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

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

4.5.2. Handling Invalid Records

Unlike TLS, DTLS is resilient in the face of invalid records (e.g., invalid formatting, length, MAC, etc.). In general, invalid records SHOULD be silently discarded, thus preserving the association;

however, an error MAY be logged for diagnostic purposes. Implementations which choose to generate an alert instead, MUST generate error alerts to avoid attacks where the attacker repeatedly probes the implementation to see how it responds to various types of error. Note that if DTLS is run over UDP, then any implementation which does this will be extremely susceptible to denial-of-service (DoS) attacks because UDP forgery is so easy. Thus, this practice is NOT RECOMMENDED for such transports, both to increase the reliability of DTLS service and to avoid the risk of spoofing attacks sending traffic to unrelated third parties.

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

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.

With these exceptions, the DTLS message formats, flows, and logic are the same as those of TLS 1.3. DTLS implementations SHOULD not use the TLS 1.3 "compatibility mode" described in [I-D.ietf-tls-tls13], Section D.4 and DTLS servers SHOULD ignore the "session_id" value generated by the client rather than sending ChangeCipherSpec messages. Implementations MUST still ignore ChangeCipherSpec messages received during the handshake and at all other times SHOULD treat them as if they had failed to deprotect.

5.1. Denial-of-Service Countermeasures

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

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

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

In order to counter both of these attacks, DTLS borrows the stateless cookie technique used by Photuris [RFC2522] and IKE [RFC7296]. When the client sends its ClientHello message to the server, the server MAY respond with a HelloRetryRequest message. The HelloRetryRequest message, as well as the cookie extension, is defined in TLS 1.3. The HelloRetryRequest message contains a stateless cookie generated using the technique of [RFC2522]. The client MUST retransmit the ClientHello with the cookie added as an extension. The server then verifies the cookie and proceeds with the handshake only if it is valid. This mechanism forces the attacker/client to be able to receive the cookie, which makes DoS attacks with spoofed IP addresses difficult. This mechanism does not provide any defence against DoS attacks mounted from valid IP addresses.

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

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

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

                      <----- HelloRetryRequest
                              + cookie

ClientHello           ------>
 + cookie

[Rest of handshake]
```

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

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

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

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

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

When the second ClientHello is received, the server can verify that the cookie is valid and that the client can receive packets at the given IP address. 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 that legitimate clients be able to handshake through the transition (e.g., they received a cookie with Secret 1 and then sent the second ClientHello after the server has changed to Secret 2), the server can have a limited window during which it accepts both secrets. [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 those cookies that were not generated within a certain amount of time.

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

If a server receives a ClientHello with an invalid cookie, it MUST NOT respond with a HelloRetryRequest. Restarting the handshake from scratch, without a cookie, allows the client to recover from a situation where it obtained a cookie that cannot be verified by the server. As described in Section 4.1.4 of [I-D.ietf-tls-tls13], clients SHOULD also abort the handshake with an "unexpected_message" alert in response to any second HelloRetryRequest which was sent in the same connection (i.e., where the ClientHello was itself in response to a HelloRetryRequest).

5.2. DTLS Handshake Message Format

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

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

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

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

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

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

5.3. ClientHello Message

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

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

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

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

legacy_version: In previous versions of DTLS, this field was used for version negotiation and represented the highest version number supported by the client. Experience has shown that many servers do not properly implement version negotiation, leading to "version intolerance" in which the server rejects an otherwise acceptable ClientHello with a version number higher than it supports. In DTLS 1.3, the client indicates its version preferences in the "supported_versions" extension (see Section 4.2.1 of [I-D.ietf-tls-tls13]) and the legacy_version field MUST be set to {254, 253}, which was the version number for DTLS 1.2.

random: Same as for TLS 1.3.

legacy_session_id: Same as for TLS 1.3.

legacy_cookie: A DTLS 1.3-only client MUST set the legacy_cookie field to zero length.

cipher_suites: Same as for TLS 1.3.

legacy_compression_methods: Same as for TLS 1.3.

extensions: Same as for TLS 1.3.

5.4. Handshake Message Fragmentation and Reassembly

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

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

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

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

5.5. DTLS Handshake Flights

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

```
Client                                             Server

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

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


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

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


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

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

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

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

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


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

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

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

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

```
  Client                                             Server

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

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


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

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

Figure 8: Message flights for the Zero-RTT handshake

```
  Client                                             Server

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

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

Figure 9: Message flights for the new session ticket message

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

5.6. Timeout and Retransmission

5.6.1. State Machine

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

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

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

             Server read retransmit
                 Retransmit ACK
```

Figure 10: DTLS timeout and retransmission state machine

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

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

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

There are four ways to exit the WAITING state:

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

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

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

a duplicate message is the likely result of timer expiry on the peer and therefore suggests that part of one's previous flight was lost.

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

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

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

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

5.6.2. Timer Values

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

Implementations SHOULD retain the current timer value until a transmission without loss occurs, at which time the value may be reset to the initial value. After a long period of idleness, no less than 10 times the current timer value, implementations may reset the timer to the initial value. One situation where this might occur is when a rehandshake is used after substantial data transfer.

5.7. CertificateVerify and Finished Messages

CertificateVerify and Finished messages have the same format as in TLS 1.3. Hash calculations include entire handshake messages, including DTLS-specific fields: message_seq, fragment_offset, and fragment_length. However, in order to remove sensitivity to handshake message fragmentation, the CertificateVerify and the Finished messages MUST be computed as if each handshake message had been sent as a single fragment following the algorithm described in Section 4.4.3 and Section 4.4.4 of [I-D.ietf-tls-tls13], respectively.

5.8. Alert Messages

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

5.9. 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 packet 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 also use some sort of connection identifier, such as the one specified in [I-D.rescorla-tls-dtls-connection-id].

6. Example of Handshake with Timeout and Retransmission

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

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

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

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


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

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


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

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

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

        Figure 11: Example DTLS exchange illustrating message loss

6.1. Epoch Values and Rekeying

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

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

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

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

- epoch value (2) is used for messages protected using keys derived from [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 traffic_secret_0. This may include handshake messages, such as post-handshake messages (e.g., a NewSessionTicket message).

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

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

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

The traffic key calculation is described in Section 7.3 of [I-D.ietf-tls-tls13].

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

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

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

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

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

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

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

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

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

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

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

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

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

                              Some time later ...
                                (Rekeying)

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

Figure 12: Example DTLS exchange with epoch information

7. ACK Message

The ACK message is used by an endpoint to indicate handshake-containing the TLS records it has received 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 it consuming space in the handshake message sequence. Note that ACKs can still be piggybacked on the same UDP datagram as handshake records.

```
struct {
       uint64 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, in numerically increasing order. ACKs only cover the current outstanding flight (this is possible because DTLS is generally a lockstep protocol). Thus, an ACK from the server would not cover both the ClientHello and the client's Certificate. Implementations can accomplish this by clearing their ACK list upon receiving the start of the next flight.

ACK records MUST be sent with an epoch that is equal to or higher than the record which is being acknowledged. Implementations SHOULD simply use the current key.

7.1. Sending ACKs

When an implementation receives a partial flight, it SHOULD generate an ACK that covers the messages from that flight which it has received so far. Implementations have some discretion about when to generate ACKs, but it is RECOMMENDED that they do so 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. Implementations MUST NOT send ACKs for handshake messages which they discard as out-of-order, because otherwise those messages will not be retransmitted.

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

In addition, implementations MUST send ACKs upon receiving all of any flight which they do not respond to with their own messages. Specifically, this means the client's final flight of the main handshake, the server's transmission of the NewSessionTicket, and KeyUpdate messages. ACKs SHOULD NOT be sent for other complete flights because they are implicitly acknowledged by the receipt of the next flight, which generally immediately follows the flight. Each NewSessionTicket or KeyUpdate is an individual flight; in particular, a KeyUpdate sent in response to a KeyUpdate with update_requested does not implicitly acknowledge that message. Implementations MAY ACK the records corresponding to each transmission of that flight or simply ACK the most recent one.

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 ACK 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 for only some messages from a flight, an implementation SHOULD retransmit the remaining 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. As noted above, the receipt of any packet responding to a given flight MUST be taken as an implicit ACK for the entire flight.

8. Key Updates

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

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

Although KeyUpdate MUST be ACKed, it is possible for the ACK to be lost, in which case the sender of the KeyUpdate will retransmit it. Implementations MUST retain the ability to ACK the KeyUpdate for up to 2MSL. It is RECOMMENDED that they do so by retaining the pre-update keying material, but they MAY do so by responding to messages which appear to be out-of-epoch with a canned ACK message; in this case, implementations SHOULD rate limit how often they send such ACKs.

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 connection ID which it wishes the other side to use in a NewConnectionId message.

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

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

cid Indicates the CID which the sender wishes the peer to use.

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

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

```
struct {
} RequestConnectionId;
```

Endpoints SHOULD respond to RequestConnectionId by sending a NewConnectionId with usage "cid_spare" as soon as possible. Note that an endpoint MAY ignore requests, which it considers excessive (though they MUST be ACKed as usual).

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

9.1. ID Example

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

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

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

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


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

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

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

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

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

Figure 13: Example DTLS 1.3 Exchange with Connection IDs

10. Application Data Protocol

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

11. Security Considerations

Security issues are discussed primarily in [I-D.ietf-tls-tls13].

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

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

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

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

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

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

- Switching connection ID based on certain events, or even regularly, helps against tracking by onpath adversaries but the sequence numbers can still allow linkability. [[OPEN ISSUE: We

need to update the document to offer sequence number encryption.
]]

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

12. Changes to DTLS 1.2

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

- New handshake pattern, which leads to a shorter message exchange
- Support for AEAD-only ciphers
- HelloRetryRequest of TLS 1.3 used instead of HelloVerifyRequest
- More flexible ciphersuite negotiation
- New session resumption mechanism
- PSK authentication redefined
- New key derivation hierarchy utilizing a new key derivation construct
- Removed support for weaker and older cryptographic algorithms
- Improved version negotation
- Optimized record layer encoding and thereby its size
- Added connection ID functionality

13. IANA Considerations

IANA is requested to allocate a new value in the "TLS ContentType" registry for the ACK message, defined in Section 7, with content type 25.

IANA is requested to allocate two values in the "TLS Handshake Type" registry, defined in [RFC5246], for RequestConnectionId (TBD), and NewConnectionId (TBD), as defined in this document.

14. References

14.1. Normative References

[I-D.ietf-tls-tls13]
Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", draft-ietf-tls-tls13-21 (work in progress), July 2017.

[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>.

14.2. Informative References

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

[I-D.rescorla-tls-dtls-connection-id]
Rescorla, E., Tschofenig, H., Fossati, T., and T. Gondrom, "The Datagram Transport Layer Security (DTLS) Connection Identifier", draft-rescorla-tls-dtls-connection-id-02 (work in progress), November 2017.

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

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

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

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

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

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

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

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

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

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

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

14.3. URIs

[1] mailto:tls@ietf.org

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

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

Appendix A. History

RFC EDITOR: PLEASE REMOVE THE THIS SECTION

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

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

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 packets 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 B. Working Group Information

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

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

Appendix C. Contributors

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

In addition, we would like to thank:

* Ilari Liusvaara
  Independent
  ilariliusvaara@welho.com

* Martin Thomson
  Mozilla
  martin.thomson@gmail.com

* Yin Xinxing
  Huawei
  yinxinxing@huawei.com

* Thomas Fossati
  Nokia
  thomas.fossati@nokia.com

* Tobias Gondrom
  Huawei
  tobias.gondrom@gondrom.org

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 N. Sullivan
Internet-Draft Cloudflare Inc.
Intended status: Standards Track June 05, 2018
Expires: December 7, 2018

Exported Authenticators in TLS
draft-ietf-tls-exported-authenticator-07

Abstract

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

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

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

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

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

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

Post-handshake authentication is defined in TLS 1.3, but it has the disadvantage of requiring additional state to be stored in the TLS state machine and it composes poorly with multiplexed connection protocols like HTTP/2 [RFC7540]. It is also only available for client authentication. This mechanism is intended to be used as part of a replacement for post-handshake authentication in applications.

2. Conventions and Terminology

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

3. Authenticator Request

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

An authenticator request message can be constructed by either the client or the server. This authenticator request uses the CertificateRequest message structure from Section 4.3.2 of [TLS13]. This message does not include the TLS record layer and is therefore not encrypted with a handshake key.

The CertificateRequest is used to define the parameters in a request for an authenticator. The definition for TLS 1.3 is:

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

certificate_request_context: An opaque string which identifies the certificate request and which will be echoed in the authenticator message. A certificate_request_context value MUST NOT be repeated for multiple authenticator requests generated by the same party within the scope of a connection (thus preventing replay of authenticators). 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. This value is unrelated to the certificate_request_context used in post-handshake authentication and collisions do not need to be avoided.

extensions: The extensions that are allowed in this structure include the extensions defined for CertificateRequest messages defined in Section 4.2. of [TLS13] and the server_name [RFC6066] extension, which is allowed for client-generated authenticator requests.

4. Authenticator

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

An authenticator message can be constructed by either the client or the server given an established TLS connection, a certificate, and a corresponding private key. Clients MUST NOT send an authenticator without a preceding authenticator request; for servers an authenticator request is optional.
The authenticator uses the message structures from Section 4.4 of [TLS13], but different parameters. These messages do not include the TLS record layer and are therefore not encrypted with a handshake key.

4.1. Authenticator Keys

Each authenticator is computed using a Handshake Context and Finished MAC Key derived from the TLS session. These values are derived using an exporter as described in [RFC5705] (for TLS 1.2) or [TLS13] (for TLS 1.3). These values use different labels depending on the role of the sender:

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

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

The context_value used for the exporter is absent (length zero) for all four values. The length of the exported value is equal to the length of the output of the hash function selected in TLS for the pseudorandom function (PRF). Cipher suites that do not use the TLS PRF MUST define a hash function that can be used for this purpose or they cannot be used. For TLS 1.3 symmetric cipher suites, the hash algorithm used with HKDF is used.

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

4.2. Authenticator Construction

An authenticator is formed from the concatenation of TLS 1.3 [TLS13] Certificate, CertificateVerify, and Finished messages.

If an authenticator request is present, the extensions used to guide the construction of these messages are taken from the authenticator request. If there is no authenticator request, the extensions are chosen from the TLS handshake. Only servers can provide an authenticator without a corresponding request. In such cases, ClientHello extensions are used to determine permissible extensions in the Certificate message.

4.2.1. Certificate

The certificate to be used for authentication and any supporting certificates in the chain. This structure is defined in [TLS13], Section 4.4.2.

The certificate message contains an opaque string called certificate_request_context, which is extracted from the authenticator request if present. If no authenticator request is provided, the certificate_request_context can be chosen arbitrarily.

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

In addition to "signature_algorithms" and "signature_algorithms_cert", the "server_name" [RFC6066], "certificate_authorities" (Section 4.2.4. of [TLS13]), and "oid_filters" (Section 4.2.5. of [TLS13]) extensions are used to guide certificate selection. The extensions, or others that might affect certificate selection, are taken from the authenticator request if present, or the TLS handshake if not.

Alternative certificate formats such as [RFC7250] Raw Public Keys are not supported in this version of the specification.

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

4.2.2. CertificateVerify

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

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

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

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

If an authenticator request is present, the signature algorithm MUST be chosen from one of the signature schemes in the authenticator request. Otherwise, the signature algorithm used should be chosen from the "signature_algorithms" sent by the peer in the TLS handshake.

The signature is computed using the over the concatenation of:

- A string that consists of octet 32 (0x20) repeated 64 times
- The context string "Exported Authenticator" (which is not NULL-terminated)

- A single 0 byte which serves as the separator

- The hashed authenticator transcript

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

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

Where Hash is the hash function negotiated by TLS. If the authenticator request is not present, it is omitted from this construction (that is, it is zero length).

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

4.2.3. Finished

A HMAC [HMAC] over the hashed authenticator transcript, which is the concatenated Handshake Context, authenticator request (if present), Certificate, and CertificateVerify:

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

The HMAC is computed using the same hash function using the Finished MAC Key as a key.

4.2.4. 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
```

A given authenticator can be validated by checking the validity of the CertificateVerify message given the authenticator request (if used) and recomputing the Finished message to see if it matches.

5. Empty Authenticator

If, given an authenticator request, the endpoint does have an appropriate certificate or does not want to return one, it constructs an authenticated refusal called an empty authenticator. This is an HMAC over the hashed authenticator transcript with a Certificate message containing no CertificateEntries and the CertificateVerify message omitted:

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

The HMAC is computed using the same hash function using the Finished MAC Key as a key.

6. API considerations

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

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

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

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

6.1. The "request" API

The "request" API takes as input:

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

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

6.2. The "get context" API

The "get context" API takes as input:

- o authenticator

It returns the certificate_request_context.

6.3. The "authenticate" API

The "authenticate" takes as input:

- o a set of certificate chains and associated extensions (OCSP, SCT, etc.)
- o a signer (either the private key associated with the certificate, or interface to perform private key operation) for each chain
- o an optional authenticator request or certificate_request_context (from 0 to 255 bytes)

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

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.

6.4. The "validate" API

The "validate" API takes as input:

- o an optional authenticator request
- o an authenticator

It returns the certificate chain and extensions and a status to indicate whether the authenticator is valid or not. If the authenticator was empty - that is, it did not contain a certificate - the certificate chain will contain no certificates.

7. IANA Considerations

This document has no IANA actions.

8. Security Considerations

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

Authenticators are independent and unidirectional. There is no explicit state change inside TLS when an authenticator is either created or validated.

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

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

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

9. Acknowledgements

Comments on this proposal were provided by Martin Thomson. Suggestions for Section 8 were provided by Karthikeyan Bhargavan.

10. References

10.1. Normative References

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

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

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

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

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

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

[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>.

[TLS13] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", draft-ietf-tls-tls13-28 (work in progress), March 2018.

10.2. Informative References

[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>.

[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: January 3, 2019

R. Barnes
Mozilla
S. Iyengar
Facebook
N. Sullivan
Cloudflare
E. Rescorla
RTFM, Inc.
July 02, 2018

Delegated Credentials for TLS
draft-ietf-tls-subcerts-01

Abstract

The organizational separation between the operator of a TLS server and the certificate authority that provides it credentials can cause problems, for example when it comes to reducing the lifetime of certificates or supporting new cryptographic algorithms. This document describes a mechanism to allow TLS server operators to create their own credential delegations without breaking compatibility with clients that do not support this specification.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

Typically, a TLS server uses a certificate provided by some entity other than the operator of the server (a "Certification Authority" or CA) [RFC5246] [RFC5280]. This organizational separation makes the TLS server operator dependent on the CA for some aspects of its operations, for example:

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

- o The server operator can only use TLS authentication schemes for which the CA will issue credentials.

These dependencies cause problems in practice. Server operators often want to create short-lived certificates for servers in low-trust zones such as CDNs or remote data centers. This allows server operators to limit the exposure of keys in cases that they do not realize a compromise has occurred. The risk inherent in cross-organizational transactions makes it operationally infeasible to rely on an external CA for such short-lived credentials. In OCSP stapling, if an operator chooses to talk frequently to the CA to obtain stapled responses, then failure to fetch an OCSP stapled response results only in degraded performance. On the other hand, failure to fetch a potentially large number of short lived certificates would result in the service not being available which creates greater operational risk.

To remove these dependencies, this document proposes a limited delegation mechanism that allows a TLS server operator to issue its own credentials within the scope of a certificate issued by an external CA. Because the above problems do not relate to the CAs inherent function of validating possession of names, it is safe to make such delegations as long as they only enable the recipient of the delegation to speak for names that the CA has authorized. For clarity, we will refer to the certificate issued by the CA as a "certificate" and the one issued by the operator as a "delegated credential".

2. Solution Overview

A delegated credential is a digitally signed data structure with the following semantic fields:

- A validity interval

- A public key (with its associated algorithm)

The signature on the credential indicates a delegation from the certificate that is issued to the TLS server operator. The secret key used to sign a credential is presumed to be one whose corresponding public key is contained in an X.509 certificate that associates one or more names to the credential.

A TLS handshake that uses credentials differs from a normal handshake in a few important ways:

- The client provides an extension in its ClientHello that indicates support for this mechanism.

- The server provides both the certificate chain terminating in its certificate as well as the credential.

- The client uses information in the server’s certificate to verify the signature on the credential and verify that the server is asserting an expected identity.

- o The client uses the public key in the credential as the server's working key for the TLS handshake.

Delegated credentials can be used either in TLS 1.3 or TLS 1.2. Differences between the use of delegated credentials in the protocols are explicitly stated.

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 prevent this type of cross-protocol attack, we define a new DelegationUsage extension to X.509 that permits use of delegated credentials. Clients MUST NOT accept delegated credentials associated with certificates without this extension.

Credentials allow the server to terminate 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 the exposure of a delegation credential compromise, servers MUST NOT issue credentials with a validity period longer than 7 days. Clients MUST NOT accept credentials with longer validity periods.

2.1. Rationale

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

- o There is no change needed to certificate validation at the PKI layer.

- o X.509 semantics are very rich. This can cause unintended consequences if a service owner creates a proxy cert where the properties differ from the leaf certificate.

- o Delegated credentials have very restricted semantics which should not conflict with X.509 semantics.

- o Proxy certificates rely on the certificate path building process to establish a binding between the proxy certificate and the server certificate. Since the cert 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.

   o Delegated credentials are bound to specific versions of TLS. This prevents them from being used for other protocols if a service owner allows multiple versions of TLS.

2.2. 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., LURK [I-D.mglt-lurk-tls-requirements]). 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 TLS 1.3 [I-D.ietf-tls-tls13].

LURK:

```
Client            Front-End            Back-End
  |----ClientHello--->|                    |
  |<---ServerHello----|                    |
  |<---Certificate----|                    |
  |                   |<-------LURK------->|
  |<---CertVerify-----|                    |
  |        ...        |                    |
```

Delegated credentials:

```
Client            Front-End            Back-End
  |                   |<--Cred Provision-->|
  |----ClientHello--->|                    |
  |<---ServerHello----|                    |
  |<---Certificate----|                    |
  |<---CertVerify-----|                    |
```

These two mechanisms can be complementary. A server could use credentials for clients that support them, while using LURK to support legacy clients.

It is possible to address the short-lived certificate concerns above by automating certificate issuance, e.g., with ACME [I-D.ietf-acme-acme]. In addition to requiring frequent operationally-critical interactions with an external party, this makes the server operator dependent on the CA’s willingness to issue certificates with sufficiently short lifetimes. It also fails to address the issues with algorithm support. Nonetheless, existing automated issuance APIs like ACME may be useful for provisioning credentials, within an operator network.

3. Client and Server behavior

This document defines the following extension code point.

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

A client which supports this document SHALL send an empty "delegated_credential" extension in its ClientHello.

If the extension is present, the server MAY send a DelegatedCredential extension. If the extension is not present, the server MUST NOT send a credential. A credential MUST NOT be provided unless a Certificate message is also sent.

When negotiating TLS 1.3, and using Delegated credentials, the server MUST send the DelegatedCredential as an extension in the CertificateEntry of its end-entity certificate. When negotiating TLS 1.2, the DelegatedCredential MUST be sent as an extension in the ServerHello.

The DelegatedCredential contains a signature from the public key in the end-entity certificate using a signature algorithm advertised by the client in the "signature_algorithms" extension. Additionally, the credential’s public key MUST be of a type that enables at least one of the supported signature algorithms. A delegated credential MUST NOT be negotiated by the server if its signature is not compatible with any of the supported signature algorithms or the credential’s public key is not usable with the supported signature algorithms of the client, even if the client advertises support for delegated credentials.

On receiving a credential and a certificate chain, the client validates the certificate chain and matches the end-entity certificate to the server’s expected identity following its normal procedures. It then takes the following steps:

- Verify that the current time is within the validity interval of the credential and that the credential’s time to live is no more than 7 days.

- o Verify that the certificate has the DelegationUsage extension, which permits the use of Delegated credentials.

- o Use the public key in the server's end-entity certificate to verify the signature on the credential.

If one or more of these checks fail, then the delegated credential is deemed invalid. Clients that receive invalid delegated credentials MUST terminate the connection with an "illegal_parameter" alert. If successful, the client uses the public key in the credential to verify a signature provided in the handshake: in particular, the CertificateVerify message in TLS 1.3 and the ServerKeyExchange in 1.2.

4. Delegated Credentials

While X.509 forbids end-entity certificates from being used as issuers for other certificates, it is perfectly fine to use them to issue other signed objects as long as the certificate contains the digitalSignature key usage (RFC5280 section 4.2.1.3). We define a new signed object format that would encode only the semantics that are needed for this application.

```
struct {
  uint32 valid_time;
  opaque public_key<0..2^16-1>;
} Credential;

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

valid_time: Relative time in seconds from the beginning of the certificate's notBefore value after which the delegated credential is no longer valid.

public_key: The delegated credential's public key, which is an encoded SubjectPublicKeyInfo [RFC5280].

scheme: The signature algorithm used to sign the delegated credential.

signature: The signature over the credential with the end-entity certificate's public key, using the scheme.

The DelegatedCredential structure is similar to the CertificateVerify structure in TLS 1.3. Since the SignatureScheme is defined in TLS 1.3, TLS 1.2 clients should translate the scheme into an appropriate group and signature algorithm to perform validation.

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".

3. A single 0 byte which serves as the separator.

4. Big endian serialized 2 bytes ProtocolVersion of the negotiated TLS version, defined by TLS.

5. DER encoded X.509 certificate used to sign the DelegatedCredential.

6. Big endian serialized 2 byte SignatureScheme scheme.

7. The Credential structure.

This signature has a few desirable properties:

- It is bound to the certificate that signed it.

- It is bound to the protocol version that is negotiated. This is intended to avoid cross-protocol attacks with signing oracles.

The code changes to create and verify delegated credentials would be localized to the TLS stack, which has the advantage of avoiding changes to security-critical and often delicate PKI code (though of course moves that complexity to the TLS stack).

4.1. Certificate Requirements

We define a new X.509 extension, DelegationUsage to be used in the certificate when the certificate permits the usage of delegated credentials. When this extension is not present the client MUST not accept a delegated credential even if it is negotiated by the server. When it is present, the client MUST follow the validation procedure.

```
id-ce-delegationUsage OBJECT IDENTIFIER ::= { TBD }

DelegationUsage ::= BIT STRING { allowed (0) }
```

Conforming CAs MUST mark this extension as non-critical. This allows the certificate to be used by service owners for clients that do not support certificate delegation as well and not need to obtain two certificates.

5. IANA Considerations

TBD

6. Security Considerations

6.1. Security of delegated private key

Delegated credentials limit the exposure of the TLS private key by limiting its validity. An attacker who compromises the private key of a delegated credential can act as a man in the middle until the delegate 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 is meant to be used between parties that have some trust relationship with each other. The secrecy of the delegated private key is thus important and several access control mechanisms SHOULD be used to protect it such as file system controls, physical security or hardware security modules.

6.2. 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 would revoke the credential. Revocation of the long term private key that signs the delegated credential also implictly revokes the delegated credential.

6.3. Privacy considerations

Delegated credentials can be valid for 7 days and it is much easier for a service to create delegated credential 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. Acknowledgements

Thanks to Kyle Nekritz, Anirudh Ramachandran, Benjamin Kaduk, Kazuho Oku, Daniel Kahn Gillmor for their discussions, ideas, and bugs they’ve found.

8. References

8.1. Normative References

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

[RFC5280] Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R., and W. Polk, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", RFC 5280, DOI 10.17487/RFC5280, May 2008, <https://www.rfc-editor.org/info/rfc5280>.

8.2. Informative References

[I-D.ietf-acme-acme]
Barnes, R., Hoffman-Andrews, J., McCarney, D., and J. Kasten, "Automatic Certificate Management Environment (ACME)", draft-ietf-acme-acme-12 (work in progress), April 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.

[I-D.mglt-lurk-tls-requirements]
Migault, D. and K. Ma, "Authentication Model and Security Requirements for the TLS/DTLS Content Provider Edge Server Split Use Case", draft-mglt-lurk-tls-requirements-00 (work in progress), January 2016.

[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/info/rfc3820>.

[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.

Authors' Addresses

Richard Barnes
Mozilla

Email: rlb@ipv.sx

Subodh Iyengar
Facebook

Email: subodh@fb.com

Nick Sullivan
Cloudflare

Email: nick@cloudflare.com

Eric Rescorla
RTFM, Inc.

Email: ekr@rtfm.com

Internet Engineering Task Force
Internet-Draft
Updates: [[List TBD]] (if approved)
Intended status: Standards Track
Expires: December 20, 2018

K. Moriarty
Dell EMC
S. Farrell
Trinity College Dublin

June 18, 2018

Deprecating TLSv1.0 and TLSv1.1
draft-moriarty-tls-oldversions-diediedie-00

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 profiiles 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

Table of Contents

1. Introduction

[[Text in double-square brackets, like this, is commentary intended to be fixed as the draft evolves. You're already seen that we need to figure out the list of RFCs that this'd update in the abstract.]]

Transport Layer Security (TLS) versions 1.0 [RFC2246] and 1.1 [RFC4346] were superceded by TLSv1.2 [RFC5246] in 2008, which has now itself been superceded by TLSv1.3 [I-D.ietf-tls-tls13]. It is therefore timely to further deprecate these old versions. The expection 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
- 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 is actively following guidance provided by NIST and the PCI Council deprecating 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] and 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.

They have provided the following rationale.

2.1. NIST 800-52r2

The following text is copied with permission from NIST SP800-52r2 [NIST800-52r2] section 1.2 History of TLS.

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.

3. Usage and Support

[[This section can be removed upon publication.]]

Usage statistics for TLSv1.0 and TLSv1.1 vary slightly, but are in general very low already and soon to decline further with the impending PCI deadline to migrate off of TLSv1.0 by June 30, 2018. As of January 2018, Stackexchange [StackExchange] quoted 4 percent of browsers using TLSv1.0.

The Alexa Top 1 Million Analysis [Alexa] from February 2018 shows that for the sites surveyed, the vast majority support TLSv1.2 (98.9 percent), with a mere 0.8 percent using TLSv1.0 and an even smaller percentage using TLSv1.1.

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

- o 3GPP 5G
- o [[Numerous web sites...]]
- o CloudFare [CloudFlare]
- o Amazon Elastic Load Balancing [Amazon]
- o GitHub [GIT]

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 or will be by July 2018 from the following standards, products, and services:

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

4. Do Not Use TLSv1.0

TLSv1.0 MUST NOT be used. Negotiation of TLSv1.0 from any version of TLS MUST NOT be permitted.

Any version of TLS is more secure then TLSv1.0 and can be configured to prevent interception, though 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.

5. Do Not Use TLSv1.1

TLSv1.1 MUST NOT be used. Negotiation of TLSv1.1 from any version of TLS MUST NOT be permitted.

Pragmatically, clients MUST NOT send a ClientHello with ClientHello.client_version set to {03,02}. Similarly, servers MUST NOT send a ServerHello with ServerHello.server_version set to {03,02}. Any party receiving a Hello message with the protocol version set to {03,02} MUST respond with a "protocol_version" alert message and close the connection.

Any newer version of TLS is more secure then TLSv1.1 and can be configured to prevent interception, though the highest version available is preferable. Support for TLSv1.1 is dwindling in libraries and will impact security going forward if mitagations for attacks cannot be easily addressed and supported in older libraries.

Historically, TLS specifications were not clear on what the record layer version number (TLSPlaintext.version) could contain when sending ClientHello. Appendix E of [RFC5246] notes that TLSPlaintext.version could be selected to maximize interoperability, though no definitive value is identified as ideal. That guidance is still applicable; therefore, TLS servers MUST accept any value {03,XX} (including {03,00}) as the record layer version number for ClientHello, but they MUST NOT negotiate TLSv1.1.

6. Updates to RFC7525

[[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:

- 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.

- 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:

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

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

7. 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.

8. Acknowledgements

Thank you to those that reviewed and improved this document, including Yoav Nir, Russ Housley, and David Black.

9. IANA Considerations

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

10. Contributors

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>.

[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>.

11.2. Informative References

[Alexa] Will be deleted before publication, "The Alexa Top 1 Million Analysis https://scotthelme.co.uk/alexa-top-1-million-analysis-february-2018/", 2018.

[Amazon] CloudFlare, "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.

[CloudFlare] CloudFlare, "CloudFlare Deprecated TLSv1.0 and TLSv1.1 https://blog.cloudflare.com/deprecating-old-tls-versions-on-cloudflare-dashboard-and-api/", 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.

[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.

[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>.

[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>.

[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.

Appendix A. Change Log

Note to RFC Editor: if this document does not obsolete an existing RFC, please remove this appendix before publication as an RFC.

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

Copyright Notice

Table of Contents

1. Introduction

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

Although TLS 1.3 [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

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

Figure 1: Shared Mode Topology

In Shared Mode, the provider is the origin server for all the domains whose DNS records point to it and clients form a TLS connection directly to that provider, which has access to the plaintext of the connection.

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

Figure 2: Split Mode Topology

In Split Mode, the provider is _not_ the origin server for private domains. Rather the DNS records for private domains point to the provider, 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:

- The client MUST only provide one KeyShareEntry

- 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:

- If it is unable to negotiate TLS 1.3 or greater, it MUST abort the connection with a "handshake_failure" alert.
- 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.]]
- 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.
- 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 servers 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 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 RECOMMEMDED 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:

- 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.

- 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.

- 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 C. Wood
Internet-Draft T. Pauly
Intended status: Informational D. Schinazi
Expires: October 14, 2018 Apple Inc.
April 12, 2018

TLS Ticket Requests
draft-wood-tls-ticketrequests-00

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 request tickets as needed during a connection.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

As per [RFC5077], and as described in [I-D.ietf-tls-tls13], 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 mechanism to request tickets on demand is desirable.

This document specifies a new TLS post-handshake message - TicketRequest - that may be used to request tickets via NewSessionTicket messages in TLS 1.3. Ticket requests may carry optional application-specific contexts to define the ways in which tickets may be used. NewSessionTicket responses reciprocate this application context in an extension.

1.1. Requirements Language

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

2. Use Cases

The ability to request one or more tickets is useful for a variety of purposes:

- Parallel HTTP connections: To minimize ticket reuse while still improving performance, it may be useful to use multiple, distinct tickets when opening parallel connections. Clients must therefore bound the number of parallel connections they initiate by the number of tickets in their possession, or risk ticket re-use.

- Connection racing: Happy Eyeballs V2 [RFC8305] describes techniques for performing connection racing. The Transport Services Architecture implementation from [I-D.brunstrom-taps-impl] also describes how connections may race across interfaces and address families. In cases where clients have early data to send and want to minimize or avoid ticket re-use, unique tickets for each unique connection attempt are useful. Moreover, as some servers may implement single-use tickets (and even session ticket encryption keys), distinct tickets will be needed to prevent premature ticket invalidation by racing.

- Connection priming: In some systems, connections may be primed or bootstrapped by a centralized service or daemon for faster connection establishment. Requesting tickets on demand allows such services to vend tickets to clients to use for accelerated handshakes with early data. (Note that if early data is not needed by these connections, this method SHOULD NOT be used. Fresh handshakes SHOULD be performed instead.)

- Less ticket waste: Currently, TLS servers use application-specific, and often implementation-specific, logic to determine how many tickets to issue. By moving the burden of ticket count to clients, servers do not generate wasteful tickets for clients.

3. Ticket Requests

TLS tickets may be requested via a TicketRequest post-handshake message, ticket_request(TBD). Its structure is shown below.

```
struct {
    opaque identifier<0..255>;
    opaque context<0..2^16-1>;
} TicketRequest;
```

- identifier: A unique value for this ticket request. Clients SHOULD fill this in with a monotonically increasing counter.

o context: An opaque context to be used when generating the ticket request. Clients and servers may use this context to implement or exchange data to be included in the ticket computation. Clients SHOULD make this field empty if it is not needed.

Upon receipt of a TicketRequest message, servers MAY reply with a NewSessionTicket message, as defined in [I-D.ietf-tls-tls13]. The latter message MUST carry two extensions, ticket_identifer and ticket_context, defined below.

```
enum {
    ...
    ticket_identifier(TBD),
    ticket_context(TBD+1),
    (65535)
} ExtensionType;
```

The value of ticket_identifier MUST match that of the corresponding TicketRequest identifier field. The value of ticket_context MAY be used by servers to convey ticket context to clients. Its value MUST be empty if the corresponding TicketRequest context field is empty.

Servers SHOULD place a limit on the number of tickets they are willing to vend to clients. Servers MUST NOT send more than 255 tickets to clients, as this is the limit imposed by the request and response identifier size. Servers SHOULD NOT send unsolicited NewSessionTickets to clients that express support for TicketRequests.

When operated over an unreliable transport, e.g., DTLS, servers may not always be able to identify retransmissions of ticket requests. In this case, when a server S receives a TicketRequest with new identifier N it MUST generate a new ticket and SHOULD cache it locally for some period of time T. If S receives a TicketRequest with identifier N within time period T, S SHOULD reply with the same ticket previously generated (and cached). If S receives a TicketRequest with identifier N outside time period T, S SHOULD reply with an empty NewSessionTicket, i.e., a NewSessionTicket with extension ticket_identifier carrying N, appropriate ticket_context extension, and empty ticket field.

4. Negotiation

Clients negotiate use of ticket requests via a new ExtensionType, ticket_request(TBD). The extension_data for this extension MUST be empty, i.e., have length of 0. Servers that support ticket requests MAY echo this extension in the EncryptedExtensions. Clients MUST NOT send ticket requests to servers that do not signal support for this message. If absent from a ClientHello, servers MUST NOT generate responses to TicketRequests issued by the client.

5. IANA Considerations

((TODO: codepoint for post-handshake message type and extensions))

6. Security Considerations

Ticket re-use is a security and privacy concern. Moreover, pre-fetching 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 pre-fetched tickets after some reasonable amount of time that mimics the ticket rotation period.

7. Acknowledgments

The authors would like to thank Eric Rescorla, Martin Thomson, and Nick Sullivan for discussions on earlier versions of this draft.

8. Normative References

[I-D.brunstrom-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-brunstrom-taps-impl-00 (work in progress), March 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.

[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>.

Authors' Addresses

Christopher A. Wood
Apple Inc.
One Apple Park Way
Cupertino, California 95014
United States of America

Email: cawood@apple.com

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