Fake Server Name Indication
draft-belyavskiy-fakesni-02

Abstract

The document provides a specification of the Fake Server Name Indication. Being implemented, the Fake SNI specification provides a way to cheat the monitoring solutions without providing any additional information to external observers.

Status of This Memo

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

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

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

This Internet-Draft will expire on September 7, 2019.

Copyright Notice

Copyright (c) 2019 IETF Trust and the persons identified as the document authors. All rights reserved.

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

Many DPI solutions use SNI information as a criterion to filter connection to various sites. Though Encrypted SNI makes impossible to read the SNI value, there is information [1] that absence of SNI looks suspicious itself and all communications are blocked.

This specification introduces a way to provide a value of SNI treated by TLS server as an alias to one of the names known by server but not matching the possibly suspicious hostname.

This specification does not save from DPI solutions but it provides one more loophole to cheat them.

2. Fake SNI design goals

The solution specified in this document is inspired by the design of Encrypted SNI. It is fully-compatible with current TLS specifications. As it does not make much sense to use it with TLS 1.0-1.2, where the original host name will be provided unencrypted in the certificate, in case of TLS 1.3 the certificate is delivered encrypted.

The provider publishes a name matching the target name to be provided in the clear text. This document defines a publication mechanism using DNS, but other mechanisms are also possible.

When a client wants to establish a TLS connection to a domain served by a Fake SNI-supporting provider, it replaces the value in "server_name" extension in the ClientHello with the value obtained by transport. The provider can then find out the desired name from its configuration and either establish the connection with the desired host or reject it.

3. Definitions

Original name - the hostname of service that is subject to hide.

Fake name - the hostname specified by server and sent by client to indicate intention to connect to host with original name.

4. Fake SNI indication

Fake SNI information is published in DNS via TXT RR. For example, the Fake SNI record for domain example.com may look like

_fakesni.example.com. 60S IN TXT "myfakerecord.com IP"
where IP address may be omitted. If present, it MUST match an IP address specified in A/AAAA record for the domain. Specifying IP address for a specific fake name may help in case when a service is hosted using more than one CDN.

The fake name specified in the Fake SNI RR MUST identify the original hostname it is valid for. Fake names for different hosts on the same IP address MUST be different to distinguish the original names.

5. Server behaviour

On receiving the value of known Fake SNI in the TLS ClientHello server MUST return the certificate matching the original hostname. Otherwise server SHOULD abort the connection.

6. Client behaviour

Client MAY use the Fake SNI record as fallback if connecting using ESNI is blocked. In this case client initiates normal TLS connection specifying the value from Fake SNI record in the server_name extension. If the certificate received from server does not match the original hostname, the client MUST abort the connection. Otherwise the client MUST follow the normal process of TLS handshake.

7. Operational considerations

Depending on the DPI modus operandi it may make sense to provide a valid fake name (e.g. from deep-level subdomain) resolving to the same IPs as original hostname does. If DPI tries to resolve the fake name, such behaviour will make distinguishing between real and fake names difficult.

8. Security considerations

As Fake SNI can be used in TLS 1.2, it does not provide any problems to DPI because in this case the original hostname is available in clear text in server certificate. TLS 1.3 encrypts the Certificate message, so it is RECOMMENDED to use Fake SNI together with TLS 1.3. To strengthen the protection, it’s recommended to obtain _fakesni RR via DoT or DoH.

As DPI solutions are able to obtain the DNS _fakesni records as legitimate clients do, it is RECOMMENDED to set reasonable TTL values for the _fakesni records. Also it is RECOMMENDED to use such values of fake names that are syntactically correct domain names. Otherwise DPI can recognise the fake names as fake ones.
9. Current version of the specification

The current version of the specification is available at GitHub repository [2].

10. References

10.1. URIs

[1] https://mailarchive.ietf.org/arch/msg/tls/WiT3oEh6PO96m90z28BNMp0YgGs


Author’s Address

Dmitry Belyavskiy
Cryptocom LTD
Kedrova st, 14/2
Moscow 117218
RU

Email: beldmit@gmail.com
Abstract

In TLS handshakes, certificate chains often take up the majority of the bytes transmitted.

This document describes how certificate chains can be compressed to reduce the amount of data transmitted and avoid some round trips.

Status of This Memo

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

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

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

This Internet-Draft will expire on October 7, 2019.

Copyright Notice

Copyright (c) 2019 IETF Trust and the persons identified as the document authors. All rights reserved.

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

In order to reduce latency and improve performance it can be useful to reduce the amount of data exchanged during a TLS handshake.

[RFC7924] describes a mechanism that allows a client and a server to avoid transmitting certificates already shared in an earlier handshake, but it doesn’t help when the client connects to a server for the first time and doesn’t already have knowledge of the server’s certificate chain.

This document describes a mechanism that would allow certificates to be compressed during full handshakes.

2. Notational Conventions

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. Negotiating Certificate Compression

This extension is only supported with TLS 1.3 and newer; if TLS 1.2 or earlier is negotiated, the peers MUST ignore this extension.

This document defines a new extension type (compress_certificate(27)), which can be used to signal the supported
compression formats for the Certificate message to the peer. Whenever it is sent by the client as a ClientHello message extension ([RFC8446], Section 4.1.2), it indicates the support for compressed server certificates. Whenever it is sent by the server as a CertificateRequest extension ([RFC8446], Section 4.3.2), it indicates the support for compressed client certificates.

By sending a compress_certificate extension, the sender indicates to the peer the certificate compression algorithms it is willing to use for decompression. The "extension_data" field of this extension SHALL contain a CertificateCompressionAlgorithms value:

```c
enum {
    zlib(1),
    brotli(2),
    zstd(3),
    (65535)
} CertificateCompressionAlgorithm;
```

```c
struct {
    CertificateCompressionAlgorithm algorithms<2..2^8-2>;
} CertificateCompressionAlgorithms;
```

There is no ServerHello extension that the server is required to echo back.

4. Compressed Certificate Message

If the peer has indicated that it supports compression, server and client MAY compress their corresponding Certificate messages and send them in the form of the CompressedCertificate message (replacing the Certificate message).

The CompressedCertificate message is formed as follows:

```c
struct {
    CertificateCompressionAlgorithm algorithm;
    uint24 uncompressed_length;
    opaque compressed_certificate_message<1..2^24-1>;
} CompressedCertificate;
```

algorithm  The algorithm used to compress the certificate. The algorithm MUST be one of the algorithms listed in the peer’s compress_certificate extension.

uncompressed_length  The length of the Certificate message once it is uncompressed. If after decompression the specified length does not match the actual length, the party receiving the invalid
message MUST abort the connection with the "bad_certificate" alert. The presence of this field allows the receiver to pre-allocate the buffer for the uncompressed Certificate message and to enforce limits on the message size before performing decompression.

compressed_certificate_message The compressed body of the Certificate message, in the same format as it would normally be expressed in. The compression algorithm defines how the bytes in the compressed_certificate_message field are converted into the Certificate message.

If the specified compression algorithm is zlib, then the Certificate message MUST be compressed with the ZLIB compression algorithm, as defined in [RFC1950]. If the specified compression algorithm is brotli, the Certificate message MUST be compressed with the Brothi compression algorithm as defined in [RFC7932]. If the specified compression algorithm is zstd, the Certificate message MUST be compressed with the Zstandard compression algorithm as defined in [RFC8478].

It is possible to define a certificate compression algorithm that uses a pre-shared dictionary to achieve higher compression ratio. This document does not define any such algorithms.

If the received CompressedCertificate message cannot be decompressed, the connection MUST be torn down with the "bad_certificate" alert.

If the format of the Certificate message is altered using the server_certificate_type or client_certificate_type extensions [RFC7250], the resulting altered message is compressed instead.

5. Security Considerations

After decompression, the Certificate message MUST be processed as if it were encoded without being compressed. This way, the parsing and the verification have the same security properties as they would have in TLS normally.

In order for certificate compression to function correctly, the underlying compression algorithm MUST be deterministic and it MUST output the same data that was provided as input by the peer.

Since certificate chains are typically presented on a per-server name or per-user basis, the attacker does not have control over any individual fragments in the Certificate message, meaning that they cannot leak information about the certificate by modifying the plaintext.
The implementations SHOULD bound the memory usage when decompressing the CompressedCertificate message.

The implementations MUST limit the size of the resulting decompressed chain to the specified uncompressed length, and they MUST abort the connection if the size exceeds that limit. TLS framing imposes 16777216 byte limit on the certificate message size, and the implementations MAY impose a limit that is lower than that; in both cases, they MUST apply the same limit as if no compression were used.

6. Middlebox Compatibility

It’s been observed that a significant number of middleboxes intercept and try to validate the Certificate message exchanged during a TLS handshake. This means that middleboxes that don’t understand the CompressedCertificate message might misbehave and drop connections that adopt certificate compression. Because of that, the extension is only supported in the versions of TLS where the certificate message is encrypted in a way that prevents middleboxes from intercepting it, that is, TLS version 1.3 [RFC8446] and higher.

7. IANA Considerations

7.1. Update of the TLS ExtensionType Registry

Create an entry, compress_certificate(27), in the existing registry for ExtensionType (defined in [RFC8446]), with "TLS 1.3" column values being set to "CH, CR", and "Recommended" column being set to "Yes".

7.2. Update of the TLS HandshakeType Registry

Create an entry, compressed_certificate(25), in the existing registry for HandshakeType (defined in [RFC8446]).

7.3. Registry for Compression Algorithms

This document establishes a registry of compression algorithms supported for compressing the Certificate message, titled "Certificate Compression Algorithm IDs", under the existing "Transport Layer Security (TLS) Extensions" heading.

The entries in the registry are:
### Algorithm Number | Description
--- | ---
0 | Reserved
1 | zlib
2 | brotli
3 | zstd
16384 to 65535 | Reserved for Experimental Use

The values in this registry shall be allocated under "IETF Review" policy for values strictly smaller than 256, under "Specification Required" policy for values 256-16383, and under "Experimental Use" otherwise (see [RFC8126] for the definition of relevant policies). Experimental Use extensions can be used both on private networks and over the open Internet.

The procedures for requesting values in the Specification Required space are specified in [RFC8447].

#### 8. Normative References


Appendix A. Acknowledgements

Certificate compression was originally introduced in the QUIC Crypto protocol, designed by Adam Langley and Wan-Teh Chang.

This document has benefited from contributions and suggestions from David Benjamin, Ryan Hamilton, Ilari Liusvaara, Piotr Sikora, Ian Swett, Martin Thomson, Sean Turner and many others.

Authors' Addresses

Alessandro Ghedini
Cloudflare, Inc.
Email: alessandro@cloudflare.com

Victor Vasiliev
Google
Email: vasilvv@google.com
The Datagram Transport Layer Security (DTLS) Protocol Version 1.3
draft-ietf-tls-dtls13-32

Abstract

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

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

Status of This Memo

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

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

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

This Internet-Draft will expire on January 9, 2020.

Copyright Notice

Copyright (c) 2019 IETF Trust and the persons identified as the document authors. All rights reserved.
Table of Contents

1. Introduction ........................................... 3
2. Conventions and Terminology .............................. 4
3. DTLS Design Rationale and Overview ...................... 5
   3.1. Packet Loss ....................................... 6
   3.2. Reordering ....................................... 7
   3.3. Message Size ....................................... 7
   3.4. Replay Detection .................................. 7
4. The DTLS Record Layer ................................... 7
   4.1. Determining the Header Format ...................... 11
   4.2. Sequence Number and Epoch .......................... 11
      4.2.1. Processing Guidelines ......................... 11
      4.2.2. Reconstructing the Sequence Number and Epoch .. 12
      4.2.3. Sequence Number Encryption ................... 13
   4.3. Transport Layer Mapping ............................ 14
   4.4. PMTU Issues ...................................... 14
   4.5. Record Payload Protection .......................... 16
      4.5.1. Anti-Replay .................................. 16
      4.5.2. Handling Invalid Records .................... 17
5. The DTLS Handshake Protocol ............................ 17
   5.1. Denial-of-Service Countermeasures .................. 18
   5.2. DTLS Handshake Message Format ..................... 21
   5.3. ClientHello Message ................................ 22
   5.4. Handshake Message Fragmentation and Reassembly ..... 23
   5.5. End Of Early Data ................................. 24
 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 use 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 based on TLS 1.3 [TLS13].

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

2. Conventions and Terminology

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

The following terms are used:
- client: The endpoint initiating the DTLS connection.
- connection: A transport-layer connection between two endpoints.
- endpoint: Either the client or server of the connection.
- handshake: An initial negotiation between client and server that establishes the parameters of their transactions.
- peer: An endpoint. When discussing a particular endpoint, "peer" refers to the endpoint that is remote to the primary subject of discussion.
- receiver: An endpoint that is receiving records.
- sender: An endpoint that is transmitting records.
- **session**: An association between a client and a server resulting from a handshake.

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

- **CID**: Connection ID

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 reordered data traffic.

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

1. TLS relies on an implicit sequence number on records. If a record is not received, then the recipient will use the wrong sequence number when attempting to remove record protection from subsequent records. DTLS solves this problem by adding sequence numbers.
2. The TLS handshake is a lock-step cryptographic handshake. Messages must be transmitted and received in a defined order; any other order is an error. DTLS handshake messages are also assigned sequence numbers to enable reassembly in the correct order in case datagrams are lost or reordered.

3. During the handshake, messages are implicitly acknowledged by other handshake messages, but the last flight of messages and post-handshake messages (such as the NewSessionTicket message) do not result in any direct response that would allow the sender to detect loss. DTLS adds an acknowledgment message to enable better loss recovery.

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

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

3.1. Packet Loss

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

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

X<-- HelloRetryRequest
(lost)

[Timer Expires]

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

Figure 1: DTLS retransmission example

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

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

3.2. Reordering

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

3.3. Message Size

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

3.4. Replay Detection

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

4. The DTLS Record Layer

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

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

3. The DTLSCiphertext structure has a variable length header.

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.

```c
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;
```

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

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

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

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

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

Connection ID: Variable length CID. The CID concept is described in [DTLS-CID]. An example can be found in Section 9.1.

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

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

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

Figure 4 illustrates different record layer header types.
The length field MAY be omitted by clearing the L bit, which means that the record consumes the entire rest of the datagram in the lower level transport. In this case it is not possible to have multiple DTLSChiphertext 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_secret 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.

When expanded, the epoch and sequence number can be combined into an unpacked RecordNumber structure, as shown below:
struct {
    uint16 epoch;
    uint48 sequence_number;
} RecordNumber;

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

The entire header value shown above is used as it appears on the wire as the additional data value for the AEAD function. Note that this design is different from the additional data calculation for DTLS 1.2 and for DTLS 1.2 with Connection ID.

4.1. Determining the Header Format

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

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

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

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

4.2. Sequence Number and Epoch

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

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

4.2.1. Processing Guidelines

Because DTLS records could be reordered, a record from epoch M may be received after epoch N (where N > M) has begun. In general, implementations SHOULD discard 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.)

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.

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

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

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

4.2.2. Reconstructing the Sequence Number and Epoch

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

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

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

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

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

\[ \text{Mask} = \text{AES-ECB}(\text{sn_key}, \text{Ciphertext}[0..15]) \]

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

\[ \text{Mask} = \text{ChaCha20}(\text{sn_key}, \text{Ciphertext}[0..12], \text{Ciphertext}[13..15]) \]

The \text{sn_key} is computed as follows:

\[ [\text{sender}]\_\text{sn_key} = \text{HKDF-Expand-Label}(\text{Secret}, \"sn\", \"\", \text{key_length}) \]

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

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

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

Note that sequence number encryption is only applied to the DTLS-Ciphertext structure and not to the DTLS-Plaintext structure, which also contains a sequence number.
4.3. Transport Layer Mapping

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

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

DTLS records, 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 CID extension defined in [DTLS-CID] 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.

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

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

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

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

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

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

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

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

4.5. Record Payload Protection

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

4.5.1. Anti-Replay

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

The received 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 check SHOULD happen after deprotecting the packet; otherwise the packet discard might itself serve as a timing channel for the record number. Note that decompressing the records number is still a potential timing channel for the record number, though a less powerful one than whether it was deprotected.

Duplicates are rejected through the use of a sliding receive window. (How the window is implemented is a local matter, but the following text describes the functionality that the implementation must exhibit.) The receiver SHOULD pick a window large enough to handle any plausible reordering, which depends on the data rate. (The receiver does not notify the sender of the window size.)
The "right" edge of the window represents the highest validated sequence number value received on the session. Records that contain sequence numbers lower than the "left" edge of the window are rejected. 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 packet is new.

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

4.5.2. Handling Invalid Records

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

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

5. The DTLS Handshake Protocol

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

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

2. Retransmission timers are introduced to handle message loss.

3. A new ACK content type has been added for reliable message delivery of handshake messages.
Note that TLS 1.3 already supports a cookie extension, which is used to prevent denial-of-service attacks. This DoS prevention mechanism is described in more detail below since UDP-based protocols are more vulnerable to amplification attacks than a connection-oriented transport like TCP that performs return-routability checks as part of the connection establishment.

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

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

5.1. Denial-of-Service Countermeasures

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

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

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

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

The DTLS 1.3 specification changes 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 reasons, the cookie field in the ClientHello is present in DTLS 1.3 but is ignored by a DTLS 1.3 compliant server implementation.

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

Client                                   Server
------                                   ------
ClientHello           ------>             HelloRetryRequest
                        + cookie
<----- HelloRetryRequest  + cookie

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

The cookie extension is defined in Section 4.2.2 of [TLS13]. When sending the initial ClientHello, the client does not have a cookie yet. In this case, the cookie extension is omitted and the legacy_cookie field in the ClientHello message 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 [TLS13].

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

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

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

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

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

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

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

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

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

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

```c
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.

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

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

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

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

random: Same as for TLS 1.3.

legacy_session_id: Same as for TLS 1.3.

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

cipher_suites: Same as for TLS 1.3.

legacy_compression_methods: Same as for TLS 1.3.

extensions: Same as for TLS 1.3.

5.4. Handshake Message Fragmentation and Reassembly

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

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

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

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

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

5.6.  DTLS Handshake Flights

DTLS messages are grouped into a series of message flights, according to the diagrams below.
Figure 6: Message flights for a full DTLS Handshake (with cookie exchange)
Figure 7: Message flights for resumption and PSK handshake (without cookie exchange)
ClientHello  
+ early_data  
+ psk_key_exchange_modes  
+ key_share*  
+ pre_shared_key  
(Application Data*)  
--->

ServerHello  
+ pre_shared_key  
+ key_share*  
(EncryptedExtensions)  
(Finished)  
[Application Data*]

(Finished)  
[Application Data*]  
--->

<--------  
[Application Data]  
[Application Data*]  
[ACK]  
[Application Data]  

Figure 8: Message flights for the Zero-RTT handshake

Client  
<--------  
[NewSessionTicket]  
[ACK]  

Figure 9: Message flights for the new session ticket message

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

5.7.1. State Machine

DTLS uses a simple timeout and retransmission scheme with the state machine shown in Figure 10. Because DTLS clients send the first message (ClientHello), they start in the PREPARING state. DTLS servers start in the WAITING state, but with empty buffers and no retransmit timer.
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 an ACK from the peer: upon receiving an ACK for a partial flight (as mentioned in Section 7.1), the implementation transitions to the SENDING state, where it retransmits the unacked portion of the flight, resets the retransmit timer, and returns to the WAITING state. Upon receiving an ACK for a complete flight, the implementation cancels all retransmissions and either remains in WAITING, or, if the ACK was for the final flight, transitions to FINISHED.

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

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

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

In addition, for at least twice the default 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.

5.7.2. Timer Values

Though timer values are the choice of the implementation, mishandling of the timer can lead to serious congestion problems; for example, if many instances of a DTLS time out early and retransmit too quickly on a congested link. Implementations SHOULD use an initial timer value of 100 msec (the minimum defined in RFC 6298 [RFC6298]) and double the value at each retransmission, up to no less than the RFC 6298 maximum of 60 seconds. Application specific profiles, such as those used for the Internet of Things environment, may recommend longer timer values. Note that a 100 msec timer is recommended rather than the 3-second RFC 6298 default in order to improve latency for time-sensitive applications. Because DTLS only uses retransmission for handshake and not dataflow, the effect on congestion should be minimal.
Implementations SHOULD retain the current timer value until a transmission without loss occurs, at which time the value may be reset to the initial value. After a long period of idleness, no less than 10 times the current timer value, implementations may reset the timer to the initial value.

5.8. CertificateVerify and Finished Messages

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

5.9. 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.

5.10. 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 CID, such as the one specified in [I-D.ietf-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.

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 [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. Note this epoch is skipped if the client does not offer early data.

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

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

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

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

The traffic key calculation is described in Section 7.3 of [TLS13]. Figure 12 illustrates the epoch values in an example DTLS handshake.

Client
------

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)

\[
\begin{align*}
\text{[NewSessionTicket]} & \quad (\text{epoch}=3) \\
\text{[ACK]} & \quad (\text{epoch}=3) \\
\text{[Application Data]} & \quad (\text{epoch}=4) \\
\end{align*}
\]

Some time later ...

(Rekeying)

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 having ACK being added to the handshake transcript. Note that ACKs can still be sent in the same UDP datagram as handshake records.

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

record_numbers: a list of the records containing handshake messages in the current flight which the endpoint has received, 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 acknowledge the records corresponding to each transmission of that flight or simply acknowledge 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 acknowledge it (as it might be damaged). If the client does not send an ACK, the server will eventually retransmit its first flight, but this might take far longer than the actual round trip time between client and server. Having the client send an empty ACK shortcuts this process.
7.2. Receiving ACKs

When an implementation receives an ACK, it SHOULD record that the messages or message fragments sent in the records being ACKed were received and omit them from any future retransmissions. Upon receipt of an ACK 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 acknowledgement 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 acknowledged, 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 [DTLS-CID], either side can send a new CID which it wishes the other side to use in a NewConnectionId message.
enum {
    cid_immediate(0), cid_spare(1), (255)
} ConnectionIdUsage;

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

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

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

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

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

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

struct {
    uint8 num_cids;
} RequestConnectionId;

num_cids  The number of CIDs desired.

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

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

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

Note: The connection_id extension is defined in [DTLS-CID], which is used in ClientHello and ServerHello messages.
If no CID is negotiated, then the receiver MUST reject any records it receives that contain a CID.
10. Application Data Protocol

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

11. Security Considerations

Security issues are discussed primarily in [TLS13].

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

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

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

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

If implementations process out-of-epoch records as recommended in Section 8, then this creates a denial of service risk since an adversary could inject 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 CID for DTLS 1.3 builds on top of what is described in [DTLS-CID]. There are, however, several improvements:

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

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

- Switching CID based on certain events, or even regularly, helps against tracking by on-path adversaries but the sequence numbers can still allow linkability. For this reason this specification defines an algorithm for encrypting sequence numbers, see Section 4.2.3. Note that sequence number encryption is used for all encrypted DTLS 1.3 records irrespectively of the use of a CID.

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

12. Changes to DTLS 1.2

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

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

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

- Removed support for weaker and older cryptographic algorithms

- HelloRetryRequest of TLS 1.3 used instead of HelloVerifyRequest

- More flexible ciphersuite negotiation

- New session resumption mechanism

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

13. IANA Considerations

IANA is requested to allocate a new value in the "TLS ContentType" registry for the ACK message, defined in Section 7, with content type 25. IANA is requested to reserve the content type range 32-63 so that content types in this range are not allocated.

IANA is requested to allocate two values in the "TLS Handshake Type" registry, defined in [TLS13], for RequestConnectionId (TBD), and NewConnectionId (TBD), as defined in this document.

14. References

14.1. Normative References


14.2. Informative References

[DTLS-CID]

[I-D.ietf-tls-dtls-connection-id]

[RFC2522]

[RFC4303]

[RFC4340]
14.3. URIs

[1] mailto:tls@ietf.org


Appendix A. Protocol Data Structures and Constant Values

This section provides the normative protocol types and constants definitions.

A.1. Record Layer

```c
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;
```

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

A.2. Handshake Protocol

```c
enum {
    hello_request_RESERVED(0),
    client_hello(1),
```
server_hello(2),
hello_verify_request_RESERVED(3),
ew_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;

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
A.3. ACKs

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

A.4. Connection ID Management

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

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

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

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

Appendix B. History

RFC EDITOR: PLEASE REMOVE THE THIS SECTION

IETF Drafts

draft-32: - Editorial improvements and clarifications.

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

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

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

draft-28: - Version bump to align with TLS 1.3 pre-RFC version.
draft-27: - Incorporated unified header format. - Added support for CIDs.

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

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

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

draft-01 - Restructured the ACK to contain a list of 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 C. 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 D. Contributors

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

In addition, we would like to thank:

* David Benjamin
  Google
davidben@google.com

* Thomas Fossati
  Nokia
  thomas.fossati@nokia.com

* Tobias Gondrom
  Huawei
tobias.gondrom@gondrom.org

* Ilari Liusvaara
  Independent
  ilariliusvaara@welho.com

* Martin Thomson
  Mozilla
  martin.thomson@gmail.com

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

* Yin Xinxing
  Huawei
  yinxinxing@huawei.com

Authors’ Addresses

Eric Rescorla
RTFM, Inc.
EMail: ekr@rtfm.com

Hannes Tschofenig
Arm Limited
EMail: hannes.tschofenig@arm.com
Abstract

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

Status of This Memo

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

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

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

This Internet-Draft will expire on January 9, 2020.

Copyright Notice

Copyright (c) 2019 IETF Trust and the persons identified as the document authors. All rights reserved.

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

Table of Contents

1. Introduction .................................................. 3
2. Conventions and Definitions ................................. 4
3. Overview ...................................................... 4
   3.1. Topologies ............................................... 4
   3.2. SNI Encryption ........................................... 5
4. Publishing the SNI Encryption Key in the DNS ............... 5
   4.1. Encrypted SNI Record .................................... 6
   4.2. Encrypted SNI DNS Resolution ............................ 8
      4.2.1. Address Set Extension ............................... 8
      4.2.2. Resolution Algorithm ................................ 9
5. The "encrypted_server_name" extension ...................... 10
   5.1. Client Behavior ......................................... 11
      5.1.1. Sending an encrypted SNI ............................ 11
      5.1.2. Handling the server response ........................ 14
      5.1.3. Authenticating for the public name ................. 15
      5.1.4. GREASE extensions .................................. 16
   5.2. Client-Facing Server Behavior ......................... 17
   5.3. Shared Mode Server Behavior ............................ 19
   5.4. Split Mode Server Behavior ............................. 19
6. Compatibility Issues ......................................... 20
   6.1. Misconfiguration and Deployment Concerns .............. 20
   6.2. Middleboxes ............................................ 20
7. Security Considerations ..................................... 21
   7.1. Why is cleartext DNS OK? ............................... 21
   7.2. Optional Record Digests and Trial Decryption .......... 21
   7.3. Encrypting other Extensions ............................. 22
   7.4. Related Privacy Leaks .................................. 22
   7.5. Comparison Against Criteria ............................ 22
      7.5.1. Mitigate against replay attacks .................... 22
      7.5.2. Avoid widely-deployed shared secrets ................ 23
      7.5.3. Prevent SNI-based DoS attacks ...................... 23
      7.5.4. Do not stick out .................................... 23
      7.5.5. Forward secrecy ..................................... 23
      7.5.6. Proper security context ............................. 23
      7.5.7. Split server spoofing ................................ 24
      7.5.8. Supporting multiple protocols ....................... 24
   7.6. Misrouting .............................................. 24
8. IANA Considerations ......................................... 24
   8.1. Update of the TLS ExtensionType Registry ............... 24
   8.2. Update of the TLS Alert Registry ........................ 24
   8.3. Update of the Resource Record (RR) TYPEs Registry .... 25
9. References .................................................... 25
   9.1. Normative References ................................... 25
1. Introduction

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

Although TLS 1.3 [RFC8446] encrypts most of the handshake, including the server certificate, there are several other channels that allow an on-path attacker to determine the domain name the client is trying to connect to, including:

- Cleartext client DNS queries.
- Visible server IP addresses, assuming the the server is not doing domain-based virtual hosting.
- 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. All TLS notation comes from [RFC8446]; Section 3.

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.
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

First, the provider publishes a public key and metadata 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 and metadata preconfigured.

When a client wants to form a TLS connection to any of the domains served by an ESNI-supporting provider, it sends 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 in the DNS

Publishing ESNI keys in the DNS requires care to ensure correct behavior. There are deployment environments in which a domain is served by multiple server operators who do not manage the ESNI keys. Because ESNI and A/AAAA lookups are independent, it is therefore possible to obtain an ESNI record which does not match the A/AAAA records. (That is, the host to which an A or AAAA record refers is not in possession of the ESNI keys.) The design of the system must
therefore allow clients to detect and recover from this situation (see Section 4.2 for more details).

Content providers operating in Split Mode SHOULD ensure that the A and AAAA records for ESNI-enabled server names do not allow identifying the server name from the IP address. This can for example be achieved by always returning the same records for all ESNI-enabled names, or by having the function that picks addresses from a pool not depend on the server name. 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 following sections describe a DNS record format that achieve these goals.

4.1. Encrypted SNI Record

SNI Encryption keys can be published using the following ESNIRecord structure.

```c
// Copied from TLS 1.3
struct {  
    NamedGroup group;
    opaque key_exchange<1..2^16-1>;
} KeyShareEntry;

struct {  
    uint16 version;
    opaque public_name<1..2^16-1>;
    KeyShareEntry keys<4..2^16-1>;
    CipherSuite cipher_suites<2..2^16-2>;
    uint16 padded_length;
    Extension extensions<0..2^16-1>;
} ESNIKeys;

struct {  
    ESNIKeys esni_keys;
    Extension dns_extensions<0..2^16-1>;
} ESNIRecord;
```

The outermost ESNIRecord structure contains the following fields:
esni_keys  An ESNIKeys structure that contains the actual keys used to encrypt the SNI as well as some metadata related to those keys.

dns_extensions  A list of extensions that the client can take into consideration when resolving the target DNS name. The format is defined in [RFC8446]; Section 4.2. The purpose of the field is to provide room for additional features in the future. An extension may be tagged as mandatory by using an extension type codepoint with the high order bit set to 1. A client which receives a mandatory extension they do not understand must reject the ESNIRecord values.

The ESNIKeys structure contains the following fields:

version  The version of the structure. For this specification, that value SHALL be 0xff03. Clients MUST ignore any ESNIKeys structure with a version they do not understand. [[NOTE: This means that the RFC will presumably have a nonzero value.]]

public_name  The non-empty name of the entity trusted to update these encryption keys. This is used to repair misconfigurations, as described in Section 5.1.2.

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 arbitrary wildcard names, it SHOULD set this value to 260. Clients SHOULD reject ESNIKeys as invalid if padded_length is greater than 260.

extensions  A list of extensions that the client can take into consideration when generating a Client Hello message. The format is defined in [RFC8446]; Section 4.2. The purpose of the field is to provide room for additional features in the future. An extension may be tagged as mandatory by using an extension type codepoint with the high order bit set to 1. A client which receives a mandatory extension they do not understand must reject the ESNIRecord value.

Any of the listed keys in the ESNIKeys value 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. Clients MUST parse the extension list and check for unsupported mandatory extensions. If an unsupported mandatory extension is present, clients MUST reject the ESNIRecord value.
The ESNIRecord structure is placed in the RRData section of an ESNI record as-is. Servers MAY supply multiple ESNIRecord values, with ESNIKeys either of the same or of different versions. This allows a server to support multiple versions at once. If the server does not supply any ESNIRecord values with an ESNIKeys version known to the client, then the client MUST behave as if no ESNI records were found.

The name of each ESNI record MUST match the query domain name or the query domain name’s canonicalized form. That is, if a client queries example.com, the ESNI Resource Record might be:

example.com. 60S IN ESNI "..." "...

In the event that ESNIKeys is corrupt in transit or otherwise invalid, servers will initiate the retry mechanism described in Section 5.2 and deliver valid ESNIKeys to clients.

Note that the length of the ESNIRecord structure MUST NOT exceed $2^{16} - 1$, as the RDLENGTH is only 16 bits [RFC1035].

4.2. Encrypted SNI DNS Resolution

This section describes a client ESNI resolution algorithm using an "address_set" extension for the ESNIRecord structure. Future specifications may introduce new ESNIRecord extensions and corresponding resolution algorithms.

4.2.1. Address Set Extension

ESNIRecord values MAY indicate one or more IP addresses for the host(s) in possession of the private key corresponding to one of the keys provided in the ESNIKeys structure, via the following mandatory "address_set" extension:

```
enum {
    address_set(0x1001), (65535)
} ExtensionType;
```

The body of this extension is encoded using the following structure.
enum {
    address_v4(4),
    address_v6(6),
} AddressType;

struct {
    AddressType address_type;
    select (address_type) {
        case address_v4: {
            opaque ipv4Address[4];
        }
        case address_v6: {
            opaque ipv6Address[16];
        }
    }
} Address;

struct {
    Address address_set<1..2^16-1>;
} AddressSet;

address_set  A set of Address structures containing IPv4 or IPv6 addresses to hosts which have the corresponding private ESNI key.

This extension MUST NOT be placed in the ESNIKeys extensions field, but only in the ESNIRecord dns_extensions field.

4.2.2. Resolution Algorithm

Clients obtain ESNI records by querying the DNS for ESNI-enabled server domains. In cases where the domain of the A or AAAA records being resolved do not match the SNI Server Name, such as when [RFC7838] is being used, the alternate domain should be used for querying the ESNI record. (See Section 2.3 of [RFC7838] for more details.)

Clients SHOULD initiate ESNI queries in parallel alongside normal A or AAAA queries to obtain address information in a timely manner in the event that ESNI is available. The following algorithm describes a procedure by which clients can process ESNI responses as they arrive to produce addresses for ESNI-capable hosts.
1. If an ESNI response containing an ESNIRecord value with an "address_set" extension arrives before an A or AAAA response, clients SHOULD initiate TLS with ESNI to the provided address(es).

2. If an A or AAAA response arrives before the ESNI response, clients SHOULD wait up to CD milliseconds before initiating TLS to either address. (Clients may begin TCP connections in this time. QUIC connections should wait.) If an ESNI response with an "address_set" extension arrives in this time, clients SHOULD initiate TLS with ESNI to the provided address(es). If an ESNI response without an "address_set" extension arrives in this time, clients MAY initiate TLS with ESNI to the address(es) in the A or AAAA response. If no ESNI response arrives in this time, clients SHOULD initiate TLS without ESNI to the available address(es).

CD (Connection Delay) is a configurable parameter. The recommended value is 50 milliseconds, as per the guidance in [RFC8305].

5. The "encrypted_server_name" extension

The encrypted SNI is carried in an "encrypted_server_name" extension, defined as follows:

```c
enum {
    encrypted_server_name(0xffce), (65535)
} ExtensionType;
```

For clients (in ClientHello), this extension contains the following ClientEncryptedSNI structure:

```c
struct {
    CipherSuite suite;
    KeyShareEntry key_share;
    opaque record_digest<0..2^16-1>;
    opaque encrypted_sni<0..2^16-1>;
} ClientEncryptedSNI;
```

- **suite**  The cipher suite used to encrypt the SNI.
- **key_share**  The KeyShareEntry carrying the client’s public ephemeral key shared used to derive the ESNI key.
- **record_digest**  A cryptographic hash of the ESNIKeys structure from which the ESNI key was obtained, i.e., from the first byte of "version" to the end of the structure. This hash is computed using the hash function associated with "suite".
- **encrypted_sni**  The ClientESNIInner structure, AEAD-encrypted using cipher suite "suite" and the key generated as described below.
For servers (in EncryptedExtensions), this extension contains the following structure:

```c
enum {
    esni_accept(0),
    esni_retry_request(1),
} ServerESNIResponseType;

struct {
    ServerESNIResponseType response_type;
    select (response_type) {
        case esni_accept:        uint8 nonce[16];
        case esni_retry_request: ESNIKeys retry_keys<1..2^16-1>;
    }
} ServerEncryptedSNI;
```

- `response_type` Indicates whether the server processed the client ESNI extension. (See Section 5.1.2 and Section 5.2.)
- `nonce` The contents of ClientESNIInner.nonce. (See Section 5.1.)
- `retry_keys` One or more ESNIKeys structures containing the keys that the client should use on subsequent connections to encrypt the ClientESNIInner structure.

This protocol also defines the "esni_required" alert, which is sent by the client when it offered an "encrypted_server_name" extension which was not accepted by the server.

```c
enum {
    esni_required(121),
} AlertDescription;
```

Finally, requirements in Section 5.1 and Section 5.2 require implementations to track, alongside each PSK established by a previous connection, whether the connection negotiated this extension with the "esni_accept" response type. If so, this is referred to as an "ESNI PSK". Otherwise, it is a "non-ESNI PSK". This may be implemented by adding a new field to client and server session states.

5.1. Client Behavior

5.1.1. Sending an encrypted SNI

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 will then be sent to the server in
the "encrypted_sni" extension and used to derive the SNI encryption key. It does not affect the (EC)DHE shared secret used in the TLS key schedule. It MUST also select an appropriate cipher suite from the list of suites offered by the server. If the client is unable to select an appropriate group or suite it SHOULD ignore that ESNIKeys value and MAY attempt to use another value provided by the server.

(Recall that servers might provide multiple ESNIRecord values in response to a ESNI record query, each containing an ESNIKeys value.) The client MUST NOT send encrypted SNI using groups or cipher suites not advertised by the server.

When offering an encrypted SNI, the client MUST NOT offer to resume any non-ESNI PSKs. It additionally MUST NOT offer to resume any sessions for TLS 1.2 or below.

Let Z be the DH shared secret derived from a key share in ESNIKeys and the corresponding client share in ClientEncryptedSNI.key_share. The SNI encryption key is computed from Z as follows:

\[
Z_x = \text{HKDF-Extract(0, } Z) \\
key = \text{HKDF-Expand-Label}(Z_x, \text{KeyLabel, Hash(ESNIContents)}, \text{key_length}) \\
v = \text{HKDF-Expand-Label}(Z_x, \text{IVLabel, Hash(ESNIContents)}, \text{iv_length})
\]

where ESNIContents is as specified below and Hash is the hash function associated with the HKDF instantiation. The salt argument for HKDF-Extract is a string consisting of Hash.length bytes set to zeros. For a client’s first ClientHello, KeyLabel = "esni key" and IVLabel = "esni iv", whereas for a client’s second ClientHello, sent in response to a HelloRetryRequest, KeyLabel = "hrr esni key" and IVLabel = "hrr esni iv". (This label variance is done to prevent nonce re-use since the client’s ESNI key share, and thus the value of Zx, does not change across ClientHello retries.)

[[TODO: label swapping fixes a bug in the spec, though this may not be the best way to deal with HRR. See https://github.com/tlswg/draft-ietf-tls-esni/issues/121 and https://github.com/tlswg/draft-ietf-tls-esni/pull/170 for more details.]]

\[
\text{struct} \\
\quad \text{opaque record_digest<0..2^{16}-1>;} \\
\quad \text{KeyShareEntry esni_key_share;} \\
\quad \text{Random client_hello_random;} \\
\quad \text{ESNIContents;}
\]

The client then creates a ClientESNIInner structure:

Rescorla, et al. Expires January 9, 2020
struct {
    opaque dns_name<1..2^16-1>;
    opaque zeros[ESNIKeys.padded_length - length(sni)];
} PaddedServerNameList;

struct {
    uint8 nonce[16];
    PaddedServerNameList realSNI;
} ClientESNIInner;

nonce  A random 16-octet value to be echoed by the server in the "encrypted_server_name" extension.

dns_name  The true SNI DNS name, that is, the HostName value that would have been sent in the plaintext "server_name" extension. (NameType values other than "host_name" are unsupported since SNI extensibility failed [SNIExtensibilityFailed]).

zeros  Zero padding whose length makes the serialized PaddedServerNameList struct have a length equal to ESNIKeys.padded_length.

This value consists of the serialized ServerNameList from the "server_name" extension, 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 ClientEncryptedSNI.encrypted_sni value is then computed using the usual TLS 1.3 AEAD:

\[
encrypted_sni = \text{AEAD-Encrypt}(key, iv, KeyShareClientHello, ClientESNIInner)
\]

Where KeyShareClientHello is the "extension_data" field of the "key_share" extension in a Client Hello (Section 4.2.8 of [RFC8446]). Including KeyShareClientHello in the AAD of AEAD-Encrypt binds the ClientEncryptedSNI value to the ClientHello and prevents cut-and-paste attacks.

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 the future you were to reuse these keys for 0-RTT priming, then you would have to worry about potentially expanding...]]
twice of $Z_{\text{extracted}}$. We should think about how to harmonize these
to make sure that we maintain key separation.]

This value is placed in an "encrypted_server_name" extension.

The client MUST place the value of ESNIKeys.public_name in the
"server_name" extension. (This is required for technical conformance
with [RFC7540]; Section 9.2.) The client MUST NOT send a
"cached_info" extension [RFC7924] with a CachedObject entry whose
CachedInformationType is "cert".

5.1.2. Handling the server response

If the server negotiates TLS 1.3 or above and provides an
"encrypted_server_name" extension in EncryptedExtensions, the client
then processes the extension’s "response_type" field:

- If the value is "esni_accept", the client MUST check that the
  extension’s "nonce" field matches ClientESNIInner.nonce and
  otherwise abort the connection with an "illegal_parameter" alert.
The client then proceeds with the connection as usual,
  authenticating the connection for the origin server.

- If the value is "esni_retry_request", the client proceeds with the
  handshake, authenticating for ESNIKeys.public_name as described in
  Section 5.1.3. If authentication or the handshake fails, the
  client MUST return a failure to the calling application. It MUST
  NOT use the retry keys.

Otherwise, when the handshake completes successfully with the
public name authenticated, the client MUST abort the connection
with an "esni_required" alert. It then processes the "retry_keys"
field from the server’s "encrypted_server_name" extension.

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

If none of the values provided in "retry_keys" contains a
supported version, the client can regard ESNI as securely disabled
by the server. As below, it SHOULD then retry the handshake with
a new transport connection and ESNI disabled.
If the field contains any other value, the client MUST abort the connection with an "illegal_parameter" alert.

If the server negotiates an earlier version of TLS, or if it does not provide an "encrypted_server_name" extension in EncryptedExtensions, the client proceeds with the handshake, authenticating for ESNIKeys.public_name as described in Section 5.1.3. If an earlier version was negotiated, the client MUST NOT enable the False Start optimization [RFC7918] for this handshake. If authentication or the handshake fails, the client MUST return a failure to the calling application. It MUST NOT treat this as a secure signal to disable ESNI.

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

[[TODO: Key replacement is significantly less scary than saying that ESNI-naive servers bounce ESNI off. Is it worth defining a strict mode toggle in the ESNI keys, for a deployment to indicate it is ready for that? ]]

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

If the server sends a HelloRetryRequest in response to the ClientHello and the client can send a second updated ClientHello per the rules in [RFC8446], the "encrypted_server_name" extension values which do not depend on the (possibly updated) KeyShareClientHello, i.e., ClientEncryptedSNI.suite, ClientEncryptedSNI.key_share, and ClientEncryptedSNI.record_digest, MUST NOT change across ClientHello messages. Moreover, ClientESNIInner MUST not change across ClientHello messages. Informally, the values of all unencrypted extension information, as well as the inner extension plaintext, must be consistent between the first and second ClientHello messages.

5.1.3. Authenticating for the public name

When the server cannot decrypt or does not process the "encrypted_server_name" extension, it continues with the handshake using the cleartext "server_name" extension instead (see Section 5.2). Clients that offer ESNI then authenticate the connection with the public name, as follows:
If the server resumed a session or negotiated a session that did not use a certificate for authentication, the client MUST abort the connection with an "illegal_parameter" alert. This case is invalid because Section 5.1.1 requires the client to only offer ESNI-established sessions, and Section 5.2 requires the server to decline ESNI-established sessions if it did not accept ESNI.

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

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

Note that authenticating a connection for the public name does not authenticate it for the origin. The TLS implementation MUST NOT report such connections as successful to the application. It additionally MUST ignore all session tickets and session IDs presented by the server. These connections are only used to trigger retries, as described in Section 5.1.2. This may be implemented, for instance, by reporting a failed connection with a dedicated error code.

5.1.4. GREASE extensions

If the client attempts to connect to a server and does not have an ESNIKeys structure available for the server, it SHOULD send a GREASE [I-D.ietf-tls-grease] "encrypted_server_name" extension as follows:

Select a supported cipher suite, named group, and padded_length value. The padded_length value SHOULD be 260 or a multiple of 16 less than

1. Set the "suite" field to the selected cipher suite. These selections SHOULD vary to exercise all supported configurations, but MAY be held constant for successive connections to the same server in the same session.

Set the "key_share" field to a randomly-generated valid public key for the named group.

Set the "record_digest" field to a randomly-generated string of hash_length bytes, where hash_length is the length of the hash function associated with the chosen cipher suite.
Set the "encrypted_sni" field to a randomly-generated string of 16 + padded_length + tag_length bytes, where tag_length is the tag length of the chosen cipher suite’s associated AEAD.

If the server sends an "encrypted_server_name" extension, the client MUST check the extension syntactically and abort the connection with a "decode_error" alert if it is invalid. If the "response_type" field contains "esni_retry_requested", the client MUST ignore the extension and proceed with the handshake. If it contains "esni_accept" or any other value, the client MUST abort the connection with an "illegal_parameter" alert.

Offering a GREASE extension is not considered offering an encrypted SNI for purposes of requirements in Section 5.1. In particular, the client MAY offer to resume sessions established without ESNI.

5.2. Client-Facing Server Behavior

Upon receiving an "encrypted_server_name" extension, the client-facing server MUST check that it is able to negotiate TLS 1.3 or greater. If not, it MUST abort the connection with a "handshake_failure" alert.

The ClientEncryptedSNI value is said to match a known ESNIKeys if there exists an ESNIKeys that can be used to successfully decrypt ClientEncryptedSNI.encrypted_sni. This matching procedure should be done using one of the following two checks:

1. Compare ClientEncryptedSNI.record_digest against cryptographic hashes of known ESNIKeys and choose the one that matches.

2. Use trial decryption of ClientEncryptedSNI.encrypted_sni with known ESNIKeys and choose the one that succeeds.

Some uses of ESNI, such as local discovery mode, may omit the ClientEncryptedSNI.record_digest since it can be used as a tracking vector. In such cases, trial decryption should be used for matching ClientEncryptedSNI to known ESNIKeys. Unless specified by the application using (D)TLS or externally configured on both sides, implementations MUST use the first method.

If the ClientEncryptedSNI value does not match any known ESNIKeys structure, it MUST ignore the extension and proceed with the connection, with the following added behavior:

- It MUST include the "encrypted_server_name" extension in EncryptedExtensions message with the "response_type" field set to "esni_retry_requested" and the "retry_keys" field set to one or
more ESNIKeys structures with up-to-date keys. Servers MAY supply multiple ESNIKeys values of different versions. This allows a server to support multiple versions at once.

- The server MUST ignore all PSK identities in the ClientHello which correspond to ESNI PSKs. ESNI PSKs offered by the client are associated with the ESNI name. The server was unable to decrypt then ESNI name, so it should not resume them when using the cleartext SNI name. This restriction allows a client to reject resumptions in Section 5.1.3.

Note that an unrecognized ClientEncryptedSNI.record_digest value may be a GREASE ESNI extension (see Section 5.1.4), so it is necessary for servers to proceed with the connection and rely on the client to abort if ESNI was required. In particular, the unrecognized value alone does not indicate a misconfigured ESNI advertisement (Section 6.1). Instead, servers can measure occurrences of the "esni_required" alert to detect this case.

If the ClientEncryptedSNI value does match a known ESNIKeys, the server performs the following checks:

- If the ClientEncryptedSNI.key_share group does not match one in the ESNIKeys.keys, 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, which also means the server MUST NOT send the "server_name" extension to the client.

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.

If the ClientHello is the result of a HelloRetryRequest, servers MUST abort the connection with an "illegal_parameter" alert if any of the ClientEncryptedSNI.suite, ClientEncryptedSNI.key_share, ClientEncryptedSNI.record_digest, or decrypted ClientESNIInner values from the second ClientHello do not match that of the first ClientHello.

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. Moreover, the server MUST include the "encrypted_server_name" extension in EncryptedExtensions with the "response_type" field set to "esni_accept" and the "nonce" field set to the decrypted PaddedServerNameList.nonce value from the client "encrypted_server_name" extension.

If the server sends a NewSessionTicket message, the corresponding ESNI PSK MUST be ignored by all other servers in the deployment when not negotiating ESNI, including servers which do not implement this specification.

This restriction provides robustness for rollbacks (see Section 6.1).

5.4. Split Mode Server Behavior

In Split Mode, the backend server must know PaddedServerNameList.nonce to echo it back in EncryptedExtensions and complete the handshake. Appendix A describes one mechanism for sending both PaddedServerNameList.sni and ClientESNIInner.nonce to the backend server. Thus, backend servers function the same as servers operating in Shared Mode.

As in Shared Mode, if the backend server sends a NewSessionTicket message, the corresponding ESNI PSK MUST be ignored by other servers in the deployment when not negotiating ESNI, including servers which do not implement this specification.
6. Compatibility Issues

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

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

6.1. Misconfiguration and Deployment Concerns

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

The retry mechanism repairs inconsistencies, provided the server is authoritative for the public name. If server and advertised keys mismatch, the server will respond with esni_retry_requested. If the server does not understand the "encrypted_server_name" extension at all, it will ignore it as required by [RFC8446]; Section 4.1.2. Provided the server can present a certificate valid for the public name, the client can safely retry with updated settings, as described in Section 5.1.2.

Unless ESNI is disabled as a result of successfully establishing a connection to the public name, the client MUST NOT fall back to cleartext SNI, as this allows a network attacker to disclose the SNI. It 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. [RFC8446]; Section 9.3 requires that such proxies remove any extensions they do not understand. The handshake will then present a certificate based on the public name, without echoing the "encrypted_server_name" extension to the client.
Depending on whether the client is configured to accept the proxy’s certificate as authoritative for the public name, this may trigger the retry logic described in Section 5.1.2 or result in a connection failure. A proxy which is not authoritative for the public name cannot forge a signal to disable ESNI.

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

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, ESNI records have no authenticity or provenance information. This means that any attacker which can inject DNS responses or poison DNS caches, which is a common scenario in client access networks, can supply clients with fake ESNI records (so that the client encrypts SNI to them) or strip the ESNI record 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 ESNI records in the clear does not make the situation significantly worse.

Clearly, DNSSEC (if the client validates and hard fails) is a defense against this form of attack, but DoH/DPRIVE are also defenses against DNS attacks by attackers on the local network, which is a common case where SNI is desired. 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. Optional Record Digests and Trial Decryption

Supporting optional record digests and trial decryption opens oneself up to DoS attacks. Specifically, an adversary may send malicious ClientHello messages, i.e., those which will not decrypt with any known ESNI key, in order to force decryption. Servers that support this feature should, for example, implement some form of rate limiting mechanism to limit the damage caused by such attacks.
7.3. Encrypting other Extensions

ESNI protects only the SNI in transit. Other ClientHello extensions, such as ALPN, might also reveal privacy-sensitive information to the network. As such, it might be desirable to encrypt other extensions alongside the SNI. However, the SNI extension is unique in that non-TLS-terminating servers or load balancers may act on its contents. Thus, using keys specifically for SNI encryption promotes key separation between client-facing servers and endpoints party to TLS connections. Moreover, the ESNI design described herein does not preclude a mechanism for generic ClientHello extension encryption.

7.4. Related Privacy Leaks

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

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

7.5. 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.

7.5.1. 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.
7.5.2. 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 ESNIRecord and ESNIKeys values with different keys using a short TTL.

7.5.3. Prevent SNI-based DoS attacks

This design requires servers to decrypt ClientHello messages with ClientEncryptedSNI 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.

7.5.4. Do not stick out

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.

Additionally, this specification allows for clients to send GREASE ESNI extensions (see Section 5.1.4), which helps ensure the ecosystem handles the values correctly.

7.5.5. Forward secrecy

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

7.5.6. Proper security context

This design permits servers operating in Split Mode to forward connections directly to backend origin servers, thereby avoiding unnecessary MiTM attacks.
7.5.7. Split server spoofing

Assuming ESNI records 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.

Validating the ESNIKeys structure additionally validates the public name. This validates any retry signals from the server because the client validates the server certificate against the public name before retrying.

7.5.8. Supporting multiple protocols

This design has no impact on application layer protocol negotiation. It may affect connection routing, server certificate selection, and client certificate verification. Thus, it is compatible with multiple protocols.

7.6. 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 [RFC8446]), with "TLS 1.3" column values being set to "CH, EE", and "Recommended" column being set to "Yes".

8.2. Update of the TLS Alert Registry

IANA is requested to create an entry, esni_required(121) in the existing registry for Alerts (defined in [RFC8446]), with the "DTLS-OK" column being set to "y".
8.3. Update of the Resource Record (RR) TYPEs Registry

IANA is requested to create an entry, ESNI(0xff9f), in the existing registry for Resource Record (RR) TYPEs (defined in [RFC6895]) with "Meaning" column value being set to "Encrypted SNI".

9. References

9.1. Normative References

[I-D.ietf-tls-exported-authenticator]


9.2. Informative References

[I-D.ietf-doh-dns-over-https]

[I-D.ietf-tls-grease]

[I-D.ietf-tls-sni-encryption]

[I-D.kazuho-protected-sni]
Oku, K., "TLS Extensions for Protecting SNI", draft-kazuho-protected-sni-00 (work in progress), July 2017.

Appendix A. Communicating SNI and Nonce to Backend Server

When operating in Split Mode, backend servers will not have access to PaddedServerNameList.sni or ClientESNIInner.nonce without access to the ESNI keys or a way to decrypt ClientEncryptedSNI.encrypted_sni.

One way to address this for a single connection, at the cost of having communication not be unmodified TLS 1.3, is as follows. Assume there is a shared (symmetric) key between the client-facing server and the backend server 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 and nonce.

Another way for backend servers to access the true SNI and nonce is by the client-facing server sharing the ESNI keys.

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:

- It protects all the extensions from ordinary eavesdroppers
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. Acknowledgements

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

Authors’ Addresses

Eric Rescorla
RTFM, Inc.
Email: ekr@rtfm.com

Kazuho Oku
Fastly
Email: kazuhooku@gmail.com

Nick Sullivan
Cloudflare
Email: nick@cloudflare.com
Christopher A. Wood
Apple, Inc.

Email: cawood@apple.com
Abstract

This document, if approved, formally deprecates Transport Layer Security (TLS) versions 1.0 [RFC2246] and 1.1 [RFC4346] and moves these documents to the historic state. These versions lack support for current and recommended cipher suites, and various government and industry profiles of applications using TLS now mandate avoiding these old TLS versions. TLSv1.2 has been the recommended version for IETF protocols since 2008, providing sufficient time to transition away from older versions. Products having to support older versions increase the attack surface unnecessarily and increase opportunities for misconfigurations. Supporting these older versions also requires additional effort for library and product maintenance.

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

This document updates many RFCs that normatively refer to TLSv1.0 or TLSv1.1 as described herein. This document also updates RFC 7525 and hence is part of BCP195.
Status of This Memo

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

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

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

This Internet-Draft will expire on December 22, 2019.

Copyright Notice

Copyright (c) 2019 IETF Trust and the persons identified as the document authors. All rights reserved.

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

Table of Contents

1.  Introduction .................................................. 3
    1.1.  RFCs Updated ........................................... 3
    1.2.  Terminology ............................................. 4
2.  Support for Deprecation ....................................... 4
3.  SHA-1 Usage Problematic in TLSv1.0 and TLSv1.1 ................ 5
4.  Do Not Use TLSv1.0 ............................................ 6
5.  Do Not Use TLSv1.1 ............................................. 6
6.  Updates to RFC7525 ............................................ 7
7.  Security Considerations ....................................... 7
8.  Acknowledgements .............................................. 8
9.  IANA Considerations ........................................... 8
10. References .................................................... 8
    10.1.  Normative References ................................... 8
    10.2.  Informative References ................................. 16
1. Introduction

Transport Layer Security (TLS) versions 1.0 [RFC2246] and 1.1 [RFC4346] were superseded by TLSv1.2 [RFC5246] in 2008, which has now itself been superseded by TLSv1.3 [RFC8446]. It is therefore timely to further deprecate these old versions. The expectation is that TLSv1.2 will continue to be used for many years alongside TLSv1.3.

The primary technical reasons for deprecating these versions include:

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

Deprecation of these versions is intended to assist developers as additional justification to no longer support older 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.

1.1. RFCs Updated

This document updates the following RFCs that normatively reference TLSv1.0 or TLSv1.1 or DTLS1.0. The update is to obsolete usage of these older versions. Fallback to these versions are prohibited through this update.

[RFC8446] [RFC8422] [RFC8261] [RFC7568] [RFC7562] [RFC7525] [RFC7507]
[RFC7465] [RFC6750] [RFC6749] [RFC6739] [RFC6460] [RFC6084] [RFC6083]
[RFC6367] [RFC6176] [RFC6042] [RFC6012] [RFC5878] [RFC5734] [RFC5469]
[RFC5456] [RFC5422] [RFC5415] [RFC5364] [RFC5281] [RFC5263] [RFC5238]
In addition these RFCs normatively refer to TLSv1.0 or TLSv1.1 and have been obsoleted: [RFC5101] [RFC5081] [RFC5077] [RFC4934] [RFC4572] [RFC4507] [RFC4492] [RFC4366] [RFC4347] [RFC4244] [RFC4132] [RFC3920] [RFC3734] [RFC3588] [RFC3489] [RFC3316] [RFC3501] [RFC3470] [RFC3436] [RFC3329] [RFC3261]

In the case of [RFC4642], that has already been updated by [RFC8143] which makes an overlapping, but not quite the same, update as this document.

This document updates DTLS [RFC6347].

1.2. Terminology

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

2. Support for Deprecation

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

NIST for example have provided the following rationale, copied with permission from NIST SP800-52r2 [NIST800-52r2], section 1.2 "History of TLS" (with references changed for RFC formatting).

TLS 1.1, specified in [RFC4346], was developed to address weaknesses discovered in TLS 1.0, primarily in the areas of initialization vector selection and padding error processing. Initialization vectors were made explicit to prevent a certain class of attacks on the Cipher Block Chaining (CBC) mode of operation used by TLS. The handling of padding errors was altered to treat a padding error as a bad message authentication code,
rather than a decryption failure. In addition, the TLS 1.1 RFC acknowledges attacks on CBC mode that rely on the time to compute the message authentication code (MAC). The TLS 1.1 specification states that to defend against such attacks, an implementation must process records in the same manner regardless of whether padding errors exist. Further implementation considerations for CBC modes (which were not included in RFC4346 [RFC4346]) are discussed in Section 3.3.2.

TLSv1.2, specified in RFC5246 [RFC5246], made several cryptographic enhancements, particularly in the area of hash functions, with the ability to use or specify the SHA-2 family algorithms for hash, MAC, and Pseudorandom Function (PRF) computations. TLSv1.2 also adds authenticated encryption with associated data (AEAD) cipher suites.

TLS 1.3, specified in TLSv1.3 [RFC8446], represents a significant change to TLS that aims to address threats that have arisen over the years. Among the changes are a new handshake protocol, a new key derivation process that uses the HMAC-based Extract-and-Expand Key Derivation Function (HKDF), and the removal of cipher suites that use static RSA or DH key exchanges, the CBC mode of operation, or SHA-1. The list of extensions that can be used with TLS 1.3 has been reduced considerably.

### 3. SHA-1 Usage Problematic in TLSv1.0 and TLSv1.1

The integrity of both TLSv1.0 and TLSv1.1 depends on a running SHA-1 hash of the exchanged messages. This makes it possible to perform a downgrade attack on the handshake by an attacker able to perform 2^77 operations, well below the acceptable modern security margin.

Similarly, the authentication of the handshake depends on signatures made using SHA-1 hash or a not stronger concatenation of MD-5 and SHA-1 hashes, allowing the attacker to impersonate a server when it is able to break the severely weakened SHA-1 hash.

Neither TLSv1.0 nor TLSv1.1 allow the peers to select a stronger hash for signatures in the ServerKeyExchange or CertificateVerify messages, making the only upgrade path the use of a newer protocol version.

See [Bhargavan2016] for additional detail.
4. Do Not Use TLSv1.0

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

Any other version of TLS is more secure than TLSv1.0. TLSv1.0 can be configured to prevent interception, though using the highest version available is preferable.

Pragmatically, clients MUST NOT send a ClientHello with ClientHello.client_version set to {03,01}. Similarly, servers MUST NOT send a ServerHello with ServerHello.server_version set to {03,01}. Any party receiving a Hello message with the protocol version set to {03,01} MUST respond with a "protocol_version" alert message and close the connection.

Historically, TLS specifications were not clear on what the record layer version number (TLSPlaintext.version) could contain when sending ClientHello. Appendix E of [RFC5246] notes that TLSPlaintext.version could be selected to maximize interoperability, though no definitive value is identified as ideal. That guidance is still applicable; therefore, TLS servers MUST accept any value {03,XX} (including {03,00}) as the record layer version number for ClientHello, but they MUST NOT negotiate TLSv1.0.

5. Do Not Use TLSv1.1

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

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

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

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

6. Updates to RFC7525

RFC7525 is BCP195, "Recommendations for Secure Use of Transport Layer
Security (TLS) and Datagram Transport Layer Security (DTLS)", is the
most recent best practice document for implementing TLS and was based
on TLSv1.2. At the time of publication, TLSv1.0 and TLSv1.1 had not
yet been deprecated. As such, this document is called out
specifically to update text implementing the deprecation
recommendations of this document.

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

- Implementations MUST NOT negotiate TLS version 1.0 [RFC2246].
  
  Rationale: TLSv1.0 (published in 1999) does not support many
  modern, strong cipher suites. In addition, TLSv1.0 lacks a per-
  record Initialization Vector (IV) for CBC-based cipher suites and
does not warn against common padding errors.

- Implementations MUST NOT negotiate TLS version 1.1 [RFC4346].
  
  Rationale: TLSv1.1 (published in 2006) is a security improvement
  over TLSv1.0 but still does not support certain stronger cipher
  suites.

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

- Implementations MUST NOT negotiate DTLS version 1.0 [RFC4347],
  [RFC6347].
  
  Version 1.0 of DTLS correlates to version 1.1 of TLS (see above).

7. Security Considerations

This document deprecates two older 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

Thanks to those that provided usage data, reviewed and/or improved this document, including: David Benjamin, David Black, Viktor Dukhovni, Julien Elie, Gary Gapinski, Alessandro Ghedini, Jeremy Harris, Russ Housley, Hubert Kario, John Mattsson, Eric Mill, Yoav Nir, Andrei Popov, Eric Rescorla, Yaron Sheffer, Robert Sparks, Martin Thomson, Loganaden Velvindron, https://github.com/yaleman, and Jakub Wilk.

[[Note to RFC editor: At least Julien Elie’s name above should have an accent on the first letter of the surname. Please fix that and any others needing a similar fix if you can, I’m not sure the tooling I have now allows that.]]

9. IANA Considerations

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

10. References

10.1. Normative References


10.2. Informative References


Internet-Draft       Deprecating TLSv1.0 and TLSv1.1           June 2019


Appendix A. Change Log

[ RFC editor: please remove this before publication. ]

From draft-ietf-tls-oldversions-deprecate-04 to draft-ietf-tls-oldversions-deprecate-05:

- Removed references to government related deprecation statements: US, Canada, and Germany. NIST documentation rationale remains as a reference describing the relevant RFCs and justification.

From draft-ietf-tls-oldversions-deprecate-02 to draft-ietf-tls-oldversions-deprecate-03:

- Added 8261 to updates list based on IETF-104 meeting.

From draft-ietf-tls-oldversions-deprecate-01 to draft-ietf-tls-oldversions-deprecate-02:

- Correction: 2nd list of referenced RFCs in Section 1.1 aren’t informatively referring to tls1.0/1.1
- Remove RFC7255 from updates list - datatracker has bad data (spotted by Robert Sparks)
- Added point about RFCs 8143 and 4642
- Added UPDATEs for RFCs that refer to 4347 and aren’t OBSOLETED
- Added note about RFC8261 to see what WG want.

From draft-ietf-tls-oldversions-deprecate-00 to draft-ietf-tls-oldversions-deprecate-01:

- PRs with typos and similar: so far just #1
- PR#2 noting msft browser announced deprecation (but this was OBE as per...)
- Implemented actions as per IETF-103 meeting:
  * Details about which RFC’s, BCP’s are affected were generated using a script in the git repo: https://github.com/tlswg/oldversions-deprecate/blob/master/nonobsnorms.sh
  * Removed the ‘measurements’ part
  * Removed SHA-1 deprecation (section 8 of -00)

From draft-moriarty-tls-oldversions-diediedie-01 to draft-ietf-tls-oldversions-deprecate-00:

- I-Ds became RFCs 8446/8447 (old-repo PR#4, for TLSv1.3)
- Accepted old-repo PR#5 fixing typos
From draft-moriarty-tls-oldversions-diediedie-00 to draft-moriarty-tls-oldversions-diediedie-01:

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

Authors’ Addresses

Kathleen Moriarty
Dell EMC
176 South Street
Hopkinton
United States
EMail: Kathleen.Moriarty.ietf@gmail.com

Stephen Farrell
Trinity College Dublin
Dublin 2
Ireland
Phone: +353-1-896-2354
EMail: stephen.farrell@cs.tcd.ie
Abstract

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

Status of This Memo

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

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

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

This Internet-Draft will expire on January 9, 2020.

Copyright Notice

Copyright (c) 2019 IETF Trust and the persons identified as the document authors. All rights reserved.
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) [RFC8446] [RFC5280]. This organizational separation makes the TLS server operator dependent on the CA for some aspects of its operations, for example:

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

- The 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 (i.e., using the Certificate Status extension types ocsp [RFC6066] or ocsp_multi [RFC6961]), 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 peer 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 CA’s 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", or "delegation certificate", and the one issued by the operator as a "delegated credential" or "DC".

1.1. Change Log

(*) indicates changes to the wire protocol.

draft-04
  o Add support for client certificates.

draft-03
  o Remove protocol version from the Credential structure. (*)

draft-02
  o Change public key type. (*)
  o Change DelegationUsage extension to be NULL and define its object identifier.
  o Drop support for TLS 1.2.
2. Solution Overview

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

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

- The initiating peer provides an extension in its ClientHello or CertificateRequest that indicates support for this mechanism.
- The peer sending the Certificate message provides both the certificate chain terminating in its certificate as well as the delegated credential.
- The authenticating initiator uses information from the peer’s certificate to verify the delegated credential and that the peer is asserting an expected identity.
- Peers accepting the delegated credential use it as the certificate’s working key for the TLS handshake.

As detailed in Section 3, the delegated credential is cryptographically bound to the end-entity certificate with which the credential may be used. This document specifies the use of delegated credentials in TLS 1.3 or later; their use in prior versions of the protocol is not allowed.

Delegated credentials allow a peer 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 exposure in case a delegated credential is compromised, peers may not issue credentials with a validity period longer than 7 days. This mechanism is described in detail in Section 3.1.
It was noted in [XPROT] that certificates in use by servers that support outdated protocols such as SSLv2 can be used to forge signatures for certificates that contain the keyEncipherment KeyUsage ([RFC5280] section 4.2.1.3). In order to prevent this type of cross-protocol attack, we define a new DelegationUsage extension to X.509 that permits use of delegated credentials. (See Section 3.2.)

2.1. Rationale

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

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

- X.509 semantics are very rich. This can cause unintended consequences if a service owner creates a proxy certificate where the properties differ from the leaf certificate. For this reason, delegated credentials have very restricted semantics that should not conflict with X.509 semantics.

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

- Each delegated credential is bound to a specific signature algorithm that may be used to sign the TLS handshake ([RFC8446] section 4.2.3). This prevents them from being used with other, perhaps unintended signature algorithms.

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---->
<---CertVerify----->

...<------LURK------->

Delegated credentials:

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

...<----DC minting---->

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. 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 KeyUsage (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. The credential has the following structure:
struct {
    uint32 valid_time;
    SignatureScheme expected_cert_verify_algorithm;
    opaque ASN1_subjectPublicKeyInfo<1..2^24-1>;
} Credential;

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

expected_cert_verify_algorithm: The signature algorithm of the
credential key pair, where the type SignatureScheme is as defined
in [RFC8446]. This is expected to be the same as
CertificateVerify.algorithm sent by the server.

ASN1_subjectPublicKeyInfo: The credential’s public key, a DER-
encoded [X690] SubjectPublicKeyInfo as defined in [RFC5280].

The delegated credential has the following structure:

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

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

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

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

1. A string that consists of octet 32 (0x20) repeated 64 times.
2. The context string "TLS, server delegated credentials" for
   servers and "TLS, client delegated credentials" for clients.
3. A single 0 byte, which serves as the separator.
4. The DER-encoded X.509 end-entity certificate used to sign the
   DelegatedCredential.
5. DelegatedCredential.cred.
6. DelegatedCredential.algorithm.

The signature effectively binds the credential to the parameters of
the handshake in which it is used. In particular, it ensures that
credentials are only used with the certificate and signature
algorithm chosen by the delegator. Minimizing their semantics in
this way is intended to mitigate the risk of cross protocol attacks
involving delegated credentials.

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

3.1. Client and Server behavior

This document defines the following extension code point.

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

3.1.1. Server authentication

A client which supports this specification SHALL send an empty
"delegated_credential" extension in its ClientHello. If the client
receives a delegated credential without indicating support, then the
client MUST abort with an "unexpected_message" alert.

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

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

The algorithm and expected_cert_verify_algorithm fields MUST be of a
type advertised by the client in the "signature_algorithms" extension
and are considered invalid otherwise. Clients that receive invalid
delegated credentials MUST terminate the connection with an
"illegal_parameter" alert.
3.1.2. Client authentication

A server which supports this specification SHALL send an empty "delegated_credential" extension in the CertificateRequest message when requesting client authentication. If the server receives a delegated credential without indicating support in its CertificateRequest, then the server MUST abort with an "unexpected_message" alert.

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

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

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

3.1.3. Validating a Delegated Credential

On receiving a delegated credential and a certificate chain, the peer validates the certificate chain and matches the end-entity certificate to the peer's expected identity in the usual way. It also takes the following steps:

1. 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. This is done by asserting that the current time is no more than the delegation certificate’s notBefore value plus DelegatedCredential.cred.valid_time.

2. Verify that expected_cert_verify_algorithm matches the scheme indicated in the peer’s CertificateVerify message.

3. Verify that the end-entity certificate satisfies the conditions in Section 3.2.

4. Use the public key in the peer’s end-entity certificate to verify the signature of the credential using the algorithm indicated by DelegatedCredential.algorithm.
If one or more of these checks fail, then the delegated credential is deemed invalid. Clients and servers that receive invalid delegated credentials MUST terminate the connection with an "illegal_parameter" alert. If successful, the participant receiving the Certificate message uses the public key in the credential to verify the signature in the peer’s CertificateVerify message.

3.2. 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.

```
id-ce-delegationUsage OBJECT IDENTIFIER ::=  { 1.3.6.1.4.1.44363.44 }
DelegationUsage ::= NULL
```

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

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

4. IANA Considerations

This document registers the "delegated_credentials" extension in the "TLS ExtensionType Values" registry. The "delegated_credentials" extension has been assigned a code point of TBD. The IANA registry lists this extension as "Recommended" (i.e., "Y") and indicates that it may appear in the ClientHello (CH), CertificateRequest (CR), or Certificate (CT) messages in TLS 1.3 [RFC8446].

5. Security Considerations

5.1. 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, including
file system controls, physical security, or hardware security modules.

5.2. Re-use of delegated credentials in multiple contexts

It is possible to use the same delegated credential for both client and server authentication if the Certificate allows it. This is safe because the context string used for delegated credentials is distinct in both contexts.

5.3. Revocation of delegated credentials

Delegated credentials do not provide any additional form of early revocation. Since it is short lived, the expiry of the delegated credential would revoke the credential. Revocation of the long term private key that signs the delegated credential also implicitly revokes the delegated credential.

5.4. 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.

6. Acknowledgements

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

7. References

7.1. Normative References

7.2. Informative References

[I-D.ietf-acme-acme]

[I-D.ietf-tls-tls13]

[I-D.mglt-lurk-tls-requirements]


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
Abstract

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

Status of This Memo

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

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

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

This Internet-Draft will expire on December 2, 2019.

Copyright Notice

Copyright (c) 2019 IETF Trust and the persons identified as the document authors. All rights reserved.

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

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

2. Terminology

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

3. Motivation and Design Rationale

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

The TLS 1.3 handshake protocol employs key agreement algorithms that could be broken by the invention of a large-scale quantum computer [I-D.hoffman-c2pq]. These algorithms include Diffie-Hellman (DH) [DH1977] 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
4. Extension Overview

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

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

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

The successful negotiation of the "tls_cert_withExtern_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, which is required by Section 4.2.11 of [RFC8446]. The hash algorithm MUST be set when the PSK is established, with a default of SHA-256 if no hash algorithm is specified during establishment.
5. Certificate with External PSK Extension

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

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

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

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

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

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

This document specifies the "tls_cert_withExternPSK" extension, adding one new type to ExtensionType:
enum {
    tls_cert_withExtern_psk(TBD), (65535)
} ExtensionType;

The "tls_cert_withExtern_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. The "tls_cert_withExtern_psk"
extension is essentially a flag to use the external PSK in the key
schedule, and it has the following syntax:

struct {
    select (Handshake.msg_type) {
        case client_hello: Empty;
        case server_hello: Empty;
    }
} CertWithExternPSK;

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

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

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

The "supported_groups" extension is defined in Section 4.2.7 of
[RFC8446].
The "pre_shared_key" extension is defined in Section 4.2.11 of [RFC8446]. The syntax is repeated below for convenience. All of the listed PSKs MUST be external PSKs.

```c
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 associated identities may be known to other parties as well. In addition, the binder validation (see below) confirms that the client and server have the same key associated with the identity.

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

The binders are a series of HMAC values, one for each external PSK offered by the client, in the same order as the identities list. The HMAC value is computed using the binder_key, which is derived from the external PSK, and a partial transcript of the current handshake. Generation of the binder_key from the external PSK is described in Section 7.1 of [RFC8446]. The partial transcript of the current handshake includes a partial ClientHello up to and including the PreSharedKeyExtension identities field as described in Section 4.2.11.2 of [RFC8446].
The selected_identity contains the external PSK identity that the server selected from the list offered by the client. If none of the offered external PSKs in the list provided by the client are acceptable to the server, then the "tls_cert_withExtern_psk" extension MUST be omitted from the ServerHello message. The server MUST validate the binder value that corresponds to the selected external PSK as described in Section 4.2.11.2 of [RFC8446]. If the binder does not validate, the server MUST abort the handshake with an "illegal_parameter" alert. Servers SHOULD NOT attempt to validate multiple binders; rather they SHOULD select one of the offered external PSKs and validate only the binder that corresponds to that external PSK.

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

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

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

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

6. IANA Considerations

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

The Security Considerations in [RFC8446] remain relevant.

TLS 1.3 [RFC8446] does not permit the server to send a CertificateRequest message when a PSK is being used. This restriction is removed when the "tls_cert_withExtern_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 generate 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.

If the external PSK is known to any party other than the client and the server, then the external PSK MUST NOT be the sole basis for authentication. The reasoning is explained in [K2016] (see Section 4.2). When this extension is used, authentication is based on certificates, not the external PSK.

In this extension, the external PSK preserves secrecy if the (EC)DH key agreement is ever broken by cryptanalysis or the future invention of a large-scale quantum computer. As long as the attacker does not know the PSK and the key derivation algorithm remains unbroken, the attacker cannot derive the session secrets even if they are able to compute the (EC)DH shared secret.
TLS 1.3 key derivation makes use of the HKDF algorithm, which depends upon the HMAC construction and a hash function. This extension provides the desired protection for the session secrets as long as HMAC with the selected hash function is a pseudorandom function (PRF) [GGM1986].

In the future, if the (EC)DH key agreement is broken by cryptanalysis or the invention of a large-scale quantum computer, the forward secrecy advantages traditionally associated with ephemeral (EC)DH keys will no longer be relevant. Although the ephemeral private keys used during a given TLS session would be destroyed at the end of a session, preventing the attacker from later accessing them, the private keys would nevertheless be recoverable due to the break in the algorithm. A more general notion of "secrecy after key material is destroyed" would still be achievable using external PSKs, of course, provided that they are managed in a way that ensures their destruction when they are no longer needed, and with the assumption that the algorithms that use the external PSKs remain quantum-safe.

This specification does not require that external PSK is known only by the client and server. The external PSK may be known to a group. Since authentication depends on the public key in a certificate, knowledge of the external PSK by other parties does not enable impersonation. Since confidentiality depends on the shared secret from (EC)DH, knowledge of the external PSK by other parties does not enable eavesdropping. However, group members can record the traffic of other members, and then decrypt it if they ever gain access to a large-scale quantum computer. Also, when many parties know the external PSK, there are many opportunities for theft of the external PSK by an attacker. Once an attacker has the external PSK, they can decrypt stored traffic if they ever gain access to a large-scale quantum computer in the same manner as a legitimate group member.

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

TLS 1.3 [RFC8446] has received careful security analysis, and some informal reasoning shows that the addition of this extension does not
introduce any security defects. This extension requires the use of certificates for authentication, but the processing of certificates is unchanged by this extension. This extension places an external PSK in the key schedule as part of the computation of the Early Secret. In the initial handshake without this extension, the Early Secret is computed as:

\[
\text{Early Secret} = \text{HKDF-Extract}(0, 0)
\]

With this extension, the Early Secret is computed as:

\[
\text{Early Secret} = \text{HKDF-Extract}(\text{External PSK}, 0)
\]

Any entropy contributed by the external PSK can only make the Early Secret better; the External PSK cannot make it worse. For these two reasons, TLS 1.3 continues to meet its security goals when this extension is used.

8. Privacy Considerations

Appendix E.6 of [RFC8446] discusses identity exposure attacks on PSKs. The guidance in this section remains relevant.

This extension makes use of external PSKs to improve resilience against attackers that gain access to a large-scale quantum computer in the future. This extension is always accompanied by the "pre_shared_key" extension to provide the PSK identities in plaintext in the ClientHello message. Passive observation of the these PSK identities will aid an attacker to track users of this extension.

9. Acknowledgments

Many thanks to Liliya Akhmetzyanova, Christian Huitema, Geoffrey Keating, Hugo Krawczyk, Nikos Mavrogiannopoulos, Nick Sullivan, Martin Thomson, and Peter Yee for their review and comments; their efforts have improved this document.

10. References

10.1. Normative References


Housley Expires December 2, 2019 [Page 10]
10.2. Informative References


Author’s Address

Russ Housley
Vigil Security, LLC
516 Dranesville Road
Herndon, VA 20170
USA

Email: housley@vigilsec.com
Abstract

This document describes a TLS 1.3 extension that can be by clients to request their public network address from a server. This information can be used for a variety of purposes, including: NAT detection, ASN identification, and privacy-driven transport protocol features.

Status of This Memo

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

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

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

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

Copyright Notice

Copyright (c) 2019 IETF Trust and the persons identified as the document authors. All rights reserved.

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

This document describes a TLS 1.3 [RFC8446] extension that can be by clients to request their public network address from a server. This has several uses, including: NAT detection, ASN identification, and privacy-driven transport protocol features. Servers that support this extension can send the perceived client address to clients. The latter may then confirm whether or not this representation matches their known public address.

Unlike the related NAT detection extension for IKE [RFC3947], clients do not send their perceived IP address to servers, even in an obfuscated form. Doing so would introduce an unwanted privacy regression for clients.

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. Client Network Address Use Cases

Knowledge of a public client network address can serve several purposes. This extension allows clients to detect the presence of a NAT or other address-transforming proxy involved in a TLS connection. The following sections describe several uses for this information.
2.1. Connection Lifetime Optimizations

Middleboxes such as NATs typically have short lifetimes for connection state. Detecting such middleboxes may help influence client connection management logic, such as the use of keep-alive messages.

Since NATs often apply to all traffic from an endhost, detection via a TLS connection may assist other non-TLS and non-TCP connections that can be more sensitive to NAT timeouts.

2.2. Privacy Stance Enhancements

Address-transforming proxies such as NATs may improve communication privacy by masking the public IP address of clients in a session. Modulo other cleartext signals such as session identifiers, the anonymity set of a connection passing through a NAT is proportional to the number of clients serviced by the NAT. Absent NAT detection, clients cannot determine if their connections are linkable via IP-layer information, such as stable source addresses. As a result, clients cannot determine if privacy-driven policies such as never resuming TLS connections improve privacy.

If clients can detect NATs, they can make informed decisions about connection reuse. As a motivating example, consider DNS-over-TLS [RFC7858][RFC8310]. Privacy-sensitive clients may wish to use fresh connections for individual queries so as to not allow recursive resolvers the ability of building client query histories. However, in the absence of a NAT, reusing a connection does not pose a significant privacy regression since such clients are generally identifiable by their IP address.

Client network awareness may also influence privacy-driven connection migration policies, such as those prescribed by QUIC [I-D.ietf-quic-transport]. For example, if clients know they are not behind a NAT, then connection ID rotation serves little value in preventing linkability.

2.3. Metric Collections

Clients may passively use their public address discovered via TLS to identify their corresponding ASN without the use of explicit probes.

3. Network Address Extension

Servers may send the perceived client IP address to its peer using the following "network_address" extension:
enum {
    network_address(TBD),  (65535)
} ExtensionType;

When sent by a client, this extension MUST be empty. A server which receives a non-empty network_address extension MUST terminate the connection with an "Illegal Parameter" alert.

Supporting servers which receive this extension may respond with a "network_address" extension, shown below, inside the EncryptedExtensions.

struct {
    opaque address<32..255>;
} NetworkAddress;

despite The client’s perceived address.

In this case, NetworkAddress.address carries the raw network-order byte-wise representation of the client IP address. (Since the extension is encrypted, there is no need to obfuscate the address for transit.) Clients which receive a non-empty NetworkAddress extension may use it to record their public IP address. Clients MUST treat empty NetworkAddress.address extensions as an error and send an Illegal Parameter alert in response.

4. IANA Considerations

IANA is requested to Create an entry, network_address(TBD), in the existing registry for ExtensionType (defined in [RFC8446]), with "TLS 1.3" column values being set to "CH, EE", and "Recommended" column being set to "Yes".

5. Security Considerations

Since NetworkAddress extension contents are encrypted, this extension introduces no (known) additional security or privacy issues.

An earlier design let clients send their address to servers in an obfuscated form, e.g., by hashing the client’s perceived IP address with ClientHello.random, so that servers could measure whether or not clients were also behind NATs. However, such obfuscation mechanisms are subject to dictionary attacks and therefore could be used by malicious on-path attackers to learn a client’s true public address. Absent this information, there are no explicit signals from a single (non-resumed) TLS connection that such attackers can use to learn the client’s public address.
In general, absent a mechanism to encrypt the client extensions, sending the client’s perceived address in any form therefore constitutes a privacy regression.

6. Normative References


Authors' Addresses

Eric Kinnear
Apple Inc.
One Apple Park Way
Cupertino, California 95014
United States of America

Email: ekinnear@apple.com
Abstract

Hybrid key exchange refers to using multiple key exchange algorithms simultaneously and combining the result with the goal of providing security even if all but one of the component algorithms is broken, and is motivated by transition to post-quantum cryptography. This document categorizes various design considerations for using hybrid key exchange in the Transport Layer Security (TLS) protocol version 1.3 and outlines two concrete instantiations for consideration.

Status of This Memo

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

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

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

This Internet-Draft will expire on January 9, 2020.

Copyright Notice

Copyright (c) 2019 IETF Trust and the persons identified as the document authors. All rights reserved.

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

Table of Contents

1. Introduction ............................................. 3
   1.1. Revision history ..................................... 3
   1.2. Terminology ......................................... 4
   1.3. Motivation for use of hybrid key exchange ............ 4
   1.4. Scope .............................................. 5
   1.5. Goals .............................................. 5
   1.6. Related work ....................................... 7
2. Overview ............................................... 8
3. Design options ......................................... 10
   3.1. (Neg) How to negotiate hybridization and component
        algorithms? ........................................ 10
      3.1.1. Key exchange negotiation in TLS 1.3 .......... 10
      3.1.2. (Neg-Ind) Negotiating component algorithms
               individually .................................. 10
      3.1.3. (Neg-Comb) Negotiating component algorithms as a
               combination ................................... 11
      3.1.4. Benefits and drawbacks ......................... 12
   3.2. (Num) How many component algorithms to combine? .... 13
      3.2.1. (Num-2) Two ................................... 13
      3.2.2. (Num-2+) Two or more ........................ 13
      3.2.3. Benefits and Drawbacks ......................... 13
   3.3. (Shares) How to convey key shares? .................. 13
      3.3.1. (Shares-Concat) Concatenate key shares ........ 13
      3.3.2. (Shares-Multiple) Send multiple key shares .... 14
      3.3.3. (Shares-Ext-Additional) Extension carrying additional
               key shares .................................... 14
      3.3.4. Benefits and Drawbacks ......................... 14
   3.4. (Comb) How to use keys? ............................ 14
      3.4.1. (Comb-Concat) Concatenate keys ................ 15
      3.4.2. (Comb-KDF-1) KDF keys ........................ 16
      3.4.3. (Comb-KDF-2) KDF keys ........................ 17
      3.4.4. (Comb-XOR) XOR keys ........................... 18
      3.4.5. (Comb-Chain) Chain of KDF applications for each key . 18
      3.4.6. (Comb-AltInput) Second shared secret in an alternate
              KDF input ..................................... 19
      3.4.7. Benefits and Drawbacks ......................... 20
      3.4.8. Open questions ................................. 21
4. Candidate instantiations ................................. 21
   4.1. Candidate Instantiation 1 .......................... 21
      4.1.1. ClientHello extension supported_groups .......... 22
1. Introduction

This document categorizes various design decisions one could make when implementing hybrid key exchange in TLS 1.3, with the goal of fostering discussion, providing options for short-term prototypes/experiments, and serving as a basis for eventual standardization. This document also includes two concrete instantiations for consideration, following two different approaches; it is not our intention that both be standardized.

This document does not propose specific post-quantum mechanisms; see Section 1.4 for more on the scope of this document.

Comments are solicited and should be addressed to the TLS working group mailing list at tls@ietf.org and/or the author(s).

1.1. Revision history

- draft-00: Initial version.
- draft-01:
  - Add (Comb-KDF-1) (Section 3.4.2) and (Comb-KDF-2) (Section 3.4.3) options.
  - Add Candidate Instantiation 1 (Section 4.1).
1.2. Terminology

For the purposes of this document, it is helpful to be able to divide cryptographic algorithms into two classes:

- "Traditional" algorithms: Algorithms which are widely deployed today, but which may be deprecated in the future. In the context of TLS 1.3 in 2019, examples of traditional key exchange algorithms include elliptic curve Diffie-Hellman using secp256r1 or x25519, or finite-field Diffie-Hellman.

- "Next-generation" (or "next-gen") algorithms: Algorithms which are not yet widely deployed, but which may eventually be widely deployed. An additional facet of these algorithms may be that we have less confidence in their security due to them being relatively new or less studied. This includes "post-quantum" algorithms.

"Hybrid" key exchange, in this context, means the use of two (or more) key exchange mechanisms based on different cryptographic assumptions (for example, one traditional algorithm and one next-gen algorithm), with the purpose of the final session key being secure as long as at least one of the component key exchange mechanisms remains unbroken. We use the term "component" algorithms to refer to the algorithms that are being combined in a hybrid key exchange.

The primary motivation of this document is preparing for post-quantum algorithms. However, it is possible that public key cryptography based on alternative mathematical constructions will be required independent of the advent of a quantum computer, for example because of a cryptanalytic breakthrough. As such we opt for the more generic term "next-generation" algorithms rather than exclusively "post-quantum" algorithms.

1.3. Motivation for use of hybrid key exchange

Ideally, one would not use hybrid key exchange: one would have confidence in a single algorithm and parameterization that will stand the test of time. However, this may not be the case in the face of quantum computers and cryptanalytic advances more generally.

Many (but not all) of the post-quantum algorithms currently under consideration are relatively new; they have not been subject to the same depth of study as RSA and finite-field / elliptic curve Diffie-Hellman, and thus we do not necessarily have as much confidence in
their fundamental security, or the concrete security level of specific parameterizations.

Early adopters eager for post-quantum security may want to use hybrid key exchange to have the potential of post-quantum security from a less-well-studied algorithm while still retaining at least the security currently offered by traditional algorithms. (They may even need to retain traditional algorithms due to regulatory constraints, for example FIPS compliance.)

Moreover, it is possible that even by the end of the NIST Post-Quantum Cryptography Standardization Project, and for a period of time thereafter, conservative users may not have full confidence in some algorithms.

As such, there may be users for whom hybrid key exchange is an appropriate step prior to an eventual transition to next-generation algorithms.

1.4. Scope

This document focuses on hybrid ephemeral key exchange in TLS 1.3 [TLS13]. It intentionally does not address:

- Selecting which next-generation algorithms to use in TLS 1.3, nor algorithm identifiers nor encoding mechanisms for next-generation algorithms. (The outcomes of the NIST Post-Quantum Cryptography Standardization Project [NIST] will inform this choice.)

- Authentication using next-generation algorithms. (If a cryptographic assumption is broken due to the advent of a quantum computer or some other cryptanalytic breakthrough, confidentiality of information can be broken retroactively by any adversary who has passively recorded handshakes and encrypted communications. But session authentication cannot be retroactively broken.)

1.5. Goals

The primary goal of a hybrid key exchange mechanism is to facilitate the establishment of a shared secret which remains secure as long as one of the component key exchange mechanisms remains unbroken.

In addition to the primary cryptographic goal, there may be several additional goals in the context of TLS 1.3:

- "Backwards compatibility:" Clients and servers who are "hybrid-aware", i.e., compliant with whatever hybrid key exchange standard is developed for TLS, should remain compatible with endpoints and
middle-boxes that are not hybrid-aware. The three scenarios to consider are:

1. Hybrid-aware client, hybrid-aware server: These parties should establish a hybrid shared secret.

2. Hybrid-aware client, non-hybrid-aware server: These parties should establish a traditional shared secret (assuming the hybrid-aware client is willing to downgrade to traditional-only).

3. Non-hybrid-aware client, hybrid-aware server: These parties should establish a traditional shared secret (assuming the hybrid-aware server is willing to downgrade to traditional-only).

Ideally backwards compatibility should be achieved without extra round trips and without sending duplicate information; see below.

- *High performance:* Use of hybrid key exchange should not be prohibitively expensive in terms of computational performance. In general this will depend on the performance characteristics of the specific cryptographic algorithms used, and as such is outside the scope of this document. See [BCNS15], [CECPQ1], [FRODO] for preliminary results about performance characteristics.

- *Low latency:* Use of hybrid key exchange should not substantially increase the latency experienced to establish a connection. Factors affecting this may include the following.
  
  * The computational performance characteristics of the specific algorithms used. See above.

  * The size of messages to be transmitted. Public key / ciphertext sizes for post-quantum algorithms range from hundreds of bytes to over one hundred kilobytes, so this impact can be substantially. See [BCNS15], [FRODO] for preliminary results in a laboratory setting, and [LANGLEY] for preliminary results on more realistic networks.

  * Additional round trips added to the protocol. See below.

- *No extra round trips:* Attempting to negotiate hybrid key exchange should not lead to extra round trips in any of the three hybrid-aware/non-hybrid-aware scenarios listed above.
1.6. Related work

Quantum computing and post-quantum cryptography in general are outside the scope of this document. For a general introduction to quantum computing, see a standard textbook such as [NIELSEN]. For an overview of post-quantum cryptography as of 2009, see [BERNSTEIN]. For the current status of the NIST Post-Quantum Cryptography Standardization Project, see [NIST]. For additional perspectives on the general transition from classical to post-quantum cryptography, see for example [ETSI] and [HOFFMAN], among others.

There have been several Internet-Drafts describing mechanisms for embedding post-quantum and/or hybrid key exchange in TLS:

- Internet-Drafts for TLS 1.2: [WHYTE12]
- Internet-Drafts for TLS 1.3: [KIEFER], [SCHANCK], [WHYTE13]

There have been several prototype implementations for post-quantum and/or hybrid key exchange in TLS:

- Experimental implementations in TLS 1.2: [BCNS15], [CECPQ1], [FRODO], [OQS-102]
- Experimental implementations in TLS 1.3: [CECPQ2], [OQS-111]

These experimental implementations have taken an ad hoc approach and not attempted to implement one of the drafts listed above.

Unrelated to post-quantum but still related to the issue of combining multiple types of keying material in TLS is the use of pre-shared keys, especially the recent TLS working group document on including an external pre-shared key [EXTERN-PSK].

Considering other IETF standards, there is work on post-quantum preshared keys in IKEv2 [IKE-PSK] and a framework for hybrid key exchange in IKEv2 [IKE-HYBRID]. The XMSS hash-based signature scheme has been published as an informational RFC by the IRTF [XMSS].

In the academic literature, [EVEN] initiated the study of combining multiple symmetric encryption schemes; [ZHANG], [DODIS], and [HARNIK] examined combining multiple public key encryption schemes, and [HARNIK] coined the term "robust combiner" to refer to a compiler that constructs a hybrid scheme from individual schemes while
preserving security properties. [GIACON] and [BINDEL] examined combining multiple key encapsulation mechanisms.

2. Overview

We identify four distinct axes along which one can make choices when integrating hybrid key exchange into TLS 1.3:

1. How to negotiate the use of hybridization in general and component algorithms specifically?
2. How many component algorithms can be combined?
3. How should multiple key shares (public keys / ciphertexts) be conveyed?
4. How should multiple shared secrets be combined?

The remainder of this document outlines various options we have identified for each of these choices. Immediately below we provide a summary list. Options are labelled with a short code in parentheses to provide easy cross-referencing.

1. (Neg) (Section 3.1) How to negotiate the use of hybridization in general and component algorithms specifically?

   * (Neg-Ind) (Section 3.1.2) Negotiating component algorithms individually

      + (Neg-Ind-1) (Section 3.1.2.1) Traditional algorithms in "ClientHello" "supported_groups" extension, next-gen algorithms in another extension

      + (Neg-Ind-2) (Section 3.1.2.2) Both types of algorithms in "supported_groups" with external mapping to tradition/next-gen.

      + (Neg-Ind-3) (Section 3.1.2.3) Both types of algorithms in "supported_groups" separated by a delimiter.

   * (Neg-Comb) (Section 3.1.3) Negotiating component algorithms as a combination

      + (Neg-Comb-1) (Section 3.1.3.1) Standardize "NamedGroup" identifiers for each desired combination.
+  (Neg-Comb-2) (Section 3.1.3.2) Use placeholder identifiers in "supported_groups" with an extension defining the combination corresponding to each placeholder.

+  (Neg-Comb-3) (Section 3.1.3.3) List combinations by inserting grouping delimiters into "supported_groups" list.

2.  (Num) (Section 3.2) How many component algorithms can be combined?
   *  (Num-2) (Section 3.2.1) Two.
   *  (Num-2+) (Section 3.2.2) Two or more.

3.  (Shares) (Section 3.3) How should multiple key shares (public keys / ciphertexts) be conveyed?
   *  (Shares-Concat) (Section 3.3.1) Concatenate each combination of key shares.
   *  (Shares-Multiple) (Section 3.3.2) Send individual key shares for each algorithm.
   *  (Shares-Ext-Additional) (Section 3.3.3) Use an extension to convey key shares for component algorithms.

4.  (Comb) (Section 3.4) How should multiple shared secrets be combined?
   *  (Comb-Concat) (Section 3.4.1) Concatenate the shared secrets then use directly in the TLS 1.3 key schedule.
   *  (Comb-KDF-1) (Section 3.4.2) and (Comb-KDF-2) (Section 3.4.3) KDF the shared secrets together, then use the output in the TLS 1.3 key schedule.
   *  (Comb-XOR) (Section 3.4.4) XOR the shared secrets then use directly in the TLS 1.3 key schedule.
   *  (Comb-Chain) (Section 3.4.5) Extend the TLS 1.3 key schedule so that there is a stage of the key schedule for each shared secret.
   *  (Comb-AltInput) (Section 3.4.6) Use the second shared secret in an alternate (otherwise unused) input in the TLS 1.3 key schedule.
3. Design options

3.1. (Neg) How to negotiate hybridization and component algorithms?

3.1.1. Key exchange negotiation in TLS 1.3

Recall that in TLS 1.3, the key exchange mechanism is negotiated via the "supported_groups" extension. The "NamedGroup" enum is a list of standardized groups for Diffie-Hellman key exchange, such as "secp256r1", "x25519", and "ffdhe2048".

The client, in its "ClientHello" message, lists its supported mechanisms in the "supported_groups" extension. The client also optionally includes the public key of one or more of these groups in the "key_share" extension as a guess of which mechanisms the server might accept in hopes of reducing the number of round trips.

If the server is willing to use one of the client’s requested mechanisms, it responds with a "key_share" extension containing its public key for the desired mechanism.

If the server is not willing to use any of the client’s requested mechanisms, the server responds with a "HelloRetryRequest" message that includes an extension indicating its preferred mechanism.

3.1.2. (Neg-Ind) Negotiating component algorithms individually

In these three approaches, the parties negotiate which traditional algorithm and which next-gen algorithm to use independently. The "NamedGroup" enum is extended to include algorithm identifiers for each next-gen algorithm.

3.1.2.1. (Neg-Ind-1)

The client advertises two lists to the server: one list containing its supported traditional mechanisms (e.g. via the existing "ClientHello" "supported_groups" extension), and a second list containing its supported next-generation mechanisms (e.g., via an additional "ClientHello" extension). A server could then select one algorithm from the traditional list, and one algorithm from the next-generation list. (This is the approach in [SCHANCK].)

3.1.2.2. (Neg-Ind-2)

The client advertises a single list to the server which contains both its traditional and next-generation mechanisms (e.g., all in the existing "ClientHello" "supported_groups" extension), but with some external table provides a standardized mapping of those mechanisms as
either "traditional" or "next-generation". A server could then select two algorithms from this list, one from each category.

3.1.2.3. (Neg-Ind-3)

The client advertises a single list to the server delimited into sublists: one for its traditional mechanisms and one for its next-generation mechanisms, all in the existing "ClientHello" "supported_groups" extension, with a special code point serving as a delimiter between the two lists. For example, "supported_groups = secp256r1, x25519, delimiter, nextgen1, nextgen4".

3.1.3. (Neg-Comb) Negotiating component algorithms as a combination

In these three approaches, combinations of key exchange mechanisms appear as a single monolithic block; the parties negotiate which of several combinations they wish to use.

3.1.3.1. (Neg-Comb-1)

The "NamedGroup" enum is extended to include algorithm identifiers for each *combination* of algorithms desired by the working group. There is no "internal structure" to the algorithm identifiers for each combination, they are simply new code points assigned arbitrarily. The client includes any desired combinations in its "ClientHello" "supported_groups" list, and the server picks one of these. This is the approach in [KIEFER] and [OQS-111].

3.1.3.2. (Neg-Comb-2)

The "NamedGroup" enum is extended to include algorithm identifiers for each next-gen algorithm. Some additional field/extension is used to convey which combinations the parties wish to use. For example, in [WHYTE13], there are distinguished "NamedGroup" called "hybrid_marker 0", "hybrid_marker 1", "hybrid_marker 2", etc. This is complemented by a "HybridExtension" which contains mappings for each numbered "hybrid_marker" to the set of component key exchange algorithms (2 or more) for that proposed combination.

3.1.3.3. (Neg-Comb-3)

The client lists combinations in "supported_groups" list, using a special delimiter to indicate combinations. For example, "supported_groups = combo_delimiter, secp256r1, nextgen1, combo_delimiter, secp256r1, nextgen4, standalone_delimiter, secp256r1, x25519" would indicate that the client’s highest preference is the combination secp256r1+nextgen1, the next highest preference is the combination secp2561+nextgen4, then the single
algorithm secp256r1, then the single algorithm x25519. A hybrid-aware server would be able to parse these; a hybrid-unaware server would see "unknown, secp256r1, unknown, unknown, secp256r1, unknown, unknown, secp256r1, x25519", which it would be able to process, although there is the potential that every "projection" of a hybrid list that is tolerable to a client does not result in list that is tolerable to the client.

3.1.4. Benefits and drawbacks

*Combinatorial explosion.* (Neg-Comb-1) (Section 3.1.3.1) requires new identifiers to be defined for each desired combination. The other 4 options in this section do not.

*Extensions.* (Neg-Ind-1) (Section 3.1.2.1) and (Neg-Comb-2) (Section 3.1.3.2) require new extensions to be defined. The other options in this section do not.

*New logic.* All options in this section except (Neg-Comb-1) (Section 3.1.3.1) require new logic to process negotiation.

*Matching security levels.* (Neg-Ind-1) (Section 3.1.2.1), (Neg-Ind-2) (Section 3.1.2.2), (Neg-Ind-3) (Section 3.1.2.3), and (Neg-Comb-2) (Section 3.1.3.2) allow algorithms of different claimed security level from their corresponding lists to be combined. For example, this could result in combining ECDH secp256r1 (classical security level 128) with NewHope-1024 (classical security level 256). Implementations dissatisfied with a mismatched security levels must either accept this mismatch or attempt to renegotiate. (Neg-Ind-1) (Section 3.1.2.1), (Neg-Ind-2) (Section 3.1.2.2), and (Neg-Ind-3) (Section 3.1.2.3) give control over the combination to the server; (Neg-Comb-2) (Section 3.1.3.2) gives control over the combination to the client. (Neg-Comb-1) (Section 3.1.3.1) only allows standardized combinations, which could be set by TLS working group to have matching security (provided security estimates do not evolve separately).

*Backwards-compability.* TLS 1.3-compliant hybrid-unaware servers should ignore unrecognized elements in "supported_groups" (Neg-Ind-2) (Section 3.1.2.2), (Neg-Ind-3) (Section 3.1.2.3), (Neg-Comb-1) (Section 3.1.3.1), (Neg-Comb-2) (Section 3.1.3.2) and unrecognized "ClientHello" extensions (Neg-Ind-1) (Section 3.1.2.1), (Neg-Comb-2) (Section 3.1.3.2). In (Neg-Ind-3) (Section 3.1.2.3) and (Neg-Comb-3) (Section 3.1.3.3), a server that is hybrid-unaware will ignore the delimiters in "supported_groups", and thus might try to negotiate an algorithm individually that is only meant to be used in combination; depending on how such an implementation is coded, it may also
encounter bugs when the same element appears multiple times in the list.

3.2. (Num) How many component algorithms to combine?

3.2.1. (Num-2) Two

Exactly two algorithms can be combined together in hybrid key exchange. This is the approach taken in [KIEFER] and [SCHANCK].

3.2.2. (Num-2+) Two or more

Two or more algorithms can be combined together in hybrid key exchange. This is the approach taken in [WHYTE13].

3.2.3. Benefits and Drawbacks

Restricting the number of component algorithms that can be hybridized to two substantially reduces the generality required. On the other hand, some adopters may want to further reduce risk by employing multiple next-gen algorithms built on different cryptographic assumptions.

3.3. (Shares) How to convey key shares?

In ECDH ephemeral key exchange, the client sends its ephemeral public key in the "key_share" extension of the "ClientHello" message, and the server sends its ephemeral public key in the "key_share" extension of the "ServerHello" message.

For a general key encapsulation mechanism used for ephemeral key exchange, we imagine that that client generates a fresh KEM public key / secret pair for each connection, sends it to the client, and the server responds with a KEM ciphertext. For simplicity and consistency with TLS 1.3 terminology, we will refer to both of these types of objects as "key shares".

In hybrid key exchange, we have to decide how to convey the client’s two (or more) key shares, and the server’s two (or more) key shares.

3.3.1. (Shares-Concat) Concatenate key shares

The client concatenates the bytes representing its two key shares and uses this directly as the "key_exchange" value in a "KeyShareEntry" in its "key_share" extension. The server does the same thing. Note that the "key_exchange" value can be an octet string of length at most 2^16-1. This is the approach taken in [KIEFER], [OQS-111], and [WHYTE13].
3.3.2. (Shares-Multiple) Send multiple key shares

The client sends multiple key shares directly in the "client_shares" vectors of the "ClientHello" "key_share" extension. The server does the same. (Note that while the existing "KeyShareClientHello" struct allows for multiple key share entries, the existing "KeyShareServerHello" only permits a single key share entry, so some modification would be required to use this approach for the server to send multiple key shares.)

3.3.3. (Shares-Ext-Additional) Extension carrying additional key shares

The client sends the key share for its traditional algorithm in the original "key_share" extension of the "ClientHello" message, and the key share for its next-gen algorithm in some additional extension in the "ClientHello" message. The server does the same thing. This is the approach taken in [SCHANCK].

3.3.4. Benefits and Drawbacks

*Backwards compatibility.* (Shares-Multiple) (Section 3.3.2) is fully backwards compatible with non-hybrid-aware servers. (Shares-Ext-Additional) (Section 3.3.3) is backwards compatible with non-hybrid-aware servers provided they ignore unrecognized extensions. (Shares-Concat) (Section 3.3.1) is backwards-compatible with non-hybrid aware servers, but may result in duplication / additional round trips (see below).

*Duplication versus additional round trips.* If a client wants to offer multiple key shares for multiple combinations in order to avoid retry requests, then the client may ended up sending a key share for one algorithm multiple times when using (Shares-Ext-Additional) (Section 3.3.3) and (Shares-Concat) (Section 3.3.1). (For example, if the client wants to send an ECDH-secp256r1 + McEliece123 key share, and an ECDH-secp256r1 + NewHope1024 key share, then the same ECDH public key may be sent twice. If the client also wants to offer a traditional ECDH-only key share for non-hybrid-aware implementations and avoid retry requests, then that same ECDH public key may be sent another time.) (Shares-Multiple) (Section 3.3.2) does not result in duplicate key shares.

3.4. (Comb) How to use keys?

Each component key exchange algorithm establishes a shared secret. These shared secrets must be combined in some way that achieves the "hybrid" property: the resulting secret is secure as long as at least one of the component key exchange algorithms is unbroken.
3.4.1. (Comb-Concat) Concatenate keys

Each party concatenates the shared secrets established by each component algorithm in an agreed-upon order, then feeds that through the TLS key schedule. In the context of TLS 1.3, this would mean using the concatenated shared secret in place of the (EC)DHE input to the second call to "HKDF-Extract" in the TLS 1.3 key schedule:

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

This is the approach used in [KIEFER], [OQS-111], and [WHYTE13].

[GIACON] analyzes the security of applying a KDF to concatenated KEM shared secrets, but their analysis does not exactly apply here since the transcript of ciphertexts is included in the KDF application (though it should follow relatively straightforwardly).

[BINDEL] analyzes the security of the (Comb-Concat) approach as abstracted in their "dualPRF" combiner. They show that, if the component KEMs are IND-CPA-secure (or IND-CCA-secure), then the values output by "Derive-Secret" are IND-CPA-secure (respectively,
IND-CCA-secure). An important aspect of their analysis is that each ciphertext is input to the final PRF calls; this holds for TLS 1.3 since the "Derive-Secret" calls that derive output keys (application traffic secrets, and exporter and resumption master secrets) include the transcript hash as input.

3.4.2. (Comb-KDF-1) KDF keys

Each party feeds the shared secrets established by each component algorithm in an agreed-upon order into a KDF, then feeds that through the TLS key schedule. In the context of TLS 1.3, this would mean first applying "HKDF-Extract" to the shared secrets, then using the output in place of the (EC)DHE input to the second call to "HKDF-Extract" in the TLS 1.3 key schedule:

```
| 0  |
| v  |
| PSK -> HKDF-Extract = Early Secret |
|-----| Derive-Secret(...) |
|-----| Derive-Secret(...) |
|-----| Derive-Secret(...) |
| Next-Gen |
| v  |
| (EC)DHE -> HKDF-Extract |
| v  |
| output -----> HKDF-Extract = Handshake Secret |
| ^^^^^^ |
|-----| Derive-Secret(...) |
|-----| Derive-Secret(...) |
|     |
| v  |
| Derive-Secret(.", "derived", ")" |
| v  |
| 0 -> HKDF-Extract = Master Secret |
|-----| Derive-Secret(...) |
|-----| Derive-Secret(...) |
|-----| Derive-Secret(...) |
|-----| Derive-Secret(...) |
```
3.4.3. (Comb-KDF-2) KDF keys

Each party concatenates the shared secrets established by each component algorithm in an agreed-upon order then feeds that into a KDF, then feeds the result through the TLS key schedule.

Compared with (Comb-KDF-1) (Section 3.4.2), this method concatenates the (2 or more) shared secrets prior to input to the KDF, whereas (Comb-KDF-1) puts the (exactly 2) shared secrets in the two different input slots to the KDF.

Compared with (Comb-Concat) (Section 3.4.1), this method has an extract KDF application. While this adds computational overhead, this may provide a cleaner abstraction of the hybridization mechanism for the purposes of formal security analysis.

```
0
  v
PSK -> HKDF-Extract = Early Secret
       +----- Derive-Secret(...)
       +----- Derive-Secret(...)
       +----- Derive-Secret(...)
         v
concatenated 0
shared secret -> HKDF-Extract
               Derive-Secret(., "derived", ")
                 v
                 v
output ----> HKDF-Extract = Handshake Secret
                 +----- Derive-Secret(...)
                 +----- Derive-Secret(...)
                 +----- Derive-Secret(...)
                   v
                   v
Derive-Secret(., "derived", ")
                     v
                     v
0 -> HKDF-Extract = Master Secret
             +----- Derive-Secret(...)
             +----- Derive-Secret(...)
             +----- Derive-Secret(...)
             +----- Derive-Secret(...)```
3.4.4. (Comb-XOR) XOR keys

Each party XORs the shared secrets established by each component algorithm (possibly after padding secrets of different lengths), then feeds that through the TLS key schedule. In the context of TLS 1.3, this would mean using the XORed shared secret in place of the (EC)DHE input to the second call to "HKDF-Extract" in the TLS 1.3 key schedule.

[GIACON] analyzes the security of applying a KDF to the XORed KEM shared secrets, but their analysis does not quite apply here since the transcript of ciphertexts is included in the KDF application (though it should follow relatively straightforwardly).

3.4.5. (Comb-Chain) Chain of KDF applications for each key

Each party applies a chain of key derivation functions to the shared secrets established by each component algorithm in an agreed-upon order; roughly speaking: "F(k1 || F(k2))". In the context of TLS 1.3, this would mean extending the key schedule to have one round of the key schedule applied for each component algorithm’s shared secret:
This is the approach used in [SCHANCK].

[BINDEL] analyzes the security of this approach as abstracted in their nested dual-PRF "N" combiner, showing a similar result as for the dualPRF combiner that it preserves IND-CPA (or IND-CCA) security. Again their analysis depends on each ciphertext being input to the final PRF ("Derive-Secret") calls, which holds for TLS 1.3.

3.4.6. (Comb-AltInput) Second shared secret in an alternate KDF input

In the context of TLS 1.3, the next-generation shared secret is used in place of a currently unused input in the TLS 1.3 key schedule, namely replacing the "0" "IKM" input to the final "HKDF-Extract":

\[
\begin{align*}
0 & \rightarrow \text{HKDF-Extract} = \text{Master Secret} \\
& \quad \downarrow \\
& \quad \downarrow \\
& \quad \downarrow \\
& \quad \downarrow \\
\text{traditional_shared_secret} & \rightarrow \text{HKDF-Extract} \\
& \quad \downarrow \\
& \quad \downarrow \\
& \quad \downarrow \\
& \quad \downarrow \\
\text{next_gen_shared_secret} & \rightarrow \text{HKDF-Extract} = \text{Handshake Secret} \\
& \quad \downarrow \\
& \quad \downarrow \\
& \quad \downarrow \\
& \quad \downarrow \\
0 & \rightarrow \text{HKDF-Extract} = \text{Master Secret} \\
& \quad \downarrow \\
& \quad \downarrow \\
& \quad \downarrow \\
& \quad \downarrow \\
\end{align*}
\]
This approach is not taken in any of the known post-quantum/hybrid TLS drafts. However, it bears some similarities to the approach for using external PSKs in [EXTERN-PSK].

3.4.7. Benefits and Drawbacks

*New logic.* While (Comb-Concat) (Section 3.4.1), (Comb-KDF-1) (Section 3.4.2), and (Comb-KDF-2) (Section 3.4.3) require new logic to compute the concatenated shared secret, this value can then be used by the TLS 1.3 key schedule without changes to the key schedule logic. In contrast, (Comb-Chain) (Section 3.4.5) requires the TLS 1.3 key schedule to be extended for each extra component algorithm.

*Philosophical.* The TLS 1.3 key schedule already applies a new stage for different types of keying material (PSK versus (EC)DHE), so (Comb-Chain) (Section 3.4.5) continues that approach.

*Efficiency.* (Comb-KDF-1) (Section 3.4.2), (Comb-KDF-2) (Section 3.4.3), and (Comb-Chain) (Section 3.4.5) increase the number
of KDF applications for each component algorithm, whereas (Comb-Concat) (Section 3.4.1) and (Comb-AltInput) (Section 3.4.6) keep the number of KDF applications the same (though with potentially longer inputs).

*Extensibility.* (Comb-AltInput) (Section 3.4.6) changes the use of an existing input, which might conflict with other future changes to the use of the input.

*More than 2 component algorithms.* The techniques in (Comb-Concat) (Section 3.4.1) and (Comb-Chain) (Section 3.4.5) can naturally accommodate more than 2 component shared secrets since there is no distinction to how each shared secret is treated. (Comb-AltInput) (Section 3.4.6) would have to make some distinct, since the 2 component shared secrets are used in different ways; for example, the first shared secret is used as the "IKM" input in the 2nd "HKDF-Extract" call, and all subsequent shared secrets are concatenated to be used as the "IKM" input in the 3rd "HKDF-Extract" call.

3.4.8. Open questions

At this point, it is unclear which, if any, of the above methods preserve FIPS compliance: i.e., if one shared secret is from a FIPS-compliant method (e.g., ECDH), and another shared secret is from a non-approved method (e.g., post-quantum), is the result still considered FIPS compliant? Guidance from NIST on this question would be helpful. Specifically, are any of these approaches acceptable under either [NIST-SP-800-56C] or [NIST-SP-800-135]?

4. Candidate instantiations

In this section, we describe two candidate instantiations of hybrid key exchange in TLS 1.3, based on the design considerations framework above. It is not our intention that both of these instantiations be standardized; we are providing two for discussion and for comparing and contrasting the two approaches.

4.1. Candidate Instantiation 1

Candidate Instantiation 1 allows for two or more component algorithms to be combined (Num-2+) (Section 3.2.2), and negotiates the combination using markers in the "NamedGroup" list as pointers to an extension listing the algorithms comprising each possible combination (Neg-Comb-2) (Section 3.1.3.2) following the approach of [WHYTE13]. The client conveys its multiple key shares individually in the "client_shares" vector of the "ClientHello" "key_share" extension (Shares-Multiple) (Section 3.3.2). The server conveys its multiple key shares concatenated together in its "KeyShareServerHello" struct
(Shares-Concat) (Section 3.3.1). The shared secrets are combined by concatenating them then feeding them through a KDF, then feeding the result into the TLS 1.3 key schedule (Comb-KDF-2) (Section 3.4.3).

4.1.1. ClientHello extension supported_groups

Following [WHYTE13] section 3.1, the "NamedGroup" enum used by the client to populate the "supported_groups" extension is extended to include new code points representing markers for hybrid combinations:

```c
enum {
    /* existing named groups */
    secp256r1 (23),
    ...,

    /* new code points eventually defined for post-quantum algorithms */
    ...

    /* new code points reserved for hybrid markers */
    hybrid_marker00 (0xFD00),
    hybrid_marker01 (0xFD01),
    ...
    hybrid_markerFF (0xFDFF),

    /* existing reserved code points */
    ffdhe_private_use (0x01FC..0x01FF),
    ecdhe_private_use (0xFE00..0xFEFF),
    (0xFFFF)
} NamedGroup;
```

"hybrid_marker" code points do not a priori represent any fixed combination. Instead, during each session establishment, the client defines what it wants each "hybrid_marker" code point to represent using the following extension.

4.1.2. ClientHello extension hybrid_extension

Following [WHYTE13] section 3.2.4, a new "ClientHello" "hybrid_extension" extension is defined. It is defined as follows:

```c
struct {
    NamedGroup hybrid_marker;
    NamedGroup components<2..10>;
} HybridMapping;

struct {
    HybridMapping map<0..255>;
} HybridExtension;
```
The "HybridExtension" contains 0 or more "HybridMapping"s. Each "HybridMapping" corresponds to one of the "hybrid_marker" included in the "supported_groups" extension, and lists the component algorithms that are meant to comprise the this hybrid combination, which can be any of the existing named groups (elliptic curve or finite field), new code points eventually defined for post-quantum algorithms, or reserved code points for private use.

4.1.3. ClientHello extension key_share

No syntactical modifications are made to the "KeyShareEntry" or "KeyShareClientHello" data structures.

Semantically, the client does not send a "KeyShareEntry" corresponding to any of the "hybrid_marker" code points. Instead, the client sends "KeyShareEntry" for each of the component algorithms listed in the "HybridMapping"s.

For example, if the list of "supported_groups" is "secp256r1", "x25519", "hybrid_marker00", and "hybrid_marker01", where "hybrid_marker00" comprises "secp256r1" with a fictional post-quantum algorithm "PQ1", and "hybrid_marker01" comprises "x25519" with "PQ1", then the client could send three "KeyShareEntry" components: one for "secp256r1", one for "x25519", and one for "PQ1".

4.1.4. ServerHello extension KeyShareServerHello

The server responds with a "KeyShareServerHello" struct containing a single "KeyShareEntry", which contains a single "NamedGroup" value and an opaque "key_exchange" string.

To complete the negotiation of a hybrid algorithm, the server responds with the "NamedGroup" value being the "hybrid_marker" code point correspond to the combination that the server was willing to agree to.

The "key_exchange" string is the octet representation of the following struct:

```c
struct {
    KeyShareEntry key_share<2..10>;
} HybridKeyShare;
```

where there is one "key_share" entry for each of the components of this hybrid combination.

Note that the "key_exchange" string has a maximum length of $2^{16}-1$ octets, which may be insufficient for some post-quantum algorithms or
for some hybridizations of multiple post-quantum algorithms. It remains an open question as to whether this length can be increased without breaking existing TLS 1.3 implementations.

4.1.5. Key schedule

The component algorithm shared secrets are combined by concatenating them, then applying a key derivation function, the output of which is then used in the TLS 1.3 key schedule in place of the (EC)DHE shared secret. The component shared secrets are concatenated in the order that they appear in the "components" vector of the "HybridMapping" extension above.

We provide two options for concatenating the shared secrets, and would like feedback from the working group in which to proceed with.

Each component algorithm’s "shared_secret" is defined by the algorithm itself, for example the DHE or ECDHE shared secrets as defined in Section 7.4 of [TLS13], or as defined by post-quantum methods once standardized in their own documents.

*Option 1: Using data structures.* Option 1 uses a full-fledged TLS 1.3 data structure to represent the list of component shared secrets. As a result, lengths of each shared secret are unambiguously encoded.

```
struct SharedSecret {
   opaque shared_secret<0..2^16-1>;
}

struct {
   SharedSecret component<2..10>;
} HybridSharedSecret;
```

The "concatenated_shared_secret" is then the octet representation of the "HybridSharedSecret " struct.

*Option 2: Direct concatenation.* Option 2 directly concatenates the shared secrets. Option 2 should only be considered if the shared secret for each algorithm is guarantees to be of a fixed length, which would imply that, once the component algorithms are fixed, concatenation is bijective.

```
concatenated_shared_secret = shared_secret0 | shared_secret1 | ...
```

In either option, the "concatenated_shared_secret" octet string is used as the IKM argument of HKDF-Extract, with the zero-length string as the salt argument. The output of HKDF-Extract is used as the IKM
argument for HKDF-Extract’s calculation of the handshake secret, as shown below.

```
        0  |
        v
PSK    HKDF-Extract = Early Secret
        v
        |
+-----> Derive-Secret(...) 
|       |
|       v
concatenated 0
shared    HKDF-Extract = Derive-Secret(., "derived", "")
^~~~~~~~
|       v
output   HKDF-Extract = Handshake Secret
^~~~~~~~
|       v
       |
+-----> Derive-Secret(...) 
|       v
|       |
|       v
 Derive-Secret(., "derived", "")
|       v
|       |
|       v
0      HKDF-Extract = Master Secret
|       |
+-----> Derive-Secret(...) 
|       v
|       |
|       v
+-----> Derive-Secret(...) 
|       v
|       |
|       v
+-----> Derive-Secret(...) 
        |
|       v
```

### 4.2. Candidate Instantiation 2

Candidate Instantiation 2 allows for exactly two component algorithms to be combined (Num-2) (Section 3.2.1), and uses code points standardized for each permissible combination. The client concatenates its multiple key shares together as a distinct entry in the "client_shares" vector of the "ClientHello" "key_share" extension (Shares-Concat) (Section 3.3.1). The server does the same. The shared secrets are combined by concatenating them then feeding them through a KDF, then feeding the result into the TLS 1.3 key schedule (Comb-KDF-2) (Section 3.4.3).
4.2.1. ClientHello extension supported_groups

The "NamedGroup" enum used by the client to populate the "supported_groups" extension is extended to include new code points representing each desired combination.

For example,

enum {
    /* existing named groups */
    secp256r1 (23),
    x25519 (0x001D),
    ...

    /* new code points eventually defined for post-quantum algorithms */
    PQ1 (0x????),
    PQ2 (0x????),
    ...

    /* new code points defined for hybrid combinations */
    secp256r1_PQ1 (0x????),
    secp256r1_PQ2 (0x????),
    x25519_PQ1 (0x????),
    x25519_PQ2 (0x????),

    /* existing reserved code points */
    ffdhe_private_use (0x01FC..0x01FF),
    ecdhe_private_use (0xFE00..0xFEFF),
    (0xFFFF)
} NamedGroup;

4.2.2. ClientHello extension KeyShareClientHello

The client sends a "KeyShareClientHello" struct containing multiple "KeyShareEntry" values, some of which may correspond to some of the hybrid combination code points it listed in the "supported_groups" extension above.

The "KeyShareEntry" for a hybrid combination code point contains an opaque "key_exchange" string which is the octet representation of the following struct:

struct {
    KeyShareEntry key_share<2..10>;
} HybridKeyShare;

where there is one "key_share" entry for each of the components of this hybrid combination.
Note that this approach may result in duplication of key shares being sent; for example, a client wanting to support either the combination "secp256r1_PQ1" or "x25519_PQ1" would send two "PQ1" key shares.

4.2.3. ServerHello extension KeyShareServerHello

The server responds with a "KeyShareServerHello" struct containing a single "KeyShareEntry", which contains a single "NamedGroup" value and an opaque "key_exchange" string. The "key_exchange" string is the octet representation of the "HybridKeyShare" struct defined above.

4.2.4. Key schedule

The key schedule is computed as in Candidate Instantiation 1 above.

4.3. Comparing Candidate Instantiation 1 and 2

CI2 requires much less change to negotiation routines — each hybrid combination is just a new key exchange method, and the concatenation of key shares and shared secrets can be handled internally to that method. This comes at the cost, however, of combinatorial explosion of code points: one code point needs to be standardized for each desired combination. We have also limited the number of hybrid algorithms to 2 in CI2 to somewhat limit the explosion of code points needing to be defined. Concatenating client key shares also risks sending duplicate key shares, increasing communication sizes.

CI1 requires more change to negotiation routines, since it introduces new data structures and has an indirect mapping between hybrid combinations and key shares. Benefits from this approach include avoiding sending duplicate key shares and not needing to standardize every possible supported combination. Implementers, however, must do the work of deciding which combinations of algorithms are meaningful / tolerable / desirable from a security perspective, potentially complicating interoperability.

5. IANA Considerations

If Candidate Instantiation 1 is selected, the TLS Supported Groups registry will have to be updated to include code points for hybrid markers.

6. Security Considerations

The majority of this document is about security considerations. As noted especially in Section 3.4, the shared secrets computed in the hybrid key exchange should be computed in a way that achieves the
"hybrid" property: the resulting secret is secure as long as at least one of the component key exchange algorithms is unbroken. While many natural approaches seem to achieve this, there can be subtleties (see for example the introduction of [GIACON]).

The rest of this section highlights a few unresolved questions related to security.

6.1. Active security

One security consideration that is not yet resolved is whether key encapsulation mechanisms used in TLS 1.3 must be secure against active attacks (IND-CCA), or whether security against passive attacks (IND-CPA) suffices. Existing security proofs of TLS 1.3 (such as [DFGS15], [DOWLING]) are formulated specifically around Diffie-Hellman and use an "actively secure" Diffie-Hellman assumption (PRF Oracle Diffie-Hellman (PRF-ODH)) rather than a "passively secure" DH assumption (e.g. decisional Diffie-Hellman (DDH)), but do not claim that the actively secure notion is required. In the context of TLS 1.2, [KPW13] show that, at least in one formalization, a passively secure assumption like DDH is insufficient (even when signatures are used for mutual authentication). Resolving this issue for TLS 1.3 is an open question.

6.2. Resumption

TLS 1.3 allows for session resumption via a pre-shared key. When a pre-shared key is used during session establishment, an ephemeral key exchange can also be used to enhance forward secrecy. If the original key exchange was hybrid, should an ephemeral key exchange in a resumption of that original key exchange be required to use the same hybrid algorithms?

6.3. Failures

Some post-quantum key exchange algorithms have non-trivial failure rates: two honest parties may fail to agree on the same shared secret with non-negligible probability. Does a non-negligible failure rate affect the security of TLS? How should such a failure be treated operationally? What is an acceptable failure rate?

7. Acknowledgements

These ideas have grown from discussions with many colleagues, including Christopher Wood, Matt Campagna, and authors of the various hybrid Internet-Drafts and implementations cited in this document. The immediate impetus for this document came from discussions with
attendees at the Workshop on Post-Quantum Software in Mountain View, California, in January 2019.

Martin Thomson suggested the (Comb-KDF-1) (Section 3.4.2) approach.

8. References

8.1. Normative References


8.2. Informative References


Authors' Addresses

Douglas Stebila
University of Waterloo
Email: dstebila@uwaterloo.ca

Scott Fluhrer
Cisco Systems
Email: sfluhrer@cisco.com

Shay Gueron
University of Haifa and Amazon Web Services
Email: shay.gueron@gmail.com
Abstract

This document describes two mechanisms for enabling the use of the OPAQUE password-authenticated key exchange in TLS 1.3.

Status of This Memo

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

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

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

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

Copyright Notice

Copyright (c) 2019 IETF Trust and the persons identified as the document authors. All rights reserved.

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

Note that this draft has not received significant security review and should not be the basis for production systems.

OPAQUE [opaque-paper] is a mutual authentication method that enables the establishment of an authenticated cryptographic key between a client and server based on a user’s memorized password, without ever exposing the password to servers or other entities other than the client machine and without relying on PKI. OPAQUE leverages a primitive called a Strong Asymmetrical Password Authenticated Key Exchange (Strong aPAKE) to provide desirable properties including resistance to pre-computation attacks in the event of a server compromise.

In some cases, it is desirable to combine password-based authentication with traditional PKI-based authentication as a defense-in-depth measure. For example, in the case of IoT devices, it may be useful to validate that both parties were issued a certificate from a certain manufacturer. Another desirable property
for password-based authentication systems is the ability to hide the client’s identity from the network. This document describes the use of OPAQUE in TLS 1.3 [TLS13] both as part of the TLS handshake and post-handshake facilitated by Exported Authenticators [I-D.ietf-tls-exported-authenticator], how the different approaches satisfy the above properties and the trade-offs associated with each design.

The in-handshake instantiations of OPAQUE can be used to authenticate a TLS handshake with a password alone, or in conjunction with certificate-based (mutual) authentication but does not provide identity hiding for the client. The Exported Authenticators instantiation of OPAQUE provides client identity hiding by default and allows the application to do password authentication at any time during the connection, but requires PKI authentication for the initial handshake and application-layer semantics to be defined for transporting authentication messages.

2. Conventions and Definitions

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

3. OPAQUE

In OPAQUE [opaque-paper], it is shown that a Strong Asymmetric Password-Authenticated Key Exchange (Strong aPAKE) can be constructed given an oblivious pseudo-random function (OPRF) and authenticated key exchange protocol that is secure against reverse impersonation (a.k.a. KCI). Unlike previous PAKE methods such as SRP [RFC2945] and SPAKE-2 [I-D.irtf-cfrg-spake2], which require a public salt value, a Strong aPAKE leverages the OPRF private key as salt, making it resistant to pre-computation attacks on the password database stored on the server.

TLS 1.3 provides a KCI-secure key agreement algorithm suitable for use with OPAQUE. This document describes three instantiations of OPAQUE in TLS 1.3: one based on digital signatures, one on Diffie-Hellman key agreement, and one based on HMQV key exchange. Of the three instantiations, the only one that has known IPR considerations is HMQV.

OPAQUE consists of two distinct phases: password registration and authentication. We will describe the mechanisms for password registration in this document but it is assumed to have been done...
outside of TLS. During password registration, the client and server establish a shared set of parameters for future authentication and two private-public key pairs are generated, one for the client and one for the server. The server keeps its private key and stores an encapsulated copy of the client’s key pair along with its own public key in an "envelope" that is encrypted with the result of the OPRF operation. Note that it is possible for the server to use the same key for multiple clients. It may be necessary to permit multiple simultaneous server keys in the even of a key rollover. The client does not store any state nor any PKI information.

We call the first instantiation OPAQUE-Sign. In OPAQUE-Sign, the key pairs generated at password registration time are digital signature keys. These signature keys are used in place of certificate keys for both server and client authentication in a TLS handshake. Client authentication is technically optional, but in practice is almost universally required. OPAQUE-Sign cannot be used alongside certificate-based handshake authentication. This instantiation can also be leveraged to do part of a post-handshake authentication using Exported Authenticators [I-D.ietf-tls-exported-authenticator] given an established TLS connection protected with certificate-based authentication.

The second and third instantiations are called OPAQUE-3DH and OPAQUE-HMQV. In these instantiations, the key pairs are Diffie-Hellman keys and are used to establish a shared secret that is fed into the key schedule for the handshake. The handshake continues to use Certificate-based authentication. The two methods for establishing the shared key are Diffie-Hellman and HMQV. These instantiations are best suited to use cases in which both password and certificate-based authentication are needed during the initial handshake, which is useful in some scenarios. There is no unilateral authentication in this context, mutual authentication is demonstrated explicitly through the finished messages.

4. Password Registration

Password registration is run between a user U and a server S. It is assumed that the user can authenticate the server during this registration phase (this is the only part in OPAQUE that requires some form of authenticated channel, either physical, out-of-band, PKI-based, etc.)

A set of parameters is chosen. This includes an AuthEnc function for key encapsulation, a group setting for the OPRF (chosen as a cipher defined in Oblivious Pseudorandom Functions (OPRFs) using Prime-Order Groups [I-D.sullivan-cfrg-voprf]), an instantiation (either OPAQUE-Sign, OPAQUE-3DH or OPAQUE-HMQV), and a key type (either a TLS

- U chooses password PwdU and a pair of private-public keys PrivU and PubU of the chosen key type.
- S chooses OPRF key kU (random and independent for each user U) and sets vU = g^kU; it also chooses its own pair of private-public keys PrivS and PubS (the server can use the same pair of keys with multiple users), and sends PubS to U.
- U and S run OPRF(kU;PwdU) as defined in with only U learning the result, denoted RwdU (mnemonics for "Randomized PwdU").
- U generates an "envelope" EnvU defined as
  \[
  EnvU = \text{AuthEnc}(RwdU; \text{PrivU}, \text{PubU}, \text{PubS})
  \]
  where AuthEnc is an authenticated encryption function with the "key committing" property and is specified below in section. In EnvU, all values require authentication and PrivU also requires encryption. PubU can be omitted from EnvU if it can be reconstructed from PrivU but while it will save bits on the wire it will come at some computational cost during client authentication.
- U sends EnvU and PubU to S and erases PwdU, RwdU and all keys. S stores (EnvU, PubS, PrivS, PubU, kU, vU) in a user-specific record. If PrivS and PubS are used for multiple users, S can store these values separately and omit them from the user’s record.

Note (salt). We note that in OPAQUE the OPRF key acts as the secret salt value that ensures the infeasibility of pre-computation attacks. No extra salt value is needed.

4.1. Implementing EnvU

The encryption for EnvU is required to be a key-committing authenticated encryption algorithm. This, unfortunately, eliminates both AES-GCM and AES-GCM-SIV as wrapping functions. It is possible to create a key-committing authenticated encryption using AES-CBC [RFC3602] or AES-CTR [RFC5930] with HMAC [RFC4886] as long as the keys for encryption and authentication are derived separately with a key domain separation mechanism such as HKDF [RFC5869].
5. TLS extensions

We define several TLS extensions to signal support for OPAQUE and transport the parameters. The extensions used here have a similar structure to those described in Usage of PAKE with TLS 1.3 [I-D.barnes-tls-pake]. The permitted messages that these extensions are allowed and the expected protocol flows are described below.

This document defines the following extension code points.

```plaintext
enum {
    ...
    opaque_client_auth(TBD),
    opaque_server_auth(TBD),
    (65535)
} ExtensionType;
```

The opaque_client_auth extension contains a PAKEClientAuthExtension struct and can only be included in the CertificateRequest and Certificate messages. The opaque_client_auth extension contains a PAKEServerAuthExtension struct and can only be included in the ClientHello, EncryptedExtensions, CertificateRequest and Certificate messages, depending on the type.

The structures contained in this extension are defined as:

```plaintext
struct {
    opaque identity<0..2^16-1>;
    opaque OPRF_1<1..2^16-1>;
} PAKEShareClient;
```
struct {
    opaque identity<0..2^16-1>;
    opaque OPRF_2<1..2^16-1>;
    opaque vU<1..2^16-1>;
    opaque EnvU<1..2^16-1>;
} PAKEShareServer;

struct {
    select (Handshake.msg_type) {
        ClientHello:
            PAKEShareClient client_shares<0..2^16-1>;
            OPAQUEType types<0..2^16-1>;
            EncryptedExtensions, Certificate:
                PAKEShareServer server_share;
                OPAQUEType type;
    }
} PAKEServerAuthExtension;

struct {
    opaque identity<0..2^16-1>;
} PAKEClientAuthExtension;

This document also defines the following set of types;

enum {
    OPAQUE-Sign(1),
    OPAQUE-3DH(2),
    OPAQUE-3DH-Cert(3),
    OPAQUE-HMQV(4),
    OPAQUE-HMQV-Cert(5),
} OPAQUEType;

The "identity" field is the unique user id used to index the user’s record on the server. The types field indicates the set of supported auth types by the client. The OPRF_1 message is as defined in Oblivious Pseudorandom Functions (OPRFs) using Prime-Order Groups [I-D.sullivan-cfrg-voprf]. The content of OPRF_1 is typically the result of the password hashed into a group element and blinded by an element known to the client. OPRF_2 is the OPRF_1 value operated on by the OPRF private key kU. vU is the public component of kU and EnvU is the envelope containing PrivU, PubS, and PubU. (Note that for groups, it may be more space efficient to only include PrivU and have the client derive PubU from PrivU). See Section 9 for details.

This document also describes a new CertificateEntry structure that corresponds to an authentication via a signature derived using OPAQUE. This structure serves as a placeholder for the PAKEServerAuthExtension extension.
struct {
    select (certificate_type) {
        case OPAQUESign:
            /* Defined in this document */
            opaque null<0>
        case RawPublicKey:
            /* From RFC 7250 ASN.1_subjectPublicKeyInfo */
            opaque ASN1_subjectPublicKeyInfo<1..2^24-1>;
        case X509:
            opaque cert_data<1..2^24-1>;
    }
    Extension extensions<0..2^16-1>;
} CertificateEntry;

We request that IANA add an additional type to the "TLS Certificate Types" registry for this OPAQUESign.

Support for the OPAQUESign Certificate type for server authentication can be negotiated using the server_certificate_type [RFC7250] and the Certificate type for client authentication can be negotiated using the client_certificate_type extension [RFC7250].

Note that there needs to be a change to the client_certificate_type row in the IANA TLS ExtensionType Values table to allow client_certificate_type extension to be used as an extension to the CertificateRequest message.

6. Use of extensions in TLS handshake flows

6.1. OPAQUE-3DH, OPAQUE-HMQV

In these two modes of operation, the OPAQUE private keys are used for key agreement algorithm and the result is fed into the TLS key schedule. Password validation is confirmed by the validation of the finished message. These modes can be used in conjunction with optional Certificate-based authentication.

It should be noted that since the identity of the client it is not encrypted as it is sent as an extension to the ClientHello. This may present a privacy problem unless a mechanism like ESNI [I-D.ietf-tls-esni] is created to protect it.

Upon receiving a PAKEServerAuth extension, the server checks to see if it has a matching record for this identity. If the record does not exist, the handshake is aborted with a TBD error message. If the record does exist, but the key type of the record does not match any
of the supported_groups sent in the key_share extension of the ClientHello, an HRR is sent containing the set of valid key types that it found records for.

Given a matching key_share and an identity with a matching supported_group, the server returns its PAKEServerAuth as an extension to its EncryptedExtensions. Both parties then derive a shared OPAQUE key using

**HMQV**

C computes \( K = (g^y \cdot PubS^e)^{(x + d \cdot PrivU)} \)

S computes \( K = (g^x \cdot PubU^d)^{(y + e \cdot PrivS)} \)

where \( d = H(g^x, IdS) \) and \( e = H(g^y, IdU) \), and \( IdU, IdS \) represent the identities of user (sent as identity in PAKEShareClient) and server (EncryptedExtension or Certificate message). TODO: be more explicit about content of IdS.

**3DH**

C computes \( K = H(g^y \cdot PrivU \ || \ PubU \ ^{x} \ || \ PubS \ ^{PrivU} \ || \ IdU \ || \ IdS) \)

S computes \( K = H(g^x \cdot PrivS \ || \ PubS \ ^{y} \ || \ PubU \ ^{PrivS} \ || \ IdU \ || \ IdS) \)

\( IdU, IdS \) represent the identities of user (sent as identity in PAKEShareClient) and server (Certificate message).

\( H \) is the HKDF function agreed upon in the TLS handshake.

The result, \( K \), is then added as an input to the Master Secret in place of the 0 value defined in TLS 1.3:

\[ 0 \rightarrow HKDF-Extract = Master \ Secret \]

becomes

\[ K \rightarrow HKDF-Extract = Master \ Secret \]

In this construction, the finished messages cannot be validated unless the OPAQUE computation was done correctly on both sides, authenticating both client and server.

For the certificate version of OPAQUE (OPAQUE-3DH-Cert, OPAQUE-HMQV-Cert), the server’s first flight contains the standard set of messages: ServerHello, EncryptedExtension, (optional)CertificateRequest, Certificate, CertificateVerify, Finished. In the non-certificate cases (OPAQUE-3DH-Cert, OPAQUE-
HMQV-Cert), the Certificate and CertificateVerify messages are omitted, similar to the PSK mode in TLS 1.3.

6.2. OPAQUE-Sign

In this modes of operation, the OPAQUE private keys are used for digital signatures and are used to define a new Certificate type and CertificateVerify algorithm. Like the 3DH and HKDF instantiations above, the identity of the client is sent in the clear in the client’s first flight unless a mechanism like ESNI [I-D.ietf-tls-esni] is created to protect it.

Upon receiving a PAKEServerAuth extension, the server checks to see if it has a matching record for this identity. If the record does not exist, the handshake is aborted with a TBD error message. If the record does exist, but the key type of the record does not match any of the supported_signatures sent in the the ClientHello, the handshake must be aborted with a TBD error.

We define a new Certificate message type for an OPAQUE-Sign authenticated handshake.

```c
enum {
    X509(0),
    RawPublicKey(2),
    OPAQUE-Sign(3),
    (255)
} CertificateType;
```

Certificates of this type have CertificateEntry structs of the form:

```c
struct {
    Extension extensions<0..2^16-1>;
} CertificateEntry;
```

Given a matching signature_scheme and an identity with a matching key type, the server returns a certificate message with type OPAQUE-Sign with PAKEServerAuth as an extension. The private key used in the CertificateVerify message is set to PrivS, and the client verifies it using PubS.

It is RECOMMENDED that the server includes a CertificateRequest message with a PAKEClientAuth and the identity originally sent in the PAKEServerAuth extension from the client hello. On receiving a CertificateRequest message with a PAKEClientAuth extension, the client returns a CertificateVerify message signed by PrivC which is validated by the server using PubC.
7. Integration into Exported Authenticators

Neither of the above mechanisms provides privacy for the user during the authentication phase, as the user id is sent in the clear. It is possible to create an encryption mechanism like ESNI [I-D.ietf-tls-esni] to protect these values, but this is not in scope for this document. Additionally, OPAQUE-Sign has the drawback that it cannot be used in conjunction with certificate-based authentication.

It is possible to address both the privacy concerns and the requirement for certificate-based authentication by using OPAQUE-Sign in Exported Authenticator [I-D.ietf-tls-exported-authenticator] flow, since exported authenticators are sent over a secure channel that is typically established with certificate-based authentication. Using Exported Authenticators for OPAQUE has the additional benefit that it can be triggered at any time after a TLS session has been established, which better fits modern web-based authentication mechanism.

The client hello contains PAKEServerAuth, PAKEClientAuth with empty identity values to indicate support for these mechanisms.

1. Client creates Authenticator Request with CR extension PAKEServerAuth (identity, OPRF_1)

2. Server creates Exported Authenticator with OPAQUE-Sign (PAKEServerAuth) and CertificateVerify (signed with PrivS)

If the server would like to then establish mutual authentication, it can do the following: 1. Server creates Authenticator Request with CH extension PAKEClientAuth (identity) 2. Client creates Exported Authenticator with OPAQUE-Sign Certificate and CertificateVerify (signed with PrivU)

Support for Exported Authenticators is negotiated at the application layer. For example, OPAQUE-Sign in EAs could be defined as an extension to Secondary Certificates in HTTP/2 [I-D.ietf-httpbis-http2-secondary-certs].

8. Summary of properties
<table>
<thead>
<tr>
<th>Variant Property</th>
<th>Identity Hiding</th>
<th>Certificate Authentication</th>
<th>Server-only Auth</th>
<th>Post-handshake auth</th>
<th>Minimum round trips</th>
</tr>
</thead>
<tbody>
<tr>
<td>OPAQUE-Sign-EA</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>2-RTT</td>
</tr>
<tr>
<td>OPAQUE-Sign</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>1-RTT</td>
</tr>
<tr>
<td>OPAQUE-3DH</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>1-RTT</td>
</tr>
<tr>
<td>OPAQUE-3DH-Cert</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>1-RTT</td>
</tr>
<tr>
<td>OPAQUE-HMQV</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>1-RTT</td>
</tr>
<tr>
<td>OPAQUE-HMQV-Cert</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>1-RTT</td>
</tr>
</tbody>
</table>

9. Example OPRF

This is an example OPRF instantiation based on the ECOPRF-P256-HKDF-SHA256-SSWU ciphersuite in [I-D.sullivan-cfrg-voprf]. We use additive group notation in this description because we specifically target the elliptic curve case. All operations can be replaced with their multiplicative group counterparts.

The example ECOPRF-P256-HKDF-SHA256-SSWU instantiation uses the following parameters:

- Curve: SECP256K1 curve
- H_1: H2C-P256-SHA256-SSWU- [I-D.sullivan-cfrg-voprf]
- label: voprf_h2c
- H_2: SHA256

See [I-D.sullivan-cfrg-voprf] for more details about how each of the above components are used. In the following we will use the functions OPRF_Blind, OPRF_Sign, OPRF_Unblind, OPRF_Finalize that are defined in the same document.
9.1. OPRF_1

Let \( p \) be the prime order of the base field of the curve that is used (e.g. \( 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1 \) for P-256). Let \( \text{I2OSP}, \text{OS2IP} \) be functions as defined in [RFC8017]. Then OPRF_1 is computed using the OPRF_Blind function on the password \( P \) follows:

1. \( r \leftarrow GF(p) \)
2. \( M := rH_1(P) \)
3. Output \((r, M)\)

\[ H_1 = \text{hash-to-curve}(P) = 1. t1 = H(\text{"h2c" || label || I2OSP(len(x), 4) || P}) 2. t2 = \text{OS2IP}(t1) 3. y = t^2 \mod p 4. H_1(P) = \text{map2curve_simple_swu}(y) 5. M = rH_1(P) \]

The client keeps the blind \( r \), and sends the OPRF_1 value \( M \) as an EllipticCurve point [TLS13].

9.2. OPRF_2

The server now computes OPRF_2 by applying OPRF_Sign on the received message \( M \):

1. \( Z := kM \)
2. Output \( Z \)  
   Note that the server should multiply \( M \) by the cofactor of the given curve before it outputs \( Z \). In the case of P-256, this cofactor is equal to 1 and so it is not necessary.

The output \( Z \) of OPRF_2 is sent as an EllipticCurve point "[]" back to the client.

When the client receives the output of OPRF_2, it derives the envelope decryption key using OPRF_Unblind followed by OPRF_Finalize.

1. \( N := (1/r)Z \) (OPRF_Unblind)
2. \( y := H_2(P, N) \) (OPRF_Finalize). Here, we require that \( N \) is serialized before it is input to \( H_2 \). The client can now stores \((P, y)\) for future usage.

10. Privacy considerations

TBD
11. Security Considerations

TODO: protecting against user enumeration

12. IANA Considerations

- Existing IANA references have not been updated yet to point to this document.

IANA is asked to register a new value in the "TLS Certificate Types" registry of Transport Layer Security (TLS) Extensions (TLS-Certificate-Types-Registry), as follows:

- Value: 4 Description: OPAQUE Authentication Reference: This RFC

Correction request: The client_certificate_type row in the IANA TLS ExtensionType Values table to allow client_certificate_type extension to be used as an extension to the CertificateRequest message.

13. References

13.1. Normative References

[I-D.ietf-httpbis-http2-secondary-certs]

[I-D.ietf-tls-exported-authenticator]


13.2. Informative References

[I-D.barnes-tls-pake]

[I-D.ietf-tls-esni]

[I-D.irtf-cfrg-spake2]

[I-D.sullivan-cfrg-voprf]

[opaque-paper]

[RFC2945]

[RFC5869]
Appendix A. Acknowledgments

Authors’ Addresses

Nick Sullivan
Cloudflare

Email: nick@cloudflare.com

Hugo Krawczyk
IBM Research

Email: hugo@ee.technion.ac.il

Owen Friel
Cisco

Email: ofriel@cisco.com

Richard Barnes
Cisco

Email: rlb@ipv.sx
Abstract

This document defines a mechanism for resuming a TLS 1.3 session across different Server Name Indications.

Status of This Memo

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

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

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

This Internet-Draft will expire on September 2, 2019.

Copyright Notice

Copyright (c) 2019 IETF Trust and the persons identified as the document authors. All rights reserved.

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

Most web transactions are short transfers that are significantly delayed by the TLS connection establishment. To accelerate the connection establishment, TLS 1.3 [RFC8446] and its predecessors provide session resumption mechanisms. They abbreviate the TLS handshake based on a shared secret exchanged during a prior TLS session between client and server. In total, these resumption handshakes significantly reduce computational overhead for cryptographic operations and save up to one round-trip compared to the full TLS connection establishment.

TLS 1.3 [RFC8446] allows resumption handshakes across Server Name Indications (SNIs) when they share the same TLS certificate. However, TLS 1.3 recommends not to use TLS resumptions across SNIs to avoid loosing a single-use ticket in case of a failed resumption attempt. This practice requires costly full TLS connection establishments in situations where a performance-optimized resumption handshake across SNI values would be possible. To illustrate this performance limitation, we describe the common situation of a redirected web request. We assume that the hostname example.com redirects to www.example.com and both hostnames are operated by the same entity and use the same certificate for their authentication. A client requesting www.example.com via this redirect requires two full TLS handshakes following the recommendation of TLS 1.3 [RFC8446]. Using resumption across SNI values, the later full handshake can be converted to a performance-optimized resumed handshake. A comprehensive study of the performance benefits of resumptions across SNI values for popular websites can be found in [PERF].
This document defines a mechanism to inform the client in between which SNI values TLS resumptions are supported. This information enables the client to use resumption across SNI values only in situations where the chance of a successful resumption handshake is high.

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 on Resumptions across SNI values

When a client wants to form a TLS connection to a server, it indicates support for the "resumption_group" extension in the ClientHello message. To signal its support for this extension type, the server returns the "resumption_group" extension with an empty data field.

The client is now aware, that all SNI values for which the presented server certificate is valid, form a TLS resumption group. Thus, resumption tickets issued by a group member are designated to be used to establish resumed connections to any member of the same group.

4. The "resumption_group" Extension

This extension carries no data as defined in the following ResumptionGroup structure:

```c
struct {
    ResumptionGroup;
} ResumptionGroup;
```

4.1. Client Behavior

To indicate support for the "resumption_group" extension, the client sends this extension type within the initial ClientHello message to the server.

Upon receiving the server’s response, the client checks whether the "resumption_group" extension is present in the extension list of the server's CertificateEntry (see Section 4.2.2 of [RFC6066]).

If this extension type is not included in the response of the server, then the client reasons that the server is not configured to support
the "resumption_group" extension and proceeds with a normal handshake.

Otherwise, the client proceeds with a normal connection establishment and associates all retrieved resumption tickets to the corresponding resumption group. This resumption group is formed of all SNI values that are valid for the presented server certificate.

To establish a resumed connection to any SNI value included in a resumption group, the client uses a resumption ticket associated to the same group. The Client Hello of a resumed handshake MUST NOT include the "resumption_group" extension.

Tickets received during a resumed connection MUST be associated to the same resumption group of the ticket that was used during the establishment of this connection.

If a SNI value is a member of multiple resumption groups, then the client is recommended to use the freshest valid ticket for a resumption handshake. It is assumed, that fresher resumption tickets are more likely to be accepted by the respective server.

According to [RFC8446], clients MUST NOT cache tickets longer than seven days.

Note, that TLS resumption enables a server to link resumed connections to the same client. A study on the feasibility of this tracking mechanism can be found in [TRAC]. To protect the client’s privacy against tracking via this mechanism, it is RECOMMENDED to cache resumption tickets only for ten minutes.

4.2. Server Behavior

Upon receiving an initial Client Hello message, the server validates if the client provided an extension of the type "resumption_group".

If the "resumption_group" extension is not listed by the client, then the server’s response MUST NOT include an entry for this extension type. Otherwise, the server includes the "resumption_group" extension in the extension list of the server’s CertificateEntry, to signal support for resumptions across SNI values. Subsequently, the server proceeds with a normal handshake.

This extension type does not affect the server behavior for resumed connection establishments.
5. Expectations on Certificates

This "resumption_group" extension forms the resumption group based on the SNI values that are valid for the server's certificate. To optimize the performance benefit of this extension, the server’s certificate is RECOMMENDED to only include SNI values that mutually support the resumption of their TLS connections. Otherwise, the client’s resumption attempt across SNI values will fail if the server does not support this practice. Note, that each failed resumption handshake uses up a single-use resumption ticket. As a result, these failed attempts might use up all cached single-use tickets, which hinders the client to establish performance-optimized resumption handshakes to legitimate SNI values.

6. Compatibility Issues with Middleboxes

[RFC8446]; Section 9.3 requires MITM proxies to remove any extensions they do not understand. If a conformant MITM proxy does not support this extension, it will remove this extension type from the Client Hello. As a result, the server reacts as if it is not supporting this extension type.

7. Security Considerations

Clients MUST only resume to a new SNI value if this SNI value is valid for the server certificate presented in the original connection. To facilitate a correct implementation of this requirement, the resumption group is identical to the list of SNI values valid for a specific server certificate. Note, that the security of TLS resumptions across different SNI values is also discussed in Section 4.6.1 of [RFC8446].

8. IANA Considerations

TODO IANA needs to be requested to create an entry, resumption_group, in the existing registry for ExtensionType (defined in [RFC8446]), with "TLS 1.3" column values being set to "CH, EE", and "Recommended" column being set to "Yes".

9. References

9.1. Normative References

9.2. Informative References


Acknowledgments

Tobias Mueller and Christian Burkert provided ideas for this document.

Author’s Address

Erik Sy
University of Hamburg

Email: tls@erik-sy.de
Abstract

The TLS protocol supports different credentials, including pre-shared keys, raw public keys, and X.509 certificates. For use with public key cryptography developers have to decide between raw public keys, which require out-of-band agreement and full-fledged X.509 certificates. For devices where the reduction of code size is important it is desirable to minimize the use of X.509-related libraries. With the CBOR Web Token (CWT) a structure has been defined that allows CBOR-encoded claims to be protected with CBOR Object Signing and Encryption (COSE).

This document registers a new value to the "TLS Certificate Types" subregistry to allow TLS and DTLS to use CWTs. Conceptually, CWTs can be seen as a certificate format (when with public key cryptography) or a Kerberos ticket (when used with symmetric key cryptography).

Status of This Memo

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

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

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

This Internet-Draft will expire on September 12, 2019.
Copyright Notice

Copyright (c) 2019 IETF Trust and the persons identified as the document authors. All rights reserved.

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

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

Table of Contents

1. Introduction .............................................. 2
2. Conventions and Terminology ............................. 3
3. The CWT Certificate Type .................................. 3
4. Representation and Verification the Identity of Application Services ............................ 4
5. Security and Privacy Considerations .................... 5
6. IANA Considerations ........................................ 5
7. References ................................................. 5
  7.1. Normative References ................................... 5
  7.2. Informative References ................................. 6
Appendix A. History .......................................... 8
Appendix B. Working Group Information .................... 8
Authors’ Addresses ........................................... 8

1. Introduction

The CBOR Web Token (CWT) [RFC8392] was defined as the CBOR-based version of the JSON Web Token (JWT) [RFC7519]. JWT is used extensibility on Web application and for use with Internet of Things
environments the believe is that a more lightweight encoding, namely CBOR, is needed. CWTs, like JWTs, contain claims and those claims are protected against modifications using COSE [RFC8152]. CWTs are flexible with regard to the use of cryptography and hence CWTs may be protected using a keyed message digest, or a digital signature. One of the claims allows keys to be included, as described in [I-D.ietf-ace-cwt-proof-of-possession]. This specification makes use of these proof-of-possession claims in CWTs.

Fundamentally, there are two types of keys that can be used with CWTs:

- Asymmetric keys: In this case a CWT contains a COSE_Key [RFC8152] representing an asymmetric public key. To protect the CWT against modifications the CWT also needs to be digitally signed.

- Symmetric keys: In this case a CWT contains a Encrypted_COSE_Key [RFC8152] representing a symmetric key encrypted to a key known to the recipient using COSE_Encrypt or COSE_Encrypt0. Again, to protect the CWT against modifications a keyed message digest is used.

The CWT also allows mixing symmetric and asymmetric crypto although this is less likely to be used in practice.

Exchanging CWTs in the TLS / DTLS handshake offers an alternative to the use of raw public keys and X.509 certificates. Compared to raw public keys, CWTs allow more information to be included via the use of claims. Compared to X.509 certificates CBOR offers an alternative encoding format, which may also be used by the application layer thereby potentially reducing the overall code size requirements.

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].

3. The CWT Certificate Type

This document defines a new value to the "TLS Certificate Types" subregistry and the value is defined as follows.
4. Representation and Verification the Identity of Application Services

RFC 6125 [RFC6125] provides guidance for matching identifiers used in X.509 certificates against a reference identifier, i.e. an identifier constructed from a source domain and optionally an application service type. Different types of identifiers have been defined over time, such as CN-IDs, DNS-IDs, SRV-IDs, and URI-IDs, and they may be carried in different fields inside the X.509 certificate, such as in the Common Name or in the subjectAltName extension.

For CWTs issued to servers the following rule applies: To claim conformance with this specification an implementation MUST populate the Subject claim with the value of the Server Name Indication (SNI) extension. The Subject claim is of type StringOrURI. If it is string an equality match is used between the Subject claim value and the SNI. If the value contains a URI then the URI schema must be matched against the service being requested and the remaining part of the URI is matched against the SNI in an equality match (since the SNI only defines Hostname types).
For CWTs issued to clients the application service interacting with the TLS/DTLS stack on the server side is responsible for authenticating the client. No specific rules apply but the Subject and the Audience claims are likely to be good candidates for authorization policy checks.

Note: Verification of the Not Before and the Expiration Time claims MUST be performed to determine the validity of the received CWT.

5. Security and Privacy Considerations

The security and privacy characteristics of this extension are best described in relationship to certificates (when asymmetric keys are used) and to Kerberos tickets (when symmetric keys are used) since the main difference is in the encoding.

When creating proof-of-possession keys the recommendations for state-of-the-art key sizes and algorithms have to be followed. For TLS/DTLS those algorithm recommendations can be found in [RFC7925] and [RFC7525].

CWTs without proof-of-possession keys MUST NOT be used.

When CWTs are used with TLS 1.3 [RFC8446] and DTLS 1.3 [I-D.ietf-tls-dtls13] additional privacy properties are provided since most handshake messages are encrypted.

6. IANA Considerations

IANA is requested to add a new value to the "TLS Certificate Types" subregistry for CWTs.

7. References

7.1. Normative References

[I-D.ietf-ace-cwt-proof-of-possession]

[I-D.ietf-tls-dtls13]
7.2. Informative References


7.3. URIs

[1] mailto:tls@ietf.org


Appendix A. History

RFC EDITOR: PLEASE REMOVE THE THIS SECTION

- 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]

Authors’ Addresses

Hannes Tschofenig
Arm Limited

EMail: hannes.tschofenig@arm.com

Mathias Brossard
Arm Limited

EMail: Mathias.Brossard@arm.com
Using Identity as Raw Public Key in Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)
draft-wang-tls-raw-public-key-with-ibc-11

Abstract

This document specifies the use of identity as a raw public key in Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS). The TLS protocol procedures are kept unchanged, but signature algorithms are extended to support Identity-based signature (IBS). A typical Identity-based signature algorithm, the ECCSI signature algorithm defined in RFC 6507, is supported in the current version.

Status of This Memo

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

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

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

This Internet-Draft will expire on April 12, 2020.

Copyright Notice

Copyright (c) 2019 IETF Trust and the persons identified as the document authors. All rights reserved.

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

DISCLAIMER: This is a personal draft and a limited security analysis is provided.

Traditionally, TLS client and server exchange public keys endorsed by PKIX [PKIX] certificates. It is considered complicated and may cause security weaknesses with the use of PKIX certificates Defeating-SSL [Defeating-SSL]. To simplify certificates exchange, using RAW public key with TLS/DTLS has been specified in [RFC 7250] and has been included in the TLS 1.3 [RFC 8446]. With RAW public key, instead of transmitting a full certificate or a certificate chain in the TLS messages, only public keys are exchanged between client and server. However, using RAW public key requires out-of-band mechanisms to verify the purported public key to the claimed entity.

Recently, 3GPP has adopted the EAP authentication framework for 5G and EAP-TLS is considered as one of the candidate authentication methods for private networks, especially for networks with a large number of IoT devices [TS33.501]. For IoT networks, TLS/DTLS with RAW public key is particularly attractive, but binding identities with public keys might be challenging. The cost to maintain a large
To simplify the binding between the public key and the entity, a better way could be using Identity-Based Cryptography (IBC), such as ECCSI public key specified in [RFC 6507], for authentication. Different from X.509 certificates and raw public keys, a public key in IBC takes the form of the entity’s identity. This eliminates the necessity of binding between a public key and the entity presenting the public key.

The concept of IBC was first proposed by Adi Shamir in 1984. As a special class of public key cryptography, IBC uses a user’s identity as public key, avoiding the hassle of public key certification in public key cryptosystems. IBC broadly includes IBE (Identity-based Encryption) and IBS (Identity-based Signature). For an IBC system to work, there exists a trusted third party, PKG (private key generator) responsible for issuing private keys to the users. In particular, the PKG has in possession a pair of Master Public Key and Master Secret Key; a private key is generated based on the user’s identity by using the Master Secret key, while the Master Public key is used together with the user’s identities for encryption (in case of IBE) and signature verification (in case of IBS). Another name of PKG is Key Management System (KMS), which is also used in some IBC system. In this document, the terms of PKG and KMS are interchangeable.

A number of IBE and IBS algorithms have been standardized by different standardization bodies, such as IETF, IEEE, ISO, etc. For example, IETF has specified several RFCs such as [RFC 5091], [RFC 6507] and [RFC 6508] for both IBE and IBS algorithms. ISO and IEEE also have a few standards on IBC algorithms, such as IBS1, IBS2, and ChineseIBS.

RFC 7250 has specified the use of raw public key with TLS/DTLS handshake. However, supporting of IBS algorithms has not been included therein. Since IBS algorithms are efficient in public key transmission and also eliminate the binding between public keys and identities, in this document, an amendment is added for supporting IBS algorithms as raw public key.

IBS algorithm exempts client and server from public key certification and identity binding by checking an entity’s signatures and its identity against the master public key of its PKG. With an IBS algorithm, a PKG generates private keys for entities based on their identities. Global parameters such as PKG’s Master Public Key (MPK) need be provisioned to both client and server. These parameters are not user specific, but PKG specific.
For a client, PKG specific parameters can be provisioned at the time PKG provisions the private key to the client. For the server, how to get the PKG specific parameters provisioned is out of the scope of this document, and it is deployment dependent.

The document is organized as follows: Section 3 defines the data structure required when identity is used as raw public key. Section 4 defines the cipher suites required to support IBS algorithm over TLS/DTLS. Section 5 explains how client and server authenticate each other when using identity as raw public key. Section 6 gives examples for using identity as raw public key over TLS/DTLS handshake procedure. Section 7 discusses the security considerations.

2. Terms

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.

3. Extension of RAW Public Key to IBC-based Public Key

To support the negotiation of using raw public between client and server, a new certificate structure is defined in RFC 7250. It is used by the client and server in the hello messages to indicate the types of certificates supported by each side.

When RawPublicKey type is selected for authentication, a data structure, subjectPublicKeyInfo, is used to carry the raw public key and its cryptographic algorithm. Within the subjectPublicKeyInfo structure, two fields, algorithm and subjectPublicKey, are defined. The algorithm is a data structure that specifies the cryptographic algorithm used with raw public key, which is represented by an object Identifiers (OID); and the parameters field provides necessary parameters associated with the algorithm. The subjectPublicKey field within the subjectPublicKeyInfo carries the raw public itself.
subjectPublicKeyInfo ::= SEQUENCE {
  algorithm            AlgorithmIdentifier,
  subjectPublicKey     BIT STRING
}

AlgorithmIdentifier ::= SEQUENCE {
  algorithm            OBJECT IDENTIFIER,
  parameters            ANY DEFINED BY algorithm OPTIONAL
}

Figure 1: SubjectPublicKeyInfo ASN.1 Structure

With IBS algorithm, identity is used as the raw public key, which can be converted to an BIT string and put into the subjectPublicKey field. The algorithm field in AlgorithmIdentifier structure is the object identifier of the IBS algorithm used. Specifically, for the ECCSI signature algorithm supported in this draft, the OBJECT IDENTIFIER is described with following data structure:

sa-eccsiWithSHA256 SIGNATURE-ALGORITHM ::= {
  IDENTIFIER id-alg-eccsi-with-sha256
  VALUE ECCSI-Sig-Value PARAMS TYPE NULL ARE absent
  HASHES { mda-sha256 }
  SMIME-CAPS { IDENTIFIED BY id-alg-eccsi-with-sha256 }
}

Figure 2: ECCSI Signature Algorithm ANSI.1 Structure

Note, in a real implementation, only IDENTIFIER part will be transmitted over the TLS negotiation protocols.

Beside OID, it is necessary to tell the peer the set of global parameters used by the signer. The information can be carried in the payload of the parameters field in AlgorithmIdentifier. On the other hand, when IBS algorithm is used for authentication, normally the global parameters in use are known to client and server, hence, instead of transmitting a full set of PKG public parameters, a hash value of them is transmitted, which is put in the parameters field of AlgorithmIdentifier data structure.

The data structure used to carry the hash value of public parameters is defined as follows:
IBSPublicParametersHash ::= SEQUENCE {
    HASHES { mda-sha256 }
}

Figure 3: IBS Global Parameters Hash ANSI.1 Structure

The hash value of the global parameters is generated by taking in the DER encoded PKG public parameters of each individual IBS algorithms as input. The data structure for each IBS algorithms supported in this draft are defined in the following.

For the ECCSI IBS signature algorithms, its PKG public parameters is specified in following Figure :

ECCSIPublicParameters ::= SEQUENCE {
    version INTEGER { v2(2) },
    curve OBJECT IDENTIFIER,
    hashfcn OBJECT IDENTIFIER,
    pointP FpPOINT,
    pointPpub FpPOINT
}

FpPoint ::= SEQUENCE {
    x INTEGER,
    y INTEGER
}

Figure 4: ECCSI Global Parameters ANSI.1 Structure

The structure to carry the ISO-IBS1/ISO-IBS2 PKG public parameters are the same and is specified in followng Figure:
ISOIBSPublicParameters ::= SEQUENCE {
    version INTEGER { v3(3) },
    curve OBJECT IDENTIFIER,
    hashfcn OBJECT IDENTIFIER,
    pairing PAIRING OPTIONAL,
    p INTEGER OPTIONAL,
    q INTEGER OPTIONAL,
    pointP FpPoint,
    pointPpub FpPoint
}

PAIRING ::= ENUMERATED{
    weil (1)  --Weil pairing
    tate (2)  --Tate pairing
    optimalAte (3)  --Optimal Ate pairing
}

Figure 5: ISO-IBS1/IBS2 Global Parameters ANSI.1 Structure

The structure to carry the ISO-SM9 PKG public parameters is specified in following Figure:
SM9PublicParameters ::= SEQUENCE {
  version INTEGER { v3(3) },
  curve OBJECT IDENTIFIER,
  hashfcn OBJECT IDENTIFIER,
  pairing PAIRING OPTIONAL,
  p INTEGER OPTIONAL,
  q INTEGER OPTIONAL,
  pointP2 FpxPoint,
  pointP2pub FpxPoint,
  y FpxElement
}

FpxPoint ::= CHOICE{
  fpPoint FpPoint,
  fp2Point [2] EXPLICIT Fp2Point,
}

Fp2Point ::= SEQUENCE{
  x Fp2Element,
  y Fp2Element
}

Fp2Element ::= SEQUENCE{
  a INTEGER,
  b INTEGER
}

FpxElement ::= CHOICE{
  fp2Elemt Fp2Element,
  fp12Elemt Fp12Element,
}

Fp12Element ::= SEQUENCE{
  a Fp6Element,
  b Fp6Element
}

Fp6Element ::= SEQUENCE{
  a Fp2Element,
  b Fp2Element,
  c Fp2Element
}

Figure 6: ISO-ChineseIBS Global Parameters ANSI.1 Structure

For ECCSIPublicParameters data structure, pointP shall be G in RFC 6507 and pointPpub shall be KPAK in RFC 6507. For
ISOIBSPublicParameters data structure, pointP and pointPpub shall be the same as defined in RFC 5091, and the pairing field shall be weil (1) or tate (2). The pairing field in SM9PublicParameters should be optimalAte (3) and the choice of v should be determined by the curve identifier. For example, for supersingular curves [RFC 5901], v shall be of type Fp2Element and for BN curves or BLS12-curves [FST10], v shall be of type Fp12Element.

To support IBS algorithm over TLS protocol, a data structure for signature value need to be defined.

Data structure for ECCSI is defined as follows (based RFC 6507):

```
ECCSI-Sig-Value ::= SEQUENCE {
    r INTEGER,
    s INTEGER,
    PVT OCTET STRING
}
```

Figure 7: ECCSI Signature Value ANSI.1 Structure

where PVT (as defined in RFC 6507) is encoded as 0x04 || x-coordinate of [v]G || y-coordinate of [v]G.

Data structure for ISO-IBS1 is defined as follows:

```
ISO-IBS1-Sig-Value ::= SEQUENCE {
    r INTEGER,
    s ECPoint
}
```

Figure 8: ISO-IBS1 Signature Value ANSI.1 Structure

Data structure for ISO-IBS2 is defined as follows:

```
ISO-IBS2-Sig-Value ::= SEQUENCE {
    r INTEGER,
    s ECPoint
}
```

Figure 9: ISO-IBS2 Signature Value ANSI.1 Structure

Data structure for ISO-ChineseIBS (SM9) is defined as follows:
SM9-Sig-Value ::= SEQUENCE {
  r INTEGER,
  s ECPoint
}

Figure 10: ISO-ChineseIBS Signature Value ANSI.1 Structure

The definition of ECPoint can be found in section 2.2 of RFC 5480.

To use a signature algorithm with TLS, OID for the signature algorithm need be provided. For ECCSI algorithm, an OID has been assigned by IANA recently. The following table shows the basic information needed for the ECCSI signature algorithm to be used for TLS.

<table>
<thead>
<tr>
<th>Key Type</th>
<th>Document</th>
<th>OID</th>
</tr>
</thead>
<tbody>
<tr>
<td>ISO/IEC 14888-3 IBS-1</td>
<td>ISO/IEC 14888-3: IBS-1</td>
<td>1.0.14888.3.0.7</td>
</tr>
<tr>
<td>ISO/IEC 14888-3 IBS-2</td>
<td>ISO/IEC 14888-3: IBS-2</td>
<td>1.0.14888.3.0.8</td>
</tr>
<tr>
<td>ISO/IEC 14888-3 ChineseIBS(SM9)</td>
<td>ISO/IEC 14888-3: ChineseIBS</td>
<td>1.2.156.10197.1.302.1</td>
</tr>
<tr>
<td>Elliptic Curve-Based Signatureless For Identity-based Encryption (ECCSI)</td>
<td>Section 5.2 in RFC 6507</td>
<td>1.3.6.1.5.5.7.6.29</td>
</tr>
</tbody>
</table>

Table 1: Algorithm Object Identifiers

4. New Signature Algorithms for IBS

To using identity as raw public key, new signature algorithms corresponding to the IBS need to be defined. With TLS 1.3, the value for signature algorithm is defined in the SignatureScheme. This document specifies how to support IBS algorithm. As a result, the SignatureScheme data structure has to be amended by including the presented IBS algorithms.
enum {
  ...

  /* IBS ECCSI signature algorithm */
  eccsi_sha256 (TBD),
  iso_ibs1 (TBD),
  iso_ibs2 (TBD),
  iso_chinese_ibs (TBD),

  /* Reserved Code Points */
  private_use (0xFE00..0xFFFF),
  (0xFFFF)
} SignatureScheme;

Figure 11: Include IBS in KeyExchangeAlgorithm

Note: The signature algorithm of eccsi_sha256 is defined in RFC6507.

Note: Other IBS signature algorithms can be added in the future.

5. Identity Format and Key Revocation

With the raw public scheme proposed in TLS 1.3 [RFC 8446], server maintains a whitelist to bind raw public key and identity. When a raw public key is revoked, then server removes the binding record from the whitelist. On the other hand, when using IBS algorithm for raw public key, there is no whitelist at server side. Instead, the server need to maintain a blacklist, which is much shorter than the whitelist, to support public key revocation. However, if we simply using the identifier as raw public key, the revocation list may keep on increasing with the time going on. Hence, to prevent the revocation list from increasing continuously, it is recommended to include a timestamp for automatic expiration of key material. With the timestamp included in the identifier, i.e. the raw public key, server can remove revoked raw public key from revocation list when it is expired.

Based on the above analysis, it is necessary to include expiration time in the identifiers for the purpose of public key management. Therefore, in this draft, we recommend both client and server take following format for the identifiers used for TLS session setup:
Identifier ::= SEQUENCE {
    version INTEGER {v1 (1)},
    identity String,
    expiration UTCTime
}

Figure 12: Identifier Format ANSI.1 Structure

Both the client and server should check the validity of the expiration field of the raw public key before verify the signature. If the expiration time is invalid, the client or the server should abort the handshake procedure.

The identities of client or server shall be unique within the domain managed by one PKG. There are many different identities domains such as email address, telephone number, Network Access Identifier (NAI), International Mobile Subscriber Identity (IMSI) etc. It is up to network operators’s choice to determine which name domain the device and server take.

6. TLS Client and Server Handshake Behavior

When IBS is used as RAW public for TLS, signature and hash algorithms are negotiated during the handshake.

The handshake between the TLS client and server follows the procedures defined in [RFC 8446], but with the support of the new signature algorithms specific to the IBS algorithms. The high-level message exchange in the following figure shows TLS handshake using raw public keys, where the client_certificate_type and server_certificate_type extensions added to the client and server hello messages (see Section 4 of [RFC 7250]).
The client hello messages tells the server the types of certificate or raw public key supported by the client, and also the certificate types that client expects to receive from server. When raw public with IBS algorithm from server is supported by the client, the client includes desired IBS signature algorithm in the client hello message based on the order of client preference.

After receiving the client hello message, server determines the client and server certificate types for handshakes. When the selected certificate type is RAW public key and IBS is the chosen signature algorithm, server uses the SubjectPublicKeyInfo structure to carry the raw public key, OID for IBS algorithm. Assuming that ECCSI is selected, the ECCSIPublicParameters data structure is used to carry global public parameters. With these information, the client knows the signature algorithm and the public parameters that should be used to verify the signature. The signature value is in the CertificateVerify message and the format of signature value is specified by the selected IBS algorithm. The data structures for PKG public parameters and signature values have been specified in the previous section of this document.

When sever specifies that RAW public key should be used by client to authenticate with server, the client_certificate_type in the server hello is set to RawPublicKey. Besides that, the server also sends Certificate Request, indicating that client should use some specific
signature and hash algorithms. When IBS is chosen as signature algorithm, the server need to indicate the required IBS signature algorithms in the signature_algorithm extension within the CertificateRequest.

After receiving the server hello, the client checks the CertificateRequest for signature algorithms. If client wants to use an IBS algorithm for signature, then the signature algorithm it intended to use must be in the list of supported signature algorithms specified by the server. Assume the IBS algorithm supported by the client is in the list, then the client response with the IBS signature algorithm and PKG information with SubjectPublicKeyInfo structure in the certificate structure and provide signatures in the certificate verify message. The format of signature in the CertificateVerify message should be specified by each individual signature algorithm.

The server verifies the signature based on the chosen IBS algorithm and the relevant PKG parameters specified by the client.

7. Examples

In the following, examples of handshake exchange using IBS algorithm under RawPublicKey are illustrated.

7.1. TLS Client and Server Use IBS algorithm

In this example, both the TLS client and server use ECCSI for authentication, and they are restricted in that they can only process ECCSI signature algorithm. As a result, the TLS client sets both the server_certificate_type and the client_certificate_type extensions to be raw public key; in addition, the client sets the signature algorithm in the client hello message to be eccsi_sha256.

When the TLS server receives the client hello, it processes the message. Since it has an ECCSI raw public key from the PKG, it indicates in (2) that it agrees to use ECCSI and provides an ECCSI key by placing the SubjectPublicKeyInfo structure into the Certificate payload back to the client (3), including the OID, the identity of server, ServerID, which is the public key of server also, and hash value of PKG public parameters. The client_certificate_type in (4) indicates that the TLS server accepts raw public key. The TLS server demands client authentication, and therefore includes a certificate_request(5), which requires the client to use eccsi_sha256 for signature. A signature value based on the eccsi_sha256 algorithm is carried in the CertificateVerify (6). The client, which has an ECCSI key, returns its ECCSI public key in the Certificate payload to the server (7), which includes an OID for the ECCSI signature.
algorithm, the PKGInfo for KMS parameters, and identity of client, ClientID, which is the public key of client also. The client also includes a signature value, ECCSI-Sig-Value, in the CertificateVerify (8) message.

When client/server receive PKG public parameters from peer, it should decide whether these parameters are acceptable or not. An example way to make decision is that a whitelist of acceptable PKG public parameters are stored locally at client/server. They can simply make a decision based on the white list stored locally.

client_hello,
+key_share                          //(1)
signature_algorithm = (eccsi_sha256)  //(1)
client_certificate_type=(RawPublicKey) // (1)
server_certificate_type=(RawPublicKey) // (1)

<- server_hello,
+  key_share
  {server_certificate_type = RawPublicKey}  // (2)
  {certificate=((1.3.6.1.5.5.7.6.29, hash value of ECCSIPublicParameters),
   serverID})                            // (3)
  {client_certificate_type = RawPublicKey} // (4)
  {certificate_request = (eccsi_sha256)}    // (5)
  {CertificateVerify = (ECCSI-Sig-Value)    // (6)
  {Finished}

{Certificate=({
  (1.3.6.1.5.5.7.6.29, hash value of ECCSIPublicParameters),
  ClientID})}                          // (7)
{CertificateVerify = (ECCSI-Sig-Value)}    // (8)
{Finished}

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

Figure 14: Basic Raw Public Key TLS Exchange

7.2. Combined Usage of Raw Public Keys and X.509 Certificates

This example combines the uses of an ECCSI key and an X.509 certificate. The TLS client uses an ECCSI key for client authentication, and the TLS server provides an X.509 certificate for server authentication.

The exchange starts with the client indicating its ability to process a raw public key, or an X.509 certificate, if provided by the server.
It prefers a raw public key with ECCSI signature algorithm since eccsi_sha256 precedes the ecdsa_secp256r1_sha256. Furthermore, the client indicates that it has a ECCSI-based raw public key for client-side authentication. Client also indicate it supports server using either ECCSI or ecdsa_secp256r1_sha256 for the certificate signature. This further indicates that server can use ecdsa_secp256r1_sha256 to sign the message.

With the received client_hello, the server chooses to provide its X.509 certificate in (3) and indicates that choice in (2). For client authentication, the server indicates in (4) that it has selected the raw public key format and requests an ECCSI certificate from the client in (4) and (5). The TLS client provides an ECCSI certificate in (6) and signature value after receiving and processing the TLS server hello message.

client_hello,
+key_share
signature_algorithms = (eccsi_sha256, ecdsa_secp256r1_sha256) // (1)
signature_algorithms_cert = (eccsi_sha256, ecdsa_secp256r1_sha256) // (1)
{client_certificate_type= (RawPublicKey)}                        // (1)
{server_certificate_type= (RawPublicKey, X.509)                  // (1)

->

<- server_hello,
+key_share
{server_certificate_type=X.509}               // (2)
{Certificate = (x.509 certificate)}       // (3)
{client_certificate_type = (RawPublicKey)}  // (4)
{CertificateRequest} = (eccsi_sha256)}    // (5)
{CertificateVerify}
{Finished}

certificate=(
(1.3.6.1.5.5.7.6.29,
ECCSIPublicParameters),
ClientID),  // (6)
{CertificateVerify = (ECCSI-Sig-Value)}     // (7)
{ Finished }
[Application Data] ---->
[Application Data] <---- [Application Data]

Figure 15: Basic Raw Public Key TLS Exchange
Handshake for other IBS algorithms can be completed similarly by including different data structures for public parameters and signature values respectively.

8. Security Considerations

Using IBS-based raw public key in TLS/DTLS does not change the message flows of TLS, hence, for the most part, the security considerations involved in using the Transport Layer Security protocol with raw public key also apply here. The additional security of the resulting protocol rests on the security of the used IBS algorithms.

IBS signature algorithm has been standardized for ten years and has been adopted in real applications. However, we would like to point out the difference between IBS signature algorithm and existing raw public key: the private key of IBS used for signature generation is generated by the PKG centre, while the private key for the existing raw public key is generated locally. Therefore, IBS mechanism may face a security risk of private key disclosure due to improper management of KMS system. The entity using IBS with TLS protocol shall be aware the above risk and an enforced key management system shall be adopted by the organization.

When using IBS algorithm, key escrow is an concern as the private key of user or devices normally is generated by PKG. PKG in the system which could generate each device’s private key. However, when IBS is used in TLS1.3, passively attack to recover the session key is not possible. Actively man-in-the-middle attack by replacing exchanged DH tokens and signatures would certainly leave traces even transiently. Similarly, a PKG could impersonate an entity to conduct a TLS session, just as the KMS in the symmetric key solution, but forensic traces could be also collected in this situation. It would be hugely risky for a PKG, which would usually be a trusted party, to launch such attacks. If such an attack is caught in red-handed, no one would trust the PKG’s service anymore.

Another worry of using IBS is about the compromising of PKG. The PKG could become operationally compromised and an attacker may obtain master secrets of a PKG. However, this security risk can be solved by protect the PKG with HSM, which is often used by CA to protect the root signing key.

Private key compromising is one security risk that need to be considered when using public key technology. When using IBS for raw public, as we have suggested in this document, a revocation list shall be maintained at the server side. At the same time, a timestamp shall be included in the public key to prevent the
revocation list from keeping on increasing. With the revocation list, server can prevent following attacks:

1) when a device using a revoked identifier for authentication, which has not been expired yet, then the server can reject the TLS session by checking the revocation list maintained at the server side. As it is in the list, then the server aborts the TLS handshake.

2) When a device using a identifier which has been expired, the server can simply verify the timestamp contained in the identifier and abort the handshake procedure immediately.

3) If the attacker changes the timestamp within the identifier, then it will cause signature verification error when server verify the signature contained in the signature_verify from client.

9. IANA Considerations

IANA has assigned 4 code points from the TLS SignatureScheme registry for the four IBS algorithms used in this document. The code points are listed as follows:

- eccsi_sha256
- iso.ibs1
- iso.ibs2
- iso.chinese.ibs

For all of these entries the Recommended field should be N, and the Reference field should be this document.

10. Acknowledgements

11. References

11.1. Normative References


11.2. Informative References

[Defeating-SSL] "New Tricks for Defeating SSL in Practice", Feb 2009,

Wang, et al. Expires April 12, 2020


Authors’ Addresses

Haiguang Wang (editor)
Huawei International Pte. Ltd.
11 North Buona Vista Dr, #17-08
Singapore 138589
SG

Phone: +65 6825 4200
Email: wang.haiguang1@huawei.com

Yanjiang Yang
Huawei International Pte. Ltd.
11 North Buona Vista Dr, #17-08
Singapore 138589
SG

Phone: +65 6825 4200
Email: yang.yanjiang@huawei.com

Xin Kang
Huawei International Pte. Ltd.
11 North Buona Vista Dr, #17-08
Singapore 138589
SG

Phone: +65 6825 4200
Email: xin.kang@huawei.com
Abstract

This document describes an interface for importing external PSK (Pre-Shared Key) into TLS.

Status of This Memo

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

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

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

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

Copyright Notice

Copyright (c) 2019 IETF Trust and the persons identified as the document authors. All rights reserved.

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

TLS 1.3 [RFC8446] supports pre-shared key (PSK) resumption, wherein PSKs can be established via session tickets from prior connections or externally via some out-of-band mechanism. The protocol mandates that each PSK only be used with a single hash function. This was done to simplify protocol analysis. TLS 1.2, in contrast, has no such requirement, as a PSK may be used with any hash algorithm and the TLS 1.2 PRF. This means that external PSKs could possibly be re-used in two different contexts with the same hash functions during key derivation. Moreover, it requires external PSKs to be provisioned for specific hash functions.

To mitigate these problems, external PSKs can be bound to a specific hash function when used in TLS 1.3, even if they are associated with a different KDF (and hash function) when provisioned. This document specifies an interface by which external PSKs may be imported for use in a TLS 1.3 connection to achieve this goal. In particular, it describes how KDF-bound PSKs can be differentiated by different hash algorithms to produce a set of candidate PSKs, each of which are bound to a specific hash function. This expands what would normally have been a single PSK identity into a set of PSK identities. However, it requires no change to the TLS 1.3 key schedule.

2. Conventions and Definitions

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

Intuitively, key importers mirror the concept of key exporters in TLS in that they diversify a key based on some contextual information before use in a connection. In contrast to key exporters, wherein differentiation is done via an explicit label and context string, the key importer defined herein uses a label and set of hash algorithms to differentiate an external PSK into one or more PSKs for use.

Imported keys do not require negotiation for use, as a client and server will not agree upon identities if not imported correctly. Thus, importers induce no protocol changes with the exception of expanding the set of PSK identities sent on the wire.

3.1. Terminology

- **External PSK (EPSK):** A PSK established or provisioned out-of-band, i.e., not from a TLS connection, which is a tuple of (Base Key, External Identity, KDF). The associated KDF (and hash function) may be undefined.
- **Base Key:** The secret value of an EPSK.
- **External Identity:** The identity of an EPSK.
- **Imported Identity:** The identity of a PSK as sent on the wire.

4. Key Import

A key importer takes as input an EPSK with external identity ‘external_identity’ and base key ‘epsk’, as defined in Section 3.1, along with an optional label, and transforms it into a set of PSKs and imported identities for use in a connection based on supported HashAlgorithms. In particular, for each supported HashAlgorithm ‘hash’, the importer constructs an ImportedIdentity structure as follows:

```c
struct {
    opaque external_identity<1...2^16-1>;
    opaque label<0..2^8-1>;
    HashAlgorithm hash;
} ImportedIdentity;
```
ImportedIdentity.label MUST be bound to the protocol for which the key is imported. Thus, TLS 1.3 and QUICv1 [I-D.ietf-quic-transport] MUST use "tls13" as the label. Similarly, TLS 1.2 and all prior TLS versions should use "tls12" as ImportedIdentity.label, as well as SHA256 as ImportedIdentity.hash. Note that this means future versions of TLS will increase the number of PSKs derived from an external PSK.

A unique and imported PSK (IPSK) with base key 'ipskx' bound to this identity is then computed as follows:

\[
\begin{align*}
\text{epskx} &= \text{HKDF-Extract}(0, \text{epsk}) \\
\text{ipskx} &= \text{HKDF-Expand-Label}(\text{epskx}, \text{"derived psk"}, \\
&\quad \text{Hash(ImportedIdentity)}, \text{Hash.length})
\end{align*}
\]

The hash function used for HKDF [RFC5869] is that which is associated with the external PSK. It is not bound to ImportedIdentity.hash. If no hash function is specified, SHA-256 MUST be used. Differentiating epsk by ImportedIdentity.hash ensures that each imported PSK is only used with at most one hash function, thus satisfying the requirements in [RFC8446]. Endpoints MUST import and derive an ipsk for each hash function used by each ciphersuite they support. For example, importing a key for TLS_AES_128_GCM_SHA256 and TLS_AES_256_GCM_SHA384 would yield two PSKs, one for SHA256 and another for SHA384. In contrast, if TLS_AES_128_GCM_SHA256 and TLS_CHACHA20_POLY1305_SHA256 are supported, only one derived key is necessary.

The resulting IPSK base key ‘ipskx’ is then used as the binder key in TLS 1.3 with identity ImportedIdentity. With knowledge of the supported hash functions, one may import PSKs before the start of a connection.

EPSKs may be imported for early data use if they are bound to protocol settings and configurations that would otherwise be required for early data with normal (ticket-based PSK) resumption. Minimally, that means ALPN, QUIC transport settings, etc., must be provisioned alongside these EPSKs.
5. Deprecating Hash Functions

If a client or server wish to deprecate a hash function and no longer use it for TLS 1.3, they may remove this hash function from the set of hashes used during while importing keys. This does not affect the KDF operation used to derive concrete PSKs.

6. Backwards Compatibility

Recall that TLS 1.2 permits computing the TLS PRF with any hash algorithm and PSK. Thus, an external PSK may be used with the same KDF (and underlying HMAC hash algorithm) as TLS 1.3 with importers. However, critically, the derived PSK will not be the same since the importer differentiates the PSK via the identity and hash function. Thus, PSKs imported for TLS 1.3 are distinct from those used in TLS 1.2, and thereby avoid cross-protocol collisions.

7. Security Considerations

This is a WIP draft and has not yet seen significant security analysis.

8. Privacy Considerations

DISCLAIMER: This section contains a sketch of a design for protecting external PSK identities. It is not meant to be implementable as written.

External PSK identities are typically static by design so that endpoints may use them to lookup keying material. For some systems and use cases, this identity may become a persistent tracking identifier. One mitigation to this problem is encryption. Future drafts may specify a way for encrypting PSK identities using a mechanism similar to that of the Encrypted SNI proposal [I-D.ietf-tls-esni]. Another approach is to replace the identity with an unpredictable or "obfuscated" value derived from the corresponding PSK. One such proposal, derived from a design outlined in [I-D.ietf-dnssd-privacy], is as follows. Let ipskx be the imported PSK with identity ImportedIdentity, and N be a unique nonce of length equal to that of ImportedIdentity.hash. With these values, construct the following "obfuscated" identity:

```c
struct {
    opaque nonce[hash.length];
    opaque obfuscated_identity<1..2^16-1>;<
    HashAlgorithm hash;
} ObfuscatedIdentity;
```
ObfuscatedIdentity.nonce carries N, ObfuscatedIdentity.obfuscated_identity carries HMAC(ipskx, N), where HMAC is computed with ImportedIdentity.hash, and ObfuscatedIdentity.hash is ImportedIdentity.hash.

Upon receipt of such an obfuscated identity, a peer must lookup the corresponding PSK by exhaustively trying to compute ObfuscatedIdentity.obfuscated_identity using ObfuscatedIdentity.nonce and each of its known imported PSKs. If N is chosen in a predictable fashion, e.g., as a timestamp, it may be possible for peers to precompute these obfuscated identities to ease the burden of trial decryption.

9. IANA Considerations

This document makes no IANA requests.

10. References

10.1. Normative References


10.2. Informative References

[I-D.ietf-dnssd-privacy]

[I-D.ietf-tls-esni]

Appendix A. Acknowledgements

The authors thank Eric Rescorla and Martin Thomson for discussions that led to the production of this document, as well as Christian Huitema for input regarding privacy considerations of external PSKs.

Authors’ Addresses

David Benjamin
Google, LLC.

Email: davidben@google.com

Christopher A. Wood
Apple, Inc.

Email: cawood@apple.com