Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: September 2, 2017

A. Ghedini
Cloudflare, Inc.
V. Vasiliev
Google

March 01, 2017

# Transport Layer Security (TLS) Certificate Compression
# draft-ghedini-tls-certificate-compression-00

## Abstract

In Transport Layer Security (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 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 2, 2017.

## Copyright Notice

Table of Contents



## 1. Introduction

In order to reduce latency and improve performance it can be useful to reduce the amount of data exchanged during a Transport Layer Security (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 server certificates to be compressed during full handshakes.

## 2. Notational Conventions

The words "MUST", "MUST NOT", "SHALL", "SHOULD", and "MAY" are used in this document. It's not shouting; when they are capitalized, they have the special meaning defined in [RFC2119].

## 3. Negotiating Certificate Compression

This document defines a new extension type (compress_server_certificates(TBD)), which is used by the client and the server to negotiate the use of compression for the server certificate chain, as well as the choice of the compression algorithm.

By sending the compress_server_certificates message, the client indicates to the server the certificate compression algorithms it

supports. The "extension_data" field of this extension in the ClientHello SHALL contain a CertificateCompressionAlgorithms value:

```
enum {
    zlib(0),
    brotli(1),
    (255)
} CertificateCompressionAlgorithm;

struct {
    CertificateCompressionAlgorithm algorithms<1..2^8>;
} CertificateCompressionAlgorithms;
```

If the server supports any of the algorithms offered in the ClientHello, it MAY respond with an extension indicating which compression algorithm it chose. In that case, the extension_data SHALL be a CertificateCompressionAlgorithm value corresponding to the chosen algorithm. If the server has chosen to not use any compression, it MUST NOT send the compress_server_certificates extension.

## 4. Server Certificate Message

If the server picks a compression algorithm and sends it in the ServerHello, the format of the Certificate message is altered as follows:

```
struct {
     uint24 uncompressed_length;
     opaque compressed_certificate_message<1..2^24-1>;
} Certificate;
```

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 client MUST abort the connection with the "bad_certificate" alert.

compressed_certificate_message The compressed body of the Certificate message, in the same format as the server would normally express it. The compression algorithm defines how the bytes in the compressed_certificate_message 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 Brotli compression algorithm as defined in [RFC7932].

If the client cannot decompress the received Certificate message from the server, it MUST tear down the connection with the "bad_certificate" alert.

The extension only affects the Certificate message from the server. It does not change the format of the Certificate message sent by the client.

If the format of the message is altered using the server_certificate_type extension [RFC7250], the resulting altered message is compressed instead.

If the server chooses to use the cached_info extension [RFC7924] to replace the Certificate message with a hash, it MUST NOT send the compress_server_certificates extension.

## 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.

Since certificate chains are typically presented on a per-server name 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 Certificate 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. Implementations MAY impose a lower limit on the chain size in addition to the 16777216 byte limit imposed by TLS framing, in which case they MUST apply the same limit to the uncompressed chain before starting to decompress it.

## 6. IANA Considerations

### 6.1. Update of the TLS ExtensionType Registry

Create an entry, compress_server_certificates(TBD), in the existing registry for ExtensionType (defined in [RFC5246]).

### 6.2. 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 | zlib |
| 1 | brotli |
| 224 to 255 | Reserved for Private Use |

The values in this registry shall be allocated under "IETF Review" policy for values strictly smaller than 64, and under "Specification Required" policy otherwise (see [RFC5226] for the definition of relevant policies).

## 7. Normative References

[RFC1950] Deutsch, P. and J-L. Gailly, "ZLIB Compressed Data Format Specification version 3.3", RFC 1950, DOI 10.17487/RFC1950, May 1996, <http://www.rfc-editor.org/info/rfc1950>.

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

[RFC4366] Blake-Wilson, S., Nystrom, M., Hopwood, D., Mikkelsen, J., and T. Wright, "Transport Layer Security (TLS) Extensions", RFC 4366, DOI 10.17487/RFC4366, April 2006, <http://www.rfc-editor.org/info/rfc4366>.

[RFC5226] Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 5226, DOI 10.17487/RFC5226, May 2008, <http://www.rfc-editor.org/info/rfc5226>.

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

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

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

[RFC7932] Alakuijala, J. and Z. Szabadka, "Brotli Compressed Data Format", RFC 7932, DOI 10.17487/RFC7932, July 2016, <http://www.rfc-editor.org/info/rfc7932>.

## Authors' Addresses

Alessandro Ghedini
Cloudflare, Inc.

Email: alessandro@cloudflare.com

Victor Vasiliev
Google

Email: vasilvv@google.com

TLS
Internet-Draft
Intended status: Standards Track
Expires: September 28, 2017

M. Shore
Fastly
R. Barnes
Mozilla
S. Huque
Salesforce
W. Toorop
NLNet Labs
March 27, 2017

# A DANE Record and DNSSEC Authentication Chain Extension for TLS
draft-ietf-tls-dnssec-chain-extension-03

## Abstract

This draft describes a new TLS extension for transport of a DNS record set serialized with the DNSSEC signatures needed to authenticate that record set. The intent of this proposal is to allow TLS clients to perform DANE authentication of a TLS server certificate without needing to perform additional DNS record lookups. It will typically not be used for general DNSSEC validation of TLS endpoint names.

## 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 28, 2017.

## Copyright Notice

Table of Contents



## 1. Requirements Notation

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

## 2. Introduction

This draft describes a new TLS [RFC5246] extension for transport of a DNS record set serialized with the DNSSEC signatures [RFC4034] needed to authenticate that record set. The intent of this proposal is to allow TLS clients to perform DANE authentication [RFC6698] of a TLS server certificate without performing additional DNS record lookups and incurring the associated latency penalty. It also provides the

ability to avoid potential problems with TLS clients being unable to look up DANE records because of an interfering or broken middlebox on the path between the client and a DNS server. And lastly, it allows a TLS client to validate DANE records itself without necessarily needing access to a validating DNS resolver to which it has a secure connection. It will typically not be used for general DNSSEC validation of endpoint names, but is more appropriate for validation of DANE TLSA records.

This mechanism is useful for TLS applications that need to address the problems described above, typically web browsers or VoIP and XMPP applications. It may not be relevant for many other applications. For example, SMTP MTAs are usually located in data centers, may tolerate extra DNS lookup latency, are on servers where it is easier to provision a validating resolver, or are less likely to experience traffic interference from misconfigured middleboxes. Furthermore, SMTP MTAs usually employ Opportunistic Security [RFC7435], in which the presence of the DNS TLSA records is used to determine whether to enforce an authenticated TLS connection. Hence DANE authentication of SMTP MTAs [RFC7672] will typically not use this mechanism.

The extension described here allows a TLS client to request in the ClientHello message that the DNS authentication chain be returned in the (extended) ServerHello message. If the server is configured for DANE authentication, then it performs the appropriate DNS queries, builds the authentication chain, and returns it to the client. The server will usually use a previously cached authentication chain, but it will need to rebuild it periodically as described in Section 5. The client then authenticates the chain using a pre-configured trust anchor.

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

As described in the DANE specification [RFC6698], this procuedure applies to the DANE authentication of X.509 certificates. Other credentials may be supported, as needed, in the future.

## 3. DNSSEC Authentication Chain Extension

### 3.1. Protocol, TLS 1.2

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

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

A client must not be able to force a server to perform lookups on arbitrary domain names using this mechanism. Therefore, a server MUST NOT construct chains for domain names other than its own.

### 3.2. Protocol, TLS 1.3

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

Servers receiving a "dnssec_chain" extension in the ClientHello, and which are capable of being authenticated via DANE, SHOULD return a serialized authentication chain in the Certificate message associated with the end entity certificate being validated, using the format described below. The authentication chain will be an extension to the certificate_list to which the certificate being authenticated belongs.

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

### 3.3. Raw Public Keys

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

name and provides a means to commit both the server and the client to using that binding.

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

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

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

The mechanism for conveying DNSSEC validation chains described in this document results in a commitment by both parties, via the TLS handshake, to a domain name which has been validated as belonging to the owner name.

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

### 3.4. DNSSEC Authentication Chain Data

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

```
opaque AuthenticationChain<0..2^16-1>
```

The AuthenticationChain structure is composed of a sequence of uncompressed wire format DNS resource record sets (RRset) and corresponding signatures (RRsig) records. The record sets and signatures are presented in the order returned by the DNS server queried by the TLS server, although they MAY be returned in validation order, starting at the target DANE record, followed by the DNSKEY and DS record sets for each intervening DNS zone up to a trust anchor chosen by the server, typically the DNS root.

This sequence of native DNS wire format records enables easier generation of the data structure on the server and easier verification of the data on client by means of existing DNS library functions. However this document describes the data structure in sufficient detail that implementers if they desire can write their own code to do this.

Each RRset in the chain is composed of a sequence of wire format DNS resource records. The format of the resource record is described in RFC 1035 [RFC1035], Section 3.2.1. The resource records SHOULD be presented in the canonical form and ordering as described in RFC 4034 [RFC4034].

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

RRs within the RRset MAY be ordered canonically, by treating the RDATA portion of each RR as a left-justified unsigned octet sequence in which the absence of an octet sorts before a zero octet.

The RRsig record is in DNS wire format as described in RFC 4034 [RFC4034], Section 3.1. The signature portion of the RDATA, as described in the same section, is the following:

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

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

The first RRset in the chain MUST contain the DANE records being presented. The subsequent RRsets MUST be a sequence of DNSKEY and DS RRsets, starting with a DNSKEY RRset. Each RRset MUST authenticate the preceding RRset:

- A DNSKEY RRset must include the DNSKEY RR containing the public key used to verify the previous RRset.
- For a DS RRset, the set of key hashes MUST overlap with the preceding set of DNSKEY records.

In addition, a DNSKEY RRset followed by a DS RRset MUST be self-signed, in the sense that its RRSIG MUST verify under one of the keys in the DNSKEY RRSET.

The final DNSKEY RRset in the authentication chain, containing the trust anchor may be omitted. If omitted, the client MUST verify that

the key tag and owner name in the final RRSIG record correspond to a trust anchor. There may however be reason to include the trust anchor RRset and signature if clients are expected to use RFC5011 compliant key rollover functions inband via the chain data. In that case, they will need to periodically inspect flags (revocation and secure entry point flags) on the trust anchor DNSKEY RRset.

For example, for an HTTPS server at www.example.com, where there are zone cuts at "com." and "example.com.", the AuthenticationChain structure would comprise the following RRsets and signatures (the data field of the records are omitted here for brevity):

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

Names that are aliased via CNAME and/or DNAME records may involve multiple branches of the DNS tree. In this case the authentication chain structure will be composed of a sequence of these multiple intersecting branches. DNAME chains should omit unsigned CNAME records that may have been synthesized in the response from a DNS resolver. Wildcard DANE records will need to include the wildcard name, and negative proof (i.e. NSEC or NSEC3 records) that no closer name exists MUST be included.

A CNAME example:

```
_443._tcp.www.example.com.   IN   CNAME    ca.example.net.
ca.example.net.              IN   TLSA     2 0 1 ...
```

Here the authentication chain structure is composed of two consecutive chains, one for _443._tcp.www.example.com/CNAME and one for ca.example.net/TLSA. The second chain can omit the record sets at the end that overlap with the first.

TLS DNSSEC chain components:

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

ca.example.net. TLSA
RRSIG(ca.example.net. TLSA)
example.net. DNSKEY
RRSIG(example.net. DNSKEY)
example.net. DS
RRSIG(example.net. DS)
net. DNSKEY
RRSIG(net. DNSKEY)
net. DS
RRSIG(net. DS)
```

Note as well that if a user has a specific TLSA record for port 443, and a different wildcard covering other ports, attackers MUST NOT be able to substitute the wildcard TLSA RRset for the more specific one for port 443. DNSSEC wildcards must not be confused with the X.509 wildcards.

## 4. Construction of Serialized Authentication Chains

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

When the goal is to perform DANE authentication [RFC6698] of the server's X.509 certificate, the DNS record set to be serialized is a TLSA record set corresponding to the server's domain name.

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

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

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

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

## 5. Caching and Regeneration of the Authentication Chain

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

## 6. Verification

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

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

## 7. Trust Anchor Maintenance

The trust anchor may change periodically, e.g. when the operator of the trust anchor zone performs a DNSSEC key rollover. Managed key rollovers typically use a process that can be tracked by verifiers allowing them to automatically update their trust anchors, as described in [RFC5011]. TLS clients using this specification are also expected to use such a mechanism to keep their trust anchors updated. Some operating systems may have a system-wide service to maintain and keep the root trust anchor up to date. In such cases, the TLS client application could simply reference that as its trust anchor, periodically checking whether it has changed.

## 8. Mandating use of this extension

A TLS server certificate MAY mandate the use of this extension by means of the X.509 TLS Feature Extension described in [RFC7633]. This X.509 certificate extension, when populated with the dnssec_chain TLS extension identifier, indicates to the client that the server must deliver the authentication chain when asked to do so. (The X.509 TLS Feature Extension is the same mechanism used to deliver other mandatory signals, such as OCSP "must staple" assertions.)

## 9. Security Considerations

The security considerations of the normatively referenced RFCs (1035, 4034, 4035, 5246, 6066, 6698, 7633, 7671) all pertain to this extension. Since the server is delivering a chain of DNS records and signatures to the client, it MUST rebuild the chain in accordance with TTL and signature expiration of the chain components as described in Section 5. TLS clients need roughly accurate time in order to properly authenticate these signatures. This could be achieved by running a time synchronization protocol like NTP

[RFC5905] or SNTP [RFC5905], which are already widely used today. TLS clients MUST support a mechanism to track and rollover the trust anchor key, or be able to avail themselves of a service that does this, as described in Section 7.

## 10. IANA Considerations

This extension requires the registration of a new value in the TLS ExtensionsType registry. The value requested from IANA is 53. If the draft is adopted by the WG, the authors expect to make an early allocation request as specified in [RFC7120].

## 11. Acknowledgments

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

## 12. References

### 12.1. Normative References

[RFC1035] Mockapetris, P., "Domain names - implementation and specification", STD 13, RFC 1035, DOI 10.17487/RFC1035, November 1987, <http://www.rfc-editor.org/info/rfc1035>.

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

[RFC4034] Arends, R., Austein, R., Larson, M., Massey, D., and S. Rose, "Resource Records for the DNS Security Extensions", RFC 4034, DOI 10.17487/RFC4034, March 2005, <http://www.rfc-editor.org/info/rfc4034>.

[RFC4035] Arends, R., Austein, R., Larson, M., Massey, D., and S. Rose, "Protocol Modifications for the DNS Security Extensions", RFC 4035, DOI 10.17487/RFC4035, March 2005, <http://www.rfc-editor.org/info/rfc4035>.

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

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

[RFC6698] Hoffman, P. and J. Schlyter, "The DNS-Based Authentication of Named Entities (DANE) Transport Layer Security (TLS) Protocol: TLSA", RFC 6698, DOI 10.17487/RFC6698, August 2012, <http://www.rfc-editor.org/info/rfc6698>.

[RFC7633] Hallam-Baker, P., "X.509v3 Transport Layer Security (TLS) Feature Extension", RFC 7633, DOI 10.17487/RFC7633, October 2015, <http://www.rfc-editor.org/info/rfc7633>.

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

## 12.2. Informative References

[RFC5011] StJohns, M., "Automated Updates of DNS Security (DNSSEC) Trust Anchors", STD 74, RFC 5011, DOI 10.17487/RFC5011, September 2007, <http://www.rfc-editor.org/info/rfc5011>.

[RFC5905] Mills, D., Martin, J., Ed., Burbank, J., and W. Kasch, "Network Time Protocol Version 4: Protocol and Algorithms Specification", RFC 5905, DOI 10.17487/RFC5905, June 2010, <http://www.rfc-editor.org/info/rfc5905>.

[RFC7120] Cotton, M., "Early IANA Allocation of Standards Track Code Points", BCP 100, RFC 7120, DOI 10.17487/RFC7120, January 2014, <http://www.rfc-editor.org/info/rfc7120>.

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

[RFC7435] Dukhovni, V., "Opportunistic Security: Some Protection Most of the Time", RFC 7435, DOI 10.17487/RFC7435, December 2014, <http://www.rfc-editor.org/info/rfc7435>.

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

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

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

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

## Appendix A. Updates from -01 and -02

- Editorial updates for style and consistency
- Updated discussion of UKS attack

## Appendix B. Updates from -01

- Added TLS 1.3 support
- Added section describing applicability to raw public keys
- Softened language about record order

## Appendix C. Updates from -00

- Edits based on comments from Rick van Rein
- Warning about not overloading X.509 wildcards on DNSSEC wildcards (from V. Dukhovny)
- Added MUST include negative proof on wildcards (from V. Dukhovny)
- Removed "TODO" on allowing the server to deliver only one signature per RRset
- Added additional minor edits suggested by Viktor Dukhovny

## Appendix D. Test vector

[data go here]

## Authors' Addresses

Melinda Shore
Fastly

EMail: mshore@fastly.com

Richard Barnes
Mozilla

EMail: rlb@ipv.sx

Shumon Huque
Salesforce

EMail: shumon.huque@gmail.com

Willem Toorop
NLNet Labs

EMail: willem@nlnetlabs.nl

Network Working Group
Internet-Draft
Obsoletes: 5077, 5246 (if approved)
Updates: 4492, 5705, 6066, 6961 (if approved)
Intended status: Standards Track
Expires: October 30, 2017

E. Rescorla
RTFM, Inc.

April 28, 2017

# The Transport Layer Security (TLS) Protocol Version 1.3
## draft-ietf-tls-tls13-20

## Abstract

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

## 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 October 30, 2017.

## Copyright Notice

Table of Contents

## 1. Introduction

DISCLAIMER: This is a WIP draft of TLS 1.3 and has not yet seen significant security analysis.

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/tls13-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 TLS is to provide a secure channel between two communicating peers. Specifically, the channel should provide the following properties:

- Authentication: The server side of the channel is always authenticated; the client side is optionally authenticated. Authentication can happen via asymmetric cryptography (e.g., RSA [RSA], ECDSA [ECDSA]) or a pre-shared key (PSK).

- Confidentiality: Data sent over the channel is only visible to the endpoints. TLS does not hide the length of the data it transmits, though endpoints are able to pad in order to obscure lengths.

- Integrity: Data sent over the channel cannot be modified by attackers.

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

TLS consists of two primary components:

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

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

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

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

This document supersedes and obsoletes previous versions of TLS including version 1.2 [RFC5246]. It also obsoletes the TLS ticket mechanism defined in [RFC5077] and replaces it with the mechanism defined in Section 2.2. Section 4.2.6 updates [RFC4492] by modifying the protocol attributes used to negotiate Elliptic Curves. Because TLS 1.3 changes the way keys are derived it updates [RFC5705] as described in Section 7.5 it also changes how OCSP messages are carried and therefore updates [RFC6066] and obsoletes [RFC6961] as described in section Section 4.4.2.1.

## 1.1. Conventions and Terminology

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

The following terms are used:

client: The endpoint initiating the TLS connection.

connection: A transport-layer connection between two endpoints.

endpoint: Either the client or server of the connection.

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

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

receiver: An endpoint that is receiving records.

sender: An endpoint that is transmitting records.

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

## 1.2. Change Log

RFC EDITOR PLEASE DELETE THIS SECTION.

(*) indicates changes to the wire protocol which may require implementations to update.

draft-20

- Add "post_handshake_auth" extension to negotiate post-handshake authentication (*).

- Shorten labels for HKDF-Expand-Label so that we can fit within one compression block (*).

- Define how RFC 7250 works (*).

- Re-enable post-handshake client authentication even when you do PSK. The previous prohibition was editorial error.

- Remove cert_type and user_mapping, which don't work on TLS 1.3 anyway.

- Added the no_application_protocol alert from [RFC7301] to the list of extensions.

- Added discussion of traffic analysis and side channel attacks.

draft-19

- Hash context_value input to Exporters (*)

- Add an additional Derive-Secret stage to Exporters (*).

- Hash ClientHello1 in the transcript when HRR is used. This reduces the state that needs to be carried in cookies. (*)

- Restructure CertificateRequest to have the selectors in extensions. This also allowed defining a "certificate_authorities" extension which can be used by the client instead of trusted_ca_keys (*).

- Tighten record framing requirements and require checking of them (*).

- Consolidate "ticket_early_data_info" and "early_data" into a single extension (*).

- Change end_of_early_data to be a handshake message (*).

- Add pre-extract Derive-Secret stages to key schedule (*).

- Remove spurious requirement to implement "pre_shared_key".

- Clarify location of "early_data" from server (it goes in EE, as indicated by the table in S 10).

- Require peer public key validation

- Add state machine diagram.

draft-18

- Remove unnecessary resumption_psk which is the only thing expanded from the resumption master secret. (*).

- Fix signature_algorithms entry in extensions table.

- Restate rule from RFC 6066 that you can’t resume unless SNI is the same.

draft-17

- Remove 0-RTT Finished and resumption_context, and replace with a psk_binder field in the PSK itself (*)

- Restructure PSK key exchange negotiation modes (*)

- Add max_early_data_size field to TicketEarlyDataInfo (*)

- Add a 0-RTT exporter and change the transcript for the regular exporter (*)

- Merge TicketExtensions and Extensions registry. Changes ticket_early_data_info code point (*)

- Replace Client.key_shares in response to HRR (*)

- Remove redundant labels for traffic key derivation (*)

- Harmonize requirements about cipher suite matching: for resumption you need to match KDF but for 0-RTT you need whole cipher suite. This allows PSKs to actually negotiate cipher suites. (*)

- Move SCT and OCSP into Certificate.extensions (*)

- Explicitly allow non-offered extensions in NewSessionTicket

- Explicitly allow predicting client Finished for NST

- Clarify conditions for allowing 0-RTT with PSK

draft-16

- Revise version negotiation (*)

- Change RSASSA-PSS and EdDSA SignatureScheme codepoints for better backwards compatibility (*)

- Move HelloRetryRequest.selected_group to an extension (*)

- Clarify the behavior of no exporter context and make it the same as an empty context.(*)

- New KeyUpdate format that allows for requesting/not-requesting an answer. This also means changes to the key schedule to support independent updates (*)

- New certificate_required alert (*)

- Forbid CertificateRequest with 0-RTT and PSK.

- Relax requirement to check SNI for 0-RTT.

draft-15

- New negotiation syntax as discussed in Berlin (*)

- Require CertificateRequest.context to be empty during handshake (*)

- Forbid empty tickets (*)

- Forbid application data messages in between post-handshake messages from the same flight (*)

- Clean up alert guidance (*)

- Clearer guidance on what is needed for TLS 1.2.

- Guidance on 0-RTT time windows.

- Rename a bunch of fields.

- Remove old PRNG text.

- Explicitly require checking that handshake records not span key changes.

draft-14

- Allow cookies to be longer (*)

- Remove the "context" from EarlyDataIndication as it was undefined and nobody used it (*)

- Remove 0-RTT EncryptedExtensions and replace the ticket_age extension with an obfuscated version. Also necessitates a change to NewSessionTicket (*).

- Move the downgrade sentinel to the end of ServerHello.Random to accommodate tlsdate (*).

- Define ecdsa_sha1 (*).

- Allow resumption even after fatal alerts. This matches current practice.

- Remove non-closure warning alerts. Require treating unknown alerts as fatal.

- Make the rules for accepting 0-RTT less restrictive.

- Clarify 0-RTT backward-compatibility rules.

- Clarify how 0-RTT and PSK identities interact.

- Add a section describing the data limits for each cipher.

- Major editorial restructuring.

- Replace the Security Analysis section with a WIP draft.

draft-13

- Allow server to send SupportedGroups.

- Remove 0-RTT client authentication

- Remove (EC)DHE 0-RTT.

- Flesh out 0-RTT PSK mode and shrink EarlyDataIndication

- Turn PSK-resumption response into an index to save room

- Move CertificateStatus to an extension

- Extra fields in NewSessionTicket.

- Restructure key schedule and add a resumption_context value.

- Require DH public keys and secrets to be zero-padded to the size of the group.

- Remove the redundant length fields in KeyShareEntry.

- Define a cookie field for HRR.

draft-12

- Provide a list of the PSK cipher suites.

- Remove the ability for the ServerHello to have no extensions (this aligns the syntax with the text).

- Clarify that the server can send application data after its first flight (0.5 RTT data)

- Revise signature algorithm negotiation to group hash, signature algorithm, and curve together. This is backwards compatible.

- Make ticket lifetime mandatory and limit it to a week.

- Make the purpose strings lower-case. This matches how people are implementing for interop.

- Define exporters.

- Editorial cleanup

draft-11

- Port the CFRG curves & signatures work from RFC4492bis.

- Remove sequence number and version from additional_data, which is now empty.

- Reorder values in HkdfLabel.

- Add support for version anti-downgrade mechanism.

- Update IANA considerations section and relax some of the policies.

- Unify authentication modes. Add post-handshake client authentication.

- Remove early_handshake content type. Terminate 0-RTT data with an alert.

- Reset sequence number upon key change (as proposed by Fournet et al.)

draft-10

- Remove ClientCertificateTypes field from CertificateRequest and add extensions.

- Merge client and server key shares into a single extension.

draft-09

- Change to RSA-PSS signatures for handshake messages.

- Remove support for DSA.

- Update key schedule per suggestions by Hugo, Hoeteck, and Bjoern Tackmann.

- Add support for per-record padding.

- Switch to encrypted record ContentType.

- Change HKDF labeling to include protocol version and value lengths.

- Shift the final decision to abort a handshake due to incompatible certificates to the client rather than having servers abort early.

- Deprecate SHA-1 with signatures.

- Add MTI algorithms.

draft-08

- Remove support for weak and lesser used named curves.

- Remove support for MD5 and SHA-224 hashes with signatures.

- Update lists of available AEAD cipher suites and error alerts.

- Reduce maximum permitted record expansion for AEAD from 2048 to 256 octets.

- Require digital signatures even when a previous configuration is used.

- Merge EarlyDataIndication and KnownConfiguration.

- Change code point for server_configuration to avoid collision with server_hello_done.

- Relax certificate_list ordering requirement to match current practice.

draft-07

- Integration of semi-ephemeral DH proposal.

- Add initial 0-RTT support.

- Remove resumption and replace with PSK + tickets.

- Move ClientKeyShare into an extension.

- Move to HKDF.

draft-06

- Prohibit RC4 negotiation for backwards compatibility.

- Freeze & deprecate record layer version field.

- Update format of signatures with context.

- Remove explicit IV.

draft-05

- Prohibit SSL negotiation for backwards compatibility.

- Fix which MS is used for exporters.

draft-04

- Modify key computations to include session hash.

- Remove ChangeCipherSpec.

- Renumber the new handshake messages to be somewhat more consistent with existing convention and to remove a duplicate registration.

- Remove renegotiation.

- Remove point format negotiation.

draft-03

- Remove GMT time.

- Merge in support for ECC from RFC 4492 but without explicit curves.

- Remove the unnecessary length field from the AD input to AEAD ciphers.

- Rename {Client,Server}KeyExchange to {Client,Server}KeyShare.

- Add an explicit HelloRetryRequest to reject the client's.

draft-02

- Increment version number.

- Rework handshake to provide 1-RTT mode.

- Remove custom DHE groups.

- Remove support for compression.

- Remove support for static RSA and DH key exchange.

- Remove support for non-AEAD ciphers.

## 1.3. Major Differences from TLS 1.2

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

- The list of supported symmetric algorithms has been pruned of all algorithms that are considered legacy. Those that remain all use Authenticated Encryption with Associated Data (AEAD) algorithms. The ciphersuite concept has been changed to separate the authentication and key exchange mechanisms from the record protection algorithm (including secret key length) and a hash to be used with the key derivation function and HMAC.

- A Zero-RTT mode was added, saving a round-trip at connection setup for some application data, at the cost of certain security properties.

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

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

- The handshake state machine has been significantly restructured to be more consistent and to remove superfluous messages such as ChangeCipherSpec.

- ECC is now in the base spec and includes new signature algorithms, such as ed25519 and ed448. TLS 1.3 removed point format negotiation in favor of a single point format for each curve.

- Other cryptographic improvements including the removal of compression and custom DHE groups, changing the RSA padding to use PSS, and the removal of DSA.

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

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

### 1.4. Updates Affecting TLS 1.2

This document defines several changes that optionally affect implementations of TLS 1.2:

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

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

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

An implementation of TLS 1.3 that also supports TLS 1.2 might need to include changes to support these changes even when TLS 1.3 is not in use. See the referenced sections for more details.

## 2. Protocol Overview

The cryptographic parameters of the connection state are produced by the TLS handshake protocol, which a TLS client and server use when first communicating to agree on a protocol version, select cryptographic algorithms, optionally authenticate each other, and establish shared secret keying material. Once the handshake is complete, the peers use the established keys to protect application layer traffic.

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

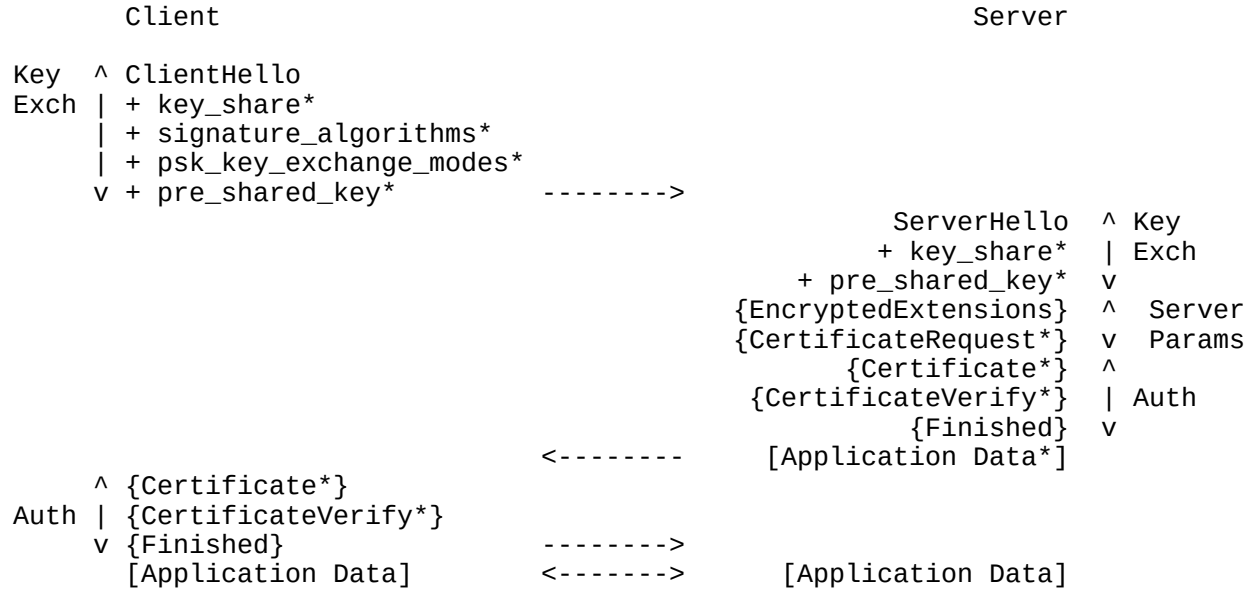
TLS supports three basic key exchange modes:

- (EC)DHE (Diffie-Hellman, both the finite field and elliptic curve varieties),

- PSK-only, and

- PSK with (EC)DHE

Figure 1 below shows the basic full TLS handshake:




+ Indicates noteworthy extensions sent in the previously noted message.

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

{} Indicates messages protected using keys derived from a [sender]_handshake_traffic_secret.

[] Indicates messages protected using keys derived from [sender]_application_traffic_secret_N

Figure 1: Message flow for full TLS Handshake

The handshake can be thought of as having three phases (indicated in the diagram above):

- Key Exchange: Establish shared keying material and select the cryptographic parameters. Everything after this phase is encrypted.

- Server Parameters: Establish other handshake parameters (whether the client is authenticated, application layer protocol support, etc.).

- Authentication: Authenticate the server (and optionally the client) and provide key confirmation and handshake integrity.

In the Key Exchange phase, the client sends the ClientHello (Section 4.1.2) message, which contains a random nonce (ClientHello.random); its offered protocol versions; a list of symmetric cipher/HKDF hash pairs; some set of Diffie-Hellman key shares (in the "key_share" extension Section 4.2.7), a set of pre-shared key labels (in the "pre_shared_key" extension Section 4.2.10) or both; and potentially some other extensions.

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

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

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

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

Finally, the client and server exchange Authentication messages. TLS uses the same set of messages every time that authentication is needed. Specifically:

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

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

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

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

At this point, the handshake is complete, and the client and server may exchange application-layer data. Application data MUST NOT be sent prior to sending the Finished message. Note that while the server may send application data prior to receiving the client's Authentication messages, any data sent at that point is, of course, being sent to an unauthenticated peer.

## 2.1. Incorrect DHE Share

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


Figure 2: Message flow for a full handshake with mismatched parameters

Note: The handshake transcript includes the initial ClientHello/ HelloRetryRequest exchange; it is not reset with the new ClientHello.

TLS also allows several optimized variants of the basic handshake, as described in the following sections.

## 2.2. Resumption and Pre-Shared Key (PSK)

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

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

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


Figure 3: Message flow for resumption and PSK

As the server is authenticating via a PSK, it does not send a Certificate or a CertificateVerify message. When a client offers resumption via PSK, it SHOULD also supply a "key_share" extension to the server to allow the server to decline resumption and fall back to a full handshake, if needed. The server responds with a

"pre_shared_key" extension to negotiate use of PSK key establishment and can (as shown here) respond with a "key_share" extension to do (EC)DHE key establishment, thus providing forward secrecy.

When PSKs are provisioned out of band, the PSK identity and the KDF to be used with the PSK MUST also be provisioned. Note: When using an out-of-band provisioned pre-shared secret, a critical consideration is using sufficient entropy during the key generation, as discussed in [RFC4086]. Deriving a shared secret from a password or other low-entropy sources is not secure. A low-entropy secret, or password, is subject to dictionary attacks based on the PSK binder. The specified PSK authentication is not a strong password-based authenticated key exchange even when used with Diffie-Hellman key establishment.

## 2.3. Zero-RTT Data

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

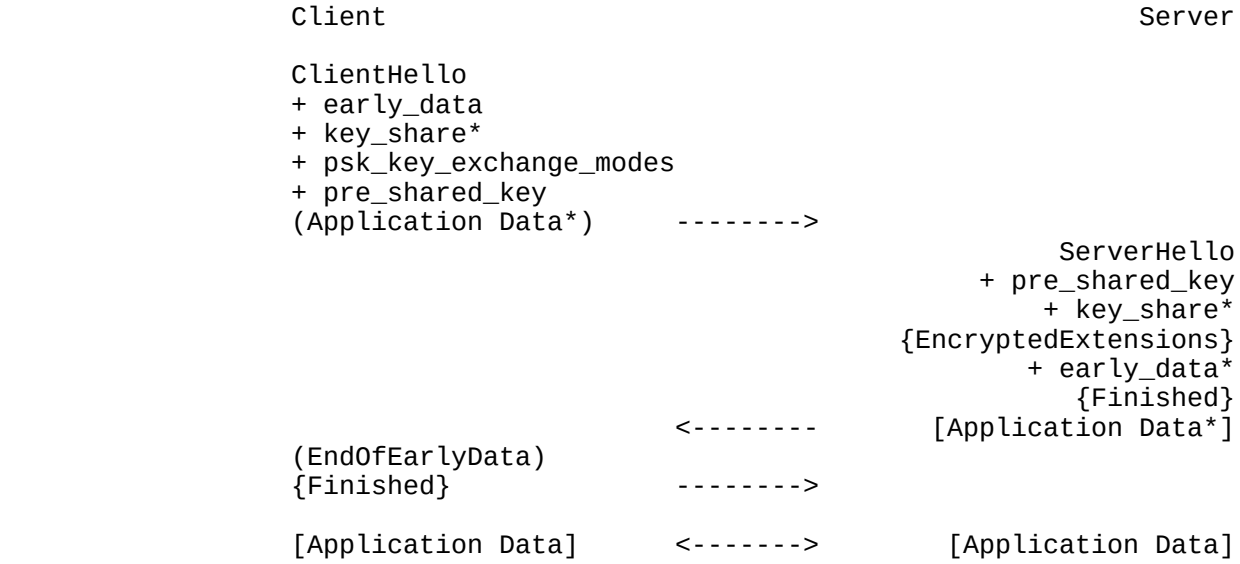
When clients use a PSK obtained externally to send early data, then the following additional information MUST be provisioned to both parties:

- The cipher suite for use with this PSK

- The Application-Layer Protocol Negotiation (ALPN) protocol, if any is to be used

- The Server Name Indication (SNI), if any is to be used

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




+ 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 client_early_traffic_secret.

{} Indicates messages protected using keys derived from a [sender]_handshake_traffic_secret.

[] Indicates messages protected using keys derived from [sender]_application_traffic_secret_N

Figure 4: Message flow for a zero round trip handshake

IMPORTANT NOTE: The security properties for 0-RTT data are weaker than those for other kinds of TLS data. Specifically:

1. This data is not forward secret, as it is encrypted solely under keys derived using the offered PSK.

2. There are no guarantees of non-replay between connections. Unless the server takes special measures outside those provided by TLS, the server has no guarantee that the same 0-RTT data was not transmitted on multiple 0-RTT connections (see Section 4.2.10.4 for more details). This is especially relevant

if the data is authenticated either with TLS client authentication or inside the application layer protocol. However, 0-RTT data cannot be duplicated within a connection (i.e., the server will not process the same data twice for the same connection) and an attacker will not be able to make 0-RTT data appear to be 1-RTT data (because it is protected with different keys.)

Protocols MUST NOT use 0-RTT data without a profile that defines its use. That profile needs to identify which messages or interactions are safe to use with 0-RTT. In addition, to avoid accidental misuse, implementations SHOULD NOT enable 0-RTT unless specifically requested. Implementations SHOULD provide special functions for 0-RTT data to ensure that an application is always aware that it is sending or receiving data that might be replayed.

The same warnings apply to any use of the early_exporter_master_secret.

The remainder of this document provides a detailed description of TLS.

# 3. Presentation Language

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

## 3.1. Basic Block Size

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

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

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

## 3.2. Miscellaneous

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

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

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

### 3.3. Vectors

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

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

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

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

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

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

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

In the following example, mandatory is a vector that must contain between 300 and 400 bytes of type opaque. It can never be empty. The actual length field consumes two bytes, a uint16, which is sufficient to represent the value 400 (see Section 3.4). Similarly, longer can represent up to 800 bytes of data, or 400 uint16 elements, and it may be empty. Its encoding will include a two-byte actual length field prepended to the vector. The length of an encoded vector must be an exact multiple of the length of a single element (e.g., a 17-byte vector of uint16 would be illegal).

```
opaque mandatory<300..400>;
      /* length field is 2 bytes, cannot be empty */
uint16 longer<0..800>;
      /* zero to 400 16-bit unsigned integers */
```

### 3.4. Numbers

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

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

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

### 3.5. Enumerateds

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

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

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

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

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

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

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

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

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

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

The names assigned to enumerateds do not need to be unique. The numerical value can describe a range over which the same name applies. The value includes the minimum and maximum inclusive values in that range, separated by two period characters. This is principally useful for reserving regions of the space.

```
enum { sad(0), meh(1..254), happy(255) } Mood;
```

### 3.6. Constructed Types

Structure types may be constructed from primitive types for convenience. Each specification declares a new, unique type. The syntax for definition is much like that of C.

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

The fields within a structure may be qualified using the type's name, with a syntax much like that available for enumerateds. For example, T.f2 refers to the second field of the previous declaration. Structure definitions may be embedded. Anonymous structs may also be defined inside other structures.

### 3.7. Constants

Fields and variables may be assigned a fixed value using "=", as in:

```
struct {
    T1 f1 = 8;  /* T.f1 must always be 8 */
    T2 f2;
} T;
```

### 3.8. Variants

Defined structures may have variants based on some knowledge that is available within the environment. The selector must be an enumerated type that defines the possible variants the structure defines. The body of the variant structure may be given a label for reference. The mechanism by which the variant is selected at runtime is not prescribed by the presentation language.

```
struct {
    T1 f1;
    T2 f2;
    ....
    Tn fn;
    select (E) {
        case e1: Te1;
        case e2: Te2;
        ....
        case en: Ten;
    } [[fv]];
} [[Tv]];
```

For example:

```
enum { apple(0), orange(1) } VariantTag;

struct {
    uint16 number;
    opaque string<0..10>; /* variable length */
} V1;

struct {
    uint32 number;
    opaque string[10];    /* fixed length */
} V2;

struct {
    VariantTag type;
    select (VariantRecord.type) {
        case apple:  V1;
        case orange: V2;
    };
} VariantRecord;
```

# 4. Handshake Protocol

The handshake protocol is used to negotiate the secure attributes of a connection. Handshake messages are supplied to the TLS record layer, where they are encapsulated within one or more TLSPlaintext or TLSCiphertext structures, which are processed and transmitted as specified by the current active connection state.

```
enum {
    client_hello(1),
    server_hello(2),
    new_session_ticket(4),
    end_of_early_data(5),
    hello_retry_request(6),
    encrypted_extensions(8),
    certificate(11),
    certificate_request(13),
    certificate_verify(15),
    finished(20),
    key_update(24),
    message_hash(254),
    (255)
} HandshakeType;

struct {
    HandshakeType msg_type;    /* handshake type */
    uint24 length;             /* bytes in message */
    select (Handshake.msg_type) {
        case client_hello:          ClientHello;
        case server_hello:          ServerHello;
        case end_of_early_data:     EndOfEarlyData;
        case hello_retry_request:   HelloRetryRequest;
        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;
```

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

New handshake message types are assigned by IANA as described in Section 10.

## 4.1. Key Exchange Messages

The key exchange messages are used to exchange security capabilities between the client and server and to establish the traffic keys used to protect the handshake and data.

### 4.1.1. Cryptographic Negotiation

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

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

- A "supported_groups" (Section 4.2.6) extension which indicates the (EC)DHE groups which the client supports and a "key_share" (Section 4.2.7) extension which contains (EC)DHE shares for some or all of these groups.

- A "signature_algorithms" (Section 4.2.3) extension which indicates the signature algorithms which the client can accept.

- A "pre_shared_key" (Section 4.2.10) extension which contains a list of symmetric key identities known to the client and a "psk_key_exchange_modes" (Section 4.2.8) extension which indicates the key exchange modes that may be used with PSKs.

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

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

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

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

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

- If PSK is not being used, then (EC)DHE and certificate-based authentication are always used.

- When (EC)DHE is in use, the server will also provide a "key_share" extension.

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

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

#### 4.1.2. Client Hello

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

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

- Removing the "early_data" extension (Section 4.2.9) if one was present. Early data is not permitted after HelloRetryRequest.

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

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

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

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

Structure of this message:

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

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

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

legacy_version In previous versions of TLS, this field was used for version negotiation and represented the highest version number supported by the client. Experience has shown that many servers do not properly implement version negotiation, leading to "version intolerance" in which the server rejects an otherwise acceptable ClientHello with a version number higher than it supports. In TLS 1.3, the client indicates its version preferences in the "supported_versions" extension (Section 4.2.1) and the legacy_version field MUST be set to 0x0303, which is the version number for TLS 1.2. (See Appendix D for details about backward compatibility.)

random 32 bytes generated by a secure random number generator. See Appendix C for additional information.

legacy_session_id Versions of TLS before TLS 1.3 supported a "session resumption" feature which has been merged with Pre-Shared Keys in this version (see Section 2.2). This field MUST be ignored by a server negotiating TLS 1.3 and MUST be set as a zero length vector (i.e., a single zero byte length field) by clients that do not have a cached session ID set by a pre-TLS 1.3 server.

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

legacy_compression_methods Versions of TLS before 1.3 supported compression with the list of supported compression methods being sent in this field. For every TLS 1.3 ClientHello, this vector MUST contain exactly one byte set to zero, which corresponds to the "null" compression method in prior versions of TLS. If a TLS 1.3 ClientHello is received with any other value in this field, the server MUST abort the handshake with an "illegal_parameter" alert. Note that TLS 1.3 servers might receive TLS 1.2 or prior ClientHellos which contain other compression methods and MUST follow the procedures for the appropriate prior version of TLS. TLS 1.3 ClientHellos are identified as having a legacy_version of 0x0303 and a supported_versions extension present with 0x0304 as the highest version indicated therein.

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

All versions of TLS allow extensions to optionally follow the compression_methods field as an extensions field. TLS 1.3 ClientHello messages always contain extensions (minimally, "supported_versions", or they will be interpreted as TLS 1.2 ClientHello messages), however TLS 1.3 servers might receive ClientHello messages without an extensions field from prior versions of TLS. The presence of extensions can be detected by determining whether there are bytes following the compression_methods field at the end of the ClientHello. Note that this method of detecting optional data differs from the normal TLS method of having a variable-length field, but it is used for compatibility with TLS before extensions were defined. TLS 1.3 servers will need to perform this check first and only attempt to negotiate TLS 1.3 if a "supported_version" extension is present. If negotiating a version of TLS prior to 1.3, a server MUST check that the message either contains no data after legacy_compression_methods or that it contains

a valid extensions block with no data following. If not, then it MUST abort the handshake with a "decode_error" alert.

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

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

#### 4.1.3. Server Hello

The server will send this message in response to a ClientHello message if it is able to find an acceptable set of parameters and the ClientHello contains sufficient information to proceed with the handshake.

Structure of this message:

```
struct {
    ProtocolVersion version;
    Random random;
    CipherSuite cipher_suite;
    Extension extensions<6..2^16-1>;
} ServerHello;
```

version This field contains the version of TLS negotiated for this connection. Servers MUST select a version from the list in ClientHello's supported_versions extension, or otherwise negotiate TLS 1.2 or previous. A client that receives a version that was not offered MUST abort the handshake. For this version of the specification, the version is 0x0304. (See Appendix D for details about backward compatibility.)

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

cipher_suite The single cipher suite selected by the server from the list in ClientHello.cipher_suites. A client which receives a cipher suite that was not offered MUST abort the handshake.

extensions A list of extensions. The ServerHello MUST only include extensions which are required to establish the cryptographic

context. Currently the only such extensions are "key_share" and "pre_shared_key". All current TLS 1.3 ServerHello messages will contain one of these two extensions, or both when using a PSK with (EC)DHE key establishment.

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

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

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

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

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

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

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

A client that receives a TLS 1.3 ServerHello during renegotiation MUST abort the handshake with a "protocol_version" alert. Note that renegotiation is only possible when a version of TLS prior to 1.3 has been negotiated.

RFC EDITOR: PLEASE REMOVE THE FOLLOWING PARAGRAPH Implementations of draft versions (see Section 4.2.1.1) of this specification SHOULD NOT implement this mechanism on either client and server. A pre-RFC client connecting to RFC servers, or vice versa, will appear to

downgrade to TLS 1.2. With the mechanism enabled, this will cause an interoperability failure.

#### 4.1.4. Hello Retry Request

The server will send this message in response to a ClientHello message if it is able to find an acceptable set of parameters but the ClientHello does not contain sufficient information to proceed with the handshake.

Structure of this message:

```
struct {
    ProtocolVersion server_version;
    CipherSuite cipher_suite;
    Extension extensions<2..2^16-1>;
} HelloRetryRequest;
```

The version, cipher_suite, and extensions fields have the same meanings as their corresponding values in the ServerHello. The server SHOULD send only the extensions necessary for the client to generate a correct ClientHello pair. As with ServerHello, a HelloRetryRequest MUST NOT contain any extensions that were not first offered by the client in its ClientHello, with the exception of optionally the "cookie" (see Section 4.2.2) extension.

Upon receipt of a HelloRetryRequest, the client MUST verify that the extensions block is not empty and otherwise MUST abort the handshake with a "decode_error" alert. Clients MUST abort the handshake with an "illegal_parameter" alert if the HelloRetryRequest would not result in any change in the ClientHello. If a client receives a second HelloRetryRequest in the same connection (i.e., where the ClientHello was itself in response to a HelloRetryRequest), it MUST abort the handshake with an "unexpected_message" alert.

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

- cookie (see Section 4.2.2)

- key_share (see Section 4.2.7)

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

offered MUST abort the handshake. Servers MUST ensure that they negotiate the same cipher suite when receiving a conformant updated ClientHello (if the server selects the cipher suite as the first step in the negotiation, then this will happen automatically). Upon receiving the ServerHello, clients MUST check that the cipher suite supplied in the ServerHello is the same as that in the HelloRetryRequest and otherwise abort the handshake with an "illegal_parameter" alert.

## 4.2. Extensions

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

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

enum {
    server_name(0),                             /* RFC 6066 */
    max_fragment_length(1),                     /* RFC 6066 */
    status_request(5),                          /* RFC 6066 */
    supported_groups(10),                       /* RFC 4492, 7919 */
    signature_algorithms(13),                   /* RFC 5246 */
    use_srtp(14),                               /* RFC 5764 */
    heartbeat(15),                              /* RFC 6520 */
    application_layer_protocol_negotiation(16), /* RFC 7301 */
    signed_certificate_timestamp(18),           /* RFC 6962 */
    client_certificate_type(19),                /* RFC 7250 */
    server_certificate_type(20)                 /* RFC 7250 */
    padding(21),                                /* RFC 7685 */
    key_share(40),                              /* [[this document]] */
    pre_shared_key(41),                         /* [[this document]] */
    early_data(42),                             /* [[this document]] */
    supported_versions(43),                     /* [[this document]] */
    cookie(44),                                 /* [[this document]] */
    psk_key_exchange_modes(45),                 /* [[this document]] */
    certificate_authorities(47),                /* [[this document]] */
    oid_filters(48),                            /* [[this document]] */
    post_handshake_auth(49),                    /* [[this document]] */
    (65535)
} ExtensionType;
```

Here:

- "extension_type" identifies the particular extension type.

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

The list of extension types is maintained by IANA as described in Section 10.

Extensions are generally structured in a request/response fashion, though some extensions are just indications with no corresponding response. The client sends its extension requests in the ClientHello message and the server sends its extension responses in the ServerHello, EncryptedExtensions and HelloRetryRequest messages. The server sends extension requests in the CertificateRequest message which a client MAY respond to with a Certificate message. The server MAY also send unsolicited extensions in the NewSessionTicket, though the client does not respond directly to these.

Implementations MUST NOT send extension responses if the remote endpoint did not send the corresponding extension requests, with the exception of the "cookie" extension in HelloRetryRequest. Upon receiving such an extension, an endpoint MUST abort the handshake with an "unsupported_extension" alert.

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

| Extension | TLS 1.3 |
|---|---|
| server_name [RFC6066] | CH, EE |
| max_fragment_length [RFC6066] | CH, EE |
| status_request [RFC6066] | CH, CR, CT |
| supported_groups [RFC7919] | CH, EE |
| signature_algorithms [RFC5246] | CH, CR |
| use_srtp [RFC5764] | CH, EE |
| heartbeat [RFC6520] | CH, EE |
| application_layer_protocol_negotiation [RFC7301] | CH, EE |
| signed_certificate_timestamp [RFC6962] | CH, CR, CT |
| client_certificate_type [RFC7250] | CH, EE |
| server_certificate_type [RFC7250] | CH, CT |
| padding [RFC7685] | CH |
| key_share [[this document]] | CH, SH, HRR |
| pre_shared_key [[this document]] | CH, SH |
| psk_key_exchange_modes [[this document]] | CH |
| early_data [[this document]] | CH, EE, NST |
| cookie [[this document]] | CH, HRR |
| supported_versions [[this document]] | CH |
| certificate_authorities [[this document]] | CH, CR |
| oid_filters [[this document]] | CR |
| post_handshake_auth [[this document]] | CH |

When multiple extensions of different types are present, the extensions MAY appear in any order, with the exception of

"pre_shared_key" Section 4.2.10 which MUST be the last extension in the ClientHello. There MUST NOT be more than one extension of the same type in a given extension block.

In TLS 1.3, unlike TLS 1.2, extensions are renegotiated with each handshake even when in resumption-PSK mode. However, 0-RTT parameters are those negotiated in the previous handshake; mismatches may require rejecting 0-RTT (see Section 4.2.9).

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

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

- Extensions should, as far as possible, be designed to prevent any attack that forces use (or non-use) of a particular feature by manipulation of handshake messages. This principle should be followed regardless of whether the feature is believed to cause a security problem. Often the fact that the extension fields are included in the inputs to the Finished message hashes will be sufficient, but extreme care is needed when the extension changes the meaning of messages sent in the handshake phase. Designers and implementors should be aware of the fact that until the handshake has been authenticated, active attackers can modify messages and insert, remove, or replace extensions.

### 4.2.1. Supported Versions

```
struct {
    ProtocolVersion versions<2..254>;
} SupportedVersions;
```

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

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

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

The server MUST NOT send the "supported_versions" extension. The server's selected version is contained in the ServerHello.version field as in previous versions of TLS.

#### 4.2.1.1. Draft Version Indicator

RFC EDITOR: PLEASE REMOVE THIS SECTION

While the eventual version indicator for the RFC version of TLS 1.3 will be 0x0304, implementations of draft versions of this specification SHOULD instead advertise 0x7f00 | draft_version in ServerHello.version, and HelloRetryRequest.server_version. For instance, draft-17 would be encoded as the 0x7f11. This allows pre-RFC implementations to safely negotiate with each other, even if they would otherwise be incompatible.

### 4.2.2. Cookie

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

Cookies serve two primary purposes:

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

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

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

#### 4.2.3. Signature Algorithms

The client uses the "signature_algorithms" extension to indicate to the server which signature algorithms may be used in digital signatures. Clients which desire the server to authenticate itself via a certificate MUST send this extension. If a server is authenticating via a certificate and the client has not sent a "signature_algorithms" extension, then the server MUST abort the handshake with a "missing_extension" alert (see Section 8.2).

The "extension_data" field of this extension in a ClientHello contains a SignatureSchemeList value:

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

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

          /* RSASSA-PSS algorithms */
          rsa_pss_sha256(0x0804),
          rsa_pss_sha384(0x0805),
          rsa_pss_sha512(0x0806),

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

          /* Legacy algorithms */
          rsa_pkcs1_sha1(0x0201),
          ecdsa_sha1(0x0203),

          /* Reserved Code Points */
          private_use(0xFE00..0xFFFF),
          (0xFFFF)
      } SignatureScheme;

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

Note: This enum is named "SignatureScheme" because there is already a "SignatureAlgorithm" type in TLS 1.2, which this replaces. We use the term "signature algorithm" throughout the text.

Each SignatureScheme value lists a single signature algorithm that the client is willing to verify. The values are indicated in descending order of preference. Note that a signature algorithm takes as input an arbitrary-length message, rather than a digest. Algorithms which traditionally act on a digest should be defined in TLS to first hash the input with a specified hash algorithm and then proceed as usual. The code point groups listed above have the following meanings:

RSASSA-PKCS1-v1_5 algorithms Indicates a signature algorithm using RSASSA-PKCS1-v1_5 [RFC8017] with the corresponding hash algorithm

as defined in [SHS]. These values refer solely to signatures which appear in certificates (see Section 4.4.2.2) and are not defined for use in signed TLS handshake messages.

ECDSA algorithms Indicates a signature algorithm using ECDSA [ECDSA], the corresponding curve as defined in ANSI X9.62 [X962] and FIPS 186-4 [DSS], and the corresponding hash algorithm as defined in [SHS]. The signature is represented as a DER-encoded [X690] ECDSA-Sig-Value structure.

RSASSA-PSS algorithms Indicates a signature algorithm using RSASSA-PSS [RFC8017] with mask generation function 1. The digest used in the mask generation function and the digest being signed are both the corresponding hash algorithm as defined in [SHS]. When used in signed TLS handshake messages, the length of the salt MUST be equal to the length of the digest output. This codepoint is new in this document and is also defined for use with TLS 1.2.

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

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

The signatures on certificates that are self-signed or certificates that are trust anchors are not validated since they begin a certification path (see [RFC5280], Section 3.2). A certificate that begins a certification path MAY use a signature algorithm that is not advertised as being supported in the "signature_algorithms" extension.

Note that TLS 1.2 defines this extension differently. TLS 1.3 implementations willing to negotiate TLS 1.2 MUST behave in accordance with the requirements of [RFC5246] when negotiating that version. In particular:

- TLS 1.2 ClientHellos MAY omit this extension.

- In TLS 1.2, the extension contained hash/signature pairs. The pairs are encoded in two octets, so SignatureScheme values have been allocated to align with TLS 1.2's encoding. Some legacy pairs are left unallocated. These algorithms are deprecated as of TLS 1.3. They MUST NOT be offered or negotiated by any implementation. In particular, MD5 [SLOTH], SHA-224, and DSA MUST NOT be used.

- ECDSA signature schemes align with TLS 1.2's ECDSA hash/signature pairs. However, the old semantics did not constrain the signing curve. If TLS 1.2 is negotiated, implementations MUST be prepared to accept a signature that uses any curve that they advertised in the "supported_groups" extension.

- Implementations that advertise support for RSASSA-PSS (which is mandatory in TLS 1.3), MUST be prepared to accept a signature using that scheme even when TLS 1.2 is negotiated. In TLS 1.2, RSASSA-PSS is used with RSA cipher suites.

#### 4.2.4. Certificate Authorities

The "certificate_authorities" extension is used to indicate the certificate authorities which an endpoint supports and which SHOULD be used by the receiving endpoint to guide certificate selection.

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

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

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

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

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

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

#### 4.2.5. Post-Handshake Client Authentication

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

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

#### 4.2.6. Negotiated Groups

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

Note: In versions of TLS prior to TLS 1.3, this extension was named "elliptic_curves" and only contained elliptic curve groups. See [RFC4492] and [RFC7919]. This extension was also used to negotiate ECDSA curves. Signature algorithms are now negotiated independently (see Section 4.2.3).

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

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

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

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

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

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

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

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

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

#### 4.2.7. Key Share

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

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

```
struct {
    NamedGroup group;
    opaque key_exchange<1..2^16-1>;
} KeyShareEntry;
```

group The named group for the key being exchanged. Finite Field Diffie-Hellman [DH] parameters are described in Section 4.2.7.1; Elliptic Curve Diffie-Hellman parameters are described in Section 4.2.7.2.

key_exchange Key exchange information. The contents of this field are determined by the specified group and its corresponding definition.

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

```
struct {
    select (Handshake.msg_type) {
        case client_hello:
            KeyShareEntry client_shares<0..2^16-1>;

        case hello_retry_request:
            NamedGroup selected_group;

        case server_hello:
            KeyShareEntry server_share;
    };
} KeyShare;
```

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

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

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

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

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

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

#### 4.2.7.1. Diffie-Hellman Parameters

Diffie-Hellman [DH] parameters for both clients and servers are encoded in the opaque key_exchange field of a KeyShareEntry in a KeyShare structure. The opaque value contains the Diffie-Hellman public value (Y = g^X mod p) for the specified group (see [RFC7919] for group definitions) encoded as a big-endian integer and padded to the left with zeros to the size of p in bytes.

Note: For a given Diffie-Hellman group, the padding results in all public keys having the same length.

Peers MUST validate each other's public key Y by ensuring that 1 < Y < p-1. This check ensures that the remote peer is properly behaved and isn't forcing the local system into a small subgroup.

#### 4.2.7.2. ECDHE Parameters

ECDHE parameters for both clients and servers are encoded in the the opaque key_exchange field of a KeyShareEntry in a KeyShare structure.

For secp256r1, secp384r1 and secp521r1, the contents are the serialized value of the following struct:

```
struct {
    uint8               legacy_form = 4;
    opaque              X[coordinate_length];
    opaque              Y[coordinate_length];
 } UncompressedPointRepresentation;
```

X and Y respectively are the binary representations of the X and Y values in network byte order. There are no internal length markers, so each number representation occupies as many octets as implied by the curve parameters. For P-256 this means that each of X and Y use 32 octets, padded on the left by zeros if necessary. For P-384 they take 48 octets each, and for P-521 they take 66 octets each.

For the curves secp256r1, secp384r1 and secp521r1, peers MUST validate each other's public value Y by ensuring that the point is a valid point on the elliptic curve. The appropriate validation procedures are defined in Section 4.3.7 of [X962] and alternatively in Section 5.6.2.6 of [KEYAGREEMENT]. This process consists of three steps: (1) verify that Y is not the point at infinity (O), (2) verify that for Y = (x, y) both integers are in the correct interval, (3) ensure that (x, y) is a correct solution to the elliptic curve equation. For these curves, implementers do not need to verify membership in the correct subgroup.

For X25519 and X448, the contents of the public value are the byte string inputs and outputs of the corresponding functions defined in [RFC7748], 32 bytes for X25519 and 56 bytes for X448.

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

#### 4.2.8. Pre-Shared Key Exchange Modes

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

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

The server MUST NOT send a "psk_key_exchange_modes" extension.

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

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

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

psk_dhe_ke PSK with (EC)DHE key establishment. In this mode, the client and servers MUST supply "key_share" values as described in Section 4.2.7.

#### 4.2.9. Early Data Indication

When a PSK is used, the client can send application data in its first flight of messages. If the client opts to do so, it MUST supply an "early_data" extension as well as the "pre_shared_key" extension.

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

```
struct {} Empty;

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

See Section 4.6.1 for the use of the max_early_data_size field.

The parameters for the 0-RTT data (symmetric cipher suite, ALPN protocol, etc.) are the same as those which were negotiated in the connection which established the PSK. The PSK used to encrypt the early data MUST be the first PSK listed in the client's "pre_shared_key" extension.

For PSKs provisioned via NewSessionTicket, a server MUST validate that the ticket age for the selected PSK identity (computed by subtracting ticket_age_add from PskIdentity.obfuscated_ticket_age modulo 2^32) is within a small tolerance of the time since the ticket was issued (see Section 4.2.10.4). If it is not, the server SHOULD proceed with the handshake but reject 0-RTT, and SHOULD NOT take any other action that assumes that this ClientHello is fresh.

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

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

- Ignore the extension and return a regular 1-RTT response. The server then ignores early data by attempting to decrypt received records in the handshake traffic keys until it is able to receive the client's second flight and complete an ordinary 1-RTT handshake, skipping records that fail to decrypt, up to the configured max_early_data_size.

- Request that the client send another ClientHello by responding with a HelloRetryRequest. A client MUST NOT include the "early_data" extension in its followup ClientHello. The server then ignores early data by skipping all records with external

content type of "application_data" (indicating that they are encrypted).

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

In order to accept early data, the server MUST have accepted a PSK cipher suite and selected the first key offered in the client's "pre_shared_key" extension. In addition, it MUST verify that the following values are consistent with those negotiated in the connection during which the ticket was established.

- The TLS version number and cipher suite.

- The selected ALPN [RFC7301] protocol, if any.

Future extensions MUST define their interaction with 0-RTT.

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

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

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

A TLS implementation SHOULD NOT automatically re-send early data; applications are in a better position to decide when re-transmission is appropriate. A TLS implementation MUST NOT automatically re-send early data unless the negotiated connection selects the same ALPN protocol.

#### 4.2.10. Pre-Shared Key Extension

The "pre_shared_key" extension is used to indicate the identity of the pre-shared key to be used with a given handshake in association with PSK key establishment.

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

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

opaque PskBinderEntry<32..255>;

struct {
    select (Handshake.msg_type) {
        case client_hello:
            PskIdentity identities<7..2^16-1>;
            PskBinderEntry binders<33..2^16-1>;

        case server_hello:
            uint16 selected_identity;
    };

} PreSharedKeyExtension;
```

identity A label for a key. For instance, a ticket defined in Appendix B.3.4, or a label for a pre-shared key established externally.

obfuscated_ticket_age An obfuscated version of the age of the key. Section 4.2.10.1 describes how to form this value for identities established via the NewSessionTicket message. For identities established externally an obfuscated_ticket_age of 0 SHOULD be used, and servers MUST ignore the value.

identities A list of the identities that the client is willing to negotiate with the server. If sent alongside the "early_data" extension (see Section 4.2.9), the first identity is the one used for 0-RTT data.

binders A series of HMAC values, one for each PSK offered in the "pre_shared_keys" extension and in the same order, computed as described below.

selected_identity The server's chosen identity expressed as a (0-based) index into the identities in the client's list.

Each PSK is associated with a single Hash algorithm. For PSKs established via the ticket mechanism (Section 4.6.1), this is the Hash used for the KDF on the connection where the ticket was established. For externally established PSKs, the Hash algorithm MUST be set when the PSK is established, or default to SHA-256 if no such algorithm is defined. The server must ensure that it selects a compatible PSK (if any) and cipher suite.

Implementor's note: the most straightforward way to implement the PSK/cipher suite matching requirements is to negotiate the cipher suite first and then exclude any incompatible PSKs. Any unknown PSKs (e.g., they are not in the PSK database or are encrypted with an unknown key) SHOULD simply be ignored. If no acceptable PSKs are found, the server SHOULD perform a non-PSK handshake if possible.

Prior to accepting PSK key establishment, the server MUST validate the corresponding binder value (see Section 4.2.10.2 below). If this value is not present or does not validate, the server MUST abort the handshake. Servers SHOULD NOT attempt to validate multiple binders; rather they SHOULD select a single PSK and validate solely the binder that corresponds to that PSK. In order to accept PSK key establishment, the server sends a "pre_shared_key" extension indicating the selected identity.

Clients MUST verify that the server's selected_identity is within the range supplied by the client, that the server selected a cipher suite indicating a Hash associated with the PSK and that a server "key_share" extension is present if required by the ClientHello "psk_key_exchange_modes". If these values are not consistent the client MUST abort the handshake with an "illegal_parameter" alert.

If the server supplies an "early_data" extension, the client MUST verify that the server's selected_identity is 0. If any other value is returned, the client MUST abort the handshake with an "illegal_parameter" alert.

This extension MUST be the last extension in the ClientHello (this facilitates implementation as described below). Servers MUST check that it is the last extension and otherwise fail the handshake with an "illegal_parameter" alert.

#### 4.2.10.1. Ticket Age

The client's view of the age of a ticket is the time since the receipt of the NewSessionTicket message. Clients MUST NOT attempt to use tickets which have ages greater than the "ticket_lifetime" value which was provided with the ticket. The "obfuscated_ticket_age" field of each PskIdentity contains an obfuscated version of the ticket age formed by taking the age in milliseconds and adding the "ticket_age_add" value that was included with the ticket, see Section 4.6.1 modulo 2^32. This addition prevents passive observers from correlating connections unless tickets are reused. Note that the "ticket_lifetime" field in the NewSessionTicket message is in seconds but the "obfuscated_ticket_age" is in milliseconds. Because ticket lifetimes are restricted to a week, 32 bits is enough to represent any plausible age, even in milliseconds.

#### 4.2.10.2. PSK Binder

The PSK binder value forms a binding between a PSK and the current handshake, as well as between the handshake in which the PSK was generated (if via a NewSessionTicket message) and the handshake where it was used. Each entry in the binders list is computed as an HMAC over a transcript hash (see Section 4.4.1) containing a partial ClientHello up to and including the PreSharedKeyExtension.identities field. That is, it includes all of the ClientHello but not the binders list itself. The length fields for the message (including the overall length, the length of the extensions block, and the length of the "pre_shared_key" extension) are all set as if binders of the correct lengths were present.

The PskBinderEntry is computed in the same way as the Finished message (Section 4.4.4) but with the BaseKey being the binder_key derived via the key schedule from the corresponding PSK which is being offered (see Section 7.1).

If the handshake includes a HelloRetryRequest, the initial ClientHello and HelloRetryRequest are included in the transcript along with the new ClientHello. For instance, if the client sends ClientHello1, its binder will be computed over:

```
Transcript-Hash(ClientHello1[truncated])
```

If the server responds with HelloRetryRequest, and the client then sends ClientHello2, its binder will be computed over:

```
Transcript-Hash(ClientHello1,
                HelloRetryRequest,
                ClientHello2[truncated])
```

The full ClientHello1 is included in all other handshake hash computations. Note that in the first flight, ClientHello1[truncated] is hashed directly, but in the second flight, ClientHello1 is hashed and then reinjected as a "handshake_hash" message, as described in Section 4.4.1.

#### 4.2.10.3. Processing Order

Clients are permitted to "stream" 0-RTT data until they receive the server's Finished, only then sending the EndOfEarlyData message. In order to avoid deadlocks, when accepting "early_data", servers MUST process the client's ClientHello and then immediately send the ServerHello, rather than waiting for the client's EndOfEarlyData message.

#### 4.2.10.4. Replay Properties

As noted in Section 2.3, TLS provides a limited mechanism for replay protection for data sent by the client in the first flight. This mechanism is intended to ensure that attackers cannot replay ClientHello messages at a time substantially after the original ClientHello was sent.

To properly validate the ticket age, a server needs to store the following values, either locally or by encoding them in the ticket:

- The time that the server generated the session ticket.

- The estimated round trip time between the client and server; this can be estimated by measuring the time between sending the Finished message and receiving the first message in the client's second flight, or potentially using information from the operating system.

- The "ticket_age_add" parameter from the NewSessionTicket message in which the ticket was established.

The server can determine the client's view of the age of the ticket by subtracting the ticket's "ticket_age_add value" from the "obfuscated_ticket_age" parameter in the client's "pre_shared_key" extension. The server can independently determine its view of the age of the ticket by subtracting the the time the ticket was issued from the current time. If the client and server clocks were running at the same rate, the client's view of would be shorter than the actual time elapsed on the server by a single round trip time. This difference is comprised of the delay in sending the NewSessionTicket message to the client, plus the time taken to send the ClientHello to the server.

The mismatch between the client's and server's views of age is thus given by:

mismatch = (client's view + RTT estimate) - (server's view)

There are several potential sources of error that make an exact measurement of time difficult. Variations in client and server clock rates are likely to be minimal, though potentially with gross time corrections. Network propagation delays are the most likely causes of a mismatch in legitimate values for elapsed time. Both the NewSessionTicket and ClientHello messages might be retransmitted and therefore delayed, which might be hidden by TCP. For browser clients on the Internet, this implies that an allowance on the order of ten seconds to account for errors in clocks and variations in measurements is advisable; other deployment scenarios may have different needs. Outside the selected range, the server SHOULD reject early data and fall back to a full 1-RTT handshake. Clock skew distributions are not symmetric, so the optimal tradeoff may involve an asymmetric range of permissible mismatch values.

### 4.3. Server Parameters

The next two messages from the server, EncryptedExtensions and CertificateRequest, contain information from the server that determines the rest of the handshake. These messages are encrypted with keys derived from the server_handshake_traffic_secret.

#### 4.3.1. Encrypted Extensions

In all handshakes, the server MUST send the EncryptedExtensions message immediately after the ServerHello message. This is the first message that is encrypted under keys derived from the server_handshake_traffic_secret.

The EncryptedExtensions message contains extensions that can be protected, i.e., any which are not needed to establish the cryptographic context, but which are not associated with individual certificates. The client MUST check EncryptedExtensions for the presence of any forbidden extensions and if any are found MUST abort the handshake with an "illegal_parameter" alert.

Structure of this message:

```
struct {
    Extension extensions<0..2^16-1>;
} EncryptedExtensions;
```

extensions A list of extensions. For more information, see the table in Section 4.2.

### 4.3.2. Certificate Request

A server which is authenticating with a certificate MAY optionally request a certificate from the client. This message, if sent, MUST follow EncryptedExtensions.

Structure of this message:

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

certificate_request_context An opaque string which identifies the certificate request and which will be echoed in the client's Certificate message. The certificate_request_context MUST be unique within the scope of this connection (thus preventing replay of client CertificateVerify messages). This field SHALL be zero length unless used for the post-handshake authentication exchanges described in Section 4.6.2. When requesting post-handshake authentication, the server SHOULD make the context unpredictable to the client (e.g., by randomly generating it) in order to prevent an attacker who has temporary access to the client's private key from pre-computing valid CertificateVerify messages.

extensions A set of extensions describing the parameters of the certificate being requested. The "signature_algorithms" extension MUST be specified, and other extensions may optionally be included if defined for this message. Clients MUST ignore unrecognized extensions.

In prior versions of TLS, the CertificateRequest message carried a list of signature algorithms and certificate authorities which the server would accept. In TLS 1.3 the former is expressed by sending the "signature_algorithms" extension. The latter is expressed by sending the "certificate_authorities" extension (see Section 4.2.4).

Servers which are authenticating with a PSK MUST NOT send the CertificateRequest message in the main handshake, though they MAY send it in post-handshake authentication (see Section 4.6.2) provided that the client has sent the "post_handshake_auth" extension (see Section 4.2.5).

#### 4.3.2.1. OID Filters

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

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

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

filters A list of certificate extension OIDs [RFC5280] with their allowed values, represented in DER-encoded [X690] format. Some certificate extension OIDs allow multiple values (e.g., Extended Key Usage). If the server has included a non-empty certificate_extensions list, the client certificate included in the response MUST contain all of the specified extension OIDs that the client recognizes. For each extension OID recognized by the client, all of the specified values MUST be present in the client certificate (but the certificate MAY have other values as well). However, the client MUST ignore and skip any unrecognized certificate extension OIDs. If the client ignored some of the required certificate extension OIDs and supplied a certificate that does not satisfy the request, the server MAY at its discretion either continue the connection without client authentication, or abort the handshake with an "unsupported_certificate" alert. PKIX RFCs define a variety of certificate extension OIDs and their corresponding value types. Depending on the type, matching certificate extension values are not necessarily bitwise-equal. It is expected that TLS implementations will rely on their PKI libraries to perform certificate selection using certificate extension OIDs. This document defines matching rules for two standard certificate extensions defined in [RFC5280]:

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

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

Separate specifications may define matching rules for other certificate extensions.

## 4.4. Authentication Messages

As discussed in Section 2, TLS generally uses a common set of messages for authentication, key confirmation, and handshake integrity: Certificate, CertificateVerify, and Finished. (The PreSharedKey binders also perform key confirmation, in a similar fashion.) These three messages are always sent as the last messages in their handshake flight. The Certificate and CertificateVerify messages are only sent under certain circumstances, as defined below. The Finished message is always sent as part of the Authentication block. These messages are encrypted under keys derived from [sender]_handshake_traffic_secret.

The computations for the Authentication messages all uniformly take the following inputs:

- The certificate and signing key to be used.

- A Handshake Context consisting of the set of messages to be included in the transcript hash.

- A base key to be used to compute a MAC key.

Based on these inputs, the messages then contain:

Certificate The certificate to be used for authentication, and any supporting certificates in the chain. Note that certificate-based client authentication is not available in the 0-RTT case.

CertificateVerify A signature over the value Transcript-Hash(Handshake Context, Certificate)

Finished A MAC over the value Transcript-Hash(Handshake Context, Certificate, CertificateVerify) using a MAC key derived from the base key.

The following table defines the Handshake Context and MAC Base Key for each scenario:

| Mode | Handshake Context | Base Key |
|---|---|---|
| Server | ClientHello ... later of EncryptedExtensions/CertificateRequest | server_handshake_traffic_secret |
| Client | ClientHello ... later of server Finished/EndOfEarlyData | client_handshake_traffic_secret |
| Post-Handshake | ClientHello ... client Finished + CertificateRequest | client_application_traffic_secret_N |

#### 4.4.1. The Transcript Hash

Many of the cryptographic computations in TLS make use of a transcript hash. This value is computed by hashing the concatenation of each included handshake message, including the handshake message header carrying the handshake message type and length fields, but not including record layer headers. I.e.,

```
Transcript-Hash(M1, M2, ... MN) = Hash(M1 || M2 ... MN)
```

As an exception to this general rule, when the server responds to a ClientHello with a HelloRetryRequest, the value of ClientHello1 is replaced with a special synthetic handshake message of handshake type "message_hash" containing Hash(ClientHello1). I.e.,

```
Transcript-Hash(ClientHello1, HelloRetryRequest, ... MN) =
    Hash(message_hash ||                // Handshake Type
         00 00 Hash.length ||   // Handshake message length
         Hash(ClientHello1) ||  // Hash of ClientHello1
         HelloRetryRequest ... MN)
```

The reason for this construction is to allow the server to do a stateless HelloRetryRequest by storing just the hash of ClientHello1 in the cookie, rather than requiring it to export the entire intermediate hash state (see Section 4.2.2).

For concreteness, the transcript hash is always taken from the following sequence of handshake messages, starting at the first ClientHello and including only those messages that were sent: ClientHello, HelloRetryRequest, ClientHello, ServerHello, EncryptedExtensions, server CertificateRequest, server Certificate,

server CertificateVerify, server Finished, EndOfEarlyData, client Certificate, client CertificateVerify, client Finished.

In general, implementations can implement the transcript by keeping a running transcript hash value based on the negotiated hash. Note, however, that subsequent post-handshake authentications do not include each other, just the messages through the end of the main handshake.

#### 4.4.2. Certificate

This message conveys the endpoint's certificate chain to the peer.

The server MUST send a Certificate message whenever the agreed-upon key exchange method uses certificates for authentication (this includes all key exchange methods defined in this document except PSK).

The client MUST send a Certificate message if and only if the server has requested client authentication via a CertificateRequest message (Section 4.3.2). If the server requests client authentication but no suitable certificate is available, the client MUST send a Certificate message containing no certificates (i.e., with the "certificate_list" field having length 0).

Structure of this message:

```
struct {
    select(certificate_type){
        case RawPublicKey:
          // From RFC 7250 ASN.1_subjectPublicKeyInfo
          opaque ASN1_subjectPublicKeyInfo<1..2^24-1>;

        case X.509:
          opaque cert_data<1..2^24-1>;
    }
    Extension extensions<0..2^16-1>;
} CertificateEntry;

struct {
    opaque certificate_request_context<0..2^8-1>;
    CertificateEntry certificate_list<0..2^24-1>;
} Certificate;
```

certificate_request_context If this message is in response to a CertificateRequest, the value of certificate_request_context in that message. Otherwise (in the case of server authentication), this field SHALL be zero length.

certificate_list This is a sequence (chain) of CertificateEntry structures, each containing a single certificate and set of extensions.

extensions: A set of extension values for the CertificateEntry. The "Extension" format is defined in Section 4.2. Valid extensions include OCSP Status extensions ([RFC6066] and [RFC6961]) and SignedCertificateTimestamps ([RFC6962]). An extension MUST only be present in a Certificate message if the corresponding ClientHello extension was presented in the initial handshake. If an extension applies to the entire chain, it SHOULD be included in the first CertificateEntry.

If the corresponding certificate type extension ("server_certificate_type" or "client_certificate_type") was not used or the X.509 certificate type was negotiated, then each CertificateEntry contains an X.509 certificate. The sender's certificate MUST come in the first CertificateEntry in the list. Each following certificate SHOULD directly certify one preceding it. Because certificate validation requires that trust anchors be distributed independently, a certificate that specifies a trust anchor MAY be omitted from the chain, provided that supported peers are known to possess any omitted certificates.

Note: Prior to TLS 1.3, "certificate_list" ordering required each certificate to certify the one immediately preceding it; however, some implementations allowed some flexibility. Servers sometimes send both a current and deprecated intermediate for transitional purposes, and others are simply configured incorrectly, but these cases can nonetheless be validated properly. For maximum compatibility, all implementations SHOULD be prepared to handle potentially extraneous certificates and arbitrary orderings from any TLS version, with the exception of the end-entity certificate which MUST be first.

If the RawPublicKey certificate type was negotiated, then the certificate_list MUST contain no more than one CertificateEntry, which contains an ASN1_subjectPublicKeyInfo value as defined in [RFC7250], Section 3.

The OpenPGP certificate type [RFC6091] MUST NOT be used with TLS 1.3.

The server's certificate_list MUST always be non-empty. A client will send an empty certificate_list if it does not have an appropriate certificate to send in response to the server's authentication request.

#### 4.4.2.1. OCSP Status and SCT Extensions

[RFC6066] and [RFC6961] provide extensions to negotiate the server sending OCSP responses to the client. In TLS 1.2 and below, the server replies with an empty extension to indicate negotiation of this extension and the OCSP information is carried in a CertificateStatus message. In TLS 1.3, the server's OCSP information is carried in an extension in the CertificateEntry containing the associated certificate. Specifically: The body of the "status_request" extension from the server MUST be a CertificateStatus structure as defined in [RFC6066], which is interpreted as defined in [RFC6960].

A server MAY request that a client present an OCSP response with its certificate by sending an empty "status_request" extension in its CertificateRequest message. If the client opts to send an OCSP response, the body of its "status_request" extension MUST be a CertificateStatus structure as defined in [RFC6066].

Similarly, [RFC6962] provides a mechanism for a server to send a Signed Certificate Timestamp (SCT) as an extension in the ServerHello in TLS 1.2 and below. In TLS 1.3, the server's SCT information is carried in an extension in CertificateEntry.

#### 4.4.2.2. Server Certificate Selection

The following rules apply to the certificates sent by the server:

- The certificate type MUST be X.509v3 [RFC5280], unless explicitly negotiated otherwise (e.g., [RFC7250]).

- The server's end-entity certificate's public key (and associated restrictions) MUST be compatible with the selected authentication algorithm (currently RSA or ECDSA).

- The certificate MUST allow the key to be used for signing (i.e., the digitalSignature bit MUST be set if the Key Usage extension is present) with a signature scheme indicated in the client's "signature_algorithms" extension.

- The "server_name" and "trusted_ca_keys" extensions [RFC6066] are used to guide certificate selection. As servers MAY require the presence of the "server_name" extension, clients SHOULD send this extension, when applicable.

All certificates provided by the server MUST be signed by a signature algorithm that appears in the "signature_algorithms" extension provided by the client, if they are able to provide such a chain (see

Section 4.2.3). Certificates that are self-signed or certificates that are expected to be trust anchors are not validated as part of the chain and therefore MAY be signed with any algorithm.

If the server cannot produce a certificate chain that is signed only via the indicated supported algorithms, then it SHOULD continue the handshake by sending the client a certificate chain of its choice that may include algorithms that are not known to be supported by the client. This fallback chain MAY use the deprecated SHA-1 hash algorithm only if the "signature_algorithms" extension provided by the client permits it. If the client cannot construct an acceptable chain using the provided certificates and decides to abort the handshake, then it MUST abort the handshake with an "unsupported_certificate" alert.

If the server has multiple certificates, it chooses one of them based on the above-mentioned criteria (in addition to other criteria, such as transport layer endpoint, local configuration and preferences).

#### 4.4.2.3. Client Certificate Selection

The following rules apply to certificates sent by the client:

- The certificate type MUST be X.509v3 [RFC5280], unless explicitly negotiated otherwise (e.g., [RFC7250]).

- If the "certificate_authorities" extension in the CertificateRequest message was present, at least one of the certificates in the certificate chain SHOULD be issued by one of the listed CAs.

- The certificates MUST be signed using an acceptable signature algorithm, as described in Section 4.3.2. Note that this relaxes the constraints on certificate-signing algorithms found in prior versions of TLS.

- If the certificate_extensions list in the CertificateRequest message was non-empty, the end-entity certificate MUST match the extension OIDs recognized by the client, as described in Section 4.3.2.

Note that, as with the server certificate, there are certificates that use algorithm combinations that cannot be currently used with TLS.

#### 4.4.2.4. Receiving a Certificate Message

In general, detailed certificate validation procedures are out of scope for TLS (see [RFC5280]). This section provides TLS-specific requirements.

If the server supplies an empty Certificate message, the client MUST abort the handshake with a "decode_error" alert.

If the client does not send any certificates, the server MAY at its discretion either continue the handshake without client authentication, or abort the handshake with a "certificate_required" alert. Also, if some aspect of the certificate chain was unacceptable (e.g., it was not signed by a known, trusted CA), the server MAY at its discretion either continue the handshake (considering the client unauthenticated) or abort the handshake.

Any endpoint receiving any certificate signed using any signature algorithm using an MD5 hash MUST abort the handshake with a "bad_certificate" alert. SHA-1 is deprecated and it is RECOMMENDED that any endpoint receiving any certificate signed using any signature algorithm using a SHA-1 hash abort the handshake with a "bad_certificate" alert. All endpoints are RECOMMENDED to transition to SHA-256 or better as soon as possible to maintain interoperability with implementations currently in the process of phasing out SHA-1 support.

Note that a certificate containing a key for one signature algorithm MAY be signed using a different signature algorithm (for instance, an RSA key signed with an ECDSA key).

### 4.4.3. Certificate Verify

This message is used to provide explicit proof that an endpoint possesses the private key corresponding to its certificate and also provides integrity for the handshake up to this point. Servers MUST send this message when authenticating via a certificate. Clients MUST send this message whenever authenticating via a certificate (i.e., when the Certificate message is non-empty). When sent, this message MUST appear immediately after the Certificate message and immediately prior to the Finished message.

Structure of this message:

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

The algorithm field specifies the signature algorithm used (see Section 4.2.3 for the definition of this field). The signature is a digital signature using that algorithm. The content that is covered under the signature is the hash output as described in Section 4.4, namely:

Transcript-Hash(Handshake Context, Certificate)

The digital signature is then computed over the concatenation of:

- A string that consists of octet 32 (0x20) repeated 64 times
- The context string
- A single 0 byte which serves as the separator
- The content to be signed

This structure is intended to prevent an attack on previous versions of TLS in which the ServerKeyExchange format meant that attackers could obtain a signature of a message with a chosen 32-byte prefix (ClientHello.random). The initial 64-byte pad clears that prefix along with the server-controlled ServerHello.random.

The context string for a server signature is "TLS 1.3, server CertificateVerify" and for a client signature is "TLS 1.3, client CertificateVerify". It is used to provide separation between signatures made in different contexts, helping against potential cross-protocol attacks.

For example, if the transcript hash was 32 bytes of 01 (this length would make sense for SHA-256), the content covered by the digital signature for a server CertificateVerify would be:

```
2020202020202020202020202020202020202020202020202020202020202020
2020202020202020202020202020202020202020202020202020202020202020
544c5320312e332c20736572766572204365727469666963617465566572696679
00
0101010101010101010101010101010101010101010101010101010101010101
```

On the sender side the process for computing the signature field of the CertificateVerify message takes as input:

- The content covered by the digital signature
- The private signing key corresponding to the certificate sent in the previous message

If the CertificateVerify message is sent by a server, the signature algorithm MUST be one offered in the client's "signature_algorithms" extension unless no valid certificate chain can be produced without unsupported algorithms (see Section 4.2.3).

If sent by a client, the signature algorithm used in the signature MUST be one of those present in the supported_signature_algorithms field of the "signature_algorithms" extension in the CertificateRequest message.

In addition, the signature algorithm MUST be compatible with the key in the sender's end-entity certificate. RSA signatures MUST use an RSASSA-PSS algorithm, regardless of whether RSASSA-PKCS1-v1_5 algorithms appear in "signature_algorithms". SHA-1 MUST NOT be used in any signatures in CertificateVerify. All SHA-1 signature algorithms in this specification are defined solely for use in legacy certificates, and are not valid for CertificateVerify signatures.

The receiver of a CertificateVerify message MUST verify the signature field. The verification process takes as input:

- The content covered by the digital signature

- The public key contained in the end-entity certificate found in the associated Certificate message.

- The digital signature received in the signature field of the CertificateVerify message

If the verification fails, the receiver MUST terminate the handshake with a "decrypt_error" alert.

#### 4.4.4. Finished

The Finished message is the final message in the authentication block. It is essential for providing authentication of the handshake and of the computed keys.

Recipients of Finished messages MUST verify that the contents are correct and if incorrect MUST terminate the connection with a "decrypt_error" alert.

Once a side has sent its Finished message and received and validated the Finished message from its peer, it may begin to send and receive application data over the connection. Early data may be sent prior to the receipt of the peer's Finished message, per Section 4.2.9.

The key used to compute the finished message is computed from the Base key defined in Section 4.4 using HKDF (see Section 7.1). Specifically:

```
finished_key =
    HKDF-Expand-Label(BaseKey, "finished", "", Hash.length)
```

Structure of this message:

```
struct {
    opaque verify_data[Hash.length];
} Finished;
```

The verify_data value is computed as follows:

```
verify_data =
    HMAC(finished_key,
         Transcript-Hash(Handshake Context,
                         Certificate*, CertificateVerify*))
```

* Only included if present.

Where HMAC [RFC2104] uses the Hash algorithm for the handshake. As noted above, the HMAC input can generally be implemented by a running hash, i.e., just the handshake hash at this point.

In previous versions of TLS, the verify_data was always 12 octets long. In TLS 1.3, it is the size of the HMAC output for the Hash used for the handshake.

Note: Alerts and any other record types are not handshake messages and are not included in the hash computations.

Any records following a 1-RTT Finished message MUST be encrypted under the appropriate application traffic key as described in Section 7.2. In particular, this includes any alerts sent by the server in response to client Certificate and CertificateVerify messages.

## 4.5. End of Early Data

```
struct {} EndOfEarlyData;
```

If the server sent an "early_data" extension, the client MUST send an EndOfEarlyData after receiving the server Finished. This indicates that all 0-RTT application_data messages, if any, have been transmitted and that the following records are protected under handshake traffic keys. Servers MUST NOT send this message and

clients receiving it MUST terminate the connection with an "unexpected_message" alert. This message is encrypted under keys derived from the client_early_traffic_secret.

### 4.6. Post-Handshake Messages

TLS also allows other messages to be sent after the main handshake. These messages use a handshake content type and are encrypted under the appropriate application traffic key.

#### 4.6.1. New Session Ticket Message

At any time after the server has received the client Finished message, it MAY send a NewSessionTicket message. This message creates a pre-shared key (PSK) binding between the ticket value and the resumption master secret.

The client MAY use this PSK for future handshakes by including the ticket value in the "pre_shared_key" extension in its ClientHello (Section 4.2.10). Servers MAY send multiple tickets on a single connection, either immediately after each other or after specific events. For instance, the server might send a new ticket after post-handshake authentication in order to encapsulate the additional client authentication state. Clients SHOULD attempt to use each ticket no more than once, with more recent tickets being used first.

Any ticket MUST only be resumed with a cipher suite that has the same KDF hash as that used to establish the original connection, and only if the client provides the same SNI value as in the original connection, as described in Section 3 of [RFC6066].

Note: Although the resumption master secret depends on the client's second flight, servers which do not request client authentication MAY compute the remainder of the transcript independently and then send a NewSessionTicket immediately upon sending its Finished rather than waiting for the client Finished. This might be appropriate in cases where the client is expected to open multiple TLS connections in parallel and would benefit from the reduced overhead of a resumption handshake, for example.

```
struct {
    uint32 ticket_lifetime;
    uint32 ticket_age_add;
    opaque ticket<1..2^16-1>;
    Extension extensions<0..2^16-2>;
} NewSessionTicket;
```

ticket_lifetime Indicates the lifetime in seconds as a 32-bit unsigned integer in network byte order from the time of ticket issuance. Servers MUST NOT use any value more than 604800 seconds (7 days). The value of zero indicates that the ticket should be discarded immediately. Clients MUST NOT cache tickets for longer than 7 days, regardless of the ticket_lifetime, and MAY delete the ticket earlier based on local policy. A server MAY treat a ticket as valid for a shorter period of time than what is stated in the ticket_lifetime.

ticket_age_add A securely generated, random 32-bit value that is used to obscure the age of the ticket that the client includes in the "pre_shared_key" extension. The client-side ticket age is added to this value modulo 2^32 to obtain the value that is transmitted by the client. The server MUST generate a fresh value for each ticket it sends.

ticket The value of the ticket to be used as the PSK identity. The ticket itself is an opaque label. It MAY either be a database lookup key or a self-encrypted and self-authenticated value. Section 4 of [RFC5077] describes a recommended ticket construction mechanism.

extensions A set of extension values for the ticket. The "Extension" format is defined in Section 4.2. Clients MUST ignore unrecognized extensions.

The sole extension currently defined for NewSessionTicket is "early_data", indicating that the ticket may be used to send 0-RTT data (Section 4.2.9)). It contains the following value:

max_early_data_size The maximum amount of 0-RTT data that the client is allowed to send when using this ticket, in bytes. Only Application Data payload (i.e., plaintext but not padding or the inner content type byte) is counted. A server receiving more than max_early_data_size bytes of 0-RTT data SHOULD terminate the connection with an "unexpected_message" alert. Note that servers that reject early data due to lack of cryptographic material will be unable to differentiate padding from content, so clients SHOULD NOT depend on being able to send large quantities of padding in early data records.

Note that in principle it is possible to continue issuing new tickets which indefinitely extend the lifetime of the keying material originally derived from an initial non-PSK handshake (which was most likely tied to the peer's certificate). It is RECOMMENDED that implementations place limits on the total lifetime of such keying material; these limits should take into account the lifetime of the

peer's certificate, the likelihood of intervening revocation, and the time since the peer's online CertificateVerify signature.

#### 4.6.2. Post-Handshake Authentication

When the client has sent the "post_handshake_auth" extension (see Section 4.2.5), a server MAY request client authentication at any time after the handshake has completed by sending a CertificateRequest message. The client MUST respond with the appropriate Authentication messages (see Section 4.4). If the client chooses to authenticate, it MUST send Certificate, CertificateVerify, and Finished. If it declines, it MUST send a Certificate message containing no certificates followed by Finished. All of the client's messages for a given response MUST appear consecutively on the wire with no intervening messages of other types.

A client that receives a CertificateRequest message without having sent the "post_handshake_auth" extension MUST send an "unexpected_message" fatal alert.

Note: Because client authentication could involve prompting the user, servers MUST be prepared for some delay, including receiving an arbitrary number of other messages between sending the CertificateRequest and receiving a response. In addition, clients which receive multiple CertificateRequests in close succession MAY respond to them in a different order than they were received (the certificate_request_context value allows the server to disambiguate the responses).

#### 4.6.3. Key and IV Update

```
enum {
    update_not_requested(0), update_requested(1), (255)
} KeyUpdateRequest;

struct {
    KeyUpdateRequest request_update;
} KeyUpdate;
```

request_update Indicates whether the recipient of the KeyUpdate should respond with its own KeyUpdate. If an implementation receives any other value, it MUST terminate the connection with an "illegal_parameter" alert.

The KeyUpdate handshake message is used to indicate that the sender is updating its sending cryptographic keys. This message can be sent by either peer after it has sent a Finished message. Implementations that receive a KeyUpdate message prior to receiving a Finished

message MUST terminate the connection with an "unexpected_message" alert. After sending a KeyUpdate message, the sender SHALL send all its traffic using the next generation of keys, computed as described in Section 7.2. Upon receiving a KeyUpdate, the receiver MUST update its receiving keys.

If the request_update field is set to "update_requested" then the receiver MUST send a KeyUpdate of its own with request_update set to "update_not_requested" prior to sending its next application data record. This mechanism allows either side to force an update to the entire connection, but causes an implementation which receives multiple KeyUpdates while it is silent to respond with a single update. Note that implementations may receive an arbitrary number of messages between sending a KeyUpdate with request_update set to update_requested and receiving the peer's KeyUpdate, because those messages may already be in flight. However, because send and receive keys are derived from independent traffic secrets, retaining the receive traffic secret does not threaten the forward secrecy of data sent before the sender changed keys.

If implementations independently send their own KeyUpdates with request_update set to "update_requested", and they cross in flight, then each side will also send a response, with the result that each side increments by two generations.

Both sender and receiver MUST encrypt their KeyUpdate messages with the old keys. Additionally, both sides MUST enforce that a KeyUpdate with the old key is received before accepting any messages encrypted with the new key. Failure to do so may allow message truncation attacks.

# 5. Record Protocol

The TLS record protocol takes messages to be transmitted, fragments the data into manageable blocks, protects the records, and transmits the result. Received data is verified and decrypted, reassembled, and then delivered to higher-level clients.

TLS records are typed, which allows multiple higher-level protocols to be multiplexed over the same record layer. This document specifies three content types: handshake, application data, and alert. Implementations MUST NOT send record types not defined in this document unless negotiated by some extension. If a TLS implementation receives an unexpected record type, it MUST terminate the connection with an "unexpected_message" alert. New record content type values are assigned by IANA in the TLS Content Type Registry as described in Section 10.

### 5.1. Record Layer

The record layer fragments information blocks into TLSPlaintext records carrying data in chunks of 2^14 bytes or less. Message boundaries are handled differently depending on the underlying ContentType. Any future content types MUST specify appropriate rules. Note that these rules are stricter than what was enforced in TLS 1.2.

Handshake messages MAY be coalesced into a single TLSPlaintext record or fragmented across several records, provided that:

- Handshake messages MUST NOT be interleaved with other record types. That is, if a handshake message is split over two or more records, there MUST NOT be any other records between them.

- Handshake messages MUST NOT span key changes. Implementations MUST verify that all messages immediately preceding a key change align with a record boundary; if not, then they MUST terminate the connection with an "unexpected_message" alert. Because the ClientHello, EndOfEarlyData, ServerHello, Finished, and KeyUpdate messages can immediately precede a key change, implementations MUST send these messages in alignment with a record boundary.

Implementations MUST NOT send zero-length fragments of Handshake types, even if those fragments contain padding.

Alert messages (Section 6) MUST NOT be fragmented across records and multiple Alert messages MUST NOT be coalesced into a single TLSPlaintext record. In other words, a record with an Alert type MUST contain exactly one message.

Application Data messages contain data that is opaque to TLS. Application Data messages are always protected. Zero-length fragments of Application Data MAY be sent as they are potentially useful as a traffic analysis countermeasure.

```
enum {
    invalid(0),
    alert(21),
    handshake(22),
    application_data(23),
    (255)
} ContentType;

struct {
    ContentType type;
    ProtocolVersion legacy_record_version;
    uint16 length;
    opaque fragment[TLSPlaintext.length];
} TLSPlaintext;
```

type The higher-level protocol used to process the enclosed fragment.

legacy_record_version This value MUST be set to 0x0301 for all records generated by a TLS 1.3 implementation. This field is deprecated and MUST be ignored for all purposes. Previous versions of TLS would use other values in this field under some circumstances.

length The length (in bytes) of the following TLSPlaintext.fragment. The length MUST NOT exceed 2^14 bytes. An endpoint that receives a record that exceeds this length MUST terminate the connection with a "record_overflow" alert.

fragment The data being transmitted. This value is transparent and is treated as an independent block to be dealt with by the higher-level protocol specified by the type field.

This document describes TLS 1.3, which uses the version 0x0304. This version value is historical, deriving from the use of 0x0301 for TLS 1.0 and 0x0300 for SSL 3.0. In order to maximize backwards compatibility, the record layer version identifies as simply TLS 1.0. Endpoints supporting multiple versions negotiate the version to use by following the procedure and requirements in Appendix D.

When record protection has not yet been engaged, TLSPlaintext structures are written directly onto the wire. Once record protection has started, TLSPlaintext records are protected and sent as described in the following section.

## 5.2. Record Payload Protection

The record protection functions translate a TLSPlaintext structure into a TLSCiphertext. The deprotection functions reverse the process. In TLS 1.3, as opposed to previous versions of TLS, all ciphers are modeled as "Authenticated Encryption with Additional Data" (AEAD) [RFC5116]. AEAD functions provide an unified encryption and authentication operation which turns plaintext into authenticated ciphertext and back again. Each encrypted record consists of a plaintext header followed by an encrypted body, which itself contains a type and optional padding.

```
struct {
    opaque content[TLSPlaintext.length];
    ContentType type;
    uint8 zeros[length_of_padding];
} TLSInnerPlaintext;

struct {
    ContentType opaque_type = 23; /* application_data */
    ProtocolVersion legacy_record_version = 0x0301; /* TLS v1.x */
    uint16 length;
    opaque encrypted_record[length];
} TLSCiphertext;
```

content The byte encoding of a handshake or an alert message, or the raw bytes of the application's data to send.

type The content type of the record.

zeros An arbitrary-length run of zero-valued bytes may appear in the cleartext after the type field. This provides an opportunity for senders to pad any TLS record by a chosen amount as long as the total stays within record size limits. See Section 5.4 for more details.

opaque_type The outer opaque_type field of a TLSCiphertext record is always set to the value 23 (application_data) for outward compatibility with middleboxes accustomed to parsing previous versions of TLS. The actual content type of the record is found in TLSInnerPlaintext.type after decryption.

legacy_record_version The legacy_record_version field is always 0x0301. TLS 1.3 TLSCiphertexts are not generated until after TLS 1.3 has been negotiated, so there are no historical compatibility concerns where other values might be received. Implementations MAY verify that the legacy_record_version field is 0x0301 and abort the connection if it is not. Note that the handshake

protocol including the ClientHello and ServerHello messages authenticates the protocol version, so this value is redundant.

length The length (in bytes) of the following TLSCiphertext.encrypted_record, which is the sum of the lengths of the content and the padding, plus one for the inner content type, plus any expansion added by the AEAD algorithm. The length MUST NOT exceed 2^14 + 256 bytes. An endpoint that receives a record that exceeds this length MUST terminate the connection with a "record_overflow" alert.

encrypted_record The AEAD-encrypted form of the serialized TLSInnerPlaintext structure.

AEAD algorithms take as input a single key, a nonce, a plaintext, and "additional data" to be included in the authentication check, as described in Section 2.1 of [RFC5116]. The key is either the client_write_key or the server_write_key, the nonce is derived from the sequence number (see Section 5.3) and the client_write_iv or server_write_iv, and the additional data input is empty (zero length). Derivation of traffic keys is defined in Section 7.3.

The plaintext input to the AEAD is the the encoded TLSInnerPlaintext structure.

The AEAD output consists of the ciphertext output from the AEAD encryption operation. The length of the plaintext is greater than the corresponding TLSPlaintext.length due to the inclusion of TLSInnerPlaintext.type and any padding supplied by the sender. The length of the AEAD output will generally be larger than the plaintext, but by an amount that varies with the AEAD algorithm. Since the ciphers might incorporate padding, the amount of overhead could vary with different lengths of plaintext. Symbolically,

```
AEADEncrypted =
    AEAD-Encrypt(write_key, nonce, plaintext)
```

In order to decrypt and verify, the cipher takes as input the key, nonce, and the AEADEncrypted value. The output is either the plaintext or an error indicating that the decryption failed. There is no separate integrity check. That is:

```
plaintext of encrypted_record =
    AEAD-Decrypt(peer_write_key, nonce, AEADEncrypted)
```

If the decryption fails, the receiver MUST terminate the connection with a "bad_record_mac" alert.

An AEAD algorithm used in TLS 1.3 MUST NOT produce an expansion greater than 255 octets. An endpoint that receives a record from its peer with TLSCiphertext.length larger than 2^14 + 256 octets MUST terminate the connection with a "record_overflow" alert. This limit is derived from the maximum TLSPlaintext length of 2^14 octets + 1 octet for ContentType + the maximum AEAD expansion of 255 octets.

### 5.3. Per-Record Nonce

A 64-bit sequence number is maintained separately for reading and writing records. Each sequence number is set to zero at the beginning of a connection and whenever the key is changed.

The appropriate sequence number is incremented by one after reading or writing each record. The first record transmitted under a particular set of traffic keys MUST use sequence number 0.

Because the size of sequence numbers is 64-bit, they should not wrap. If a TLS implementation would need to wrap a sequence number, it MUST either re-key (Section 4.6.3) or terminate the connection.

Each AEAD algorithm will specify a range of possible lengths for the per-record nonce, from N_MIN bytes to N_MAX bytes of input ([RFC5116]). The length of the TLS per-record nonce (iv_length) is set to the larger of 8 bytes and N_MIN for the AEAD algorithm (see [RFC5116] Section 4). An AEAD algorithm where N_MAX is less than 8 bytes MUST NOT be used with TLS. The per-record nonce for the AEAD construction is formed as follows:

1. The 64-bit record sequence number is encoded in network byte order and padded to the left with zeros to iv_length.

2. The padded sequence number is XORed with the static client_write_iv or server_write_iv, depending on the role.

The resulting quantity (of length iv_length) is used as the per-record nonce.

Note: This is a different construction from that in TLS 1.2, which specified a partially explicit nonce.

### 5.4. Record Padding

All encrypted TLS records can be padded to inflate the size of the TLSCiphertext. This allows the sender to hide the size of the traffic from an observer.

When generating a TLSCiphertext record, implementations MAY choose to pad. An unpadded record is just a record with a padding length of zero. Padding is a string of zero-valued bytes appended to the ContentType field before encryption. Implementations MUST set the padding octets to all zeros before encrypting.

Application Data records may contain a zero-length TLSInnerPlaintext.content if the sender desires. This permits generation of plausibly-sized cover traffic in contexts where the presence or absence of activity may be sensitive. Implementations MUST NOT send Handshake or Alert records that have a zero-length TLSInnerPlaintext.content; if such a message is received, the receiving implementation MUST terminate the connection with an "unexpected_message" alert.

The padding sent is automatically verified by the record protection mechanism; upon successful decryption of a TLSCiphertext.encrypted_record, the receiving implementation scans the field from the end toward the beginning until it finds a non-zero octet. This non-zero octet is the content type of the message. This padding scheme was selected because it allows padding of any encrypted TLS record by an arbitrary size (from zero up to TLS record size limits) without introducing new content types. The design also enforces all-zero padding octets, which allows for quick detection of padding errors.

Implementations MUST limit their scanning to the cleartext returned from the AEAD decryption. If a receiving implementation does not find a non-zero octet in the cleartext, it MUST terminate the connection with an "unexpected_message" alert.

The presence of padding does not change the overall record size limitations - the full encoded TLSInnerPlaintext MUST not exceed 2^14 octets. If the maximum fragment length is reduced, such as by the max_fragment_length extension from [RFC6066], then the reduced limit applies to the full plaintext, including the padding.

Selecting a padding policy that suggests when and how much to pad is a complex topic, and is beyond the scope of this specification. If the application layer protocol atop TLS has its own padding, it may be preferable to pad application_data TLS records within the application layer. Padding for encrypted handshake and alert TLS records must still be handled at the TLS layer, though. Later documents may define padding selection algorithms, or define a padding policy request mechanism through TLS extensions or some other means.

### 5.5. Limits on Key Usage

There are cryptographic limits on the amount of plaintext which can be safely encrypted under a given set of keys. [AEAD-LIMITS] provides an analysis of these limits under the assumption that the underlying primitive (AES or ChaCha20) has no weaknesses. Implementations SHOULD do a key update as described in Section 4.6.3 prior to reaching these limits.

For AES-GCM, up to 2^24.5 full-size records (about 24 million) may be encrypted on a given connection while keeping a safety margin of approximately 2^-57 for Authenticated Encryption (AE) security. For ChaCha20/Poly1305, the record sequence number would wrap before the safety limit is reached.

## 6. Alert Protocol

One of the content types supported by the TLS record layer is the alert type. Like other messages, alert messages are encrypted as specified by the current connection state.

Alert messages convey a description of the alert and a legacy field that conveyed the severity of the message in previous versions of TLS. In TLS 1.3, the severity is implicit in the type of alert being sent, and the 'level' field can safely be ignored. The "close_notify" alert is used to indicate orderly closure of the connection. Upon receiving such an alert, the TLS implementation SHOULD indicate end-of-data to the application.

Error alerts indicate abortive closure of the connection (see Section 6.2). Upon receiving an error alert, the TLS implementation SHOULD indicate an error to the application and MUST NOT allow any further data to be sent or received on the connection. Servers and clients MUST forget keys and secrets associated with a failed connection. Stateful implementations of tickets (as in many clients) SHOULD discard tickets associated with failed connections.

All the alerts listed in Section 6.2 MUST be sent as fatal and MUST be treated as fatal regardless of the AlertLevel in the message. Unknown alert types MUST be treated as fatal.

Note: TLS defines two generic alerts (see Section 6) to use upon failure to parse a message. Peers which receive a message which cannot be parsed according to the syntax (e.g., have a length extending beyond the message boundary or contain an out-of-range length) MUST terminate the connection with a "decode_error" alert. Peers which receive a message which is syntactically correct but

semantically invalid (e.g., a DHE share of p - 1, or an invalid enum) MUST terminate the connection with an "illegal_parameter" alert.

```
enum { warning(1), fatal(2), (255) } AlertLevel;

enum {
    close_notify(0),
    unexpected_message(10),
    bad_record_mac(20),
    record_overflow(22),
    handshake_failure(40),
    bad_certificate(42),
    unsupported_certificate(43),
    certificate_revoked(44),
    certificate_expired(45),
    certificate_unknown(46),
    illegal_parameter(47),
    unknown_ca(48),
    access_denied(49),
    decode_error(50),
    decrypt_error(51),
    protocol_version(70),
    insufficient_security(71),
    internal_error(80),
    inappropriate_fallback(86),
    user_canceled(90),
    missing_extension(109),
    unsupported_extension(110),
    certificate_unobtainable(111),
    unrecognized_name(112),
    bad_certificate_status_response(113),
    bad_certificate_hash_value(114),
    unknown_psk_identity(115),
    certificate_required(116),
    no_application_protocol(120),
    (255)
} AlertDescription;

struct {
    AlertLevel level;
    AlertDescription description;
} Alert;
```

### 6.1. Closure Alerts

The client and the server must share knowledge that the connection is ending in order to avoid a truncation attack.

close_notify This alert notifies the recipient that the sender will not send any more messages on this connection. Any data received after a closure MUST be ignored.

user_canceled This alert notifies the recipient that the sender is canceling the handshake for some reason unrelated to a protocol failure. If a user cancels an operation after the handshake is complete, just closing the connection by sending a "close_notify" is more appropriate. This alert SHOULD be followed by a "close_notify". This alert is generally a warning.

Either party MAY initiate a close by sending a "close_notify" alert. Any data received after a closure alert MUST be ignored. If a transport-level close is received prior to a "close_notify", the receiver cannot know that all the data that was sent has been received.

Each party MUST send a "close_notify" alert before closing the write side of the connection, unless some other fatal alert has been transmitted. The other party MUST respond with a "close_notify" alert of its own and close down the connection immediately, discarding any pending writes. The initiator of the close need not wait for the responding "close_notify" alert before closing the read side of the connection.

If the application protocol using TLS provides that any data may be carried over the underlying transport after the TLS connection is closed, the TLS implementation MUST receive the responding "close_notify" alert before indicating to the application layer that the TLS connection has ended. If the application protocol will not transfer any additional data, but will only close the underlying transport connection, then the implementation MAY choose to close the transport without waiting for the responding "close_notify". No part of this standard should be taken to dictate the manner in which a usage profile for TLS manages its data transport, including when connections are opened or closed.

Note: It is assumed that closing a connection reliably delivers pending data before destroying the transport.

## 6.2. Error Alerts

Error handling in the TLS Handshake Protocol is very simple. When an error is detected, the detecting party sends a message to its peer. Upon transmission or receipt of a fatal alert message, both parties MUST immediately close the connection.

Whenever an implementation encounters a fatal error condition, it SHOULD send an appropriate fatal alert and MUST close the connection without sending or receiving any additional data. In the rest of this specification, when the phrases "terminate the connection" and "abort the handshake" are used without a specific alert it means that the implementation SHOULD send the alert indicated by the descriptions below. The phrases "terminate the connection with a X alert" and "abort the handshake with a X alert" mean that the implementation MUST send alert X if it sends any alert. All alerts defined in this section below, as well as all unknown alerts, are universally considered fatal as of TLS 1.3 (see Section 6).

The following error alerts are defined:

unexpected_message An inappropriate message (e.g., the wrong handshake message, premature application data, etc.) was received. This alert should never be observed in communication between proper implementations.

bad_record_mac This alert is returned if a record is received which cannot be deprotected. Because AEAD algorithms combine decryption and verification, and also to avoid side channel attacks, this alert is used for all deprotection failures. This alert should never be observed in communication between proper implementations, except when messages were corrupted in the network.

record_overflow A TLSCiphertext record was received that had a length more than 2^14 + 256 bytes, or a record decrypted to a TLSPlaintext record with more than 2^14 bytes. This alert should never be observed in communication between proper implementations, except when messages were corrupted in the network.

handshake_failure Receipt of a "handshake_failure" alert message indicates that the sender was unable to negotiate an acceptable set of security parameters given the options available.

bad_certificate A certificate was corrupt, contained signatures that did not verify correctly, etc.

unsupported_certificate A certificate was of an unsupported type.

certificate_revoked A certificate was revoked by its signer.

certificate_expired A certificate has expired or is not currently valid.

certificate_unknown Some other (unspecified) issue arose in processing the certificate, rendering it unacceptable.

illegal_parameter A field in the handshake was incorrect or inconsistent with other fields. This alert is used for errors which conform to the formal protocol syntax but are otherwise incorrect.

unknown_ca A valid certificate chain or partial chain was received, but the certificate was not accepted because the CA certificate could not be located or could not be matched with a known trust anchor.

access_denied A valid certificate or PSK was received, but when access control was applied, the sender decided not to proceed with negotiation.

decode_error A message could not be decoded because some field was out of the specified range or the length of the message was incorrect. This alert is used for errors where the message does not conform to the formal protocol syntax. This alert should never be observed in communication between proper implementations, except when messages were corrupted in the network.

decrypt_error A handshake (not record-layer) cryptographic operation failed, including being unable to correctly verify a signature or validate a Finished message or a PSK binder.

protocol_version The protocol version the peer has attempted to negotiate is recognized but not supported. (see Appendix D)

insufficient_security Returned instead of "handshake_failure" when a negotiation has failed specifically because the server requires parameters more secure than those supported by the client.

internal_error An internal error unrelated to the peer or the correctness of the protocol (such as a memory allocation failure) makes it impossible to continue.

inappropriate_fallback Sent by a server in response to an invalid connection retry attempt from a client (see [RFC7507]).

missing_extension Sent by endpoints that receive a hello message not containing an extension that is mandatory to send for the offered TLS version or other negotiated parameters.

unsupported_extension Sent by endpoints receiving any hello message containing an extension known to be prohibited for inclusion in the given hello message, or including any extensions in a ServerHello or Certificate not first offered in the corresponding ClientHello.

certificate_unobtainable Sent by servers when unable to obtain a certificate from a URL provided by the client via the "client_certificate_url" extension (see [RFC6066]).

unrecognized_name Sent by servers when no server exists identified by the name provided by the client via the "server_name" extension (see [RFC6066]).

bad_certificate_status_response Sent by clients when an invalid or unacceptable OCSP response is provided by the server via the "status_request" extension (see [RFC6066]).

bad_certificate_hash_value Sent by servers when a retrieved object does not have the correct hash provided by the client via the "client_certificate_url" extension (see [RFC6066]).

unknown_psk_identity Sent by servers when PSK key establishment is desired but no acceptable PSK identity is provided by the client. Sending this alert is OPTIONAL; servers MAY instead choose to send a "decrypt_error" alert to merely indicate an invalid PSK identity.

certificate_required Sent by servers when a client certificate is desired but none was provided by the client.

no_application_protocol Sent by servers when a client "application_layer_protocol_negotiation" extension advertises protocols that the server does not support.

New Alert values are assigned by IANA as described in Section 10.

# 7. Cryptographic Computations

The TLS handshake establishes one or more input secrets which are combined to create the actual working keying material, as detailed below. The key derivation process incorporates both the input secrets and the handshake transcript. Note that because the handshake transcript includes the random values in the Hello messages, any given handshake will have different traffic secrets, even if the same input secrets are used, as is the case when the same PSK is used for multiple connections

## 7.1. Key Schedule

The key derivation process makes use of the HKDF-Extract and HKDF-Expand functions as defined for HKDF [RFC5869], as well as the functions defined below:

```
HKDF-Expand-Label(Secret, Label, HashValue, Length) =
     HKDF-Expand(Secret, HkdfLabel, Length)

Where HkdfLabel is specified as:

struct {
    uint16 length = Length;
    opaque label<7..255> = "tls13 " + Label;
    opaque hash_value<0..255> = HashValue;
} HkdfLabel;

Derive-Secret(Secret, Label, Messages) =
     HKDF-Expand-Label(Secret, Label,
                       Transcript-Hash(Messages), Hash.length)
```

The Hash function and the HKDF hash are the cipher suite hash algorithm. Hash.length is its output length in bytes. Messages are the concatenation of the indicated handshake messages, including the handshake message type and length fields, but not including record layer headers. Note that in some cases a zero-length HashValue (indicated by "") is passed to HKDF-Expand-Label.

Note: with common hash functions, any label longer than 12 characters requires an additional iteration of the hash function to compute. The labels in this specification have all been chosen to fit within this limit.

Given a set of n InputSecrets, the final "master secret" is computed by iteratively invoking HKDF-Extract with InputSecret_1, InputSecret_2, etc. The initial secret is simply a string of Hash.length zero bytes. Concretely, for the present version of TLS 1.3, secrets are added in the following order:

- PSK (a pre-shared key established externally or a resumption_master_secret value from a previous connection)

- (EC)DHE shared secret (Section 7.4)

This produces a full key derivation schedule shown in the diagram below. In this diagram, the following formatting conventions apply:

- HKDF-Extract is drawn as taking the Salt argument from the top and the IKM argument from the left.

- Derive-Secret’s Secret argument is indicated by the incoming arrow. For instance, the Early Secret is the Secret for generating the client_early_traffic_secret.


0
|
v
PSK -> HKDF-Extract = Early Secret
|
+-----> Derive-Secret(.,
| "ext binder" |
| "res binder",
| "")
| = binder_key
|
+-----> Derive-Secret(., "c e traffic",
| ClientHello)
| = client_early_traffic_secret
|
+-----> Derive-Secret(., "e exp master",
| ClientHello)
| = early_exporter_master_secret
v
Derive-Secret(., "derived", "")
|
v
(EC)DHE -> HKDF-Extract = Handshake Secret
|
+-----> Derive-Secret(., "c hs traffic",
| ClientHello...ServerHello)
| = client_handshake_traffic_secret
|
+-----> Derive-Secret(., "s hs traffic",
| ClientHello...ServerHello)
| = server_handshake_traffic_secret
v
Derive-Secret(., "derived", "")
|
v
0 -> HKDF-Extract = Master Secret
|
+-----> Derive-Secret(., "c ap traffic",
| ClientHello...server Finished)
| = client_application_traffic_secret_0
|
+-----> Derive-Secret(., "s ap traffic",
| ClientHello...server Finished)
| = server_application_traffic_secret_0
|
+-----> Derive-Secret(., "exp master",
| ClientHello...server Finished)
| = exporter_master_secret

```
                |
                +-----> Derive-Secret(., "res master",
                                      ClientHello...client Finished)
                                      = resumption_master_secret
```

The general pattern here is that the secrets shown down the left side of the diagram are just raw entropy without context, whereas the secrets down the right side include handshake context and therefore can be used to derive working keys without additional context. Note that the different calls to Derive-Secret may take different Messages arguments, even with the same secret. In a 0-RTT exchange, Derive-Secret is called with four distinct transcripts; in a 1-RTT-only exchange with three distinct transcripts.

If a given secret is not available, then the 0-value consisting of a string of Hash.length zero bytes is used. Note that this does not mean skipping rounds, so if PSK is not in use Early Secret will still be HKDF-Extract(0, 0). For the computation of the binder_secret, the label is "ext binder" for external PSKs (those provisioned outside of TLS) and "res binder" for resumption PSKs (those provisioned as the resumption master secret of a previous handshake). The different labels prevent the substitution of one type of PSK for the other.

There are multiple potential Early Secret values depending on which PSK the server ultimately selects. The client will need to compute one for each potential PSK; if no PSK is selected, it will then need to compute the early secret corresponding to the zero PSK.

## 7.2. Updating Traffic Keys and IVs

Once the handshake is complete, it is possible for either side to update its sending traffic keys using the KeyUpdate handshake message defined in Section 4.6.3. The next generation of traffic keys is computed by generating client_/server_application_traffic_secret_N+1 from client_/server_application_traffic_secret_N as described in this section then re-deriving the traffic keys as described in Section 7.3.

The next-generation application_traffic_secret is computed as:

```
application_traffic_secret_N+1 =
    HKDF-Expand-Label(application_traffic_secret_N,
                      "traffic upd", "", Hash.length)
```

Once client/server_application_traffic_secret_N+1 and its associated traffic keys have been computed, implementations SHOULD delete client_/server_application_traffic_secret_N and its associated traffic keys.

### 7.3. Traffic Key Calculation

The traffic keying material is generated from the following input values:

- A secret value

- A purpose value indicating the specific value being generated

- The length of the key

The traffic keying material is generated from an input traffic secret value using:

```
 [sender]_write_key = HKDF-Expand-Label(Secret, "key", "", key_length)
 [sender]_write_iv = HKDF-Expand-Label(Secret, "iv", "", iv_length)
```

[sender] denotes the sending side. The Secret value for each record type is shown in the table below.

| Record Type | Secret |
|---|---|
| 0-RTT Application | client_early_traffic_secret |
| Handshake | [sender]_handshake_traffic_secret |
| Application Data | [sender]_application_traffic_secret_N |

All the traffic keying material is recomputed whenever the underlying Secret changes (e.g., when changing from the handshake to application data keys or upon a key update).

### 7.4. (EC)DHE Shared Secret Calculation

#### 7.4.1. Finite Field Diffie-Hellman

For finite field groups, a conventional Diffie-Hellman computation is performed. The negotiated key (Z) is converted to a byte string by encoding in big-endian and padded with zeros up to the size of the prime. This byte string is used as the shared secret, and is used in the key schedule as specified above.

Note that this construction differs from previous versions of TLS which remove leading zeros.

#### 7.4.2. Elliptic Curve Diffie-Hellman

For secp256r1, secp384r1 and secp521r1, ECDH calculations (including parameter and key generation as well as the shared secret calculation) are performed according to [IEEE1363] using the ECKAS-DH1 scheme with the identity map as key derivation function (KDF), so that the shared secret is the x-coordinate of the ECDH shared secret elliptic curve point represented as an octet string. Note that this octet string (Z in IEEE 1363 terminology) as output by FE2OSP, the Field Element to Octet String Conversion Primitive, has constant length for any given field; leading zeros found in this octet string MUST NOT be truncated.

(Note that this use of the identity KDF is a technicality. The complete picture is that ECDH is employed with a non-trivial KDF because TLS does not directly use this secret for anything other than for computing other secrets.)

ECDH functions are used as follows:

- The public key to put into the KeyShareEntry.key_exchange structure is the result of applying the ECDH function to the secret key of appropriate length (into scalar input) and the standard public basepoint (into u-coordinate point input).

- The ECDH shared secret is the result of applying the ECDH function to the secret key (into scalar input) and the peer's public key (into u-coordinate point input). The output is used raw, with no processing.

For X25519 and X448, implementations SHOULD use the approach specified in [RFC7748] to calculate the Diffie-Hellman shared secret. Implementations MUST check whether the computed Diffie-Hellman shared secret is the all-zero value and abort if so, as described in Section 6 of [RFC7748]. If implementers use an alternative implementation of these elliptic curves, they SHOULD perform the additional checks specified in Section 7 of [RFC7748].

### 7.5. Exporters

[RFC5705] defines keying material exporters for TLS in terms of the TLS pseudorandom function (PRF). This document replaces the PRF with HKDF, thus requiring a new construction. The exporter interface remains the same.

The exporter value is computed as:

```
HKDF-Expand-Label(Derive-Secret(Secret, label, ""),
                  "exporter", Hash(context_value), key_length)
```

Where Secret is either the early_exporter_master_secret or the exporter_master_secret. Implementations MUST use the exporter_master_secret unless explicitly specified by the application. The early_exporter_master_secret is define for use in settings where an exporter is needed for 0-RTT data. A separate interface for the early exporter is RECOMMENDED, especially on a server where a single interface can make the early exporter inaccessible.

If no context is provided, the context_value is zero-length. Consequently, providing no context computes the same value as providing an empty context. This is a change from previous versions of TLS where an empty context produced a different output to an absent context. As of this document's publication, no allocated exporter label is used both with and without a context. Future specifications MUST NOT define a use of exporters that permit both an empty context and no context with the same label. New uses of exporters SHOULD provide a context in all exporter computations, though the value could be empty.

Requirements for the format of exporter labels are defined in section 4 of [RFC5705].

# 8. Compliance Requirements

## 8.1. Mandatory-to-Implement Cipher Suites

In the absence of an application profile standard specifying otherwise, a TLS-compliant application MUST implement the TLS_AES_128_GCM_SHA256 [GCM] cipher suite and SHOULD implement the TLS_AES_256_GCM_SHA384 [GCM] and TLS_CHACHA20_POLY1305_SHA256 [RFC7539] cipher suites. (see Appendix B.4)

A TLS-compliant application MUST support digital signatures with rsa_pkcs1_sha256 (for certificates), rsa_pss_sha256 (for CertificateVerify and certificates), and ecdsa_secp256r1_sha256. A TLS-compliant application MUST support key exchange with secp256r1 (NIST P-256) and SHOULD support key exchange with X25519 [RFC7748].

## 8.2. Mandatory-to-Implement Extensions

In the absence of an application profile standard specifying otherwise, a TLS-compliant application MUST implement the following TLS extensions:

- Supported Versions ("supported_versions"; Section 4.2.1)

- Cookie ("cookie"; Section 4.2.2)

- Signature Algorithms ("signature_algorithms"; Section 4.2.3)

- Negotiated Groups ("supported_groups"; Section 4.2.6)

- Key Share ("key_share"; Section 4.2.7)

- Server Name Indication ("server_name"; Section 3 of [RFC6066])

All implementations MUST send and use these extensions when offering applicable features:

- "supported_versions" is REQUIRED for all ClientHello messages.

- "signature_algorithms" is REQUIRED for certificate authentication.

- "supported_groups" and "key_share" are REQUIRED for DHE or ECDHE key exchange.

- "pre_shared_key" is REQUIRED for PSK key agreement.

A client is considered to be attempting to negotiate using this specification if the ClientHello contains a "supported_versions" extension 0x0304 the highest version number contained in its body. Such a ClientHello message MUST meet the following requirements:

- If not containing a "pre_shared_key" extension, it MUST contain both a "signature_algorithms" extension and a "supported_groups" extension.

- If containing a "supported_groups" extension, it MUST also contain a "key_share" extension, and vice versa. An empty KeyShare.client_shares vector is permitted.

Servers receiving a ClientHello which does not conform to these requirements MUST abort the handshake with a "missing_extension" alert.

Additionally, all implementations MUST support use of the "server_name" extension with applications capable of using it. Servers MAY require clients to send a valid "server_name" extension. Servers requiring this extension SHOULD respond to a ClientHello lacking a "server_name" extension by terminating the connection with a "missing_extension" alert.

## 9. Security Considerations

Security issues are discussed throughout this memo, especially in Appendix C, Appendix D, and Appendix E.

## 10. IANA Considerations

This document uses several registries that were originally created in [RFC4346]. IANA has updated these to reference this document. The registries and their allocation policies are below:

- TLS Cipher Suite Registry: Values with the first byte in the range 0-254 (decimal) are assigned via Specification Required [RFC5226]. Values with the first byte 255 (decimal) are reserved for Private Use [RFC5226].

  IANA [SHALL add/has added] the cipher suites listed in Appendix B.4 to the registry. The "Value" and "Description" columns are taken from the table. The "DTLS-OK" and "Recommended" columns are both marked as "Yes" for each new cipher suite. [[This assumes [I-D.ietf-tls-iana-registry-updates] has been applied.]]

- TLS ContentType Registry: Future values are allocated via Standards Action [RFC5226].

- TLS Alert Registry: Future values are allocated via Standards Action [RFC5226]. IANA [SHALL update/has updated] this registry to include values for "missing_extension" and "certificate_required".

- TLS HandshakeType Registry: Future values are allocated via Standards Action [RFC5226]. IANA [SHALL update/has updated] this registry to rename item 4 from "NewSessionTicket" to "new_session_ticket" and to add the "hello_retry_request", "encrypted_extensions", "end_of_early_data", "key_update", and "handshake_hash" values.

This document also uses the TLS ExtensionType Registry originally created in [RFC4366]. IANA has updated it to reference this document. The registry and its allocation policy is listed below:

- IANA [SHALL update/has updated] this registry to include the "key_share", "pre_shared_key", "psk_key_exchange_modes", "early_data", "cookie", "supported_versions", "certificate_authorities", "oid_filters", and "post_handshake_auth" extensions with the values defined in this document and the Recommended value of "Yes".

- IANA [SHALL update/has updated] this registry to include a "TLS 1.3" column which lists the messages in which the extension may appear. This column [SHALL be/has been] initially populated from the table in Section 4.2 with any extension not listed there marked as "-" to indicate that it is not used by TLS 1.3.

In addition, this document defines a new registry to be maintained by IANA:

- TLS SignatureScheme Registry: Values with the first byte in the range 0-254 (decimal) are assigned via Specification Required [RFC5226]. Values with the first byte 255 (decimal) are reserved for Private Use [RFC5226]. Values with the first byte in the range 0-6 or with the second byte in the range 0-3 that are not currently allocated are reserved for backwards compatibility. This registry SHALL have a "Recommended" column. The registry [shall be/ has been] initially populated with the values described in Section 4.2.3. The following values SHALL be marked as "Recommended": ecdsa_secp256r1_sha256, ecdsa_secp384r1_sha384, rsa_pss_sha256, rsa_pss_sha384, rsa_pss_sha512, ed25519.

# 11. References

## 11.1. Normative References

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

[GCM] Dworkin, M., "Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC", NIST Special Publication 800-38D, November 2007.

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

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

[RFC5226] Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 5226, DOI 10.17487/RFC5226, May 2008, <http://www.rfc-editor.org/info/rfc5226>.

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

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

[RFC5869] Krawczyk, H. and P. Eronen, "HMAC-based Extract-and-Expand Key Derivation Function (HKDF)", RFC 5869, DOI 10.17487/RFC5869, May 2010, <http://www.rfc-editor.org/info/rfc5869>.

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

[RFC6655] McGrew, D. and D. Bailey, "AES-CCM Cipher Suites for Transport Layer Security (TLS)", RFC 6655, DOI 10.17487/RFC6655, July 2012, <http://www.rfc-editor.org/info/rfc6655>.

[RFC6960] Santesson, S., Myers, M., Ankney, R., Malpani, A., Galperin, S., and C. Adams, "X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP", RFC 6960, DOI 10.17487/RFC6960, June 2013, <http://www.rfc-editor.org/info/rfc6960>.

[RFC6961] Pettersen, Y., "The Transport Layer Security (TLS) Multiple Certificate Status Request Extension", RFC 6961, DOI 10.17487/RFC6961, June 2013, <http://www.rfc-editor.org/info/rfc6961>.

[RFC6962] Laurie, B., Langley, A., and E. Kasper, "Certificate Transparency", RFC 6962, DOI 10.17487/RFC6962, June 2013, <http://www.rfc-editor.org/info/rfc6962>.

[RFC6979] Pornin, T., "Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA)", RFC 6979, DOI 10.17487/RFC6979, August 2013, <http://www.rfc-editor.org/info/rfc6979>.

[RFC7507] Moeller, B. and A. Langley, "TLS Fallback Signaling Cipher Suite Value (SCSV) for Preventing Protocol Downgrade Attacks", RFC 7507, DOI 10.17487/RFC7507, April 2015, <http://www.rfc-editor.org/info/rfc7507>.

[RFC7539] Nir, Y. and A. Langley, "ChaCha20 and Poly1305 for IETF Protocols", RFC 7539, DOI 10.17487/RFC7539, May 2015, <http://www.rfc-editor.org/info/rfc7539>.

[RFC7748] Langley, A., Hamburg, M., and S. Turner, "Elliptic Curves for Security", RFC 7748, DOI 10.17487/RFC7748, January 2016, <http://www.rfc-editor.org/info/rfc7748>.

[RFC7919] Gillmor, D., "Negotiated Finite Field Diffie-Hellman Ephemeral Parameters for Transport Layer Security (TLS)", RFC 7919, DOI 10.17487/RFC7919, August 2016, <http://www.rfc-editor.org/info/rfc7919>.

[RFC8017] Moriarty, K., Ed., Kaliski, B., Jonsson, J., and A. Rusch, "PKCS #1: RSA Cryptography Specifications Version 2.2", RFC 8017, DOI 10.17487/RFC8017, November 2016, <http://www.rfc-editor.org/info/rfc8017>.

[RFC8032] Josefsson, S. and I. Liusvaara, "Edwards-Curve Digital Signature Algorithm (EdDSA)", RFC 8032, DOI 10.17487/RFC8032, January 2017, <http://www.rfc-editor.org/info/rfc8032>.

[SHS] National Institute of Standards and Technology, U.S. Department of Commerce, "Secure Hash Standard", NIST FIPS PUB 180-4, March 2012.

[X690] ITU-T, "Information technology - ASN.1 encoding Rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)", ISO/IEC 8825-1:2002, 2002.

[X962] ANSI, "Public Key Cryptography For The Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA)", ANSI X9.62, 1998.

## 11.2. Informative References

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

[BBFKZG16]
Bhargavan, K., Brzuska, C., Fournet, C., Kohlweiss, M., Zanella-Beguelin, S., and M. Green, "Downgrade Resilience in Key-Exchange Protocols", Proceedings of IEEE Symposium on Security and Privacy (Oakland) 2016 , 2016.

[BBK17] Bhargavan, K., Blanchet, B., and N. Kobeissi, "Verified Models and Reference Implementations for the TLS 1.3 Standard Candidate", Proceedings of IEEE Symposium on Security and Privacy (Oakland) 2017 , 2017.

[BDFKPPRSZZ16]
Bhargavan, K., Delignat-Lavaud, A., Fournet, C., Kohlweiss, M., Pan, J., Protzenko, J., Rastogi, A., Swamy, N., Zanella-Beguelin, S., and J. Zinzindohoue, "Implementing and Proving the TLS 1.3 Record Layer", Proceedings of IEEE Symposium on Security and Privacy (Oakland) 2017 , December 2016, <https://eprint.iacr.org/2016/1178>.

[BMMT15] Badertscher, C., Matt, C., Maurer, U., and B. Tackmann, "Augmented Secure Channels and the Goal of the TLS 1.3 Record Layer", ProvSec 2015 , September 2015, <https://eprint.iacr.org/2015/394>.

[BT16] Bellare, M. and B. Tackmann, "The Multi-User Security of Authenticated Encryption: AES-GCM in TLS 1.3", Proceedings of CRYPTO 2016 , 2016, <https://eprint.iacr.org/2016/564>.

[CCG16] Cohn-Gordon, K., Cremers, C., and L. Garratt, "On Post-Compromise Security", IEEE Computer Security Foundations Symposium , 2015.

[CHHSV17] Cremers, C., Horvat, M., Hoyland, J., van der Merwe, T., and S. Scott, "Awkward Handshake: Possible mismatch of client/server view on client authentication in post-handshake mode in Revision 18", 2017, <https://www.ietf.org/mail-archive/web/tls/current/msg22382.html>.

[CHSV16] Cremers, C., Horvat, M., Scott, S., and T. van der Merwe, "Automated Analysis and Verification of TLS 1.3: 0-RTT, Resumption and Delayed Authentication", Proceedings of IEEE Symposium on Security and Privacy (Oakland) 2016 , 2016, <http://ieeexplore.ieee.org/document/7546518/>.

[CK01] Canetti, R. and H. Krawczyk, "Analysis of Key-Exchange Protocols and Their Use for Building Secure Channels", Proceedings of Eurocrypt 2001 , 2001.

[CLINIC] Miller, B., Huang, L., Joseph, A., and J. Tygar, "I Know Why You Went to the Clinic: Risks and Realization of HTTPS Traffic Analysis", Privacy Enhancing Technologies pp. 143-163, DOI 10.1007/978-3-319-08506-7_8, 2014.

[DFGS15] Dowling, B., Fischlin, M., Guenther, F., and D. Stebila, "A Cryptographic Analysis of the TLS 1.3 draft-10 Full and Pre-shared Key Handshake Protocol", Proceedings of ACM CCS 2015 , 2015, <https://eprint.iacr.org/2015/914>.

[DFGS16] Dowling, B., Fischlin, M., Guenther, F., and D. Stebila, "A Cryptographic Analysis of the TLS 1.3 draft-10 Full and Pre-shared Key Handshake Protocol", TRON 2016 , 2016, <https://eprint.iacr.org/2016/081>.

[DOW92] Diffie, W., van Oorschot, P., and M. Wiener, ""Authentication and authenticated key exchanges"", Designs, Codes and Cryptography , 1992.

[DSS] National Institute of Standards and Technology, U.S. Department of Commerce, "Digital Signature Standard, version 4", NIST FIPS PUB 186-4, 2013.

[ECDSA] American National Standards Institute, "Public Key Cryptography for the Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA)", ANSI ANS X9.62-2005, November 2005.

[FG17] Fischlin, M. and F. Guenther, "Replay Attacks on Zero Round-Trip Time: The Case of the TLS 1.3 Handshake Candidates", Proceedings of Euro S"P 2017 , 2017, <https://eprint.iacr.org/2017/082>.

[FGSW16] Fischlin, M., Guenther, F., Schmidt, B., and B. Warinschi, "Key Confirmation in Key Exchange: A Formal Treatment and Implications for TLS 1.3", Proceedings of IEEE Symposium on Security and Privacy (Oakland) 2016 , 2016, <http://ieeexplore.ieee.org/document/7546517/>.

[FW15] Florian Weimer, ., "Factoring RSA Keys With TLS Perfect Forward Secrecy", September 2015.

[HCJ16] Husak, M., Čermak, M., Jirsik, T., and P. Čeleda, "HTTPS traffic analysis and client identification using passive SSL/TLS fingerprinting", EURASIP Journal on Information Security Vol. 2016, DOI 10.1186/s13635-016-0030-7, February 2016.

[HGFS15] Hlauschek, C., Gruber, M., Fankhauser, F., and C. Schanes, "Prying Open Pandora's Box: KCI Attacks against TLS", Proceedings of USENIX Workshop on Offensive Technologies , 2015.

[I-D.ietf-tls-iana-registry-updates] Salowey, J. and S. Turner, "D/TLS IANA Registry Updates", draft-ietf-tls-iana-registry-updates-01 (work in progress), April 2017.

[I-D.ietf-tls-tls13-vectors] Thomson, M., "Example Handshake Traces for TLS 1.3", draft-ietf-tls-tls13-vectors-00 (work in progress), January 2017.

[IEEE1363] IEEE, "Standard Specifications for Public Key Cryptography", IEEE 1363 , 2000.

[KEYAGREEMENT] Barker, E., Lily Chen, ., Roginsky, A., and M. Smid, "Recommendation for Pair-Wise Key Establishment Schemes Using Discrete Logarithm Cryptography", NIST Special Publication 800-38D, May 2013.

[Kraw10] Krawczyk, H., "Cryptographic Extraction and Key Derivation: The HKDF Scheme", Proceedings of CRYPTO 2010 , 2010, <https://eprint.iacr.org/2010/264>.

[Kraw16] Krawczyk, H., "A Unilateral-to-Mutual Authentication Compiler for Key Exchange (with Applications to Client Authentication in TLS 1.3", Proceedings of ACM CCS 2016 , 2016, <https://eprint.iacr.org/2016/711>.

[KW16] Krawczyk, H. and H. Wee, "The OPTLS Protocol and TLS 1.3", Proceedings of Euro S"P 2016 , 2016, <https://eprint.iacr.org/2015/978>.

[LXZFH16] Li, X., Xu, J., Feng, D., Zhang, Z., and H. Hu, "Multiple Handshakes Security of TLS 1.3 Candidates", Proceedings of IEEE Symposium on Security and Privacy (Oakland) 2016 , 2016, <http://ieeexplore.ieee.org/document/7546519/>.

[PSK-FINISHED]
Cremers, C., Horvat, M., van der Merwe, T., and S. Scott, "Revision 10: possible attack if client authentication is allowed during PSK", 2015, <https://www.ietf.org/mail-archive/web/tls/current/msg18215.html>.

[REKEY] Abdalla, M. and M. Bellare, "Increasing the Lifetime of a Key: A Comparative Analysis of the Security of Re-keying Techniques", ASIACRYPT2000 , October 2000.

[RFC3552] Rescorla, E. and B. Korver, "Guidelines for Writing RFC Text on Security Considerations", BCP 72, RFC 3552, DOI 10.17487/RFC3552, July 2003, <http://www.rfc-editor.org/info/rfc3552>.

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

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

[RFC4366] Blake-Wilson, S., Nystrom, M., Hopwood, D., Mikkelsen, J., and T. Wright, "Transport Layer Security (TLS) Extensions", RFC 4366, DOI 10.17487/RFC4366, April 2006, <http://www.rfc-editor.org/info/rfc4366>.

[RFC4492] Blake-Wilson, S., Bolyard, N., Gupta, V., Hawk, C., and B. Moeller, "Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS)", RFC 4492, DOI 10.17487/RFC4492, May 2006, <http://www.rfc-editor.org/info/rfc4492>.

[RFC4681] Santesson, S., Medvinsky, A., and J. Ball, "TLS User Mapping Extension", RFC 4681, DOI 10.17487/RFC4681, October 2006, <http://www.rfc-editor.org/info/rfc4681>.

[RFC5077] Salowey, J., Zhou, H., Eronen, P., and H. Tschofenig, "Transport Layer Security (TLS) Session Resumption without Server-Side State", RFC 5077, DOI 10.17487/RFC5077, January 2008, <http://www.rfc-editor.org/info/rfc5077>.

[RFC5116] McGrew, D., "An Interface and Algorithms for Authenticated Encryption", RFC 5116, DOI 10.17487/RFC5116, January 2008, <http://www.rfc-editor.org/info/rfc5116>.

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

[RFC5764] McGrew, D. and E. Rescorla, "Datagram Transport Layer Security (DTLS) Extension to Establish Keys for the Secure Real-time Transport Protocol (SRTP)", RFC 5764, DOI 10.17487/RFC5764, May 2010, <http://www.rfc-editor.org/info/rfc5764>.

[RFC5929] Altman, J., Williams, N., and L. Zhu, "Channel Bindings for TLS", RFC 5929, DOI 10.17487/RFC5929, July 2010, <http://www.rfc-editor.org/info/rfc5929>.

[RFC6091] Mavrogiannopoulos, N. and D. Gillmor, "Using OpenPGP Keys for Transport Layer Security (TLS) Authentication", RFC 6091, DOI 10.17487/RFC6091, February 2011, <http://www.rfc-editor.org/info/rfc6091>.

[RFC6176] Turner, S. and T. Polk, "Prohibiting Secure Sockets Layer (SSL) Version 2.0", RFC 6176, DOI 10.17487/RFC6176, March 2011, <http://www.rfc-editor.org/info/rfc6176>.

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

[RFC6520] Seggelmann, R., Tuexen, M., and M. Williams, "Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS) Heartbeat Extension", RFC 6520, DOI 10.17487/RFC6520, February 2012, <http://www.rfc-editor.org/info/rfc6520>.

[RFC7230] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing", RFC 7230, DOI 10.17487/RFC7230, June 2014, <http://www.rfc-editor.org/info/rfc7230>.

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

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

[RFC7465] Popov, A., "Prohibiting RC4 Cipher Suites", RFC 7465, DOI 10.17487/RFC7465, February 2015, <http://www.rfc-editor.org/info/rfc7465>.

[RFC7568] Barnes, R., Thomson, M., Pironti, A., and A. Langley, "Deprecating Secure Sockets Layer Version 3.0", RFC 7568, DOI 10.17487/RFC7568, June 2015, <http://www.rfc-editor.org/info/rfc7568>.

[RFC7627] Bhargavan, K., Ed., Delignat-Lavaud, A., Pironti, A., Langley, A., and M. Ray, "Transport Layer Security (TLS) Session Hash and Extended Master Secret Extension", RFC 7627, DOI 10.17487/RFC7627, September 2015, <http://www.rfc-editor.org/info/rfc7627>.

[RFC7685] Langley, A., "A Transport Layer Security (TLS) ClientHello Padding Extension", RFC 7685, DOI 10.17487/RFC7685, October 2015, <http://www.rfc-editor.org/info/rfc7685>.

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

[RSA] Rivest, R., Shamir, A., and L. Adleman, "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems", Communications of the ACM v. 21, n. 2, pp. 120-126., February 1978.

[SIGMA] Krawczyk, H., "SIGMA: the 'SIGn-and-MAc' approach to authenticated Diffie-Hellman and its use in the IKE protocols", Proceedings of CRYPTO 2003 , 2003.

[SLOTH] Bhargavan, K. and G. Leurent, "Transcript Collision Attacks: Breaking Authentication in TLS, IKE, and SSH", Network and Distributed System Security Symposium (NDSS 2016) , 2016.

[SSL2] Hickman, K., "The SSL Protocol", February 1995.

[SSL3] Freier, A., Karlton, P., and P. Kocher, "The SSL 3.0 Protocol", November 1996.

[TIMING] Boneh, D. and D. Brumley, "Remote timing attacks are practical", USENIX Security Symposium, 2003.

[X501] "Information Technology - Open Systems Interconnection - The Directory: Models", ITU-T X.501, 1993.

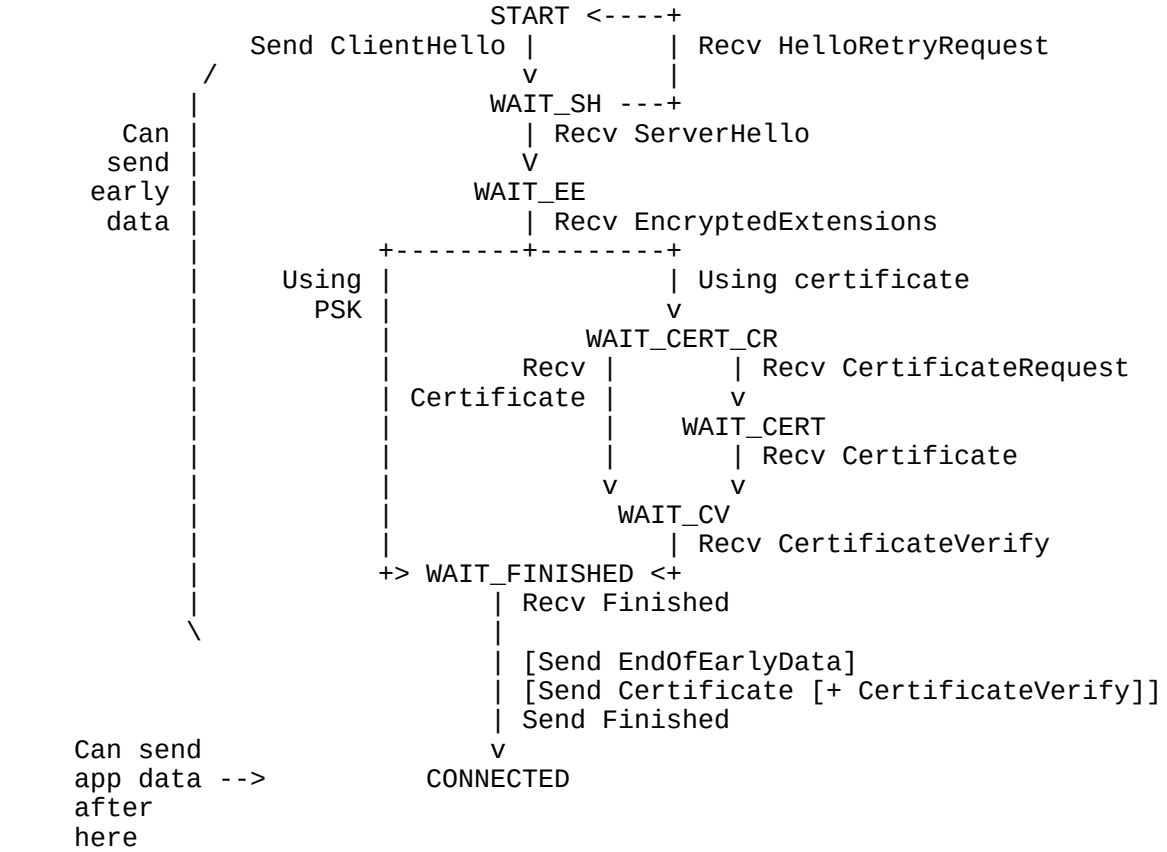
### 11.3. URIs

[1] mailto:tls@ietf.org

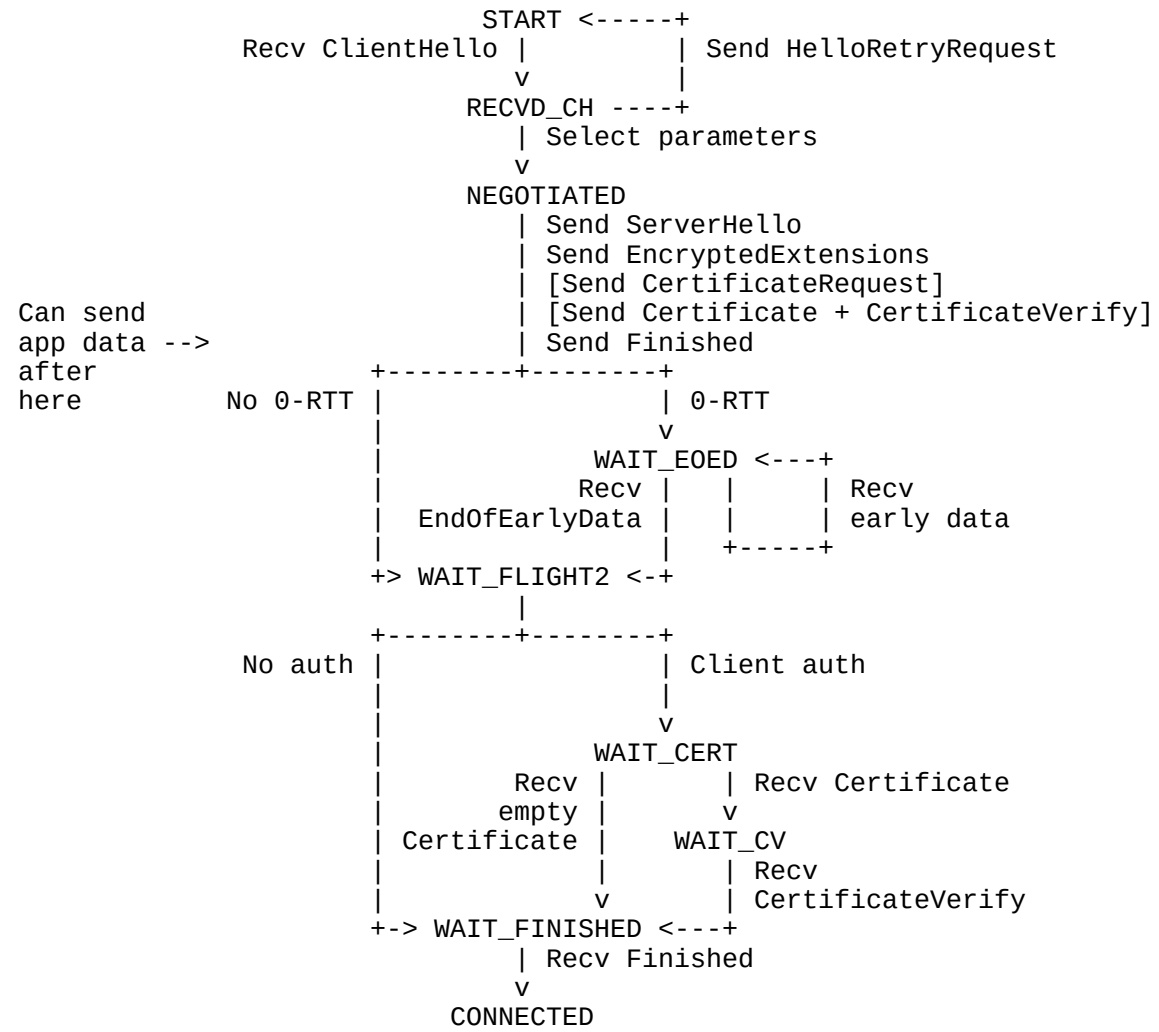
## Appendix A. State Machine

This section provides a summary of the legal state transitions for the client and server handshakes. State names (in all capitals, e.g., START) have no formal meaning but are provided for ease of comprehension. Messages which are sent only sometimes are indicated in [].

### A.1. Client




### A.2. Server




## Appendix B. Protocol Data Structures and Constant Values

This section describes protocol types and constants. Values listed as _RESERVED were used in previous versions of TLS and are listed here for completeness. TLS 1.3 implementations MUST NOT send them but might receive them from older TLS implementations.

### B.1. Record Layer

```
      enum {
          invalid(0),
          change_cipher_spec_RESERVED(20),
          alert(21),
          handshake(22),
          application_data(23),
          (255)
      } ContentType;

      struct {
          ContentType type;
          ProtocolVersion legacy_record_version;
          uint16 length;
          opaque fragment[TLSPlaintext.length];
      } TLSPlaintext;

      struct {
          opaque content[TLSPlaintext.length];
          ContentType type;
          uint8 zeros[length_of_padding];
      } TLSInnerPlaintext;

      struct {
          ContentType opaque_type = 23; /* application_data */
          ProtocolVersion legacy_record_version = 0x0301; /* TLS v1.x */
          uint16 length;
          opaque encrypted_record[length];
      } TLSCiphertext;
```

## B.2. Alert Messages

```
      enum { warning(1), fatal(2), (255) } AlertLevel;

      enum {
          close_notify(0),
          unexpected_message(10),
          bad_record_mac(20),
          decryption_failed_RESERVED(21),
          record_overflow(22),
          decompression_failure_RESERVED(30),
          handshake_failure(40),
          no_certificate_RESERVED(41),
          bad_certificate(42),
          unsupported_certificate(43),
          certificate_revoked(44),
          certificate_expired(45),
          certificate_unknown(46),
          illegal_parameter(47),
          unknown_ca(48),
          access_denied(49),
          decode_error(50),
          decrypt_error(51),
          export_restriction_RESERVED(60),
          protocol_version(70),
          insufficient_security(71),
          internal_error(80),
          inappropriate_fallback(86),
          user_canceled(90),
          no_renegotiation_RESERVED(100),
          missing_extension(109),
          unsupported_extension(110),
          certificate_unobtainable(111),
          unrecognized_name(112),
          bad_certificate_status_response(113),
          bad_certificate_hash_value(114),
          unknown_psk_identity(115),
          certificate_required(116),
          no_application_protocol(120),
          (255)
      } AlertDescription;

      struct {
          AlertLevel level;
          AlertDescription description;
      } Alert;
```

### B.3. Handshake Protocol

```
      enum {
          hello_request_RESERVED(0),
          client_hello(1),
          server_hello(2),
          hello_verify_request_RESERVED(3),
          new_session_ticket(4),
          end_of_early_data(5),
          hello_retry_request(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 */
          select (Handshake.msg_type) {
              case client_hello:          ClientHello;
              case server_hello:          ServerHello;
              case end_of_early_data:     EndOfEarlyData;
              case hello_retry_request:   HelloRetryRequest;
              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;
```

#### B.3.1. Key Exchange Messages

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

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

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

   struct {
       ProtocolVersion version;
       Random random;
       CipherSuite cipher_suite;
       Extension extensions<6..2^16-1>;
   } ServerHello;

   struct {
       ProtocolVersion server_version;
       CipherSuite cipher_suite;
       Extension extensions<2..2^16-1>;
   } HelloRetryRequest;

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

   enum {
       server_name(0),                             /* RFC 6066 */
       max_fragment_length(1),                     /* RFC 6066 */
       status_request(5),                          /* RFC 6066 */
       supported_groups(10),                       /* RFC 4492, 7919 */
       signature_algorithms(13),                   /* RFC 5246 */
       use_srtp(14),                               /* RFC 5764 */
       heartbeat(15),                              /* RFC 6520 */
       application_layer_protocol_negotiation(16), /* RFC 7301 */
       signed_certificate_timestamp(18),           /* RFC 6962 */
       client_certificate_type(19),                /* RFC 7250 */
       server_certificate_type(20)                 /* RFC 7250 */
       padding(21),                                /* RFC 7685 */
       key_share(40),                              /* [[this document]] */
       pre_shared_key(41),                         /* [[this document]] */
       early_data(42),                             /* [[this document]] */
       supported_versions(43),                     /* [[this document]] */
       cookie(44),                                 /* [[this document]] */
       psk_key_exchange_modes(45),                 /* [[this document]] */
       certificate_authorities(47),                /* [[this document]] */
       oid_filters(48),                            /* [[this document]] */
```

```
        post_handshake_auth(49),                    /* [[this document]] */
        (65535)
    } ExtensionType;

    struct {
        NamedGroup group;
        opaque key_exchange<1..2^16-1>;
    } KeyShareEntry;

    struct {
        select (Handshake.msg_type) {
            case client_hello:
                KeyShareEntry client_shares<0..2^16-1>;

            case hello_retry_request:
                NamedGroup selected_group;

            case server_hello:
                KeyShareEntry server_share;
        };
    } KeyShare;

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

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

    struct {} Empty;

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

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

    opaque PskBinderEntry<32..255>;

    struct {
        select (Handshake.msg_type) {
            case client_hello:
```

```
            PskIdentity identities<7..2^16-1>;
            PskBinderEntry binders<33..2^16-1>;

        case server_hello:
            uint16 selected_identity;
    };

} PreSharedKeyExtension;
```

#### B.3.1.1. Version Extension

```
struct {
    ProtocolVersion versions<2..254>;
} SupportedVersions;
```

#### B.3.1.2. Cookie Extension

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

#### B.3.1.3. Signature Algorithm Extension

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

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

          /* RSASSA-PSS algorithms */
          rsa_pss_sha256(0x0804),
          rsa_pss_sha384(0x0805),
          rsa_pss_sha512(0x0806),

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

          /* Legacy algorithms */
          rsa_pkcs1_sha1(0x0201),
          ecdsa_sha1(0x0203),

          /* Reserved Code Points */
          obsolete_RESERVED(0x0000..0x0200),
          dsa_sha1_RESERVED(0x0202),
          obsolete_RESERVED(0x0204..0x0400),
          dsa_sha256_RESERVED(0x0402),
          obsolete_RESERVED(0x0404..0x0500),
          dsa_sha384_RESERVED(0x0502),
          obsolete_RESERVED(0x0504..0x0600),
          dsa_sha512_RESERVED(0x0602),
          obsolete_RESERVED(0x0604..0x06FF),
          private_use(0xFE00..0xFFFF),
          (0xFFFF)
      } SignatureScheme;

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

#### B.3.1.4. Supported Groups Extension

```
      enum {
          /* Elliptic Curve Groups (ECDHE) */
          obsolete_RESERVED(0x0001..0x0016),
          secp256r1(0x0017), secp384r1(0x0018), secp521r1(0x0019),
          obsolete_RESERVED(0x001A..0x001C),
          x25519(0x001D), x448(0x001E),

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

          /* Reserved Code Points */
          ffdhe_private_use(0x01FC..0x01FF),
          ecdhe_private_use(0xFE00..0xFEFF),
          obsolete_RESERVED(0xFF01..0xFF02),
          (0xFFFF)
      } NamedGroup;

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

Values within "obsolete_RESERVED" ranges are used in previous versions of TLS and MUST NOT be offered or negotiated by TLS 1.3 implementations. The obsolete curves have various known/theoretical weaknesses or have had very little usage, in some cases only due to unintentional server configuration issues. They are no longer considered appropriate for general use and should be assumed to be potentially unsafe. The set of curves specified here is sufficient for interoperability with all currently deployed and properly configured TLS implementations.

### B.3.2. Server Parameters Messages

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

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

      struct {
          Extension extensions<0..2^16-1>;
      } EncryptedExtensions;

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

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

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

#### B.3.3. Authentication Messages

```
      struct {
          select(certificate_type){
              case RawPublicKey:
                // From RFC 7250 ASN.1_subjectPublicKeyInfo
                opaque ASN1_subjectPublicKeyInfo<1..2^24-1>;

              case X.509:
                opaque cert_data<1..2^24-1>;
          }
          Extension extensions<0..2^16-1>;
      } CertificateEntry;

      struct {
          opaque certificate_request_context<0..2^8-1>;
          CertificateEntry certificate_list<0..2^24-1>;
      } Certificate;

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

      struct {
          opaque verify_data[Hash.length];
      } Finished;
```

#### B.3.4. Ticket Establishment

```
      struct {
          uint32 ticket_lifetime;
          uint32 ticket_age_add;
          opaque ticket<1..2^16-1>;
          Extension extensions<0..2^16-2>;
      } NewSessionTicket;
```

#### B.3.5. Updating Keys

```
      struct {} EndOfEarlyData;

      enum {
          update_not_requested(0), update_requested(1), (255)
      } KeyUpdateRequest;

      struct {
          KeyUpdateRequest request_update;
      } KeyUpdate;
```

### B.4. Cipher Suites

A symmetric cipher suite defines the pair of the AEAD algorithm and hash algorithm to be used with HKDF. Cipher suite names follow the naming convention:

```
CipherSuite TLS_AEAD_HASH = VALUE;
```

| Component | Contents |
|---|---|
| TLS | The string "TLS" |
| AEAD | The AEAD algorithm used for record protection |
| HASH | The hash algorithm used with HKDF |
| VALUE | The two byte ID assigned for this cipher suite |

This specification defines the following cipher suites for use with TLS 1.3.

| Description | Value |
|---|---|
| TLS_AES_128_GCM_SHA256 | {0x13,0x01} |
| TLS_AES_256_GCM_SHA384 | {0x13,0x02} |
| TLS_CHACHA20_POLY1305_SHA256 | {0x13,0x03} |
| TLS_AES_128_CCM_SHA256 | {0x13,0x04} |
| TLS_AES_128_CCM_8_SHA256 | {0x13,0x05} |

The corresponding AEAD algorithms AEAD_AES_128_GCM, AEAD_AES_256_GCM, and AEAD_AES_128_CCM are defined in [RFC5116]. AEAD_CHACHA20_POLY1305 is defined in [RFC7539]. AEAD_AES_128_CCM_8 is defined in [RFC6655]. The corresponding hash algorithms are defined in [SHS].

Although TLS 1.3 uses the same cipher suite space as previous versions of TLS, TLS 1.3 cipher suites are defined differently, only specifying the symmetric ciphers, and cannot be used for TLS 1.2. Similarly, TLS 1.2 and lower cipher suites cannot be used with TLS 1.3.

New cipher suite values are assigned by IANA as described in Section 10.

# Appendix C. Implementation Notes

The TLS protocol cannot prevent many common security mistakes. This section provides several recommendations to assist implementors. [I-D.ietf-tls-tls13-vectors] provides test vectors for TLS 1.3 handshakes.

## C.1. API considerations for 0-RTT

0-RTT data has very different security properties from data transmitted after a completed handshake: it can be replayed. Implementations SHOULD provide different functions for reading and writing 0-RTT data and data transmitted after the handshake, and SHOULD NOT automatically resend 0-RTT data if it is rejected by the server.

## C.2. Random Number Generation and Seeding

TLS requires a cryptographically secure pseudorandom number generator (PRNG). In most cases, the operating system provides an appropriate facility such as /dev/urandom, which should be used absent other (performance) concerns. It is generally preferable to use an existing PRNG implementation in preference to crafting a new one, and many adequate cryptographic libraries are already available under favorable license terms. Should those prove unsatisfactory, [RFC4086] provides guidance on the generation of random values.

## C.3. Certificates and Authentication

Implementations are responsible for verifying the integrity of certificates and should generally support certificate revocation messages. Absent a specific indication from an application profile, Certificates should always be verified to ensure proper signing by a trusted Certificate Authority (CA). The selection and addition of trust anchors should be done very carefully. Users should be able to view information about the certificate and trust anchor. Applications SHOULD also enforce minimum and maximum key sizes. For example, certification paths containing keys or signatures weaker than 2048-bit RSA or 224-bit ECDSA are not appropriate for secure applications.

## C.4. Implementation Pitfalls

Implementation experience has shown that certain parts of earlier TLS specifications are not easy to understand, and have been a source of interoperability and security problems. Many of these areas have been clarified in this document, but this appendix contains a short list of the most important things that require special attention from implementors.

TLS protocol issues:

- Do you correctly handle handshake messages that are fragmented to multiple TLS records (see Section 5.1)? Including corner cases like a ClientHello that is split to several small fragments? Do you fragment handshake messages that exceed the maximum fragment size? In particular, the Certificate and CertificateRequest handshake messages can be large enough to require fragmentation.

- Do you ignore the TLS record layer version number in all unencrypted TLS records? (see Appendix D)

- Have you ensured that all support for SSL, RC4, EXPORT ciphers, and MD5 (via the "signature_algorithms" extension) is completely removed from all possible configurations that support TLS 1.3 or later, and that attempts to use these obsolete capabilities fail correctly? (see Appendix D)

- Do you handle TLS extensions in ClientHello correctly, including unknown extensions?

- When the server has requested a client certificate, but no suitable certificate is available, do you correctly send an empty Certificate message, instead of omitting the whole message (see Section 4.4.2.3)?

- When processing the plaintext fragment produced by AEAD-Decrypt and scanning from the end for the ContentType, do you avoid scanning past the start of the cleartext in the event that the peer has sent a malformed plaintext of all-zeros?

- Do you properly ignore unrecognized cipher suites (Section 4.1.2), hello extensions (Section 4.2), named groups (Section 4.2.6), key shares Section 4.2.7, supported versions Section 4.2.1, and signature algorithms (Section 4.2.3) in the ClientHello?

- As a server, do you send a HelloRetryRequest to clients which support a compatible (EC)DHE group but do not predict it in the

"key_share" extension? As a client, do you correctly handle a HelloRetryRequest from the server?

Cryptographic details:

- What countermeasures do you use to prevent timing attacks [TIMING]?

- When using Diffie-Hellman key exchange, do you correctly preserve leading zero bytes in the negotiated key (see Section 7.4.1)?

- Does your TLS client check that the Diffie-Hellman parameters sent by the server are acceptable, (see Section 4.2.7.1)?

- Do you use a strong and, most importantly, properly seeded random number generator (see Appendix C.2) when generating Diffie-Hellman private values, the ECDSA "k" parameter, and other security-critical values? It is RECOMMENDED that implementations implement "deterministic ECDSA" as specified in [RFC6979].

- Do you zero-pad Diffie-Hellman public key values to the group size (see Section 4.2.7.1)?

- Do you verify signatures after making them to protect against RSA-CRT key leaks? [FW15]

### C.5. Client Tracking Prevention

Clients SHOULD NOT reuse a ticket for multiple connections. Reuse of a ticket allows passive observers to correlate different connections. Servers that issue tickets SHOULD offer at least as many tickets as the number of connections that a client might use; for example, a web browser using HTTP/1.1 [RFC7230] might open six connections to a server. Servers SHOULD issue new tickets with every connection. This ensures that clients are always able to use a new ticket when creating a new connection.

### C.6. Unauthenticated Operation

Previous versions of TLS offered explicitly unauthenticated cipher suites based on anonymous Diffie-Hellman. These modes have been deprecated in TLS 1.3. However, it is still possible to negotiate parameters that do not provide verifiable server authentication by several methods, including:

- Raw public keys [RFC7250].

- Using a public key contained in a certificate but without validation of the certificate chain or any of its contents.

Either technique used alone is vulnerable to man-in-the-middle attacks and therefore unsafe for general use. However, it is also possible to bind such connections to an external authentication mechanism via out-of-band validation of the server's public key, trust on first use, or a mechanism such as channel bindings (though the channel bindings described in [RFC5929] are not defined for TLS 1.3). If no such mechanism is used, then the connection has no protection against active man-in-the-middle attack; applications MUST NOT use TLS in such a way absent explicit configuration or a specific application profile.

## Appendix D. Backward Compatibility

The TLS protocol provides a built-in mechanism for version negotiation between endpoints potentially supporting different versions of TLS.

TLS 1.x and SSL 3.0 use compatible ClientHello messages. Servers can also handle clients trying to use future versions of TLS as long as the ClientHello format remains compatible and the client supports the highest protocol version available in the server.

Prior versions of TLS used the record layer version number for various purposes. (TLSPlaintext.legacy_record_version and TLSCiphertext.legacy_record_version) As of TLS 1.3, this field is deprecated. The value of TLSPlaintext.legacy_record_version MUST be ignored by all implementations. The value of TLSCiphertext.legacy_record_version MAY be ignored, or MAY be validated to match the fixed constant value. Version negotiation is performed using only the handshake versions (ClientHello.legacy_version, ClientHello "supported_versions" extension, and ServerHello.version). In order to maximize interoperability with older endpoints, implementations that negotiate the use of TLS 1.0-1.2 SHOULD set the record layer version number to the negotiated version for the ServerHello and all records thereafter.

For maximum compatibility with previously non-standard behavior and misconfigured deployments, all implementations SHOULD support validation of certification paths based on the expectations in this document, even when handling prior TLS versions' handshakes. (see Section 4.4.2.2)

TLS 1.2 and prior supported an "Extended Master Secret" [RFC7627] extension which digested large parts of the handshake transcript into

the master secret. Because TLS 1.3 always hashes in the transcript up to the server CertificateVerify, implementations which support both TLS 1.3 and earlier versions SHOULD indicate the use of the Extended Master Secret extension in their APIs whenever TLS 1.3 is used.

### D.1. Negotiating with an older server

A TLS 1.3 client who wishes to negotiate with servers that do not support TLS 1.3 will send a normal TLS 1.3 ClientHello containing 0x0303 (TLS 1.2) in ClientHello.legacy_version but with the correct version in the "supported_versions" extension. If the server does not support TLS 1.3 it will respond with a ServerHello containing an older version number. If the client agrees to use this version, the negotiation will proceed as appropriate for the negotiated protocol. A client using a ticket for resumption SHOULD initiate the connection using the version that was previously negotiated.

Note that 0-RTT data is not compatible with older servers and SHOULD NOT be sent absent knowledge that the server supports TLS 1.3. See Appendix D.3.

If the version chosen by the server is not supported by the client (or not acceptable), the client MUST abort the handshake with a "protocol_version" alert.

Some legacy server implementations are known to not implement the TLS specification properly and might abort connections upon encountering TLS extensions or versions which they are not aware of. Interoperability with buggy servers is a complex topic beyond the scope of this document. Multiple connection attempts may be required in order to negotiate a backwards compatible connection; however, this practice is vulnerable to downgrade attacks and is NOT RECOMMENDED.

### D.2. Negotiating with an older client

A TLS server can also receive a ClientHello indicating a version number smaller than its highest supported version. If the "supported_versions" extension is present, the server MUST negotiate using that extension as described in Section 4.2.1. If the "supported_versions" extension is not present, the server MUST negotiate the minimum of ClientHello.legacy_version and TLS 1.2. For example, if the server supports TLS 1.0, 1.1, and 1.2, and legacy_version is TLS 1.0, the server will proceed with a TLS 1.0 ServerHello. If the "supported_versions" extension is absent and the server only supports versions greater than

ClientHello.legacy_version, the server MUST abort the handshake with a "protocol_version" alert.

Note that earlier versions of TLS did not clearly specify the record layer version number value in all cases (TLSPlaintext.legacy_record_version). Servers will receive various TLS 1.x versions in this field, but its value MUST always be ignored.

### D.3. Zero-RTT backwards compatibility

0-RTT data is not compatible with older servers. An older server will respond to the ClientHello with an older ServerHello, but it will not correctly skip the 0-RTT data and will fail to complete the handshake. This can cause issues when a client attempts to use 0-RTT, particularly against multi-server deployments. For example, a deployment could deploy TLS 1.3 gradually with some servers implementing TLS 1.3 and some implementing TLS 1.2, or a TLS 1.3 deployment could be downgraded to TLS 1.2.

A client that attempts to send 0-RTT data MUST fail a connection if it receives a ServerHello with TLS 1.2 or older. A client that attempts to repair this error SHOULD NOT send a TLS 1.2 ClientHello, but instead send a TLS 1.3 ClientHello without 0-RTT data.

To avoid this error condition, multi-server deployments SHOULD ensure a uniform and stable deployment of TLS 1.3 without 0-RTT prior to enabling 0-RTT.

### D.4. Backwards Compatibility Security Restrictions

Implementations negotiating use of older versions of TLS SHOULD prefer forward secure and AEAD cipher suites, when available.

The security of RC4 cipher suites is considered insufficient for the reasons cited in [RFC7465]. Implementations MUST NOT offer or negotiate RC4 cipher suites for any version of TLS for any reason.

Old versions of TLS permitted the use of very low strength ciphers. Ciphers with a strength less than 112 bits MUST NOT be offered or negotiated for any version of TLS for any reason.

The security of SSL 3.0 [SSL3] is considered insufficient for the reasons enumerated in [RFC7568], and MUST NOT be negotiated for any reason.

The security of SSL 2.0 [SSL2] is considered insufficient for the reasons enumerated in [RFC6176], and MUST NOT be negotiated for any reason.

Implementations MUST NOT send an SSL version 2.0 compatible CLIENT-HELLO. Implementations MUST NOT negotiate TLS 1.3 or later using an SSL version 2.0 compatible CLIENT-HELLO. Implementations are NOT RECOMMENDED to accept an SSL version 2.0 compatible CLIENT-HELLO in order to negotiate older versions of TLS.

Implementations MUST NOT send a ClientHello.legacy_version or ServerHello.version set to 0x0300 or less. Any endpoint receiving a Hello message with ClientHello.legacy_version or ServerHello.version set to 0x0300 MUST abort the handshake with a "protocol_version" alert.

Implementations MUST NOT send any records with a version less than 0x0300. Implementations SHOULD NOT accept any records with a version less than 0x0300 (but may inadvertently do so if the record version number is ignored completely).

Implementations MUST NOT use the Truncated HMAC extension, defined in Section 7 of [RFC6066], as it is not applicable to AEAD algorithms and has been shown to be insecure in some scenarios.

# Appendix E. Overview of Security Properties

A complete security analysis of TLS is outside the scope of this document. In this section, we provide an informal description the desired properties as well as references to more detailed work in the research literature which provides more formal definitions.

We cover properties of the handshake separately from those of the record layer.

## E.1. Handshake

The TLS handshake is an Authenticated Key Exchange (AKE) protocol which is intended to provide both one-way authenticated (server-only) and mutually authenticated (client and server) functionality. At the completion of the handshake, each side outputs its view of the following values:

- A set of "session keys" (the various secrets derived from the master secret) from which can be derived a set of working keys.

- A set of cryptographic parameters (algorithms, etc.)

- The identities of the communicating parties.

We assume that the attacker has complete control of the network in between the parties [RFC3552]. Even under these conditions, the

handshake should provide the properties listed below. Note that these properties are not necessarily independent, but reflect the protocol consumers' needs.

Establishing the same session keys. The handshake needs to output the same set of session keys on both sides of the handshake, provided that it completes successfully on each endpoint (See [CK01]; defn 1, part 1).

Secrecy of the session keys. The shared session keys should be known only to the communicating parties, not to the attacker (See [CK01]; defn 1, part 2). Note that in a unilaterally authenticated connection, the attacker can establish its own session keys with the server, but those session keys are distinct from those established by the client.

Peer Authentication. The client's view of the peer identity should reflect the server's identity. If the client is authenticated, the server's view of the peer identity should match the client's identity.

Uniqueness of the session keys: Any two distinct handshakes should produce distinct, unrelated session keys. Individual session keys produced by a handshake should also be distinct and unrelated.

Downgrade protection. The cryptographic parameters should be the same on both sides and should be the same as if the peers had been communicating in the absence of an attack (See [BBFKZG16]; defns 8 and 9}).

Forward secret with respect to long-term keys If the long-term keying material (in this case the signature keys in certificate-based authentication modes or the external/resumption PSK in PSK with (EC)DHE modes) is compromised after the handshake is complete, this does not compromise the security of the session key (See [DOW92]). The forward secrecy property is not satisfied when PSK is used in the "psk_ke" PskKeyExchangeMode.

Key Compromise Impersonation (KCI) resistance In a mutually-authenticated connection with certificates, peer authentication should hold even if the local long-term secret was compromised before the connection was established (see [HGFS15]). For example, if a client's signature key is compromised, it should not be possible to impersonate arbitrary servers to that client in subsequent handshakes.

Protection of endpoint identities. The server's identity (certificate) should be protected against passive attackers. The

client's identity should be protected against both passive and active attackers.

Informally, the signature-based modes of TLS 1.3 provide for the establishment of a unique, secret, shared, key established by an (EC)DHE key exchange and authenticated by the server's signature over the handshake transcript, as well as tied to the server's identity by a MAC. If the client is authenticated by a certificate, it also signs over the handshake transcript and provides a MAC tied to both identities. [SIGMA] describes the analysis of this type of key exchange protocol. If fresh (EC)DHE keys are used for each connection, then the output keys are forward secret.

The external PSK and resumption PSK bootstrap from a long-term shared secret into a unique per-connection set of short-term session keys. This secret may have been established in a previous handshake. If PSK with (EC)DHE key establishment is used, these session keys will also be forward secret. The resumption PSK has been designed so that the resumption master secret computed by connection N and needed to form connection N+1 is separate from the traffic keys used by connection N, thus providing forward secrecy between the connections.

The PSK binder value forms a binding between a PSK and the current handshake, as well as between the session where the PSK was established and the session where it was used. This binding transitively includes the original handshake transcript, because that transcript is digested into the values which produce the Resumption Master Secret. This requires that both the KDF used to produce the resumption master secret and the MAC used to compute the binder be collision resistant. See Appendix E.1.1 for more on this. Note: The binder does not cover the binder values from other PSKs, though they are included in the Finished MAC.

Note: TLS does not currently permit the server to send a certificate_request message in non-certificate-based handshakes (e.g., PSK). If this restriction were to be relaxed in future, the client's signature would not cover the server's certificate directly. However, if the PSK was established through a NewSessionTicket, the client's signature would transitively cover the server's certificate through the PSK binder. [PSK-FINISHED] describes a concrete attack on constructions that do not bind to the server's certificate. It is unsafe to use certificate-based client authentication when the client might potentially share the same PSK/key-id pair with two different endpoints. Implementations MUST NOT combine external PSKs with certificate-based authentication of either the client or the server.

If an exporter is used, then it produces values which are unique and secret (because they are generated from a unique session key).

Exporters computed with different labels and contexts are computationally independent, so it is not feasible to compute one from another or the session secret from the exported value. Note: exporters can produce arbitrary-length values. If exporters are to be used as channel bindings, the exported value MUST be large enough to provide collision resistance. The exporters provided in TLS 1.3 are derived from the same handshake contexts as the early traffic keys and the application traffic keys respectively, and thus have similar security properties. Note that they do not include the client's certificate; future applications which wish to bind to the client's certificate may need to define a new exporter that includes the full handshake transcript.

For all handshake modes, the Finished MAC (and where present, the signature), prevents downgrade attacks. In addition, the use of certain bytes in the random nonces as described in Section 4.1.3 allows the detection of downgrade to previous TLS versions.

As soon as the client and the server have exchanged enough information to establish shared keys, the remainder of the handshake is encrypted, thus providing protection against passive attackers. Because the server authenticates before the client, the client can ensure that it only reveals its identity to an authenticated server. Note that implementations must use the provided record padding mechanism during the handshake to avoid leaking information about the identities due to length. The client's proposed PSK identities are not encrypted, nor is the one that the server selects.

### E.1.1. Key Derivation and HKDF

Key derivation in TLS 1.3 uses the HKDF function defined in [RFC5869] and its two components, HKDF-Extract and HKDF-Expand. The full rationale for the HKDF construction can be found in [Kraw10] and the rationale for the way it is used in TLS 1.3 in [KW16]. Throughout this document, each application of HKDF-Extract is followed by one or more invocations of HKDF-Expand. This ordering should always be followed (including in future revisions of this document), in particular, one SHOULD NOT use an output of HKDF-Extract as an input to another application of HKDF-Extract without an HKDF-Expand in between. Consecutive applications of HKDF-Expand are allowed as long as these are differentiated via the key and/or the labels.

Note that HKDF-Expand implements a pseudorandom function (PRF) with both inputs and outputs of variable length. In some of the uses of HKDF in this document (e.g., for generating exporters and the resumption_master_secret), it is necessary that the application of HKDF-Expand be collision-resistant, namely, it should be infeasible to find two different inputs to HKDF-Expand that output the same

value. This requires the underlying hash function to be collision resistant and the output length from HKDF-Expand to be of size at least 256 bits (or as much as needed for the hash function to prevent finding collisions).

#### E.1.2. Client Authentication

A client that has sent authentication data to a server, either during the handshake or in post-handshake authentication, cannot be sure if the server afterwards considers the client to be authenticated or not. If the client needs to determine if the server considers the connection to be unilaterally or mutually authenticated, this has to be provisioned by the application layer. See [CHHSV17] for details. In addition, the analysis of post-handshake authentication from [Kraw16] shows that the client identified by the certificate sent in the post-handshake phase possesses the traffic key. This party is therefore the client that participated in the original handshake or one to whom the original client delegated the traffic key (assuming that the traffic key has not been compromised).

#### E.1.3. 0-RTT

The 0-RTT mode of operation generally provides the same security properties as 1-RTT data, with the two exceptions that the 0-RTT encryption keys do not provide full forward secrecy and that the server is not able to guarantee full uniqueness of the handshake (non-replayability) without keeping potentially undue amounts of state. See Section 4.2.9 for one mechanism to limit the exposure to replay.

#### E.1.4. Post-Compromise Security

TLS does not provide security for handshakes which take place after the peer’s long-term secret (signature key or external PSK) is compromised. It therefore does not provide post-compromise security [CCG16], sometimes also referred to as backwards or future security. This is in contrast to KCI resistance, which describes the security guarantees that a party has after its own long-term secret has been compromised.

#### E.1.5. External References

The reader should refer to the following references for analysis of the TLS handshake: [DFGS15] [CHSV16] [DFGS16] [KW16] [Kraw16] [FGSW16] [LXZFH16] [FG17] [BBK17].

### E.2. Record Layer

The record layer depends on the handshake producing strong traffic secrets which can be used to derive bidirectional encryption keys and nonces. Assuming that is true, and the keys are used for no more data than indicated in Section 5.5 then the record layer should provide the following guarantees:

Confidentiality. An attacker should not be able to determine the plaintext contents of a given record.

Integrity. An attacker should not be able to craft a new record which is different from an existing record which will be accepted by the receiver.

Order protection/non-replayability An attacker should not be able to cause the receiver to accept a record which it has already accepted or cause the receiver to accept record N+1 without having first processed record N.

Length concealment. Given a record with a given external length, the attacker should not be able to determine the amount of the record that is content versus padding.

Forward security after key change. If the traffic key update mechanism described in Section 4.6.3 has been used and the previous generation key is deleted, an attacker who compromises the endpoint should not be able to decrypt traffic encrypted with the old key.

Informally, TLS 1.3 provides these properties by AEAD-protecting the plaintext with a strong key. AEAD encryption [RFC5116] provides confidentiality and integrity for the data. Non-replayability is provided by using a separate nonce for each record, with the nonce being derived from the record sequence number (Section 5.3), with the sequence number being maintained independently at both sides thus records which are delivered out of order result in AEAD deprotection failures.

The re-keying technique in TLS 1.3 (see Section 7.2) follows the construction of the serial generator in [REKEY], which shows that re-keying can allow keys to be used for a larger number of encryptions than without re-keying. This relies on the security of the HKDF-Expand-Label function as a pseudorandom function (PRF). In addition, as long as this function is truly one way, it is not possible to compute traffic keys from prior to a key change (forward secrecy).

TLS does not provide security for data which is communicated on a connection after a traffic secret of that connection is compromised. That is, TLS does not provide post-compromise security/future secrecy/backward secrecy with respect to the traffic secret. Indeed, an attacker who learns a traffic secret can compute all future traffic secrets on that connection. Systems which want such guarantees need to do a fresh handshake and establish a new connection with an (EC)DHE exchange.

#### E.2.1. External References

The reader should refer to the following references for analysis of the TLS record layer: [BMMT15] [BT16] [BDFKPPRSZZ16] [BBK17].

### E.3. Traffic Analysis

TLS is susceptible to a variety of traffic analysis attacks based on observing the length and timing of encrypted packets [CLINIC] [HCJ16]. This is particularly easy when there is a small set of possible messages to be distinguished, such as for a video server hosting a fixed corpus of content, but still provides usable information even in more complicated scenarios.

TLS does not provide any specific defenses against this form of attack but does include a padding mechanism for use by applications: The plaintext protected by the AEAD function consists of content plus variable-length padding, which allows the application to produce arbitrary length encrypted records as well as padding-only cover traffic to conceal the difference between periods of transmission and periods of silence. Because the padding is encrypted alongside the actual content, an attacker cannot directly determine the length of the padding, but may be able to measure it indirectly by the use of timing channels exposed during record processing (i.e., seeing how long it takes to process a record or trickling in records to see which ones elicit a response from the server). In general, it is not known how to remove all of these channels because even a constant time padding removal function will then feed the content into data-dependent functions.

Note: Robust traffic analysis defences will likely lead to inferior performance due to delay in transmitting packets and increased traffic volume.

### E.4. Side Channel Attacks

In general, TLS does not have specific defenses against side-channel attacks (i.e., those which attack the communications via secondary channels such as timing) leaving those to the implementation of the

relevant cryptographic primitives. However, certain features of TLS are designed to make it easier to write side-channel resistant code:

- Unlike previous versions of TLS which used a composite MAC-then-encrypt structure, TLS 1.3 only uses AEAD algorithms, allowing implementations to use self-contained constant-time implementations of those primitives.

- TLS uses a uniform "bad_record_mac" alert for all decryption errors, which is intended to prevent an attacker from gaining piecewise insight into portions of the message. Additional resistance is provided by terminating the connection on such errors; a new connection will have different cryptographic material, preventing attacks against the cryptographic primitives that require multiple trials.

Information leakage through side channels can occur at layers above TLS, in application protocols and the applications that use them. Resistance to side-channel attacks depends on applications and application protocols separately ensuring that confidential information is not inadvertently leaked.

## Appendix F. 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://www.ietf.org/mailman/listinfo/tls

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

## Appendix G. Contributors

- Martin Abadi
  University of California, Santa Cruz
  abadi@cs.ucsc.edu

- Christopher Allen (co-editor of TLS 1.0)
  Alacrity Ventures
  ChristopherA@AlacrityManagement.com

- Steven M. Bellovin
  Columbia University
  smb@cs.columbia.edu

- David Benjamin
  Google

davidben@google.com

- Benjamin Beurdouche
  INRIA & Microsoft Research - Joint Center
  benjamin.beurdouche@ens.fr

- Karthikeyan Bhargavan (co-author of [RFC7627])
  INRIA
  karthikeyan.bhargavan@inria.fr

- Simon Blake-Wilson (co-author of [RFC4492])
  BCI
  sblakewilson@bcisse.com

- Nelson Bolyard (co-author of [RFC4492])
  Sun Microsystems, Inc.
  nelson@bolyard.com

- Ran Canetti
  IBM
  canetti@watson.ibm.com

- Pete Chown
  Skygate Technology Ltd
  pc@skygate.co.uk

- Katriel Cohn-Gordon
  University of Oxford
  me@katriel.co.uk

- Cas Cremers
  University of Oxford
  cas.cremers@cs.ox.ac.uk

- Antoine Delignat-Lavaud (co-author of [RFC7627])
  INRIA
  antoine.delignat-lavaud@inria.fr

- Tim Dierks (co-editor of TLS 1.0, 1.1, and 1.2)
  Independent
  tim@dierks.org

- Taher Elgamal
  Securify
  taher@securify.com

- Pasi Eronen
  Nokia

pasi.eronen@nokia.com

- Cedric Fournet
  Microsoft
  fournet@microsoft.com

- Anil Gangolli
  anil@busybuddha.org

- David M. Garrett
  dave@nulldereference.com

- Alessandro Ghedini
  Cloudflare Inc.
  alessandro@cloudflare.com

- Daniel Kahn Gillmor
  ACLU
  dkg@fifthhorseman.net

- Matthew Green
  Johns Hopkins University
  mgreen@cs.jhu.edu

- Jens Guballa
  ETAS
  jens.guballa@etas.com

- Felix Guenther
  TU Darmstadt
  mail@felixguenther.info

- Vipul Gupta (co-author of [RFC4492])
  Sun Microsystems Laboratories
  vipul.gupta@sun.com

- Chris Hawk (co-author of [RFC4492])
  Corriente Networks LLC
  chris@corriente.net

- Kipp Hickman

- Alfred Hoenes

- David Hopwood
  Independent Consultant
  david.hopwood@blueyonder.co.uk

- Marko Horvat
  MPI-SWS
  mhorvat@mpi-sws.org

- Jonathan Hoyland
  Royal Holloway, University of London

- Subodh Iyengar
  Facebook
  subodh@fb.com

- Benjamin Kaduk
  Akamai
  kaduk@mit.edu

- Hubert Kario
  Red Hat Inc.
  hkario@redhat.com

- Phil Karlton (co-author of SSL 3.0)

- Leon Klingele
  Independent
  mail@leonklingele.de

- Paul Kocher (co-author of SSL 3.0)
  Cryptography Research
  paul@cryptography.com

- Hugo Krawczyk
  IBM
  hugo@ee.technion.ac.il

- Adam Langley (co-author of [RFC7627])
  Google
  agl@google.com

- Olivier Levillain
  ANSSI
  olivier.levillain@ssi.gouv.fr

- Xiaoyin Liu
  University of North Carolina at Chapel Hill
  xiaoyin.l@outlook.com

- Ilari Liusvaara
  Independent
  ilariliusvaara@welho.com

- Atul Luykx
  K.U. Leuven
  atul.luykx@kuleuven.be

- Carl Mehner
  USAA
  carl.mehner@usaa.com

- Jan Mikkelsen
  Transactionware
  janm@transactionware.com

- Bodo Moeller (co-author of [RFC4492])
  Google
  bodo@openssl.org

- Kyle Nekritz
  Facebook
  knekritz@fb.com

- Erik Nygren
  Akamai Technologies
  erik+ietf@nygren.org

- Magnus Nystrom
  Microsoft
  mnystrom@microsoft.com

- Kazuho Oku
  DeNA Co., Ltd.
  kazuhooku@gmail.com

- Kenny Paterson
  Royal Holloway, University of London
  kenny.paterson@rhul.ac.uk

- Alfredo Pironti (co-author of [RFC7627])
  INRIA
  alfredo.pironti@inria.fr

- Andrei Popov
  Microsoft
  andrei.popov@microsoft.com

- Marsh Ray (co-author of [RFC7627])
  Microsoft
  maray@microsoft.com

- Robert Relyea
  Netscape Communications
  relyea@netscape.com

- Kyle Rose
  Akamai Technologies
  krose@krose.org

- Jim Roskind
  Amazon
  jroskind@amazon.com

- Michael Sabin

- Joe Salowey
  Tableau Software
  joe@salowey.net

- Rich Salz
  Akamai
  rsalz@akamai.com

- Sam Scott
  Royal Holloway, University of London
  me@samjs.co.uk

- Dan Simon
  Microsoft, Inc.
  dansimon@microsoft.com

- Brian Sniffen
  Akamai Technologies
  ietf@bts.evenmere.org

- Nick Sullivan
  Cloudflare Inc.
  nick@cloudflare.com

- Bjoern Tackmann
  University of California, San Diego
  btackmann@eng.ucsd.edu

- Tim Taubert
  Mozilla
  ttaubert@mozilla.com

- Martin Thomson
  Mozilla

mt@mozilla.com

- Sean Turner
  sn3rd
  sean@sn3rd.com

- Filippo Valsorda
  Cloudflare Inc.
  filippo@cloudflare.com

- Thyla van der Merwe
  Royal Holloway, University of London
  tjvdmerwe@gmail.com

- Tom Weinstein

- Hoeteck Wee
  Ecole Normale Superieure, Paris
  hoeteck@alum.mit.edu

- David Wong
  NCC Group
  david.wong@nccgroup.trust

- Tim Wright
  Vodafone
  timothy.wright@vodafone.com

- Kazu Yamamoto
  Internet Initiative Japan Inc.
  kazu@iij.ad.jp

## Author's Address

Eric Rescorla
RTFM, Inc.

EMail: ekr@rtfm.com

TLS
Internet-Draft
Obsoletes: 6347 (if approved)
Intended status: Standards Track
Expires: September 14, 2017

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

March 13, 2017

# The Datagram Transport Layer Security (DTLS) Protocol Version 1.3
## draft-rescorla-tls-dtls13-01

## Abstract

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

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

Table of Contents

## 1. Introduction

RFC EDITOR: PLEASE REMOVE THE FOLLOWING PARAGRAPH

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

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

There are applications that utilize UDP 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 was originally defined as a delta from TLS 1.1 and DTLS 1.2 was defined as a series of deltas to TLS 1.2. There is no DTLS 1.1; that version number was skipped in order to harmonize version numbers with TLS. This specification describes the most current version of the DTLS protocol aligning with the efforts around TLS 1.3.

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

## 2. Conventions and Terminology

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

The following terms are used:

- client: The endpoint initiating the TLS connection.

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

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

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

The reader is assumed to be familiar with the TLS 1.3 specification since this document defined as a delta from TLS 1.3.

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

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

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

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

- ’[]’ indicates messages protected using keysderived from traffic_secret_N.

## 3. DTLS Design Rational and Overview

The basic design philosophy of DTLS is to construct "TLS over datagram transport". Datagram transport does not require or provide reliable or in-order delivery of data. The DTLS protocol preserves this property for application data. Applications such as media streaming, Internet telephony, and online gaming use datagram transport for communication due to the delay-sensitive nature of transported data. The behavior of such applications is unchanged when the DTLS protocol is used to secure communication, since the DTLS protocol does not compensate for lost or re-ordered data traffic.

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

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

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

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

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

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

### 3.1. Packet Loss

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

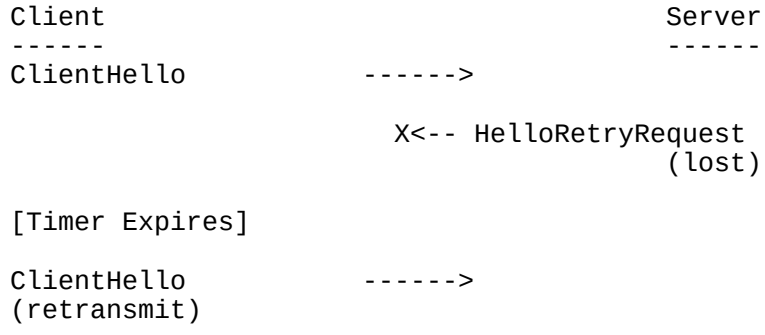



Figure 1: DTLS Retransmission Example.

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

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

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

#### 3.1.1. Reordering

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

#### 3.1.2. Message Size

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

bytes of a handshake message can reassemble the original unfragmented message.

### 3.2. Replay Detection

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

## 4. The DTLS Record Layer

The DTLS record layer is similar to that of TLS 1.3 unless noted otherwise. The only change is the inclusion of an explicit epoch and sequence number in the record. This sequence number allows the recipient to correctly verify the TLS MAC. The DTLS record format is shown below:

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

   struct {
       ContentType opaque_type = 23; /* application_data */
       ProtocolVersion legacy_record_version = {254,253); // DTLSv1.2
       uint16 epoch;                          // DTLS-related field
       uint48 sequence_number;                // DTLS-related field
       uint16 length;
       opaque encrypted_record[length];
} DTLSCiphertext;
```

type: The content type of the record.

legacy_record_version: This field is redundant and it is treated in the same way as specified in the TLS 1.3 specification. The DTLS version 1.2 version number is reused, namely { 254, 253 }. This field is deprecated and MUST be ignored.

epoch: A counter value that is incremented on every cipher state change.

sequence_number: The sequence number for this record.

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

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

## 4.1. Sequence Number Handling

DTLS uses an explicit sequence number, rather than an implicit one, carried in the sequence_number field of the record. Sequence numbers are maintained separately for each epoch, with each sequence_number initially being 0 for each epoch. For instance, if a handshake message from epoch 0 is retransmitted, it might have a sequence number after a message from epoch 1, even if the message from epoch 1 was transmitted first. Note that some care needs to be taken during the handshake to ensure that retransmitted messages use the right epoch and keying material.

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 5.9. In order to ensure that any given sequence/ epoch pair is unique, implementations MUST NOT allow the same epoch value to be reused within two times the TCP maximum segment lifetime.

Note that because DTLS records may be reordered, a record from epoch 1 may be received after epoch 2 has begun. In general, implementations SHOULD discard packets from earlier epochs, but if packet loss causes noticeable problems they 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, not that they attempt to interrogate the MSL that the system TCP stack is using.) Until the handshake has completed, implementations MUST accept packets from the old epoch.

Conversely, it is possible for records that are protected by the newly negotiated context 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), they SHOULD be buffered and processed once the handshake completes. Note that TLS's restrictions on when packets may be sent still apply, and the receiver treats the packets as if they were sent in the right order. In particular, it is still impermissible to send data prior to completion of the first handshake.

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. Transport Layer Mapping

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.

Note that unlike IPsec, DTLS records do not contain any association identifiers. 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.

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

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.

### 4.3. PMTU Issues

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

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

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

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

- The DTLS handshake messages can exceed the PMTU.

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

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

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

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

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

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

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

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

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

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

The final issue is the DTLS handshake protocol. From the perspective of the DTLS record layer, this is merely another upper layer protocol. However, DTLS handshakes occur infrequently and involve only a few round trips; therefore, the handshake protocol PMTU handling places a premium on rapid completion over accurate PMTU

discovery. In order to allow connections under these circumstances, DTLS implementations SHOULD follow the following rules:

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

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

### 4.4. 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.

#### 4.4.1. Anti-Replay

DTLS records contain 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 receiver packet counter for this session MUST be initialized to zero when the 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 life of this session. This SHOULD be the first check applied to a packet after it has been matched to a session, to speed rejection of duplicate records.

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

The "right" edge of the window represents the highest validated sequence number value received on this session. Records that contain sequence numbers lower than the "left" edge of the window are rejected. Packets falling within the window are checked against a list of received packets within the window. An efficient means for

performing this check, based on the use of a bit mask, is described in Section 3.4.3 of [RFC4303].

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

#### 4.4.2. 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.

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. The TLS 1.3 KeyUpdate message is not used in DTLS 1.3 for re-keying.

4. A new ACK message has been added for reliable message delivery of certain handshake messages.

Note that TLS 1.3 already supports a cookie extension, which used to prevent denial-of-service attacks. This DoS prevention mechanism is described in more detail below since UDP-based protocols are more vulnerable to amplification attacks than a connection-oriented

transport like TCP that performs return-routability checks as part of the connection establishment.

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

### 5.1. Denial-of-Service Countermeasures

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

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

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

In order to counter both of these attacks, DTLS borrows the stateless cookie technique used by Photuris [RFC2522] and IKE [RFC5996]. When the client sends its ClientHello message to the server, the server MAY respond with a HelloRetryRequest message. The HelloRetryRequest message, as well as the cookie extension, is defined in TLS 1.3. The HelloRetryRequest message contains a stateless cookie generated using the technique of [RFC2522]. The client MUST retransmit the ClientHello with the cookie added as an extension. The server then verifies the cookie and proceeds with the handshake only if it is valid. This mechanism forces the attacker/client to be able to receive the cookie, which makes DoS attacks with spoofed IP addresses difficult. This mechanism does not provide any defence against DoS attacks mounted from valid IP addresses.

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

The exchange is shown in Figure 2. Note that the figure focuses on the cookie exchange; all other extensions are omitted.

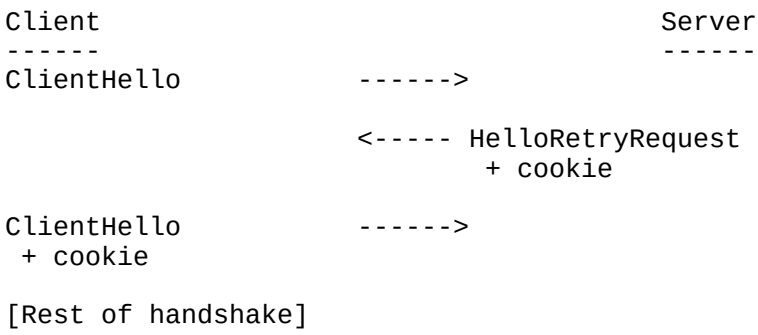



Figure 2: DTLS Exchange with HelloRetryRequest contain the Cookie Extension

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

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

The server SHOULD use information received in the ClientHello to generate its cookie, such as version, random, ciphersuites. The server MUST use the same version number in the HelloRetryRequest that it would use when sending a ServerHello. Upon receipt of the ServerHello, the client MUST verify that the server version values match and MUST terminate the connection with an "illegal_parameter" alert otherwise.

If the HelloRetryRequest message is used, the initial ClientHello and the HelloRetryRequest are included in the calculation of the handshake_messages (for the CertificateVerify message) and verify_data (for the Finished message). However, the computation of the message hash for the HelloRetryRequest is done according to the description in Section 4.4.1 of [I-D.ietf-tls-tls13].

The handshake transcript is not reset with the second ClientHello and a stateless server-cookie implementation requires the transcript of the HelloRetryRequest to be stored in the cookie or the internal

state of the hash algorithm, since only the hash of the transcript is required for the handshake to complete.

When the second ClientHello is received, the server can verify that the cookie is valid and that the client can receive packets at the given IP address.

One potential attack on this scheme is for the attacker to collect a number of cookies from different addresses and then reuse them to attack the server. The server can defend against this attack by changing the secret value frequently, thus invalidating those cookies. If the server wishes that legitimate clients be able to handshake through the transition (e.g., they received a cookie with Secret 1 and then sent the second ClientHello after the server has changed to Secret 2), the server can have a limited window during which it accepts both secrets. [RFC5996] suggests adding a key identifier to cookies to detect this case. An alternative approach is simply to try verifying with both secrets. It is RECOMMENDED that servers implement a key rotation scheme that allows the server to manage keys with overlapping lifetime.

Alternatively, the server can store timestamps in the cookie and reject those cookies that were not generated within a certain amount of time.

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

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

### 5.2. DTLS Handshake Message Format

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

```
  enum {
      hello_request_RESERVED(0),
      client_hello(1),
      server_hello(2),
      hello_verify_request_RESERVED(3),
      new_session_ticket(4),
      end_of_early_data(5),
      hello_retry_request(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_RESERVED(24),
      ack([[TBD RFC Editor -- Proposal: 25]]),
      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 hello_retry_request:   HelloRetryRequest;
          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; /* reserved */
          case ack:                   ACK; /* DTLS-required field */
      } body;
  } Handshake;
```

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.

A DTLS 1.3 MUST NOT use the KeyUpdate message to change keying material used for the protection of traffic data. Instead the epoch field is used, which is explained in Section 5.9.

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

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

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

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

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

random: Same as for TLS 1.3

legacy_session_id: Same as for TLS 1.3

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

cipher_suites: Same as for TLS 1.3

legacy_compression_methods: Same as for TLS 1.3

extensions: Same as for TLS 1.3

The first message each side transmits in each handshake 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.

Here is an example:

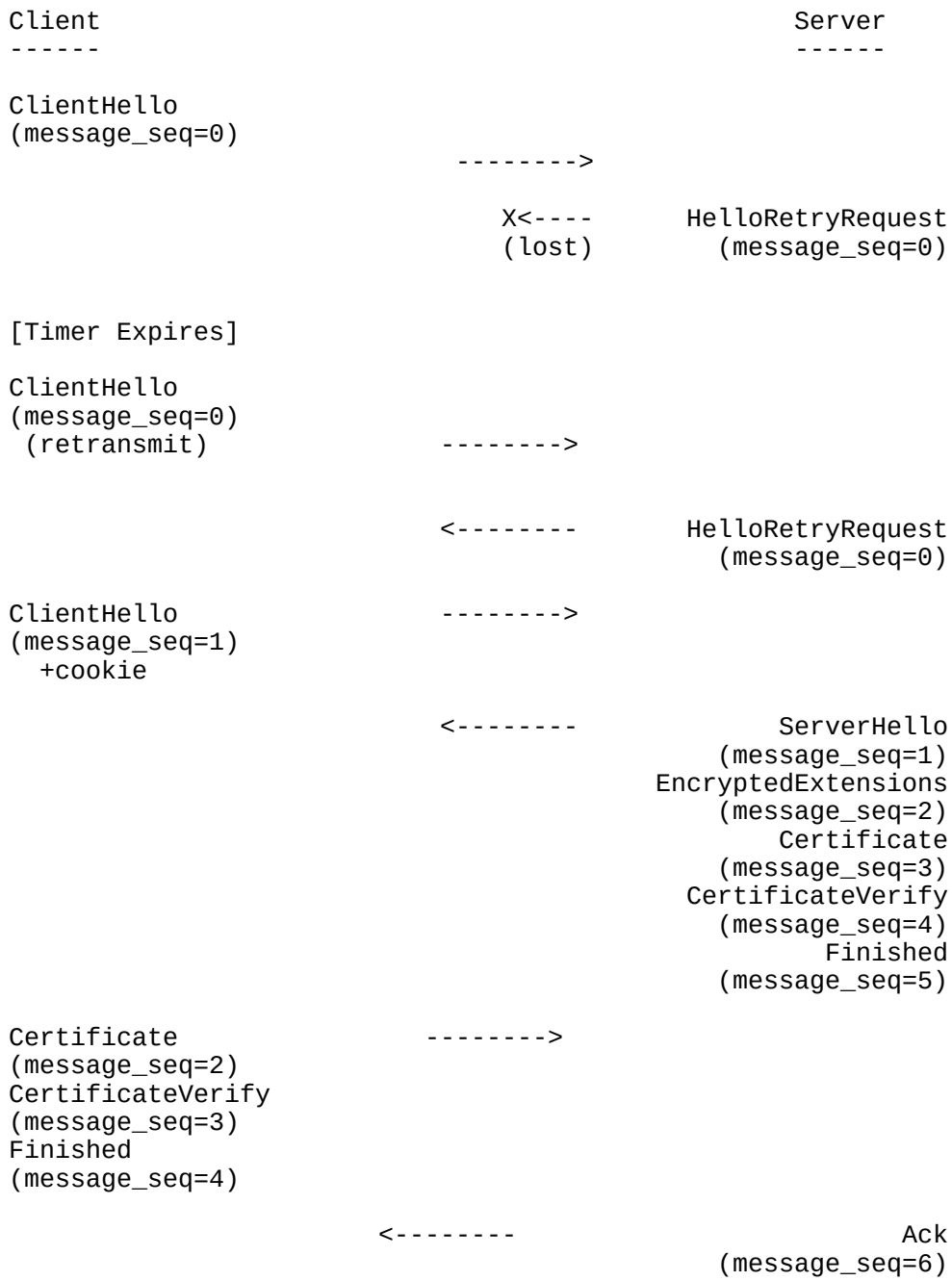



Figure 3: Example DTLS Exchange illustrating Message Loss

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 message is received, if its sequence number 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).

### 5.3. ACK Message

```
struct {} ACK;
```

The ACK handshake message is used by an endpoint to respond to a message where the TLS 1.3 handshake does not foresee such return message. With the use of the ACK message the sender is able to determine whether a transmitted request has been lost and needs to be retransmitted. Since the ACK message does not contain any correlation information the sender MUST only have one such message outstanding at a time.

The ACK message uses a handshake content type and is encrypted under the appropriate application traffic key. [[OPEN ISSUE: It seems odd to have the ACK that responds to CFIN encrypted under the application key. Also, what do you do about ACKs that have to deal with key changes.]]

### 5.4. Handshake Message Fragmentation and Reassembly

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

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

the length field of the original message. An unfragmented message is a degenerate case with fragment_offset=0 and fragment_length=length.

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

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

### 5.5. Timeout and Retransmission

DTLS messages are grouped into a series of message flights, according to the diagrams below. Although each flight of messages may consist of a number of messages, they should be viewed as monolithic for the purpose of timeout and retransmission.


Figure 4: Message Flights for full DTLS Handshake (with Cookie Exchange)

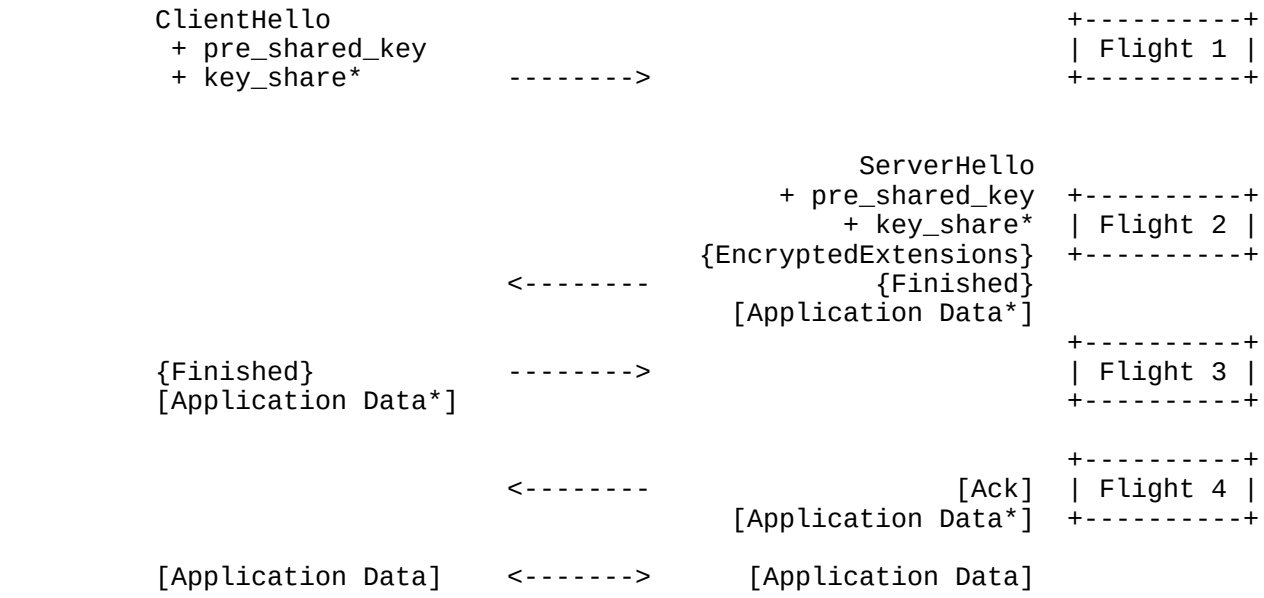



Figure 5: Message Flights for Resumption and PSK Handshake (without Cookie Exchange)

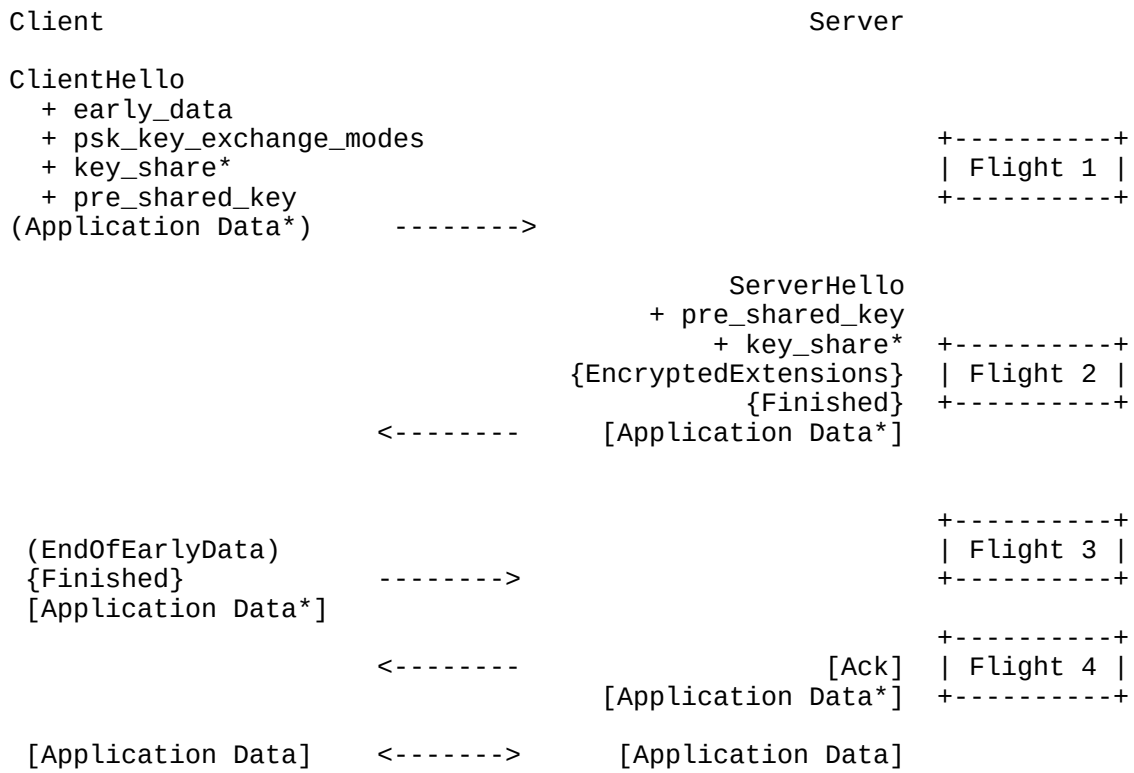



Figure 6: Message Flights for the Zero-RTT Handshake

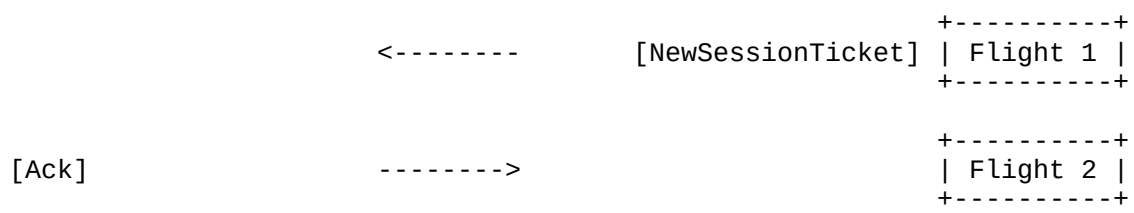



Figure 7: Message Flights for New Session Ticket Message

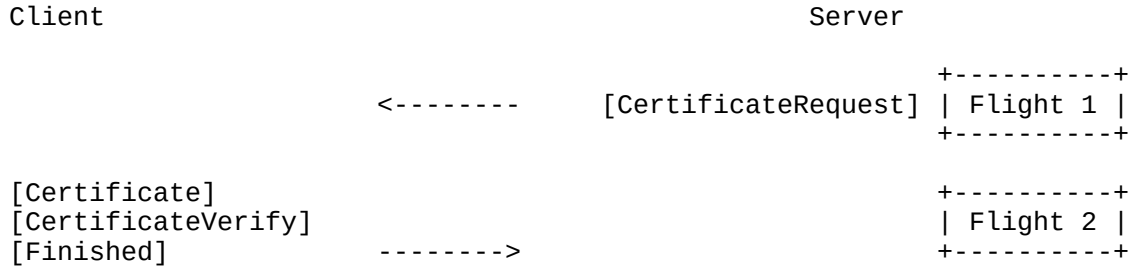



Figure 8: Message Flights for Post-Handshake Authentication (Success)

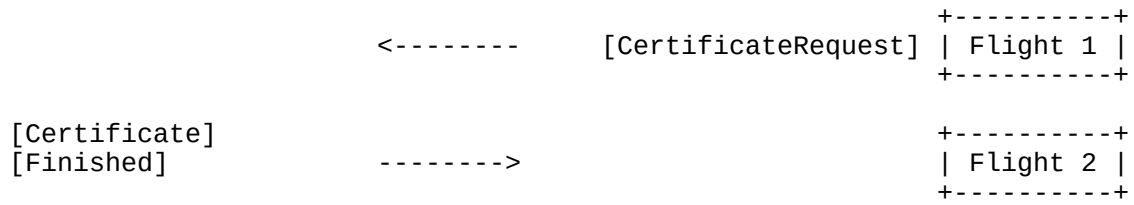



Figure 9: Message Flights for Post-Handshake Authentication (Decline)

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

### 5.5.1. 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.

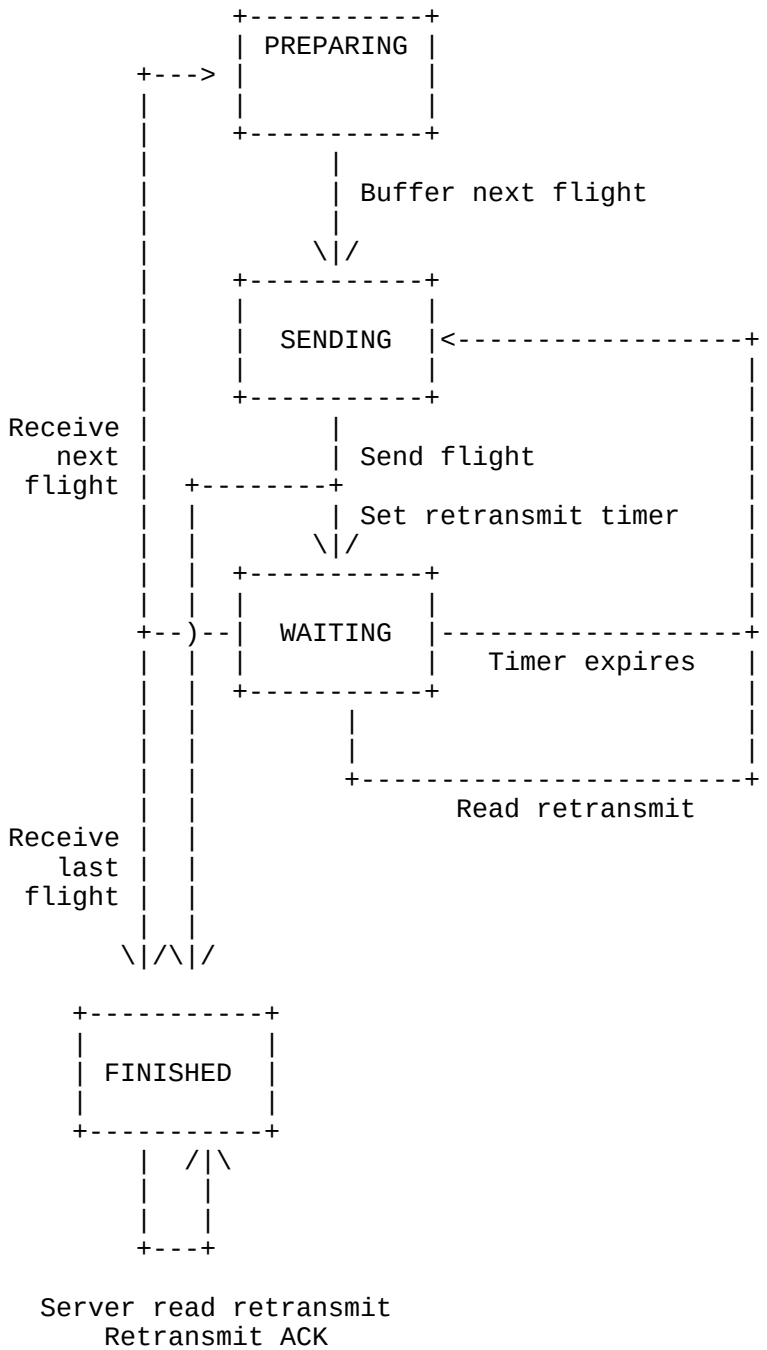



Figure 10: DTLS Timeout and Retransmission State Machine

The state machine has three basic states.

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. 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 three ways to exit the WAITING state:

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

2. The implementation reads a 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.

3. The implementation receives the next flight of messages: if this is the final flight of messages, the implementation transitions to FINISHED. If the implementation needs to send a new flight, it transitions to the PREPARING state. Partial reads (whether partial messages or only some of the messages in the flight) do not cause state transitions or timer resets.

   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.

#### 5.5.2. 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 we recommend a 100 msec timer rather than the 3-second RFC 6298 default in order to improve latency for time-sensitive applications. Because DTLS only uses retransmission for handshake and not dataflow, the effect on congestion should be minimal.

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

### 5.6. CertificateVerify and Finished Messages

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

### 5.7. 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.

## 5.8. 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.

## 5.9. Epoch Values and Rekeying

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

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

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

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

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

- epoch value (3) is used for payloads protected using keys derived from the initial traffic_secret_0. This may include handshake messages, such as post-handshake messages (e.g., a NewSessionTicket message).

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

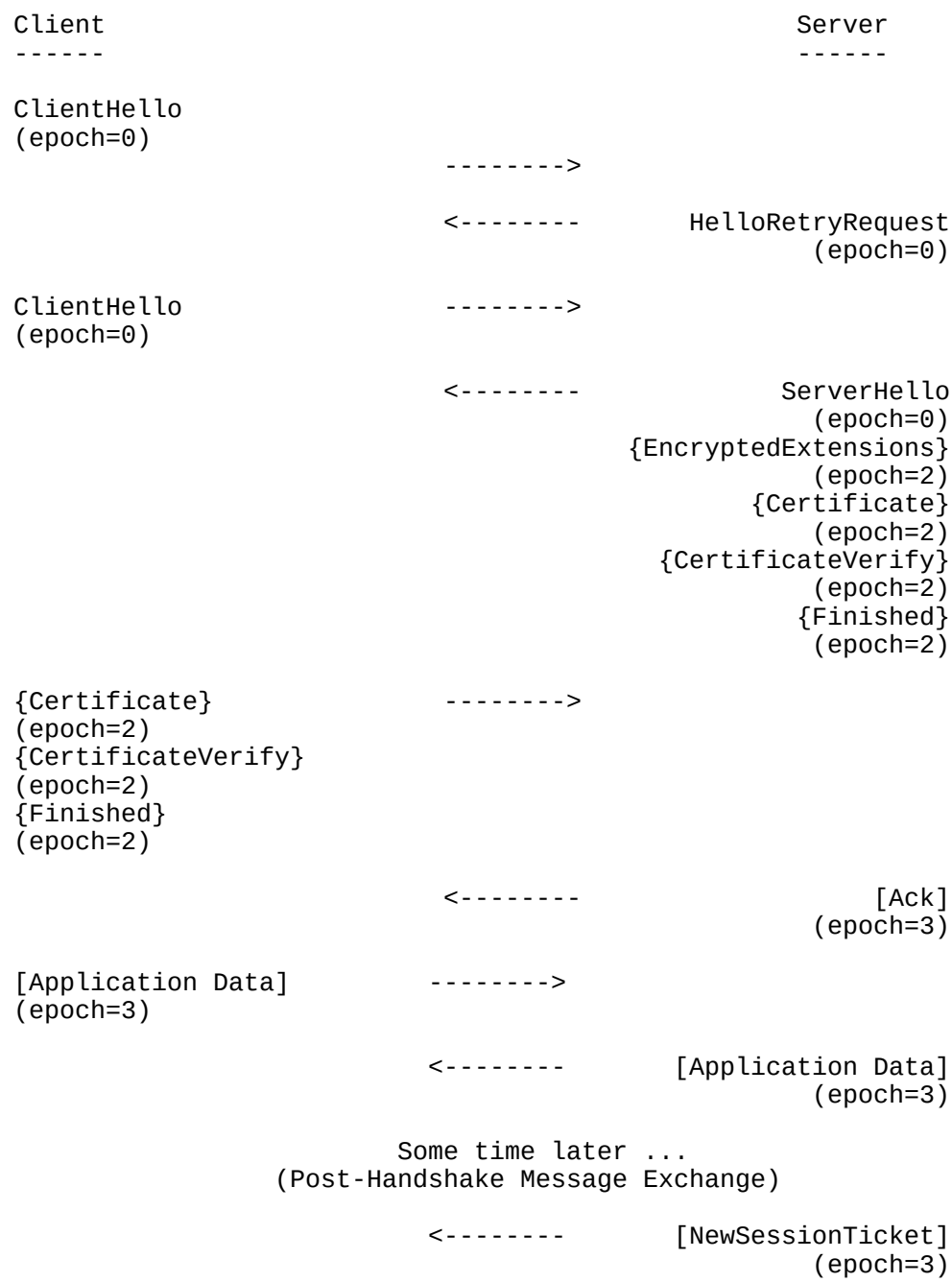
Using these reserved epoch values a receiver knows what cipher state has been used to encrypt and integrity protect a message. Implementations that receive a payload with an epoch value for which no corresponding cipher state can be determined MUST generate a "unexpected_message" alert. For example,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

Increasing the epoch value by a sender (starting with value 4 upwards) corresponds semantically to rekeying using the KeyUpdate message in TLS 1.3. Instead of utilizing an dedicated message in DTLS 1.3 the sender uses an increase in the epoch value to signal rekeying. Hence, a sender that decides to increment the epoch value MUST send all its traffic using the next generation of keys, computed as described in Section 7.2 of [I-D.ietf-tls-tls13]. Upon receiving a payload with such a new epoch value, the receiver MUST update their receiving keys and if they have not already updated their sending state up to or past the then current receiving generation MUST send messages with the new epoch value prior to sending any other messages. For epoch values lower than 4 the key schedule described in Section 7.1 of [I-D.ietf-tls-tls13] is applicable. As a difference to the functionality of the KeyUpdate in TLS 1.3 the sender forces the receiver to increase the epoch value for outgoing data as well.

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

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

Figure 11 illustrates the epoch values in an example DTLS handshake.



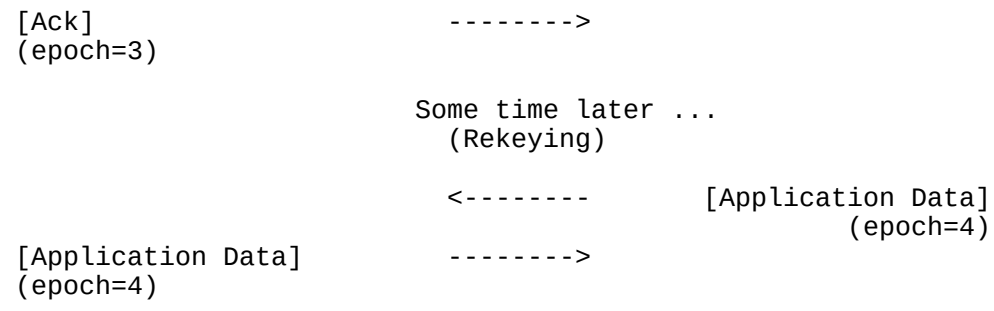



Figure 11: Example DTLS Exchange with Epoch Information

## 6. 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.

## 7. Security Considerations

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

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

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

## 8. Changes to DTLS 1.2

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

- New handshake pattern, which leads to a shorter message exchange
- Support for AEAD-only ciphers
- HelloRetryRequest of TLS 1.3 used instead of HelloVerifyRequest

- More flexible ciphersuite negotiation
- New session resumption mechanism
- PSK authentication redefined
- New key derivation hierarchy utilizing a new key derivation construct
- Removed support for weaker and older cryptographic algorithms
- Improved version negotation

## 9. IANA Considerations

IANA is requested to allocate a new value in the TLS HandshakeType Registry for the ACK message defined in Section 5.3.

## 10. References

### 10.1. Normative References

[I-D.ietf-tls-tls13] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", draft-ietf-tls-tls13-19 (work in progress), March 2017.

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

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

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

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

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

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

## 10.2. Informative References

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

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

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

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

[RFC5996] Kaufman, C., Hoffman, P., Nir, Y., and P. Eronen, "Internet Key Exchange Protocol Version 2 (IKEv2)", RFC 5996, DOI 10.17487/RFC5996, September 2010, <http://www.rfc-editor.org/info/rfc5996>.

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

## 10.3. URIs

[1] mailto:tls@ietf.org

## Appendix A. History

RFC EDITOR: PLEASE REMOVE THE THIS SECTION

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

draft-00

- Initial version using TLS 1.3 as a baseline.

- Use of epoch values instead of KeyUpdate message

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

- Added ACK message

- Text about sequence number handling

## Appendix B. Working Group Information

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

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

## Appendix C. Contributors

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

For this version of the document we would like to thank:

* Ilari Liusvaara
  Independent
  ilariliusvaara@welho.com

* Martin Thomson
  Mozilla
  martin.thomson@gmail.com

Authors' Addresses

Eric Rescorla
RTFM, Inc.

EMail: ekr@rtfm.com

Hannes Tschofenig
ARM Limited

EMail: hannes.tschofenig@arm.com

Nagendra Modadugu
Google, Inc.

EMail: nagendra@cs.stanford.edu

Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: September 9, 2017

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

March 08, 2017

# Delegated Credentials for TLS
# draft-rescorla-tls-subcerts-01

## Abstract

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

## Status of This Memo

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 9, 2017.

Table of Contents



## 1. Introduction

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

- 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. The risk inherent in cross-organizational transactions makes it infeasible to rely on an external CA for such short-lived credentials.

To remove these dependencies, this document proposes a limited delegation mechanism that allows a TLS server operator to issue its own credentials within the scope of a certificate issued by an external CA. Because the above problems do not relate to the CAs inherent function of validating possession of names, it is safe to

make such delegations as long as they only enable the recipient of the delegation to speak for names that the CA has authorized. For clarity, we will refer to the certificate issued by the CA as a "certificate" and the one issued by the operator as a "Delegated credential".

## 2. Solution Overview

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

- A validity interval
- A public key (with its associated algorithm)

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

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

- The client provides an extension in its ClientHello that indicates support for this mechanism.
- The server provides both the certificate chain terminating in its certificate as well as the credential.
- The client uses information in the server’s certificate to verify the signature on the credential and verify that the server is asserting an expected identity.
- The client uses the public key in the credential as the server’s working key for the TLS handshake.

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

It was noted in [XPROT] that certificates in use by servers that support outdated protocols such as SSLv2 can be used to forge signatures for certificates that contain the keyEncipherment KeyUsage ([RFC5280] section 4.2.1.3) In order to prevent this type of cross-protocol attack, we define a new DelegationUsage extension to X.509 which permits use of delegated credentials. Clients MUST NOT accept

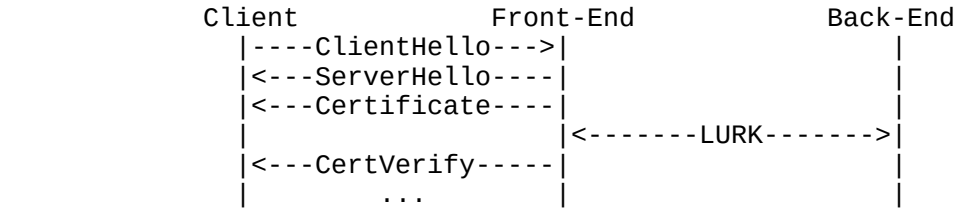
delegated credentials associated with certificates without this extension.

Credentials allow the server to terminate TLS connections on behalf of the certificate owner. If a credential is stolen, there is no mechanism for revoking it without revoking the certificate itself. To limit the exposure of a delegation credential compromise, servers MUST NOT issue credentials with a validity period longer than 7 days. Clients MUST NOT accept credentials with longer validity periods.

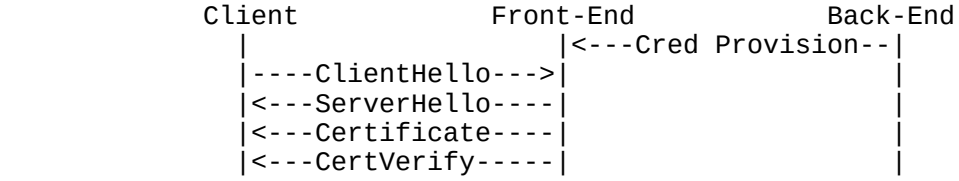
## 3. Related Work

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:




Delegated credentials:




These two classes of mechanism 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.

## 4. Client and Server behavior

This document defines the following extension code point.

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

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

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

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

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

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

- Verify that the current time is within the validity interval of the credential.
- Use the public key in the server's end-entity certificate to verify the signature on the credential.
- Use the public key in the credential to verify the CertificateVerify message provided in the handshake.
- Verify that the certificate has the correct extensions that allow the use of Delegated credentials.

Clients that receive Delegated credentials that are valid for more than 7 days MUST terminate the connection with an "illegal_parameter" alert.

## 5. Delegated Credentials

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

```
struct {
  uint32 validTime;
  opaque publicKey<0..2^24-1>;
} DelegatedCredentialParams;

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

validTime: Relative time in seconds from the beginning of the certificate's notBefore value after which the Delegated Credential is no longer valid.

publicKey: The Delegated Credential's public key which is an encoded SubjectPublicKeyInfo [RFC5280].

scheme: The Signature algorithm and scheme used to sign the Delegated credential.

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

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

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

- A string that consists of octet 32 (0x20) repeated 64 times.
- The context string "TLS, server delegated credentials".
- Big endian serialized 2 bytes ProtocolVersion of the negotiated TLS version, defined by TLS.
- DER encoded X.509 certificate used to sign the DelegatedCredential.
- Big endian serialized 2 byte SignatureScheme scheme.
- The DelegatedCredentialParams structure.

This signature has a few desirable properties:

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

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

Delegated credentials present a better alternative from 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 cert where the properties differ from the leaf certificate. Delegated credentials have very restricted semantics which 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 cert path building process is not cryptographically protected, it is possible that a proxy certificate could be bound to another certificate with the same public key, with different X.509 parameters. Delegated credentials, which rely on a cryptographic binding between the entire certificate and the Delegated credential, cannot.

- Delegated credentials allow signed messages to be bound to specific versions of TLS. This prevents them from being used for other protocols if a service owner allows multiple versions of TLS.

### 5.1. Certificate Requirements

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

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

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

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

## 6. IANA Considerations

## 7. Security Considerations

## 8. References

### 8.1. Normative References

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

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

## 8.2. Informative References

[I-D.ietf-acme-acme]
Barnes, R., Hoffman-Andrews, J., and J. Kasten, "Automatic Certificate Management Environment (ACME)", draft-ietf-acme-acme-05 (work in progress), February 2017.

[I-D.ietf-tls-tls13]
Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", draft-ietf-tls-tls13-18 (work in progress), October 2016.

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

[RFC3820] Tuecke, S., Welch, V., Engert, D., Pearlman, L., and M. Thompson, "Internet X.509 Public Key Infrastructure (PKI) Proxy Certificate Profile", RFC 3820, DOI 10.17487/RFC3820, June 2004, <http://www.rfc-editor.org/info/rfc3820>.

[XPROT] Jager, T., Schwenk, J., and J. Somorovsky, "On the Security of TLS 1.3 and QUIC Against Weaknesses in PKCS#1 v1.5 Encryption", Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security , 2015.

## Authors' Addresses

Richard Barnes
Mozilla

Email: rlb@ipv.sx

Subodh Iyengar
Facebook

Email: subodh@fb.com

Nick Sullivan
Cloudflare

Email: nick@cloudflare.com

Eric Rescorla
RTFM, Inc.

Email: ekr@rtfm.com