Network Working Group
Internet-Draft
Obsoletes: 5077, 5246, 5746 (if approved)
Updates: 4492 (if approved)
Intended status: Standards Track
Expires: April 21, 2016

E. Rescorla
RTFM, Inc.

October 19, 2015

The Transport Layer Security (TLS) Protocol Version 1.3
draft-ietf-tls-tls13-10

Abstract

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

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 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. At the lowest level, layered on top of some reliable transport protocol (e.g., TCP [RFC0793]), is the TLS Record Protocol. The TLS Record Protocol provides connection security that has two basic properties:

- The connection is private. Symmetric cryptography is used for data encryption (e.g., AES [AES]). The keys for this symmetric encryption are generated uniquely for each connection and are based on a secret negotiated by another protocol (such as the TLS Handshake Protocol).

- The connection is reliable. Messages include an authentication tag which protects them against modification.

Note: The TLS Record Protocol can operate in an insecure mode but is generally only used in this mode while another protocol is using the TLS Record Protocol as a transport for negotiating security parameters.

The TLS Record Protocol is used for encapsulation of various higher-level protocols. One such encapsulated protocol, the TLS Handshake Protocol, allows the server and client to authenticate each other and to negotiate an encryption algorithm and cryptographic keys before the application protocol transmits or receives its first byte of data. The TLS Handshake Protocol provides connection security that has three basic properties:

- The peer's identity can be authenticated using asymmetric, or public key, cryptography (e.g., RSA [RSA], ECDSA [ECDSA]). This authentication can be made optional, but is generally required for at least one of the peers.

- The negotiation of a shared secret is secure: the negotiated secret is unavailable to eavesdroppers, and for any authenticated connection the secret cannot be obtained, even by an attacker who can place himself in the middle of the connection.

- The negotiation is reliable: no attacker can modify the negotiation communication without being detected by the parties to the communication.

One advantage of TLS is that it is application protocol independent. Higher-level protocols can layer on top of the TLS protocol transparently. The TLS standard, however, does not specify how protocols add security with TLS; the decisions on 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.

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 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.

1.2. Major Differences from TLS 1.2

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.

2. Goals

The goals of the TLS protocol, in order of priority, are as follows:

1. Cryptographic security: TLS should be used to establish a secure connection between two parties.

2. Interoperability: Independent programmers should be able to develop applications utilizing TLS that can successfully exchange cryptographic parameters without knowledge of one another's code.

3. Extensibility: TLS seeks to provide a framework into which new public key and record protection methods can be incorporated as necessary. This will also accomplish two sub-goals: preventing the need to create a new protocol (and risking the introduction of possible new weaknesses) and avoiding the need to implement an entire new security library.

4. Relative efficiency: Cryptographic operations tend to be highly CPU intensive, particularly public key operations. For this reason, the TLS protocol has incorporated an optional session caching scheme to reduce the number of connections that need to be established from scratch. Additionally, care has been taken to reduce network activity.

3. Goals of This Document

This document and the TLS protocol itself have evolved from the SSL 3.0 Protocol Specification as published by Netscape. The differences between this version and previous versions are significant enough that the various versions of TLS and SSL 3.0 do not interoperate (although each protocol incorporates a mechanism by which an implementation can back down to prior versions). This document is intended primarily for readers who will be implementing the protocol and for those doing cryptographic analysis of it. The specification has been written with this in mind, and it is intended to reflect the needs of those two groups. For that reason, many of the algorithm-dependent data structures and rules are included in the body of the text (as opposed to in an appendix), providing easier access to them.

This document is not intended to supply any details of service definition or of interface definition, although it does cover select areas of policy as they are required for the maintenance of solid security.

4. 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. The syntax draws from several sources in its structure. Although it resembles the programming language "C" in its syntax and XDR [RFC4506] in both its syntax and intent, it would be risky to draw too many parallels. The purpose of this presentation language is to document TLS only; it has no general application beyond that particular goal.

4.1. 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.

4.2. 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.

4.3. 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 4.4). On the other hand, 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 even multiple of the length of a single element (for example, 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 */
```

4.4. 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 4.1 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.

Note that in some cases (e.g., DH parameters) it is necessary to represent integers as opaque vectors. In such cases, they are represented as unsigned integers (i.e., leading zero octets are not required even if the most significant bit is set).

4.5. Enumerateds

An additional sparse data type is available called enum. A field of type enum can only assume the values declared in the definition. 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;
```

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.

```
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 */
```

For enumerateds that are never converted to external representation, the numerical information may be omitted.

```
enum { low, medium, high } Amount;
```

4.6. 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.

4.6.1. 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. There must be a case arm for every element of the enumeration declared in the select. Case arms have limited fall-through: if two case arms follow in immediate succession with no fields in between, then they both contain the same fields. Thus, in the example below, "orange" and "banana" both contain V2. Note that this is a new piece of syntax in TLS 1.2.

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 e3: case e4: Te3;
         ....
         case en: Ten;
     } [[fv]];
} [[Tv]];
```

For example:

```
enum { apple, orange, banana } VariantTag;

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

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

struct {
    select (VariantTag) { /* value of selector is implicit */
        case apple:
          V1;   /* VariantBody, tag = apple */
        case orange:
        case banana:
          V2;   /* VariantBody, tag = orange or banana */
    } variant_body;       /* optional label on variant */
} VariantRecord;
```

4.7. Constants

Typed constants can be defined for purposes of specification by declaring a symbol of the desired type and assigning values to it.

Under-specified types (opaque, variable-length vectors, and structures that contain opaque) cannot be assigned values. No fields of a multi-element structure or vector may be elided.

For example:

```
struct {
    uint8 f1;
    uint8 f2;
} Example1;

Example1 ex1 = {1, 4};  /* assigns f1 = 1, f2 = 4 */
```

4.8. Primitive Types

The following common primitive types are defined and used subsequently:

```
enum { false(0), true(1) } Boolean;
```

4.9. Cryptographic Attributes

The two cryptographic operations -- digital signing, and authenticated encryption with additional data (AEAD) -- are designated digitally-signed, and aead-ciphered, respectively. A field's cryptographic processing is specified by prepending an appropriate key word designation before the field's type specification. Cryptographic keys are implied by the current session state (see Section 5.1).

4.9.1. Digital Signing

A digitally-signed element is encoded as a struct DigitallySigned:

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

The algorithm field specifies the algorithm used (see Section 6.3.2.1 for the definition of this field). Note that the algorithm field was introduced in TLS 1.2, and is not in earlier versions. The signature is a digital signature using those algorithms over the contents of the element. The contents themselves do not appear on the wire but are simply calculated. The length of the signature is specified by the signing algorithm and key.

In previous versions of TLS, the ServerKeyExchange format meant that attackers can obtain a signature of a message with a chosen, 32-byte prefix. Because TLS 1.3 servers are likely to also implement prior versions, the contents of the element always start with 64 bytes of octet 32 in order to clear that chosen-prefix.

Following that padding is a NUL-terminated context string in order to disambiguate signatures for different purposes. The context string will be specified whenever a digitally-signed element is used.

Finally, the specified contents of the digitally-signed structure follow the NUL at the end of the context string. (See the example at the end of this section.)

In RSA signing, the opaque vector contains the signature generated using the RSASSA-PSS signature scheme defined in [RFC3447] with MGF1. The digest used in the mask generation function MUST be the same as the digest which is being signed (i.e., what appears in algorithm.signature). The length of the salt MUST be equal to the octet length of the digest output. Note that previous versions of TLS used RSASSA-PKCS1-v1_5, not RSASSA-PSS.

All ECDSA computations MUST be performed according to ANSI X9.62 [X962] or its successors. Data to be signed/verified is hashed, and the result run directly through the ECDSA algorithm with no additional hashing. The SignatureAndHashAlgorithm parameter in the DigitallySigned object indicates the digest algorithm which was used in the signature.

In the following example

```
struct {
    uint8 field1;
    uint8 field2;
    digitally-signed opaque {
      uint8 field3<0..255>;
      uint8 field4;
    };
} UserType;
```

Assume that the context string for the signature was specified as "Example". The input for the signature/hash algorithm would be:

```
2020202020202020202020202020202020202020202020202020202020202020
2020202020202020202020202020202020202020202020202020202020202020
4578616d706c6500
```

followed by the encoding of the inner struct (field3 and field4).

The length of the structure, in bytes, would be equal to two bytes for field1 and field2, plus two bytes for the signature and hash algorithm, plus two bytes for the length of the signature, plus the length of the output of the signing algorithm. The length of the signature is known because the algorithm and key used for the signing are known prior to encoding or decoding this structure.

4.9.2. Authenticated Encryption with Additional Data (AEAD)

In AEAD encryption, the plaintext is simultaneously encrypted and integrity protected. The input may be of any length, and aead-ciphered output is generally larger than the input in order to accommodate the integrity check value.

5. The TLS Record Protocol

The TLS Record Protocol is a layered protocol. At each layer, messages may include fields for length, description, and content. 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 decrypted and verified, reassembled, and then delivered to higher-level clients.

Three protocols that use the TLS Record Protocol are described in this document: the TLS Handshake Protocol, the Alert Protocol, and the application data protocol. In order to allow extension of the TLS protocol, additional record content types can be supported by the TLS Record Protocol. New record content type values are assigned by IANA in the TLS Content Type Registry as described in Section 11.

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 send an "unexpected_message" alert.

Any protocol designed for use over TLS must be carefully designed to deal with all possible attacks against it. As a practical matter, this means that the protocol designer must be aware of what security properties TLS does and does not provide and cannot safely rely on the latter.

Note in particular that the length of a record or absence of traffic itself is not protected by encryption unless the sender uses the supplied padding mechanism - see Section 5.2.3 for more details.

5.1. Connection States

[[TODO: I plan to totally rewrite or remove this. IT seems like just cruft.]]

A TLS connection state is the operating environment of the TLS Record Protocol. It specifies a record protection algorithm and its parameters as well as the record protection keys and IVs for the connection in both the read and the write directions. The security parameters are set by the TLS Handshake Protocol, which also determines when new cryptographic keys are installed and used for record protection. The initial current state always specifies that records are not protected.

The security parameters for a TLS Connection read and write state are set by providing the following values:

connection end
: Whether this entity is considered the "client" or the "server" in this connection.

Hash algorithm

An algorithm used to generate keys from the appropriate secret (see Section 7.1 and Section 7.2).

record protection algorithm
The algorithm to be used for record protection. This algorithm must be of the AEAD type and thus provides integrity and confidentiality as a single primitive. This specification includes the key size of this algorithm and of the nonce for the AEAD algorithm.

master secret
A 48-byte secret shared between the two peers in the connection and used to generate keys for protecting data.

client random
A 32-byte value provided by the client.

server random
A 32-byte value provided by the server.

These parameters are defined in the presentation language as:

```
enum { server, client } ConnectionEnd;

enum { tls_kdf_sha256, tls_kdf_sha384 } KDFAlgorithm;

enum { aes_gcm } RecordProtAlgorithm;

/* The algorithms specified in KDFAlgorithm and
   RecordProtAlgorithm may be added to. */

struct {
    ConnectionEnd          entity;
    KDFAlgorithm           kdf_algorithm;
    RecordProtAlgorithm    record_prot_algorithm;
    uint8                  enc_key_length;
    uint8                  iv_length;
    opaque                 hs_master_secret[48];
    opaque                 master_secret[48];
    opaque                 client_random[32];
    opaque                 server_random[32];
} SecurityParameters;
```

[TODO: update this to handle new key hierarchy.]

The connection state will use the security parameters to generate the following four items:

client write key
server write key
client write iv
server write iv

The client write parameters are used by the server when receiving and processing records and vice versa. The algorithm used for generating these items from the security parameters is described in Section 7.2.

Once the security parameters have been set and the keys have been generated, the connection states can be instantiated by making them the current states. These current states MUST be updated for each record processed. Each connection state includes the following elements:

cipher state
The current state of the encryption algorithm. This will consist of the scheduled key for that connection.

sequence number
Each connection state contains a sequence number, which is maintained separately for read and write states. The sequence number is set to zero at the beginning of a connection and incremented by one thereafter. Sequence numbers are of type uint64 and MUST NOT exceed 2^64-1. Sequence numbers do not wrap. If a TLS implementation would need to wrap a sequence number, it MUST terminate the connection. A sequence number is incremented after each record: specifically, the first record transmitted under a particular connection state MUST use sequence number 0. NOTE: This is a change from previous versions of TLS, where sequence numbers were reset whenever keys were changed.

5.2. Record Layer

The TLS record layer receives uninterpreted data from higher layers in non-empty blocks of arbitrary size.

5.2.1. Fragmentation

The record layer fragments information blocks into TLSPlaintext records carrying data in chunks of 2^14 bytes or less. Client message boundaries are not preserved in the record layer (i.e., multiple client messages of the same ContentType MAY be coalesced into a single TLSPlaintext record, or a single message MAY be fragmented across several records). Alert messages Section 6.1 MUST NOT be fragmented across records.

```
    struct {
        uint8 major;
        uint8 minor;
    } ProtocolVersion;

    enum {
        alert(21),
        handshake(22),
        application_data(23),
        early_handshake(25),
        (255)
    } ContentType;

    struct {
        ContentType type;
        ProtocolVersion record_version = { 3, 1 };    /* TLS v1.x */
        uint16 length;
        opaque fragment[TLSPlaintext.length];
    } TLSPlaintext;
```

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

record_version
: The protocol version the current record is compatible with. This value MUST be set to { 3, 1 } for all records. This field is deprecated and MUST be ignored for all purposes.

length
: The length (in bytes) of the following TLSPlaintext.fragment. The length MUST NOT exceed 2^14.

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

This document describes TLS Version 1.3, which uses the version { 3, 4 }. The version value 3.4 is historical, deriving from the use of { 3, 1 } for TLS 1.0 and { 3, 0 } for SSL 3.0. In order to maximize backwards compatibility, the record layer version identifies as simply TLS 1.0. Endpoints supporting other versions negotiate the version to use by following the procedure and requirements in Appendix C.

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

fragments of Application data MAY be sent as they are potentially useful as a traffic analysis countermeasure.

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.

5.2.2. 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 a unified encryption and authentication operation which turns plaintext into authenticated ciphertext and back again.

AEAD ciphers 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.

```
struct {
    ContentType opaque_type = application_data(23); /* see fragment.type */
    ProtocolVersion record_version = { 3, 1 };    /* TLS v1.x */
    uint16 length;
    aead-ciphered struct {
       opaque content[TLSPlaintext.length];
       ContentType type;
       uint8 zeros[length_of_padding];
    } fragment;
} TLSCiphertext;
```

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 used to parsing previous versions of TLS. The actual content type of the record is found in fragment.type after decryption.

record_version
: The record_version field is identical to TLSPlaintext.record_version and is always { 3, 1 }. 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.fragment. The length MUST NOT exceed 2^14 + 256. An endpoint that receives a record that exceeds this length MUST generate a fatal "record_overflow" alert.

fragment.content
The cleartext of TLSPlaintext.fragment.

fragment.type
The actual content type of the record.

fragment.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 5.2.3 for more details.

fragment
The AEAD encrypted form of TLSPlaintext.fragment + TLSPlaintext.type + zeros, where "+" denotes concatenation.

The length of the per-record nonce (iv_length) is set to max(8 bytes, 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 padded to the left with zeroes 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.

The plaintext is the concatenation of TLSPlaintext.fragment and TLSPlaintext.type.

The additional authenticated data, which we denote as additional_data, is defined as follows:

additional_data = seq_num + TLSPlaintext.record_version

where "+" denotes concatenation.

Note: In versions of TLS prior to 1.3, the additional_data included a length field. This presents a problem for cipher constructions with data-dependent padding (such as CBC). TLS 1.3 removes the length field and relies on the AEAD cipher to provide integrity for the length of the data.

The AEAD output consists of the ciphertext output by the AEAD encryption operation. The length of the plaintext is greater than TLSPlaintext.length due to the inclusion of TLSPlaintext.type and however much padding is supplied by the sender. The length of aead_output will generally be larger than the plaintext, but by an amount that varies with the AEAD cipher. 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 of fragment,
                             additional_data)
```

In order to decrypt and verify, the cipher takes as input the key, nonce, the "additional_data", 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 fragment = AEAD-Decrypt(write_key, nonce,
                                        AEADEncrypted,
                                        additional_data)
```

If the decryption fails, a fatal "bad_record_mac" alert MUST be generated.

An AEAD cipher MUST NOT produce an expansion of greater than 255 bytes. An endpoint that receives a record from its peer with TLSCipherText.length larger than 2^14 + 256 octets MUST generate a fatal "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.

5.2.3. 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 fragment.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 fragment.content.

The padding sent is automatically verified by the record protection mechanism: Upon successful decryption of a TLSCiphertext.fragment, 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.

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 should treat the record as having an unexpected ContentType, sending an "unexpected_message" alert.

The presence of padding does not change the overall record size limitations - the full fragment plaintext may not exceed 2^14 octets.

Versions of TLS prior to 1.3 had limited support for padding. 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.

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 permits 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.

6. The TLS Handshaking Protocols

TLS has three subprotocols that are used to allow peers to agree upon security parameters for the record layer, to authenticate themselves, to instantiate negotiated security parameters, and to report error conditions to each other.

The TLS Handshake Protocol is responsible for negotiating a session, which consists of the following items:

peer certificate
: X509v3 [RFC5280] certificate of the peer. This element of the state may be null.

cipher spec
: Specifies the authentication and key establishment algorithms, the hash for use with HKDF to generate keying material, and the record protection algorithm (See Appendix A.5 for formal definition.)

resumption master secret
: a secret shared between the client and server that can be used as a PSK in future connections.

These items are then used to create security parameters for use by the record layer when protecting application data. Many connections can be instantiated using the same session using a PSK established in an initial handshake.

6.1. Alert Protocol

One of the content types supported by the TLS record layer is the alert type. Alert messages convey the severity of the message (warning or fatal) and a description of the alert. Alert messages with a level of fatal result in the immediate termination of the connection. In this case, other connections corresponding to the session may continue, but the session identifier MUST be invalidated, preventing the failed session from being used to establish new connections. Like other messages, alert messages are encrypted as specified by the current connection state.

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

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

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

6.1.1. Closure Alerts

The client and the server must share knowledge that the connection is ending in order to avoid a truncation attack. Failure to properly close a connection does not prohibit a session from being resumed.

close_notify
: This message 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 message 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 is 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.

6.1.2. 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 immediately close the connection. Servers and clients MUST forget any session-identifiers, keys, and secrets associated with a failed connection. Thus, any connection terminated with a fatal alert MUST NOT be resumed.

Whenever an implementation encounters a condition which is defined as a fatal alert, it MUST send the appropriate alert prior to closing

the connection. For all errors where an alert level is not explicitly specified, the sending party MAY determine at its discretion whether to treat this as a fatal error or not. If the implementation chooses to send an alert but intends to close the connection immediately afterwards, it MUST send that alert at the fatal alert level.

If an alert with a level of warning is sent and received, generally the connection can continue normally. If the receiving party decides not to proceed with the connection (e.g., after having received a "user_canceled" alert that it is not willing to accept), it SHOULD send a fatal alert to terminate the connection. Given this, the sending peer cannot, in general, know how the receiving party will behave. Therefore, warning alerts are not very useful when the sending party wants to continue the connection, and thus are sometimes omitted. For example, if a party decides to accept an expired certificate (perhaps after confirming this with the user) and wants to continue the connection, it would not generally send a "certificate_expired" alert.

The following error alerts are defined:

unexpected_message
: An inappropriate message was received. This alert is always fatal and 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, this alert is used for all deprotection failures. This alert is always fatal and 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 is always fatal and should never be observed in communication between proper implementations (except when messages were corrupted in the network).

handshake_failure
: Reception of a "handshake_failure" alert message indicates that the sender was unable to negotiate an acceptable set of security parameters given the options available. This alert is always fatal.

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 out of range or inconsistent with other fields. This alert is always fatal.

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 couldn't be matched with a known, trusted CA. This alert is always fatal.

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

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 always fatal and should never be observed in communication between proper implementations (except when messages were corrupted in the network).

decrypt_error
A handshake cryptographic operation failed, including being unable to correctly verify a signature or validate a Finished message. This alert is always fatal.

protocol_version
The protocol version the peer has attempted to negotiate is recognized but not supported. (For example, old protocol versions

might be avoided for security reasons.) This alert is always fatal.

insufficient_security
Returned instead of "handshake_failure" when a negotiation has failed specifically because the server requires ciphers more secure than those supported by the client. This alert is always fatal.

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. This alert is always fatal.

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

missing_extension
Sent by endpoints that receive a hello message not containing an extension that is mandatory to send for the offered TLS version. This message is always fatal. [[TODO: IANA Considerations.]]

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

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

unrecognized_name
Sent by servers when no server exists identified by the name provided by the client via the "server_name" extension [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 [RFC6066]. This alert is always fatal.

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 [RFC6066]. This alert is always fatal.

unknown_psk_identity
Sent by servers when a PSK cipher suite is selected 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. [[TODO: This doesn't really make sense with the current PSK negotiation scheme where the client provides multiple PSKs in flight 1. https://github.com/tlswg/tls13-spec/issues/230]]

New Alert values are assigned by IANA as described in Section 11.

6.2. Handshake Protocol Overview

The cryptographic parameters of the session state are produced by the TLS Handshake Protocol, which operates on top of the TLS record layer. When a TLS client and server first start communicating, they agree on a protocol version, select cryptographic algorithms, optionally authenticate each other, and establish shared secret keying material.

TLS supports three basic key exchange modes:

- Diffie-Hellman (of both the finite field and elliptic curve varieties).

- A pre-shared symmetric key (PSK)

- A combination of a symmetric key and Diffie-Hellman

Which mode is used depends on the negotiated cipher suite. Conceptually, the handshake establishes two secrets which are used to derive all the keys.

Ephemeral Secret (ES): A secret which is derived from fresh (EC)DHE shares for this connection. Keying material derived from ES is intended to be forward secure (with the exception of pre-shared key only modes).

Static Secret (SS): A secret which may be derived from static or semi-static keying material, such as a pre-shared key or the server's semi-static (EC)DH share.

In some cases, as with the DH handshake shown in Figure 1, these secrets are the same, but having both allows for a uniform key derivation scheme for all cipher modes.

The basic TLS Handshake for DH is shown in Figure 1:

+ Indicates extensions sent in the previously noted message.

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

{} Indicates messages protected using keys derived from the ephemeral secret.

[] Indicates messages protected using keys derived from the master secret.

Figure 1: Message flow for full TLS Handshake

The first message sent by the client is the ClientHello Section 6.3.1.1 which contains a random nonce (ClientHello.random), its offered protocol version, cipher suite, and extensions, and one or more Diffie-Hellman key shares in the KeyShare extension Section 6.3.2.3.

The server processes the ClientHello and determines the appropriate cryptographic parameters for the connection. It then responds with the following messages:

ServerHello
indicates the negotiated connection parameters. [Section 6.3.1.2] If DH is in use, this will contain a KeyShare extension with the server's ephemeral Diffie-Hellman share which MUST be in the same group as one of the shares offered by the client. The server's KeyShare and the client's KeyShare corresponding to the negotiated key exchange are used together to derive the Static Secret and

Ephemeral Secret (in this mode they are the same). [Section 6.3.2.3]

ServerConfiguration
: supplies a configuration for 0-RTT handshakes (see Section 6.2.2). [Section 6.3.6]

EncryptedExtensions
: responses to any extensions which are not required in order to determine the cryptographic parameters. [Section 6.3.3]

Certificate
: the server certificate. This message will be omitted if the server is not authenticating via a certificates. [Section 6.3.4]

CertificateRequest
: if certificate-based client authentication is desired, the desired parameters for that certificate. This message will be omitted if client authentication is not desired. [[OPEN ISSUE: See https://github.com/tlswg/tls13-spec/issues/184]]. [Section 6.3.5]

CertificateVerify
: a signature over the entire handshake using the public key in the Certificate message. This message will be omitted if the server is not authenticating via a certificate. [Section 6.3.7]

Finished
: a MAC over the entire handshake computed using the Static Secret. This message provides key confirmation and In some modes (see Section 6.2.2) it also authenticates the handshake using the the Static Secret. [Section 6.3.8]

Upon receiving the server's messages, the client responds with his final flight of messages:

Certificate
: the client's certificate. This message will be omitted if the client is not authenticating via a certificates. [Section 6.3.9]

CertificateVerify
: a signature over the entire handshake using the private key corresponding to the public key in the Certificate message. This message will be omitted if the client is not authenticating via a certificate. [Section 6.3.10]

Finished
: a MAC over the entire handshake computed using the Static Secret and providing key confirmation. [Section 6.3.8]

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. If client authentication is requested, the server MUST NOT send application data before it receives the client's Finished.

[[TODO: Move this elsewhere? Note that higher layers should not be overly reliant on whether TLS always negotiates the strongest possible connection between two endpoints. There are a number of ways in which a man-in-the-middle attacker can attempt to make two entities drop down to the least secure method they support (i.e., perform a downgrade attack). The TLS protocol has been designed to minimize this risk, but there are still attacks available: for example, an attacker could block access to the port a secure service runs on, or attempt to get the peers to negotiate an unauthenticated connection. The fundamental rule is that higher levels must be cognizant of what their security requirements are and never transmit information over a channel less secure than what they require. The TLS protocol is secure in that any cipher suite offers its promised level of security: if you negotiate AES-GCM [GCM] with a 255-bit ECDHE key exchange with a host whose certificate chain you have verified, you can expect that to be reasonably "secure" against algorithmic attacks, at least in the year 2015.]]

6.2.1. Incorrect DHE Share

If the client has not provided an appropriate KeyShare extension (e.g. it includes only DHE or ECDHE groups unacceptable or unsupported by the server), the server corrects the mismatch with a HelloRetryRequest and the client will need to restart the handshake with an appropriate KeyShare extension, as shown in Figure 2:

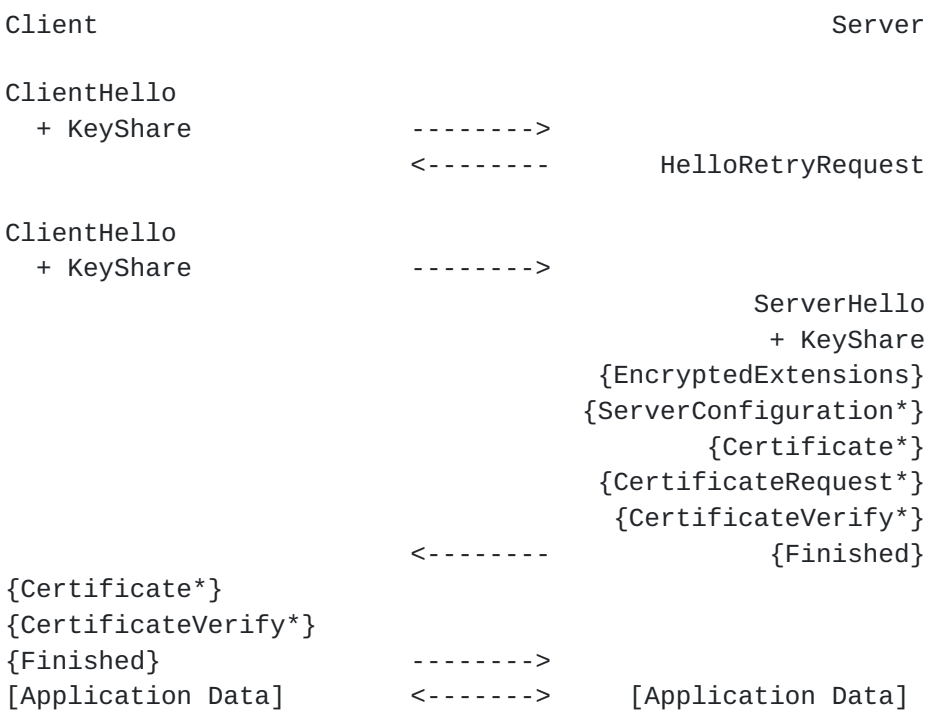

Figure 2: Message flow for a full handshake with mismatched parameters

[[OPEN ISSUE: Should we restart the handshake hash? https://github.com/tlswg/tls13-spec/issues/104.]] [[OPEN ISSUE: We need to make sure that this flow doesn't introduce downgrade issues. Potential options include continuing the handshake hashes (as long as clients don't change their opinion of the server's capabilities with aborted handshakes) and requiring the client to send the same ClientHello (as is currently done) and then checking you get the same negotiated parameters.]]

If no common cryptographic parameters can be negotiated, the server will send a "handshake_failure" or "insufficient_security" fatal alert (see Section 6.1).

TLS also allows several optimized variants of the basic handshake, as described below.

6.2.2. Zero-RTT Exchange

TLS 1.3 supports a "0-RTT" mode in which the client can send application data as well as its Certificate and CertificateVerify (if client authentication is requested) on its first flight, thus reducing handshake latency. In order to enable this functionality, the server provides a ServerConfiguration message containing a long-

term (EC)DH share. On future connections to the same server, the client can use that share to encrypt the first-flight data.

Figure 3: Message flow for a zero round trip handshake

Note: because sequence numbers continue to increment between the initial (early) application data and the application data sent after the handshake has completed, an attacker cannot remove early application data messages.

IMPORTANT NOTE: The security properties for 0-RTT data (regardless of the cipher suite) are weaker than those for other kinds of TLS data. Specifically:

1. This data is not forward secure, because it is encrypted solely with the server's semi-static (EC)DH share.

2. There are no guarantees of non-replay between connections. Unless the server takes special measures outside those provided by TLS (See Section 6.3.2.5.2), the server has no guarantee that the same 0-RTT data was not transmitted on multiple 0-RTT connections. 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 also cannot be sent as if it were ordinary TLS data.

3. If the server key is compromised, and client authentication is used, then the attacker can impersonate the client to the server (as it knows the traffic key).

6.2.3. Resumption and PSK

Finally, TLS provides a pre-shared key (PSK) mode which allows a client and server who share an existing secret (e.g., a key established out of band) to establish a connection authenticated by that key. PSKs can also be established in a previous session and then reused ("session resumption"). Once a handshake has completed, the server can send the client a PSK identity which corresponds to a key derived from the initial handshake (See Section 6.3.11). 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 original connection is tied to the new connection. In TLS 1.2 and below, this functionality was provided by "session resumption" and "session tickets" [RFC5077]. Both mechanisms are obsoleted in TLS 1.3.

PSK cipher suites can either use PSK in combination with an (EC)DHE exchange in order to provide forward secrecy in combination with shared keys, or can use PSKs alone, at the cost of losing forward secrecy.

Figure 4 shows a pair of handshakes in which the first establishes a PSK and the second uses it:

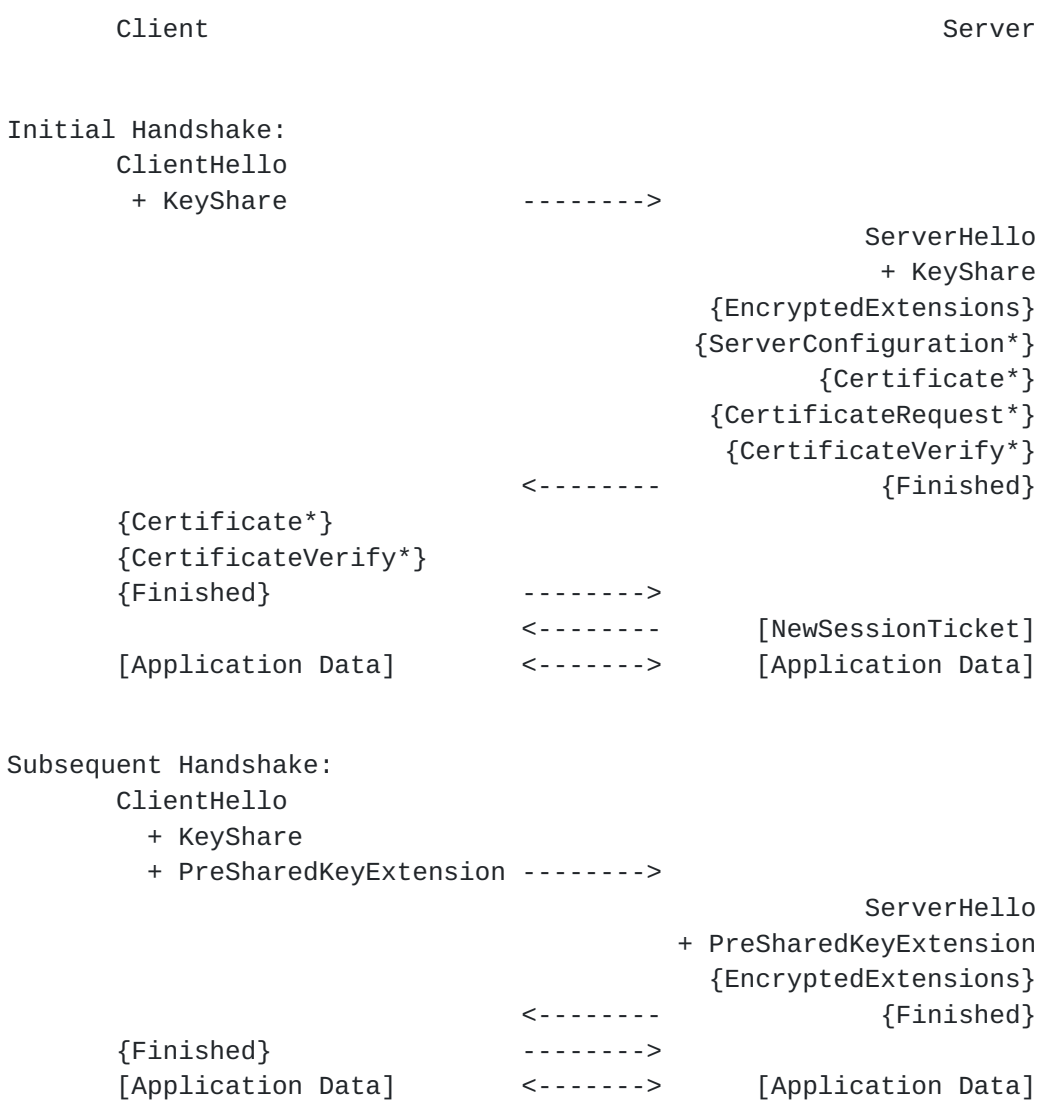

Figure 4: Message flow for resumption and PSK

As the server is authenticating via a PSK, it does not send a Certificate or a CertificateVerify. PSK-based resumption cannot be used to provide a new ServerConfiguration. Note that the client supplies a KeyShare to the server as well, which allows the server to decline resumption and fall back to a full handshake.

The contents and significance of each message will be presented in detail in the following sections.

6.3. Handshake Protocol

The TLS Handshake Protocol is one of the defined higher-level clients of the TLS Record Protocol. This protocol is used to negotiate the secure attributes of a session. 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 session state.

```
enum {
    client_hello(1),
    server_hello(2),
    session_ticket(4),
    hello_retry_request(6),
    encrypted_extensions(8),
    certificate(11),
    certificate_request(13),
    certificate_verify(15),
    server_configuration(17),
    finished(20),
    (255)
} HandshakeType;

struct {
    HandshakeType msg_type;    /* handshake type */
    uint24 length;             /* bytes in message */
    select (HandshakeType) {
        case client_hello:        ClientHello;
        case server_hello:        ServerHello;
        case hello_retry_request: HelloRetryRequest;
        case encrypted_extensions: EncryptedExtensions;
        case server_configuration:ServerConfiguration;
        case certificate:         Certificate;
        case certificate_request: CertificateRequest;
        case certificate_verify:  CertificateVerify;
        case finished:            Finished;
        case session_ticket:      NewSessionTicket;
    } body;
} Handshake;
```

The TLS Handshake Protocol messages are presented below in the order they MUST be sent; sending handshake messages in an unexpected order results in an "unexpected_message" fatal error. Unneeded handshake messages can be omitted, however.

New handshake message types are assigned by IANA as described in Section 11.

6.3.1. Hello Messages

The hello phase messages are used to exchange security enhancement capabilities between the client and server. When a new session

begins, the record layer's connection state AEAD algorithm is initialized to NULL_NULL.

6.3.1.1. Client Hello

When this message will be sent:

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 ServerHello that selects cryptographic parameters that don't match the client's KeyShare extension. In that case, the client MUST send the same ClientHello (without modification) except including a new KeyShareEntry as the lowest priority share (i.e., appended to the list of shares in the KeyShare message). [[OPEN ISSUE: New random values? See: https://github.com/tlswg/tls13-spec/issues/185]] If a server receives a ClientHello at any other time, it MUST send a fatal "unexpected_message" alert and close the connection.

Structure of this message:

The ClientHello message includes a random structure, which is used later in the protocol.

```
struct {
    opaque random_bytes[32];
} Random;
```

random_bytes
: 32 bytes generated by a secure random number generator. See Appendix B for additional information.

Note: Versions of TLS prior to TLS 1.3 used the top 32 bits of the Random value to encode the time since the UNIX epoch.

The cipher suite list, passed from the client to the server in the ClientHello message, contains the combinations of cryptographic algorithms supported by the client in order of the client's preference (favorite choice first). Each cipher suite defines a key exchange algorithm, a record protection algorithm (including secret key length) and a hash to be used with HKDF. The server will select a cipher suite or, if no acceptable choices are presented, return a "handshake_failure" alert and close the connection. 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.

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

    enum { null(0), (255) } CompressionMethod;

    struct {
        ProtocolVersion client_version = { 3, 4 };    /* TLS v1.3 */
        Random random;
        SessionID session_id;
        CipherSuite cipher_suites<2..2^16-2>;
        CompressionMethod compression_methods<1..2^8-1>;
        Extension extensions<0..2^16-1>;
    } ClientHello;
```

TLS allows extensions to follow the compression_methods field in an extensions block. The presence of extensions can be detected by determining whether there are bytes following the compression_methods 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.

client_version
: The version of the TLS protocol by which the client wishes to communicate during this session. This SHOULD be the latest (highest valued) version supported by the client. For this version of the specification, the version will be { 3, 4 }. (See Appendix C for details about backward compatibility.)

random
: A client-generated random structure.

session_id
: Versions of TLS prior to TLS 1.3 supported a session resumption feature which has been merged with Pre-Shared Keys in this version (see Section 6.2.3). This field MUST be ignored by a server negotiating TLS 1.3 and should be set as a zero length vector (i.e., a single zero byte length field) by clients which do not have a cached session_id set by a pre-TLS 1.3 server.

cipher_suites
: This is a list of the cryptographic options supported by the client, with the client's first preference first. Values are defined in Appendix A.4.

compression_methods
: Versions of TLS before 1.3 supported compression and the list of compression methods was supplied in this field. For any TLS 1.3 ClientHello, this field MUST contain only the "null" compression

method with the code point of 0. If a TLS 1.3 ClientHello is received with any other value in this field, the server MUST generate a fatal "illegal_parameter" alert. Note that TLS 1.3 servers may receive TLS 1.2 or prior ClientHellos which contain other compression methods and MUST follow the procedures for the appropriate prior version of TLS.

extensions
Clients MAY request extended functionality from servers by sending data in the extensions field. The actual "Extension" format is defined in Section 6.3.2.

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. A server MUST accept ClientHello messages both with and without the extensions field, and (as for all other messages) it MUST check that the amount of data in the message precisely matches one of these formats; if not, then it MUST send a fatal "decode_error" alert.

After sending the ClientHello message, the client waits for a ServerHello or HelloRetryRequest message.

6.3.1.2. Server Hello

When this message will be sent:

The server will send this message in response to a ClientHello message when it was able to find an acceptable set of algorithms and the client's KeyShare extension was acceptable. If the client proposed groups are not acceptable by the server, it will respond with a "handshake_failure" fatal alert.

Structure of this message:

```
struct {
    ProtocolVersion server_version;
    Random random;
    CipherSuite cipher_suite;
    select (extensions_present) {
        case false:
            struct {};
        case true:
            Extension extensions<0..2^16-1>;
    };
} ServerHello;
```

The presence of extensions can be detected by determining whether there are bytes following the cipher_suite field at the end of the ServerHello.

server_version
: This field will contain the lower of that suggested by the client in the ClientHello and the highest supported by the server. For this version of the specification, the version is { 3, 4 }. (See Appendix C for details about backward compatibility.)

random
: 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. For resumed sessions, this field is the value from the state of the session being resumed. [[TODO: interaction with PSK.]]

extensions
: A list of extensions. Note that only extensions offered by the client can appear in the server's list. In TLS 1.3 as opposed to previous versions of TLS, the server's extensions are split between the ServerHello and the EncryptedExtensions Section 6.3.3 message. The ServerHello MUST only include extensions which are required to establish the cryptographic context.

6.3.1.3. Hello Retry Request

When this message will be sent:

> Servers send this message in response to a ClientHello message when it was able to find an acceptable set of algorithms and groups that are mutually supported, but the client's KeyShare did not contain an acceptable offer. If it cannot find such a match, it will respond with a "handshake_failure" alert.

Structure of this message:

```
struct {
    ProtocolVersion server_version;
    CipherSuite cipher_suite;
    NamedGroup selected_group;
    Extension extensions<0..2^16-1>;
} HelloRetryRequest;
```

[[OPEN ISSUE: Merge in DTLS Cookies?]]

selected_group
The group which the client MUST use for its new ClientHello.

The "server_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.

Upon receipt of a HelloRetryRequest, the client MUST first verify that the "selected_group" field corresponds to a group which was provided in the "supported_groups" extension in the original ClientHello. It MUST then verify that 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 a fatal "handshake_failure" alert. Clients SHOULD also abort with "handshake_failure" 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).

Otherwise, the client MUST send a ClientHello with a new KeyShare extension to the server. The client MUST append a new KeyShareEntry list which is consistent with the "selected_group" field to the groups in its original KeyShare.

Upon re-sending the ClientHello and receiving the server's ServerHello/KeyShare, the client MUST verify that the selected CipherSuite and NamedGroup match that supplied in the HelloRetryRequest.

[[OPEN ISSUE: https://github.com/tlswg/tls13-spec/issues/104]]

6.3.2. Hello Extensions

The extension format is:

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

enum {
    supported_groups(10),
    signature_algorithms(13),
    early_data(TBD),
    pre_shared_key(TBD),
    key_share(TBD),
    (65535)
} ExtensionType;
```

Here:

- "extension_type" identifies the particular extension type.

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

The initial set of extensions is defined in [RFC6066]. The list of extension types is maintained by IANA as described in Section 11.

An extension type MUST NOT appear in the ServerHello or HelloRetryRequest unless the same extension type appeared in the corresponding ClientHello. If a client receives an extension type in ServerHello or HelloRetryRequest that it did not request in the associated ClientHello, it MUST abort the handshake with an "unsupported_extension" fatal alert.

Nonetheless, "server-oriented" extensions may be provided in the future within this framework. Such an extension (say, of type x) would require the client to first send an extension of type x in a ClientHello with empty extension_data to indicate that it supports the extension type. In this case, the client is offering the capability to understand the extension type, and the server is taking the client up on its offer.

When multiple extensions of different types are present in the ClientHello or ServerHello messages, the extensions MAY appear in any order. There MUST NOT be more than one extension of the same type.

Finally, note that extensions can be sent both when starting a new session and when requesting session resumption or 0-RTT mode. Indeed, a client that requests session resumption does not in general know whether the server will accept this request, and therefore it

SHOULD send the same extensions as it would send if it were not attempting resumption.

In general, the specification of each extension type needs to describe the effect of the extension both during full handshake and session resumption. Most current TLS extensions are relevant only when a session is initiated: when an older session is resumed, the server does not process these extensions in ClientHello, and does not include them in ServerHello. However, some extensions may specify different behavior during session resumption. [[TODO: update this and the previous paragraph to cover PSK-based resumption.]]

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.

- It would be technically possible to use extensions to change major aspects of the design of TLS; for example the design of cipher suite negotiation. This is not recommended; it would be more appropriate to define a new version of TLS -- particularly since the TLS handshake algorithms have specific protection against version rollback attacks based on the version number, and the possibility of version rollback should be a significant consideration in any major design change.

6.3.2.1. Signature Algorithms

The client uses the "signature_algorithms" extension to indicate to the server which signature/hash algorithm pairs may be used in digital signatures.

Clients which offer one or more cipher suites which use certificate authentication (i.e., any non-PSK cipher suite) MUST send the "signature_algorithms" extension. If this extension is not provided and no alternative cipher suite is available, the server MUST close the connection with a fatal "missing_extension" alert. (see Section 8.2)

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

```
   enum {
       none(0),
       sha1(2),
       sha256(4), sha384(5), sha512(6),
       (255)
   } HashAlgorithm;

   enum {
       rsa(1),
       dsa(2),
       ecdsa(3),
       rsapss(4),
       (255)
   } SignatureAlgorithm;

   struct {
       HashAlgorithm hash;
       SignatureAlgorithm signature;
   } SignatureAndHashAlgorithm;

   SignatureAndHashAlgorithm
     supported_signature_algorithms<2..2^16-2>;
```

[[TODO: IANA considerations for new SignatureAlgorithm value]]

Each SignatureAndHashAlgorithm value lists a single hash/signature pair that the client is willing to verify. The values are indicated in descending order of preference.

Note: Because not all signature algorithms and hash algorithms may be accepted by an implementation (e.g., ECDSA with SHA-256, but not SHA-384), algorithms here are listed in pairs.

hash
: This field indicates the hash algorithms which may be used. The values indicate support for unhashed data, SHA-1, SHA-256, SHA-384, and SHA-512 [SHS], respectively. The "none" value is provided for future extensibility, in case of a signature algorithm which does not require hashing before signing. Previous versions of TLS supported MD5 and SHA-1. These algorithms are now deprecated and MUST NOT be offered by TLS 1.3 implementations. SHA-1 SHOULD NOT be offered, however clients willing to negotiate use of TLS 1.2 MAY offer support for SHA-1 for backwards compatibility with old servers.

signature
: This field indicates the signature algorithm that may be used. The values indicate RSASSA-PKCS1-v1_5 [RFC3447], DSA [DSS], ECDSA [ECDSA], and RSASSA-PSS [RFC3447] respectively. Because all RSA signatures used in signed TLS handshake messages (see Section 4.9.1), as opposed to those in certificates, are RSASSA-PSS, the "rsa" value refers solely to signatures which appear in certificates. The use of DSA and anonymous is deprecated. Previous versions of TLS supported DSA. DSA is deprecated as of TLS 1.3 and SHOULD NOT be offered or negotiated by any implementation.

The semantics of this extension are somewhat complicated because the cipher suite indicates permissible signature algorithms but not hash algorithms. Section 6.3.4 and Section 6.3.2.3 describe the appropriate rules.

Clients offering support for SHA-1 for TLS 1.2 servers MUST do so by listing those hash/signature pairs as the lowest priority (listed after all other pairs in the supported_signature_algorithms vector). TLS 1.3 servers MUST NOT offer a SHA-1 signed certificate unless no valid certificate chain can be produced without it (see Section 6.3.4).

Note: TLS 1.3 servers MAY receive TLS 1.2 ClientHellos which do not contain this extension. If those servers are willing to negotiate TLS 1.2, they MUST behave in accordance with the requirements of [RFC5246] when negotiating that version.

6.3.2.2. Negotiated Groups

When sent by the client, the "supported_groups" extension indicates the named groups which the client supports, 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 [I-D.ietf-tls-negotiated-ff-dhe].

Clients which offer one or more (EC)DHE cipher suites MUST send at least one supported NamedGroup value and servers MUST NOT negotiate any of these cipher suites unless a supported value was provided. If this extension is not provided and no alternative cipher suite is available, the server MUST close the connection with a fatal "missing_extension" alert. (see Section 8.2) If the extension is provided, but no compatible group is offered, the server MUST NOT negotiate a cipher suite of the relevant type. For instance, if a client supplies only ECDHE groups, the server MUST NOT negotiate finite field Diffie-Hellman. If no acceptable group can be selected across all cipher suites, then the server MUST generate a fatal "handshake_failure" alert.

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

```
enum {
    // Elliptic Curve Groups.
    secp256r1 (23), secp384r1 (24), secp521r1 (25),

    // Finite Field Groups.
    ffdhe2048 (256), ffdhe3072 (257), ffdhe4096 (258),
    ffdhe6144 (259), ffdhe8192 (260),

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

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

secp256r1, etc.
: Indicates support of the corresponding named curve. Note that some curves are also recommended in ANSI X9.62 [X962] and FIPS 186-4 [DSS]. Values 0xFE00 through 0xFEFF are reserved for private use.

ffdhe2048, etc.
: Indicates support of the corresponding finite field group, defined in [I-D.ietf-tls-negotiated-ff-dhe]. Values 0x01FC through 0x01FF are reserved for private use.

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

As an example, a client that only supports secp256r1 (aka NIST P-256; value 23 = 0x0017) and secp384r1 (aka NIST P-384; value 24 = 0x0018) and prefers to use secp256r1 would include a TLS extension consisting of the following octets. Note that the first two octets indicate the extension type (Supported Group Extension):

```
00 0A 00 06 00 04 00 17 00 18
```

NOTE: A server participating in an ECDHE-ECDSA key exchange may use different curves for (i) the ECDSA key in its certificate, and (ii) the ephemeral ECDH key in its KeyShare extension. The server must consider the supported groups in both cases.

[[TODO: IANA Considerations.]]

6.3.2.3. Key Share

The "key_share" extension contains the endpoint's cryptographic parameters for non-PSK key establishment methods (currently DHE or ECDHE).

Clients which offer one or more (EC)DHE cipher suites MUST send at least one supported KeyShare value and servers MUST NOT negotiate any of these cipher suites unless a supported value was provided. If this extension is not provided in a ServerHello or retried ClientHello, and the peer is offering (EC)DHE cipher suites, then the endpoint MUST close the connection with a fatal "missing_extension" alert. (see Section 8.2)

```
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 6.3.2.3.1; Elliptic Curve Diffie-Hellman parameters are described in Section 6.3.2.3.2.

key_exchange
: Key exchange information. The contents of this field are determined by the specified group and its corresponding definition. Endpoints MUST NOT send empty or otherwise invalid key_exchange values for any reason.

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

```
struct {
    select (role) {
        case client:
            KeyShareEntry client_shares<4..2^16-1>;

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

client_shares
: A list of offered KeyShareEntry values in descending order of client preference. This vector MUST NOT be empty. Clients not providing a KeyShare MUST instead omit this extension from the ClientHello.

server_shares
: A single KeyShareEntry value for the negotiated cipher suite. Servers MUST NOT send a KeyShareEntry value for a group not offered by the client.

Servers offer exactly one KeyShareEntry value, which corresponds to the key exchange used for the negotiated cipher suite.

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 integer DH groups. The key_exchange values for each KeyShareEntry MUST by generated independently. Clients MUST NOT offer multiple KeyShareEntry values for the same parameters. Clients MAY omit this extension from the ClientHello, and in response to this, servers MUST send a HelloRetryRequest requesting use of one of the groups the client offered support for in its "supported_groups" extension. If no common supported group is available, the server MUST produce a fatal "handshake_failure" alert. (see Section 6.3.1.3)

[[TODO: Recommendation about what the client offers. Presumably which integer DH groups and which curves.]]

6.3.2.3.1. 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 (dh_Y = g^X mod p), encoded as a big-endian integer.

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

6.3.2.3.2. 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. The opaque value conveys the Elliptic Curve Diffie-Hellman public value (ecdh_Y) represented as a byte string ECPoint.point.

```
opaque point <1..2^8-1>;
```

point
: This is the byte string representation of an elliptic curve point following the conversion routine in Section 4.3.6 of ANSI X9.62 [X962].

Although X9.62 supports multiple point formats, any given curve MUST specify only a single point format. All curves currently specified in this document MUST only be used with the uncompressed point format.

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

[[OPEN ISSUE: We will need to adjust the compressed/uncompressed point issue if we have new curves that don't need point compression. This depends on the CFRG's recommendations. The expectation is that future curves will come with defined point formats and that existing curves conform to X9.62.]]

6.3.2.4. 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 a PSK or (EC)DHE-PSK cipher suite (see [RFC4279] for background).

Clients which offer one or more PSK cipher suites MUST send at least one supported psk_identity value and servers MUST NOT negotiate any of these cipher suites unless a supported value was provided. If this extension is not provided and no alternative cipher suite is available, the server MUST close the connection with a fatal "missing_extension" alert. (see Section 8.2)

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

```
opaque psk_identity<0..2^16-1>;

struct {
    select (Role) {
        case client:
            psk_identity identities<2..2^16-1>;

        case server:
            psk_identity identity;
    }
} PreSharedKeyExtension;
```

identity
An opaque label for the pre-shared key.

If no suitable identity is provided, the server MUST NOT negotiate a PSK cipher suite and MAY respond with an "unknown_psk_identity" alert message. Sending this alert is OPTIONAL; servers MAY instead choose to send a "decrypt_error" alert to merely indicate an invalid PSK identity or instead negotiate use of a non-PSK cipher suite, if available.

If the server selects a PSK cipher suite, it MUST send a PreSharedKeyExtension with the identity that it selected. The client MUST verify that the server has selected one of the identities that the client supplied. If any other identity is returned, the client MUST generate a fatal "unknown_psk_identity" alert and close the connection.

6.3.2.5. Early Data Indication

In cases where TLS clients have previously interacted with the server and the server has supplied a ServerConfiguration Section 6.3.6, the client can send application data and its Certificate/ CertificateVerify messages (if client authentication is required). If the client opts to do so, it MUST supply an Early Data Indication extension.

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

```
      enum { client_authentication(1), early_data(2),
             client_authentication_and_data(3), (255) } EarlyDataType;

      struct {
          select (Role) {
              case client:
                  opaque configuration_id<1..2^16-1>;
                  CipherSuite cipher_suite;
                  Extension extensions<0..2^16-1>;
                  opaque context<0..255>;
                  EarlyDataType type;

              case server:
                 struct {};
          }
      } EarlyDataIndication;
```

configuration_id
: The label for the configuration in question.

cipher_suite
: The cipher suite which the client is using to encrypt the early data.

extensions
: The extensions required to define the cryptographic configuration for the clients early data (see below for details).

context
: An optional context value that can be used for anti-replay (see below).

type
: The type of early data that is being sent. "client_authentication" means that only handshake data is being sent. "early_data" means that only data is being sent. "client_authentication_and_data" means that both are being sent.

The client specifies the cryptographic configuration for the 0-RTT data using the "configuration", "cipher_suite", and "extensions" values. For configurations received in-band (in a previous TLS connection) the client MUST:

- Send the same cryptographic determining parameters (Section Section 6.3.2.5.1) with the previous connection. If a 0-RTT handshake is being used with a PSK that was negotiated via a non-PSK handshake, then the client MUST use the same symmetric

cipher parameters as were negotiated on that handshake but with a PSK cipher suite.

- Indicate the same parameters as the server indicated in that connection.

If TLS client authentication is being used, then either "early_handshake" or "early_handshake_and_data" MUST be indicated in order to send the client authentication data on the first flight. In either case, the client Certificate and CertificateVerify (assuming that the Certificate is non-empty) MUST be sent on the first flight. A server which receives an initial flight with only "early_data" and which expects certificate-based client authentication MUST NOT accept early data.

In order to allow servers to readily distinguish between messages sent in the first flight and in the second flight (in cases where the server does not accept the EarlyDataIndication extension), the client MUST send the handshake messages as content type "early_handshake". A server which does not accept the extension proceeds by skipping all records after the ClientHello and until the next client message of type "handshake". [[OPEN ISSUE: This needs replacement when we add encrypted content types.]]

A server which receives an EarlyDataIndication extension can behave in one of two ways:

- Ignore the extension and return no response. This indicates that the server has ignored any early data and an ordinary 1-RTT handshake is required.

- Return an empty extension, 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.

Prior to accepting the EarlyDataIndication extension, the server MUST perform the following checks:

- The configuration_id matches a known server configuration.

- The client's cryptographic determining parameters match the parameters that the server has negotiated based on the rest of the ClientHello.

If any of these checks fail, the server MUST NOT respond with the extension and must discard all the remaining first flight data (thus falling back to 1-RTT).

[[TODO: How does the client behave if the indication is rejected.]]

[[OPEN ISSUE: This just specifies the signaling for 0-RTT but not the the 0-RTT cryptographic transforms, including:

- What is in the handshake hash (including potentially some speculative data from the server).

- What is signed in the client's CertificateVerify.

- Whether we really want the Finished to not include the server's data at all.

What's here now needs a lot of cleanup before it is clear and correct.]]

6.3.2.5.1. Cryptographic Determining Parameters

In order to allow the server to decrypt 0-RTT data, the client needs to provide enough information to allow the server to decrypt the traffic without negotiation. This is accomplished by having the client indicate the "cryptographic determining parameters" in its ClientHello, which are necessary to decrypt the client's packets. This includes the following values:

- The cipher suite identifier.

- If PSK is being used, the server's version of the PreSharedKey extension (indicating the PSK the client is using).

[[TODO: Are there other extensions we need? I've gone over the list and I don't see any, but...]] [[TODO: This should be the same list as what you need for !EncryptedExtensions. Consolidate this list.]]

6.3.2.5.2. Replay Properties

As noted in Section 6.2.2, TLS does not provide any inter-connection mechanism for replay protection for data sent by the client in the first flight. As a special case, implementations where the server configuration, is delivered out of band (as has been proposed for DTLS-SRTP [RFC5763]), MAY use a unique server configuration identifier for each connection, thus preventing replay. Implementations are responsible for ensuring uniqueness of the identifier in this case.

6.3.3. Encrypted Extensions

When this message will be sent:

If this message is sent, it MUST be sent immediately after the ServerHello message. This is the first message that is encrypted under keys derived from ES.

Meaning of this message:

The EncryptedExtensions message simply contains any extensions which should be protected, i.e., any which are not needed to establish the cryptographic context. The same extension types MUST NOT appear in both the ServerHello and EncryptedExtensions. If the same extension appears in both locations, the client MUST rely only on the value in the EncryptedExtensions block. [[OPEN ISSUE: Should we just produce a canonical list of what goes where and have it be an error to have it in the wrong place? That seems simpler. Perhaps have a whitelist of which extensions can be unencrypted and everything else MUST be encrypted.]]

Structure of this message:

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

extensions
A list of extensions.

6.3.4. Server Certificate

When this message will be sent:

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). This message will always immediately follow the EncryptedExtensions message.

Meaning of this message:

This message conveys the server's certificate chain to the client.

The certificate MUST be appropriate for the negotiated cipher suite's key exchange algorithm and any negotiated extensions.

Structure of this message:

```
opaque ASN1Cert<1..2^24-1>;

struct {
    ASN1Cert certificate_list<0..2^24-1>;
} Certificate;
```

certificate_list
This is a sequence (chain) of certificates. The sender's certificate MUST come first 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.

The same message type and structure will be used for the client's response to a certificate request message. Note that a client MAY send no certificates if it does not have an appropriate certificate to send in response to the server's authentication request.

Note: PKCS #7 [PKCS7] is not used as the format for the certificate vector because PKCS #6 [PKCS6] extended certificates are not used. Also, PKCS #7 defines a SET rather than a SEQUENCE, making the task of parsing the list more difficult.

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., [RFC5081]).

- The server's end-entity certificate's public key (and associated restrictions) MUST be compatible with the selected key exchange algorithm.

Key Exchange Alg.	Certificate Key Type
DHE_RSA ECDHE_RSA	RSA public key; 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 the signature scheme and hash algorithm that will be employed in the server's KeyShare extension. Note: ECDHE_RSA is defined in [RFC4492].
ECDHE_ECDSA	ECDSA-capable public key; the certificate MUST allow the key to be used for signing with the hash algorithm that will be employed in the server's KeyShare extension. The public key MUST use a curve and point format supported by the client, as described in [RFC4492].

- 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.

All certificates provided by the server MUST be signed by a hash/signature algorithm pair that appears in the "signature_algorithms" extension provided by the client, if they are able to provide such a chain (see Section 6.3.2.1). If the server cannot produce a certificate chain that is signed only via the indicated supported pairs, 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. If the client cannot construct an acceptable chain using the provided certificates and decides to abort the handshake, then it MUST send an "unsupported_certificate" alert message and close the connection.

Any endpoint receiving any certificate signed using any signature algorithm using an MD5 hash MUST send a "bad_certificate" alert message and close the connection.

As SHA-1 and SHA-224 are deprecated, support for them is NOT RECOMMENDED. Endpoints that reject chains due to use of a deprecated hash MUST send a fatal "bad_certificate" alert message before closing the connection. All servers 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 a ECDSA key).

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). If the server has a single certificate, it SHOULD attempt to validate that it meets these criteria.

As cipher suites that specify new key exchange methods are specified for the TLS protocol, they will imply the certificate format and the required encoded keying information.

6.3.5. Certificate Request

When this message will be sent:

A non-anonymous server can optionally request a certificate from the client, if appropriate for the selected cipher suite. This message, if sent, will immediately follow the server's Certificate message.

Structure of this message:

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

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

struct {
    SignatureAndHashAlgorithm
      supported_signature_algorithms<2..2^16-2>;
    DistinguishedName certificate_authorities<0..2^16-1>;
    CertificateExtension certificate_extensions<0..2^16-1>;
} CertificateRequest;
```

supported_signature_algorithms
A list of the hash/signature algorithm pairs that the server is able to verify, listed in descending order of preference. Any certificates provided by the client MUST be signed using a hash/signature algorithm pair found in supported_signature_algorithms.

certificate_authorities
A list of the distinguished names [X501] of acceptable certificate_authorities, represented in DER-encoded [X690] format.

These distinguished names may specify a desired distinguished name for a root CA or for a subordinate CA; thus, this message can be used to describe known roots as well as a desired authorization space. If the certificate_authorities list is empty, then the client MAY send any certificate that meets the rest of the selection criteria in the CertificateRequest, unless there is some external arrangement to the contrary.

certificate_extensions
A list of certificate extension OIDs [RFC5280] with their allowed values, represented in DER-encoded 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 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 has 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 session without client authentication, or terminate the session with a fatal 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.

Note: It is a fatal "handshake_failure" alert for an anonymous server to request client authentication.

6.3.6. Server Configuration

When this message will be sent:

This message is used to provide a server configuration which the client can use in future to skip handshake negotiation and (optionally) to allow 0-RTT handshakes. The ServerConfiguration message is sent as the last message before the CertificateVerify.

Structure of this Message:

```
enum { (65535) } ConfigurationExtensionType;

struct {
    ConfigurationExtensionType extension_type;
    opaque extension_data<0..2^16-1>;
} ConfigurationExtension;

struct {
    opaque configuration_id<1..2^16-1>;
    uint32 expiration_date;
    NamedGroup group;
    opaque server_key<1..2^16-1>;
    EarlyDataType early_data_type;
    ConfigurationExtension extensions<0..2^16-1>;
} ServerConfiguration;
```

configuration_id
: The configuration identifier to be used in 0-RTT mode.

group
: The group for the long-term DH key that is being established for this configuration.

expiration_date
: The last time when this configuration is expected to be valid (in seconds since the Unix epoch). Servers MUST NOT use any value more than 604800 seconds (7 days) in the future. Clients MUST NOT cache configurations for longer than 7 days, regardless of the expiration_date. [[OPEN ISSUE: Is this the right value? The idea is just to minimize exposure.]]

server_key
: The long-term DH key that is being established for this configuration.

early_data_type

The type of 0-RTT handshake that this configuration is to be used for (see Section 6.3.2.5). If "client_authentication" or "client_authentication_and_data", then the client should select the certificate for future handshakes based on the CertificateRequest parameters supplied in this handshake. The server MUST NOT send either of these two options unless it also requested a certificate on this handshake. [[OPEN ISSUE: Should we relax this?]]

extensions
This field is a placeholder for future extensions to the ServerConfiguration format.

The semantics of this message are to establish a shared state between the client and server for use with the "known_configuration" extension with the key specified in key and with the handshake parameters negotiated by this handshake.

When the ServerConfiguration message is sent, the server MUST also send a Certificate message and a CertificateVerify message, even if the "known_configuration" extension was used for this handshake, thus requiring a signature over the configuration before it can be used by the client. Clients MUST NOT rely on the ServerConfiguration message until successfully receiving and processing the server's Certificate, CertificateVerify, and Finished. If there is a failure in processing those messages, the client MUST discard the ServerConfiguration.

6.3.7. Server Certificate Verify

When this message will be sent:

This message is used to provide explicit proof that the server possesses the private key corresponding to its certificate and also provides integrity for the handshake up to this point. This message is sent when the server is authenticated via a certificate. When sent, it MUST be the last server handshake message prior to the Finished.

Structure of this message:

```
struct {
     digitally-signed struct {
        opaque handshake_hash[hash_length];
     };
} CertificateVerify;
```

Where handshake_hash is as described in Section 7.2.1 and includes the messages sent or received, starting at ClientHello and up to,

but not including, this message, including the type and length fields of the handshake messages. This is a digest of the concatenation of all the Handshake structures (as defined in Section 6.3) exchanged thus far. The digest MUST be the Hash used as the basis for HKDF.

The context string for the signature is "TLS 1.3, server CertificateVerify". A hash of the handshake messages is signed rather than the messages themselves because the digitally-signed format requires padding and context bytes at the beginning of the input. Thus, by signing a digest of the messages, an implementation need only maintain one running hash per hash type for CertificateVerify, Finished and other messages.

The signature algorithm and hash algorithm MUST be a pair offered in the client's "signature_algorithms" extension unless no valid certificate chain can be produced without unsupported algorithms (see Section 6.3.2.1). Note that there is a possibility for inconsistencies here. For instance, the client might offer ECDHE_ECDSA key exchange but omit any ECDSA pairs from its "signature_algorithms" extension. In order to negotiate correctly, the server MUST check any candidate cipher suites against the "signature_algorithms" extension before selecting them. This is somewhat inelegant but is a compromise designed to minimize changes to the original cipher suite design.

In addition, the hash and signature algorithms MUST be compatible with the key in the server's end-entity certificate. RSA keys MAY be used with any permitted hash algorithm, subject to restrictions in the certificate, if any. RSA signatures MUST be based on RSASSA-PSS, regardless of whether RSASSA-PKCS-v1_5 appears in "signature_algorithms". SHA-1 MUST NOT be used in any signatures in CertificateVerify, regardless of whether SHA-1 appears in "signature_algorithms".

6.3.8. Server Finished

When this message will be sent:

The Server's Finished message is the final message sent by the server and is essential for providing authentication of the server side of the handshake and computed keys.

Meaning of this message:

Recipients of Finished messages MUST verify that the contents are correct. 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. This data will be protected under keys derived from the ephemeral secret (see Section 7).

Structure of this message:

```
struct {
    opaque verify_data[verify_data_length];
} Finished;
```

The verify_data value is computed as follows:

verify_data
HMAC(finished_secret, finished_label + '\0' + handshake_hash) where HMAC [RFC2104] uses the Hash algorithm for the handshake. See Section 7.2.1 for the definition of handshake_hash.

finished_label
For Finished messages sent by the client, the string "client finished". For Finished messages sent by the server, the string "server finished".

In previous versions of TLS, the verify_data was always 12 octets long. In the current version of TLS, 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. Also, HelloRequest messages and the Finished message are omitted from handshake hashes.

6.3.9. Client Certificate

When this message will be sent:

This message is the first handshake message the client can send after receiving the server's Finished. This message is only sent if the server requests a certificate. If no suitable certificate is available, the client MUST send a certificate message containing no certificates. That is, the certificate_list structure has a length of zero. If the client does not send any certificates, the server MAY at its discretion either continue the handshake without client authentication, or respond with a fatal "handshake_failure" 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 send a fatal alert.

Client certificates are sent using the Certificate structure defined in Section 6.3.4.

Meaning of this message:

This message conveys the client's certificate chain to the server; the server will use it when verifying the CertificateVerify message (when the client authentication is based on signing). The certificate MUST be appropriate for the negotiated cipher suite's key exchange algorithm, and any negotiated extensions.

In particular:

- The certificate type MUST be X.509v3 [RFC5280], unless explicitly negotiated otherwise (e.g., [RFC5081]).

- If the certificate_authorities list in the certificate request message was non-empty, 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 hash/signature algorithm pair, as described in Section 6.3.5. Note that this relaxes the constraints on certificate-signing algorithms found in prior versions of TLS.

- If the certificate_extensions list in the certificate request message was non-empty, the end-entity certificate MUST match the extension OIDs recognized by the client, as described in Section 6.3.5.

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

6.3.10. Client Certificate Verify

When this message will be sent:

This message is used to provide explicit verification of a client certificate. This message is only sent following a client certificate that has signing capability (i.e., all certificates except those containing fixed Diffie-Hellman parameters). When sent, it MUST immediately follow the client's Certificate message. The contents of the message are computed as described in Section 6.3.7, except that the context string is "TLS 1.3, client CertificateVerify".

The hash and signature algorithms used in the signature MUST be one of those present in the supported_signature_algorithms field of the CertificateRequest message. In addition, the hash and signature algorithms MUST be compatible with the key in the client's end-entity certificate. RSA keys MAY be used with any permitted hash algorithm, subject to restrictions in the certificate, if any. RSA signatures MUST be based on RSASSA-PSS, regardless of whether RSASSA-PKCS-v1_5 appears in "signature_algorithms". SHA-1 MUST NOT be used in any signatures in CertificateVerify, regardless of whether SHA-1 appears in "signature_algorithms".

6.3.11. New Session Ticket Message

After the server has received the client Finished message, it MAY send a NewSessionTicket message. This message MUST be sent before the server sends any application data traffic, and is encrypted under the application traffic key. This message creates a pre-shared key (PSK) binding between the resumption master secret and the ticket label. The client MAY use this PSK for future handshakes by including it in the "pre_shared_key" extension in its ClientHello (Section 6.3.2.4) and supplying a suitable PSK cipher suite.

```
struct {
    uint32 ticket_lifetime_hint;
    opaque ticket<0..2^16-1>;
} NewSessionTicket;
```

ticket_lifetime_hint
Indicates the lifetime in seconds as a 32-bit unsigned integer in network byte order from the time of ticket issuance. A value of zero is reserved to indicate that the lifetime of the ticket is unspecified.

ticket
The value of the ticket to be used as the PSK identifier.

The ticket lifetime hint is informative only. A client SHOULD delete the ticket and associated state when the time expires. It MAY delete the ticket earlier based on local policy. A server MAY treat a ticket as valid for a shorter or longer period of time than what is stated in the ticket_lifetime_hint.

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.

[[TODO: Should we require that tickets be bound to the existing symmetric cipher suite. See the TODO above about early_data and PSK.??]

7. Cryptographic Computations

In order to begin connection protection, the TLS Record Protocol requires specification of a suite of algorithms, a master secret, and the client and server random values. The authentication, key exchange, and record protection algorithms are determined by the cipher_suite selected by the server and revealed in the ServerHello message. The random values are exchanged in the hello messages. All that remains is to calculate the key schedule.

7.1. Key Schedule

The TLS handshake establishes secret keying material which is then used to protect traffic. This keying material is derived from the two input secret values: Static Secret (SS) and Ephemeral Secret (ES).

The exact source of each of these secrets depends on the operational mode (DHE, ECDHE, PSK, etc.) and is summarized in the table below:

Key Exchange	Static Secret (SS)	Ephemeral Secret (ES)
(EC)DHE (full handshake)	Client ephemeral w/ server ephemeral	Client ephemeral w/ server ephemeral
(EC)DHE (w/ 0-RTT)	Client ephemeral w/ server static	Client ephemeral w/ server ephemeral
PSK	Pre-Shared Key	Pre-shared key
PSK + (EC)DHE	Pre-Shared Key	Client ephemeral w/ server ephemeral

These shared secret values are used to generate cryptographic keys as shown below.

The derivation process is as follows, where L denotes the length of the underlying hash function for HKDF [RFC5869]. SS and ES denote the sources from the table above. Whilst SS and ES may be the same in some cases, the extracted xSS and xES will not.

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

Where HkdfLabel is specified as:

```
struct HkdfLabel {
  uint16 length;
  opaque hash_value<0..255>;
  opaque label<9..255>;
};
```

Where:
- HkdfLabel.length is Length
- HkdfLabel.hash_value is HashValue.
- HkdfLabel.label is "TLS 1.3, " + Label

1. xSS = HKDF-Extract(0, SS). Note that HKDF-Extract always produces a value the same length as the underlying hash function.

2. xES = HKDF-Extract(0, ES)

3. mSS = HKDF-Expand-Label(xSS, "expanded static secret", handshake_hash, L)

4. mES = HKDF-Expand-Label(xES, "expanded ephemeral secret", handshake_hash, L)

5. master_secret = HKDF-Extract(mSS, mES)

6. finished_secret = HKDF-Expand-Label(xSS, "finished secret", handshake_hash, L)

Where handshake_hash includes all the messages in the client's first flight and the server's flight, excluding the Finished messages (which are never included in the hashes).

5. resumption_secret = HKDF-Expand-Label(master_secret, "resumption master secret" session_hash, L)

Where session_hash is as defined in {{the-handshake-hash}}.

6. exporter_secret = HKDF-Expand-Label(master_secret, "exporter master secret", session_hash, L)

Where session_hash is the session hash as defined in {{the-handshake-hash}} (i.e., the entire handshake except

for Finished).

The traffic keys are computed from xSS, xES, and the master_secret as described in Section 7.2 below.

Note: although the steps above are phrased as individual HKDF-Extract and HKDF-Expand operations, because each HKDF-Expand operation is paired with an HKDF-Extract, it is possible to implement this key schedule with a black-box HKDF API, albeit at some loss of efficiency as some HKDF-Extract operations will be repeated.

7.2. Traffic Key Calculation

[[OPEN ISSUE: This needs to be revised. Most likely we'll extract each key component separately. See https://github.com/tlswg/tls13-spec/issues/5]]

The Record Protocol requires an algorithm to generate keys required by the current connection state (see Appendix A.5) from the security parameters provided by the handshake protocol.

The traffic key computation takes four input values and returns a key block of sufficient size to produce the needed traffic keys:

- A secret value

- A string label that indicates the purpose of keys being generated.

- The current handshake hash.

- The total length in octets of the key block.

The keying material is computed using:

```
key_block = HKDF-Expand-Label(Secret, Label,
                              handshake_hash,
                              total_length)
```

The key_block is partitioned as follows:

```
client_write_key[SecurityParameters.enc_key_length]
server_write_key[SecurityParameters.enc_key_length]
client_write_IV[SecurityParameters.iv_length]
server_write_IV[SecurityParameters.iv_length]
```

The following table describes the inputs to the key calculation for each class of traffic keys:

Record Type	Secret	Label	Handshake Hash
Early data	xSS	"early data key expansion"	ClientHello
Handshake	xES	"handshake key expansion"	ClientHello... ServerHello
Application	master secret	"application data key expansion"	All handshake messages but Finished (session_hash)

7.2.1. The Handshake Hash

```
handshake_hash = Hash(
                      Hash(handshake_messages) ||
                      Hash(configuration)
                     )
```

handshake_messages
: All handshake messages sent or received, starting at ClientHello up to the present time, with the exception of the Finished message, including the type and length fields of the handshake messages. This is the concatenation of all the exchanged Handshake structures in plaintext form (even if they were encrypted on the wire).

configuration
: When 0-RTT is in use (Section 6.3.2.5) this contains the concatenation of the ServerConfiguration and Certificate messages from the handshake where the configuration was established (including the type and length fields). Note that this requires the client and server to memorize these values.

This final value of the handshake hash is referred to as the "session hash" because it contains all the handshake messages required to establish the session. Note that if client authentication is not used, then the session hash is complete at the point when the server has sent its first flight. Otherwise, it is only complete when the client has sent its first flight, as it covers the client's Certificate and CertificateVerify.

7.2.2. Diffie-Hellman

A conventional Diffie-Hellman computation is performed. The negotiated key (Z) is used as the shared secret, and is used in the key schedule as specified above. Leading bytes of Z that contain all zero bits are stripped before it is used as the input to HKDF.

7.2.3. Elliptic Curve Diffie-Hellman

All ECDH calculations (including parameter and key generation as well as the shared secret calculation) are performed according to [6] 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.)

8. Mandatory Algorithms

8.1. MTI Cipher Suites

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

```
TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256
TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
```

These cipher suites MUST support both digital signatures and key exchange with secp256r1 (NIST P-256) and SHOULD support key exchange with X25519 [I-D.irtf-cfrg-curves].

A TLS-compliant application SHOULD implement the following cipher suites:

```
TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384
TLS_ECDHE_ECDSA_WITH_CHACHA20_POLY1305
TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305
```

8.2. MTI Extensions

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

- Signature Algorithms ("signature_algorithms"; Section 6.3.2.1)

- Negotiated Groups ("supported_groups"; Section 6.3.2.2)

- Key Share ("key_share"; Section 6.3.2.3)

- Pre-Shared Key Extension ("pre_shared_key"; Section 6.3.2.4)

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

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

- "signature_algorithms" is REQUIRED for certificate authenticated cipher suites

- "supported_groups" and "key_share" are REQUIRED for DHE or ECDHE cipher suites

- "pre_shared_key" is REQUIRED for PSK cipher suites

When negotiating use of applicable cipher suites, endpoints MUST abort the connection with a "missing_extension" alert if the required extension was not provided. Any endpoint that receives any invalid combination of cipher suites and extensions MAY abort the connection with a "missing_extension" alert, regardless of negotiated parameters.

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 with a fatal "missing_extension" alert.

Some of these extensions exist only for the client to provide additional data to the server in a backwards-compatible way and thus have no meaning when sent from a server. The client-only extensions defined in this document are: "Signature Algorithms" & "Negotiated Groups". Servers MUST NOT send these extensions. Clients receiving any of these extensions MUST respond with a fatal "unsupported_extension" alert and close the connection.

9. 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.

10. Security Considerations

Security issues are discussed throughout this memo, especially in Appendices B, C, and D.

11. IANA Considerations

[[TODO: Update https://github.com/tlswg/tls13-spec/issues/62]]
[[TODO: Rename "RSA" in TLS SignatureAlgorithm Registry to RSASSA-PKCS1-v1_5]]

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 (unchanged from [RFC4346]) are listed below.

- TLS Cipher Suite Registry: Future values with the first byte in the range 0-191 (decimal) inclusive are assigned via Standards Action [RFC2434]. Values with the first byte in the range 192-254 (decimal) are assigned via Specification Required [RFC2434]. Values with the first byte 255 (decimal) are reserved for Private Use [RFC2434].

- TLS ContentType Registry: Future values are allocated via Standards Action [RFC2434].

- TLS Alert Registry: Future values are allocated via Standards Action [RFC2434].

- TLS HandshakeType Registry: Future values are allocated via Standards Action [RFC2434].

This document also uses a registry originally created in [RFC4366]. IANA has updated it to reference this document. The registry and its allocation policy (unchanged from [RFC4366]) is listed below:

- TLS ExtensionType Registry: Future values are allocated via IETF Consensus [RFC2434]. IANA has updated this registry to include the "signature_algorithms" extension and its corresponding value (see Section 6.3.2).

This document also uses two registries originally created in [RFC4492]. IANA [should update/has updated] it to reference this document. The registries and their allocation policies are listed below.

- TLS NamedCurve registry: Future values are allocated via IETF Consensus [RFC2434].

- TLS ECPointFormat Registry: Future values are allocated via IETF Consensus [RFC2434].

In addition, this document defines two new registries to be maintained by IANA:

- TLS SignatureAlgorithm Registry: The registry has been initially populated with the values described in Section 6.3.2.1. Future values in the range 0-63 (decimal) inclusive are assigned via Standards Action [RFC2434]. Values in the range 64-223 (decimal) inclusive are assigned via Specification Required [RFC2434]. Values from 224-255 (decimal) inclusive are reserved for Private Use [RFC2434].

- TLS HashAlgorithm Registry: The registry has been initially populated with the values described in Section 6.3.2.1. Future values in the range 0-63 (decimal) inclusive are assigned via Standards Action [RFC2434]. Values in the range 64-223 (decimal) inclusive are assigned via Specification Required [RFC2434]. Values from 224-255 (decimal) inclusive are reserved for Private Use [RFC2434].

12. References

12.1. Normative References

[AES] National Institute of Standards and Technology, "Specification for the Advanced Encryption Standard (AES)", NIST FIPS 197, November 2001.

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

[I-D.ietf-tls-chacha20-poly1305] Langley, A., Chang, W., Mavrogiannopoulos, N., Strombergson, J., and S. Josefsson, "The ChaCha20-Poly1305 AEAD Cipher for Transport Layer Security", draft-ietf-tls-chacha20-poly1305-00 (work in progress), June 2015.

[I-D.irtf-cfrg-curves] Langley, A. and M. Hamburg, "Elliptic Curves for Security", draft-irtf-cfrg-curves-08 (work in progress), September 2015.

[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>.

[RFC2434] Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", RFC 2434, DOI 10.17487/RFC2434, October 1998, <http://www.rfc-editor.org/info/rfc2434>.

[RFC3447] Jonsson, J. and B. Kaliski, "Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1", RFC 3447, DOI 10.17487/RFC3447, February 2003, <http://www.rfc-editor.org/info/rfc3447>.

[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>.

[RFC5288] Salowey, J., Choudhury, A., and D. McGrew, "AES Galois Counter Mode (GCM) Cipher Suites for TLS", RFC 5288, DOI 10.17487/RFC5288, August 2008, <http://www.rfc-editor.org/info/rfc5288>.

[RFC5289] Rescorla, E., "TLS Elliptic Curve Cipher Suites with SHA-256/384 and AES Galois Counter Mode (GCM)", RFC 5289, DOI 10.17487/RFC5289, August 2008, <http://www.rfc-editor.org/info/rfc5289>.

[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>.

[RFC6209] Kim, W., Lee, J., Park, J., and D. Kwon, "Addition of the ARIA Cipher Suites to Transport Layer Security (TLS)", RFC 6209, DOI 10.17487/RFC6209, April 2011, <http://www.rfc-editor.org/info/rfc6209>.

[RFC6367] Kanno, S. and M. Kanda, "Addition of the Camellia Cipher Suites to Transport Layer Security (TLS)", RFC 6367, DOI 10.17487/RFC6367, September 2011, <http://www.rfc-editor.org/info/rfc6367>.

[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>.

[RFC7251] McGrew, D., Bailey, D., Campagna, M., and R. Dugal, "AES-CCM Elliptic Curve Cryptography (ECC) Cipher Suites for TLS", RFC 7251, DOI 10.17487/RFC7251, June 2014, <http://www.rfc-editor.org/info/rfc7251>.

[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.

12.2. Informative References

[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.

[FI06] "Bleichenbacher's RSA signature forgery based on implementation error", August 2006, <http://www.imc.org/ietf-openpgp/mail-archive/msg14307.html>.

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

[I-D.ietf-tls-negotiated-ff-dhe] Gillmor, D., "Negotiated Finite Field Diffie-Hellman Ephemeral Parameters for TLS", draft-ietf-tls-negotiated-ff-dhe-10 (work in progress), June 2015.

[PKCS6] RSA Laboratories, "PKCS #6: RSA Extended Certificate Syntax Standard, version 1.5", November 1993.

[PKCS7] RSA Laboratories, "PKCS #7: RSA Cryptographic Message Syntax Standard, version 1.5", November 1993.

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

[RFC1948] Bellovin, S., "Defending Against Sequence Number Attacks", RFC 1948, DOI 10.17487/RFC1948, May 1996, <http://www.rfc-editor.org/info/rfc1948>.

[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>.

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

[RFC4302] Kent, S., "IP Authentication Header", RFC 4302, DOI 10.17487/RFC4302, December 2005, <http://www.rfc-editor.org/info/rfc4302>.

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

[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>.

[RFC4506] Eisler, M., Ed., "XDR: External Data Representation Standard", STD 67, RFC 4506, DOI 10.17487/RFC4506, May 2006, <http://www.rfc-editor.org/info/rfc4506>.

[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>.

[RFC5081] Mavrogiannopoulos, N., "Using OpenPGP Keys for Transport Layer Security (TLS) Authentication", RFC 5081, DOI 10.17487/RFC5081, November 2007, <http://www.rfc-editor.org/info/rfc5081>.

[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>.

[RFC5763] Fischl, J., Tschofenig, H., and E. Rescorla, "Framework for Establishing a Secure Real-time Transport Protocol (SRTP) Security Context Using Datagram Transport Layer Security (DTLS)", RFC 5763, DOI 10.17487/RFC5763, May 2010, <http://www.rfc-editor.org/info/rfc5763>.

[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>.

[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>.

[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>.

[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>.

[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.

[SSL2] Netscape Communications Corp., "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.

12.3. URIs

[1] mailto:tls@ietf.org

Appendix A. 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 may receive them from older TLS implementations.

A.1. Record Layer

```
struct {
    uint8 major;
    uint8 minor;
} ProtocolVersion;

enum {
    invalid_RESERVED(0),
    change_cipher_spec_RESERVED(20),
    alert(21),
    handshake(22),
    application_data(23),
    early_handshake(25),
    (255)
} ContentType;

struct {
    ContentType type;
    ProtocolVersion record_version = { 3, 1 };    /* TLS v1.x */
    uint16 length;
    opaque fragment[TLSPlaintext.length];
} TLSPlaintext;

struct {
    ContentType opaque_type = application_data(23); /* see fragment.type */
    ProtocolVersion record_version = { 3, 1 };    /* TLS v1.x */
    uint16 length;
    aead-ciphered struct {
       opaque content[TLSPlaintext.length];
       ContentType type;
       uint8 zeros[length_of_padding];
    } fragment;
} TLSCiphertext;
```

A.2. Alert Messages

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

      enum {
          close_notify(0),
          unexpected_message(10),               /* fatal */
          bad_record_mac(20),                   /* fatal */
          decryption_failed_RESERVED(21),       /* fatal */
          record_overflow(22),                  /* fatal */
          decompression_failure_RESERVED(30),   /* fatal */
          handshake_failure(40),                /* fatal */
          no_certificate_RESERVED(41),          /* fatal */
          bad_certificate(42),
          unsupported_certificate(43),
          certificate_revoked(44),
          certificate_expired(45),
          certificate_unknown(46),
          illegal_parameter(47),                /* fatal */
          unknown_ca(48),                       /* fatal */
          access_denied(49),                    /* fatal */
          decode_error(50),                     /* fatal */
          decrypt_error(51),                    /* fatal */
          export_restriction_RESERVED(60),      /* fatal */
          protocol_version(70),                 /* fatal */
          insufficient_security(71),            /* fatal */
          internal_error(80),                   /* fatal */
          inappropriate_fallback(86),           /* fatal */
          user_canceled(90),
          no_renegotiation_RESERVED(100),       /* fatal */
          missing_extension(109),               /* fatal */
          unsupported_extension(110),           /* fatal */
          certificate_unobtainable(111),
          unrecognized_name(112),
          bad_certificate_status_response(113), /* fatal */
          bad_certificate_hash_value(114),      /* fatal */
          unknown_psk_identity(115),
          (255)
      } AlertDescription;

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

A.3. Handshake Protocol

```
      enum {
          hello_request_RESERVED(0),
          client_hello(1),
          server_hello(2),
          session_ticket(4),
          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),
          server_configuration(17),
          finished(20),
          (255)
      } HandshakeType;

      struct {
          HandshakeType msg_type;    /* handshake type */
          uint24 length;             /* bytes in message */
          select (HandshakeType) {
              case client_hello:        ClientHello;
              case server_hello:        ServerHello;
              case hello_retry_request: HelloRetryRequest;
              case encrypted_extensions: EncryptedExtensions;
              case server_configuration:ServerConfiguration;
              case certificate:         Certificate;
              case certificate_request: CertificateRequest;
              case certificate_verify:  CertificateVerify;
              case finished:            Finished;
              case session_ticket:      NewSessionTicket;
          } body;
      } Handshake;
```

A.3.1. Hello Messages

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

      enum { null(0), (255) } CompressionMethod;

      struct {
          ProtocolVersion client_version = { 3, 4 };    /* TLS v1.3 */
          Random random;
          SessionID session_id;
          CipherSuite cipher_suites<2..2^16-2>;
          CompressionMethod compression_methods<1..2^8-1>;
          Extension extensions<0..2^16-1>;
```

```
      } ClientHello;

      struct {
          ProtocolVersion server_version;
          Random random;
          CipherSuite cipher_suite;
          select (extensions_present) {
              case false:
                  struct {};
              case true:
                  Extension extensions<0..2^16-1>;
          };
      } ServerHello;

      struct {
          ProtocolVersion server_version;
          CipherSuite cipher_suite;
          NamedGroup selected_group;
          Extension extensions<0..2^16-1>;
      } HelloRetryRequest;

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

      enum {
          supported_groups(10),
          signature_algorithms(13),
          early_data(TBD),
          pre_shared_key(TBD),
          key_share(TBD),
          (65535)
      } ExtensionType;

      opaque psk_identity<0..2^16-1>;

      struct {
          select (Role) {
              case client:
                  psk_identity identities<2..2^16-1>;

              case server:
                  psk_identity identity;
          }
      } PreSharedKeyExtension;

      enum { client_authentication(1), early_data(2),
```

```
            client_authentication_and_data(3), (255) } EarlyDataType;

      struct {
          select (Role) {
              case client:
                  opaque configuration_id<1..2^16-1>;
                  CipherSuite cipher_suite;
                  Extension extensions<0..2^16-1>;
                  opaque context<0..255>;
                  EarlyDataType type;

              case server:
                 struct {};
          }
      } EarlyDataIndication;

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

         enum { (65535) } ConfigurationExtensionType;

         struct {
             ConfigurationExtensionType extension_type;
             opaque extension_data<0..2^16-1>;
         } ConfigurationExtension;

         struct {
             opaque configuration_id<1..2^16-1>;
             uint32 expiration_date;
             NamedGroup group;
             opaque server_key<1..2^16-1>;
             EarlyDataType early_data_type;
             ConfigurationExtension extensions<0..2^16-1>;
         } ServerConfiguration;
```

A.3.1.1. Signature Algorithm Extension

```
      enum {
          none(0),
          md5_RESERVED(1),
          sha1(2),
          sha224_RESERVED(3),
          sha256(4), sha384(5), sha512(6),
          (255)
      } HashAlgorithm;

      enum {
          anonymous_RESERVED(0),
          rsa(1),
          dsa(2),
          ecdsa(3),
          rsapss(4),
          (255)
      } SignatureAlgorithm;

      struct {
          HashAlgorithm hash;
          SignatureAlgorithm signature;
      } SignatureAndHashAlgorithm;

      SignatureAndHashAlgorithm
        supported_signature_algorithms<2..2^16-2>;
```

A.3.1.2. Named Group Extension

```
      enum {
          // Elliptic Curve Groups.
          obsolete_RESERVED (1..22),
          secp256r1 (23), secp384r1 (24), secp521r1 (25),

          // Finite Field Groups.
          ffdhe2048 (256), ffdhe3072 (257), ffdhe4096 (258),
          ffdhe6144 (259), ffdhe8192 (260),

          // Reserved Code Points.
          ffdhe_private_use (0x01FC..0x01FF),
          ecdhe_private_use (0xFE00..0xFEFF),
          obsolete_RESERVED (0xFF01..0xFF02),
          (0xFFFF)
      } NamedGroup;

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

Values within "obsolete_RESERVED" ranges were 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.

A.3.2. Key Exchange Messages

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

struct {
    select (role) {
        case client:
            KeyShareEntry client_shares<4..2^16-1>;

        case server:
            KeyShareEntry server_share;
    }
} KeyShare;

opaque dh_Y<1..2^16-1>;

opaque point <1..2^8-1>;
```

A.3.3. Authentication Messages

```
      opaque ASN1Cert<1..2^24-1>;

      struct {
          ASN1Cert certificate_list<0..2^24-1>;
      } Certificate;

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

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

      struct {
          SignatureAndHashAlgorithm
            supported_signature_algorithms<2..2^16-2>;
          DistinguishedName certificate_authorities<0..2^16-1>;
          CertificateExtension certificate_extensions<0..2^16-1>;
      } CertificateRequest;

      struct {
           digitally-signed struct {
              opaque handshake_hash[hash_length];
           };
      } CertificateVerify;
```

A.3.4. Handshake Finalization Messages

```
      struct {
          opaque verify_data[verify_data_length];
      } Finished;
```

A.3.5. Ticket Establishment

```
     struct {
         uint32 ticket_lifetime_hint;
         opaque ticket<0..2^16-1>;
     } NewSessionTicket;
```

A.4. Cipher Suites

A cipher suite defines a cipher specification supported in TLS and negotiated via hello messages in the TLS handshake. Cipher suite names follow a general naming convention composed of a series of component algorithm names separated by underscores:

```
CipherSuite TLS_KEA_SIGN_WITH_CIPHER_HASH = VALUE;

Component      Contents
TLS            The string "TLS"
KEA            The key exchange algorithm
SIGN           The signature algorithm
WITH           The string "WITH"
CIPHER         The symmetric cipher used for record protection
HASH           The hash algorithm used with HKDF
VALUE          The two byte ID assigned for this cipher suite
```

The "CIPHER" component commonly has sub-components used to designate the cipher name, bits, and mode, if applicable. For example, "AES_256_GCM" represents 256-bit AES in the GCM mode of operation. Cipher suite names that lack a "HASH" value that are defined for use with TLS 1.2 or later use the SHA-256 hash algorithm by default.

The primary key exchange algorithm used in TLS is Ephemeral Diffie-Hellman [DH]. The finite field based version is denoted "DHE" and the elliptic curve based version is denoted "ECDHE". Prior versions of TLS supported non-ephemeral key exchanges, however these are not supported by TLS 1.3.

See the definitions of each cipher suite in its specification document for the full details of each combination of algorithms that is specified.

The following is a list of standards track server-authenticated (and optionally client-authenticated) cipher suites which are currently available in TLS 1.3:

Cipher Suite Name	Value	Specification
TLS_DHE_RSA_WITH_AES_128_GCM_SHA256	{0x00,0x9E}	[RFC5288]
TLS_DHE_RSA_WITH_AES_256_GCM_SHA384	{0x00,0x9F}	[RFC5288]
TLS_DHE_RSA_WITH_AES_128_CCM	{0xC0,0x9E}	[RFC6655]
TLS_DHE_RSA_WITH_AES_256_CCM	{0xC0,0x9F}	[RFC6655]
TLS_DHE_RSA_WITH_AES_128_CCM_8	{0xC0,0xA2}	[RFC6655]
TLS_DHE_RSA_WITH_AES_256_CCM_8	{0xC0,0xA3}	[RFC6655]
TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305	{TBD,TBD}	[I-D.ietf-tls-chacha20-poly1305]
TLS_ECDHE_ECDSA_WITH_CHACHA20_POLY1305	{TBD,TBD}	[I-D.ietf-tls-chacha20-poly1305]
TLS_DHE_RSA_WITH_CHACHA20_POLY1305	{TBD,TBD}	[I-D.ietf-tls-chacha20-poly1305]

[[TODO: CHACHA20_POLY1305 cipher suite IDs are TBD.]]

The following is a list of non-standards track server-authenticated (and optionally client-authenticated) cipher suites which are

currently available in TLS 1.3:

Cipher Suite Name	Value	Specification
TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256	{0xC0,0x2B}	[RFC5289]
TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384	{0xC0,0x2C}	[RFC5289]
TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256	{0xC0,0x2F}	[RFC5289]
TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384	{0xC0,0x30}	[RFC5289]
TLS_ECDHE_ECDSA_WITH_AES_128_CCM	{0xC0,0xAC}	[RFC7251]
TLS_ECDHE_ECDSA_WITH_AES_256_CCM	{0xC0,0xAD}	[RFC7251]
TLS_ECDHE_ECDSA_WITH_AES_128_CCM_8	{0xC0,0xAE}	[RFC7251]
TLS_ECDHE_ECDSA_WITH_AES_256_CCM_8	{0xC0,0xAF}	[RFC7251]
TLS_DHE_RSA_WITH_ARIA_128_GCM_SHA256	{0xC0,0x52}	[RFC6209]
TLS_DHE_RSA_WITH_ARIA_256_GCM_SHA384	{0xC0,0x53}	[RFC6209]
TLS_ECDHE_ECDSA_WITH_ARIA_128_GCM_SHA256	{0xC0,0x5C}	[RFC6209]
TLS_ECDHE_ECDSA_WITH_ARIA_256_GCM_SHA384	{0xC0,0x5D}	[RFC6209]
TLS_ECDHE_RSA_WITH_ARIA_128_GCM_SHA256	{0xC0,0x60}	[RFC6209]
TLS_ECDHE_RSA_WITH_ARIA_256_GCM_SHA384	{0xC0,0x61}	[RFC6209]
TLS_DHE_RSA_WITH_CAMELLIA_128_GCM_SHA256	{0xC0,0x7C}	[RFC6367]
TLS_DHE_RSA_WITH_CAMELLIA_256_GCM_SHA384	{0xC0,0x7D}	[RFC6367]
TLS_ECDHE_ECDSA_WITH_CAMELLIA_128_GCM_SHA256	{0xC0,0x86}	[RFC6367]
TLS_ECDHE_ECDSA_WITH_CAMELLIA_256_GCM_SHA384	{0xC0,0x87}	[RFC6367]
TLS_ECDHE_RSA_WITH_CAMELLIA_128_GCM_SHA256	{0xC0,0x8A}	[RFC6367]
TLS_ECDHE_RSA_WITH_CAMELLIA_256_GCM_SHA384	{0xC0,0x8B}	[RFC6367]

ECDHE AES GCM is not yet standards track, however it is already widely deployed.

Note: In the case of the CCM mode of AES, two variations exist: "CCM_8" which uses an 8-bit authentication tag and "CCM" which uses a 16-bit authentication tag. Both use the default hash, SHA-256.

All cipher suites in this section are specified for use with both TLS 1.2 and TLS 1.3, as well as the corresponding versions of DTLS. (see Appendix C)

New cipher suite values are assigned by IANA as described in Section 11.

A.4.1. Unauthenticated Operation

Previous versions of TLS offered explicitly unauthenticated cipher suites base on anonymous Diffie-Hellman. These cipher suites have been deprecated in TLS 1.3. However, it is still possible to negotiate cipher suites that do not provide verifiable server authentication by serveral 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 are 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 channel bindings [RFC5929]. [[NOTE: TLS 1.3 needs a new channel binding definition that has not yet been defined.]] 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.

A.5. The Security Parameters

These security parameters are determined by the TLS Handshake Protocol and provided as parameters to the TLS record layer in order to initialize a connection state. SecurityParameters includes:

```
enum { server, client } ConnectionEnd;

enum { tls_kdf_sha256, tls_kdf_sha384 } KDFAlgorithm;

enum { aes_gcm } RecordProtAlgorithm;

/* The algorithms specified in KDFAlgorithm and
   RecordProtAlgorithm may be added to. */

struct {
    ConnectionEnd          entity;
    KDFAlgorithm           kdf_algorithm;
    RecordProtAlgorithm    record_prot_algorithm;
    uint8                  enc_key_length;
    uint8                  iv_length;
    opaque                 hs_master_secret[48];
    opaque                 master_secret[48];
    opaque                 client_random[32];
    opaque                 server_random[32];
} SecurityParameters;
```

A.6. Changes to RFC 4492

RFC 4492 [RFC4492] adds Elliptic Curve cipher suites to TLS. This document changes some of the structures used in that document. This section details the required changes for implementors of both RFC 4492 and TLS 1.2. Implementors of TLS 1.2 who are not implementing RFC 4492 do not need to read this section.

This document adds a "signature_algorithm" field to the digitally-signed element in order to identify the signature and digest

algorithms used to create a signature. This change applies to digital signatures formed using ECDSA as well, thus allowing ECDSA signatures to be used with digest algorithms other than SHA-1, provided such use is compatible with the certificate and any restrictions imposed by future revisions of [RFC5280].

As described in Section 6.3.4, the restrictions on the signature algorithms used to sign certificates are no longer tied to the cipher suite. Thus, the restrictions on the algorithm used to sign certificates specified in Sections 2 and 3 of RFC 4492 are also relaxed. As in this document, the restrictions on the keys in the end-entity certificate remain.

Appendix B. Implementation Notes

The TLS protocol cannot prevent many common security mistakes. This section provides several recommendations to assist implementors.

B.1. Random Number Generation and Seeding

TLS requires a cryptographically secure pseudorandom number generator (PRNG). Care must be taken in designing and seeding PRNGs. PRNGs based on secure hash operations, most notably SHA-256, are acceptable, but cannot provide more security than the size of the random number generator state.

To estimate the amount of seed material being produced, add the number of bits of unpredictable information in each seed byte. For example, keystroke timing values taken from a PC compatible 18.2 Hz timer provide 1 or 2 secure bits each, even though the total size of the counter value is 16 bits or more. Seeding a 128-bit PRNG would thus require approximately 100 such timer values.

[RFC4086] provides guidance on the generation of random values.

B.2. Certificates and Authentication

Implementations are responsible for verifying the integrity of certificates and should generally support certificate revocation messages. Certificates should always be verified to ensure proper signing by a trusted Certificate Authority (CA). The selection and addition of trusted CAs should be done very carefully. Users should be able to view information about the certificate and root CA.

B.3. Cipher Suite Support

TLS supports a range of key sizes and security levels, including some that provide no or minimal security. A proper implementation will probably not support many cipher suites. Applications SHOULD also enforce minimum and maximum key sizes. For example, certificate chains containing keys or signatures weaker than 2048-bit RSA or 224-bit ECDSA are not appropriate for secure applications. See also Appendix C.3.

B.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 5.2.1)? 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 certificate request handshake messages can be large enough to require fragmentation.

- Do you ignore the TLS record layer version number in all TLS records? (see Appendix C)

- 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 C)

- Do you handle TLS extensions in ClientHello correctly, including omitting the extensions field completely?

- 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 6.3.9)?

- 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?

Cryptographic details:

- What countermeasures do you use to prevent timing attacks against RSA signing operations [TIMING]?

- When verifying RSA signatures, do you accept both NULL and missing parameters (see Section 4.9)? Do you verify that the RSA padding doesn't have additional data after the hash value? [FI06]

- When using Diffie-Hellman key exchange, do you correctly strip leading zero bytes from the negotiated key (see Section 7.2.2)?

- Does your TLS client check that the Diffie-Hellman parameters sent by the server are acceptable (see Appendix D.1.1.1)?

- Do you use a strong and, most importantly, properly seeded random number generator (see Appendix B.1) Diffie-Hellman private values, the ECDSA "k" parameter, and other security-critical values?

Appendix C. 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.record_version & TLSCiphertext.record_version) As of TLS 1.3, this field is deprecated and its value MUST be ignored by all implementations. Version negotiation is performed using only the handshake versions. (ClientHello.client_version & ServerHello.server_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 certificate chains based on the expectations in this

document, even when handling prior TLS versions' handshakes. (see Section 6.3.4)

C.1. Negotiating with an older server

A TLS 1.3 client who wishes to negotiate with such older servers will send a normal TLS 1.3 ClientHello containing { 3, 4 } (TLS 1.3) in ClientHello.client_version. If the server does not support this version 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 resuming a session SHOULD initiate the connection using the version that was previously negotiated.

If the version chosen by the server is not supported by the client (or not acceptable), the client MUST send a "protocol_version" alert message and close the connection.

If a TLS server receives a ClientHello containing a version number greater than the highest version supported by the server, it MUST reply according to the highest version supported by the server.

Some legacy server implementations are known to not implement the TLS specification properly and might abort connections upon encountering TLS extensions or versions which it is 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.

C.2. Negotiating with an older client

A TLS server can also receive a ClientHello containing a version number smaller than the highest supported version. If the server wishes to negotiate with old clients, it will proceed as appropriate for the highest version supported by the server that is not greater than ClientHello.client_version. For example, if the server supports TLS 1.0, 1.1, and 1.2, and client_version is TLS 1.0, the server will proceed with a TLS 1.0 ServerHello. If the server only supports versions greater than client_version, it MUST send a "protocol_version" alert message and close the connection.

Note that earlier versions of TLS did not clearly specify the record layer version number value in all cases (TLSPlaintext.record_version). Servers will receive various TLS 1.x versions in this field, however its value MUST always be ignored.

C.3. Backwards Compatibility Security Restrictions

If an implementation negotiates use of TLS 1.2, then negotiation of cipher suites also supported by TLS 1.3 SHOULD be preferred, if 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 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 or accept any records with a version less than { 3, 0 }.

The security of SSL 3.0 [SSL3] is considered insufficient for the reasons enumerated in [RFC7568], and MUST NOT be negotiated for any reason.

Implementations MUST NOT send a ClientHello.client_version or ServerHello.server_version set to { 3, 0 } or less. Any endpoint receiving a Hello message with ClientHello.client_version or ServerHello.server_version set to { 3, 0 } MUST respond with a "protocol_version" alert message and close the connection.

Implementations MUST NOT use the Truncated HMAC extension, defined in Section 7 of [RFC6066], as it is not applicable to AEAD ciphers and has been shown to be insecure in some scenarios.

Appendix D. Security Analysis

[[TODO: The entire security analysis needs a rewrite.]]

The TLS protocol is designed to establish a secure connection between a client and a server communicating over an insecure channel. This document makes several traditional assumptions, including that

attackers have substantial computational resources and cannot obtain secret information from sources outside the protocol. Attackers are assumed to have the ability to capture, modify, delete, replay, and otherwise tamper with messages sent over the communication channel. This appendix outlines how TLS has been designed to resist a variety of attacks.

D.1. Handshake Protocol

The TLS Handshake Protocol is responsible for selecting a cipher spec and generating a master secret, which together comprise the primary cryptographic parameters associated with a secure session. The TLS Handshake Protocol can also optionally authenticate parties who have certificates signed by a trusted certificate authority.

D.1.1. Authentication and Key Exchange

TLS supports three authentication modes: authentication of both parties, server authentication with an unauthenticated client, and total anonymity. Whenever the server is authenticated, the channel is secure against man-in-the-middle attacks, but completely anonymous sessions are inherently vulnerable to such attacks. Anonymous servers cannot authenticate clients. If the server is authenticated, its certificate message must provide a valid certificate chain leading to an acceptable certificate authority. Similarly, authenticated clients must supply an acceptable certificate to the server. Each party is responsible for verifying that the other's certificate is valid and has not expired or been revoked.

[[TODO: Rewrite this because the master_secret is not used this way any more after Hugo's changes.]] The general goal of the key exchange process is to create a master_secret known to the communicating parties and not to attackers (see Section 7.1). The master_secret is required to generate the Finished messages and record protection keys (see Section 6.3.8 and Section 7.2). By sending a correct Finished message, parties thus prove that they know the correct master_secret.

D.1.1.1. Diffie-Hellman Key Exchange with Authentication

When Diffie-Hellman key exchange is used, the client and server use the KeyShare extension to send temporary Diffie-Hellman parameters. The signature in the certificate verify message (if present) covers the entire handshake up to that point and thus attests the certificate holder's desire to use the the ephemeral DHE keys.

Peers SHOULD validate each other's public key Y (dh_Ys offered by the server or DH_Yc offered by the client) by ensuring that 1 < Y < p-1.

This simple check ensures that the remote peer is properly behaved and isn't forcing the local system into a small subgroup.

Additionally, using a fresh key for each handshake provides Perfect Forward Secrecy. Implementations SHOULD generate a new X for each handshake when using DHE cipher suites.

D.1.2. Version Rollback Attacks

Because TLS includes substantial improvements over SSL Version 2.0, attackers may try to make TLS-capable clients and servers fall back to Version 2.0. This attack can occur if (and only if) two TLS-capable parties use an SSL 2.0 handshake. (See also Appendix C.3.)

Although the solution using non-random PKCS #1 block type 2 message padding is inelegant, it provides a reasonably secure way for Version 3.0 servers to detect the attack. This solution is not secure against attackers who can brute-force the key and substitute a new ENCRYPTED-KEY-DATA message containing the same key (but with normal padding) before the application-specified wait threshold has expired. Altering the padding of the least-significant 8 bytes of the PKCS padding does not impact security for the size of the signed hashes and RSA key lengths used in the protocol, since this is essentially equivalent to increasing the input block size by 8 bytes.

D.1.3. Detecting Attacks Against the Handshake Protocol

An attacker might try to influence the handshake exchange to make the parties select different encryption algorithms than they would normally choose.

For this attack, an attacker must actively change one or more handshake messages. If this occurs, the client and server will compute different values for the handshake message hashes. As a result, the parties will not accept each others' Finished messages. Without the static secret, the attacker cannot repair the Finished messages, so the attack will be discovered.

D.2. Protecting Application Data

The shared secrets are hashed with the handshake transcript to produce unique record protection secrets for each connection.

Outgoing data is protected using an AEAD algorithm before transmission. The authentication data includes the sequence number, message type, message length, and the message contents. The message type field is necessary to ensure that messages intended for one TLS record layer client are not redirected to another. The sequence

number ensures that attempts to delete or reorder messages will be detected. Since sequence numbers are 64 bits long, they should never overflow. Messages from one party cannot be inserted into the other's output, since they use independent keys.

D.3. Denial of Service

TLS is susceptible to a number of denial-of-service (DoS) attacks. In particular, an attacker who initiates a large number of TCP connections can cause a server to consume large amounts of CPU doing asymmetric crypto operations. However, because TLS is generally used over TCP, it is difficult for the attacker to hide his point of origin if proper TCP SYN randomization is used [RFC1948] by the TCP stack.

Because TLS runs over TCP, it is also susceptible to a number of DoS attacks on individual connections. In particular, attackers can forge RSTs, thereby terminating connections, or forge partial TLS records, thereby causing the connection to stall. These attacks cannot in general be defended against by a TCP-using protocol. Implementors or users who are concerned with this class of attack should use IPsec AH [RFC4302] or ESP [RFC4303].

D.4. Final Notes

For TLS to be able to provide a secure connection, both the client and server systems, keys, and applications must be secure. In addition, the implementation must be free of security errors.

The system is only as strong as the weakest key exchange and authentication algorithm supported, and only trustworthy cryptographic functions should be used. Short public keys and anonymous servers should be used with great caution. Implementations and users must be careful when deciding which certificates and certificate authorities are acceptable; a dishonest certificate authority can do tremendous damage.

Appendix E. 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 F. 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

Benjamin Beurdouche

Karthikeyan Bhargavan (co-author of [RFC7627])
INRIA
karthikeyan.bhargavan@inria.fr

Simon Blake-Wilson (co-author of [RFC4492])
BCI
sblakewilson@bcisse.com

Nelson Bolyard
Sun Microsystems, Inc.
nelson@bolyard.com (co-author of [RFC4492])

Ran Canetti
IBM
canetti@watson.ibm.com

Pete Chown
Skygate Technology Ltd
pc@skygate.co.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

Anil Gangolli
anil@busybuddha.org

David M. Garrett

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

Daniel Kahn Gillmor
ACLU
dkg@fifthhorseman.net

Phil Karlton (co-author of SSL 3.0)

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

Ilari Liusvaara
ilari.liusvaara@elisanet.fi

Jan Mikkelsen
Transactionware

janm@transactionware.com

Bodo Moeller (co-author of [RFC4492])
Google
bodo@openssl.org

Erik Nygren
Akamai Technologies
erik+ietf@nygren.org

Magnus Nystrom
RSA Security
magnus@rsasecurity.com

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

Jim Roskind
Netscape Communications
jar@netscape.com

Michael Sabin

Dan Simon
Microsoft, Inc.
dansimon@microsoft.com

Bjoern Tackmann
University of California, San Diego
btackmann@eng.ucsd.edu

Martin Thomson
Mozilla
mt@mozilla.com

Tom Weinstein

Hoeteck Wee
Ecole Normale Superieure, Paris
hoeteck@alum.mit.edu

Tim Wright
Vodafone
timothy.wright@vodafone.com

Author's Address

Eric Rescorla
RTFM, Inc.

EMail: ekr@rtfm.com