DECLARATION

I, Alexa Morris, based on my personal knowledge and information, hereby declare as follows:

1.   I am Managing Director of the IETF Administration LLC and have held that position since the LLC was formed in August 2018. Prior to that, starting on January 1, 2008, I was the Executive Director of the Internet Engineering Task Force, which was an activity of the Internet Society. Since the business of IETF did not change in any materially relevant manner with the formation of the LLC, I will collectively refer to both the activity and the LLC as IETF.

2.   One of my responsibilities with IETF has been to act as the custodian of Internet-Drafts and records relating to Internet-Drafts. I am familiar with the record keeping practices relating to Internet-Drafts, including the creation and maintenance of such records.

3.   I hereby declare that all statements made herein are of my own knowledge and information contained in the business records of IETF and are true, and that all statements made on information and belief are believed to be true; and further that these statements were made with the knowledge that willful false statements may be punishable by fine or imprisonment, or both, under Section 1001 of Title 18 of the United States Code.

4.   If depositions regarding the information in this declaration are required, the deposition should be taken by phone or videoconference or, if it must be in person, should be in California.

5.   Since 1998, it has been the regular practice of the IETF to publish Internet-Drafts and make them available to the public on its website at www.ietf.org (the IETF website). The IETF maintains copies of Internet-Drafts in the ordinary course of its regularly conducted activities.

6.   Any Internet-Draft published on the IETF website was reasonably accessible to the public and was disseminated or otherwise available to the extent that persons interested and ordinarily skilled in the subject matter or art exercising reasonable diligence could have located it. In particular, the Internet-Drafts were indexed and searchable on the IETF website.

7.   Internet-Drafts are posted to an IETF online directory. When an Internet-Draft is published, an announcement of its publication that describes the Internet-Draft is disseminated. Typically, that dated announcement is made within 24 hours of the publication of the Internet-Draft. The announcement is kept in the IETF email archive and the date is affixed automatically.

8.   The records of posting the Internet-Drafts in the IETF online repository are kept in the course of the IETF's regularly conducted activity and ordinary course of business. The records are made pursuant to established procedures and are relied upon by the IETF in the performance of its functions.

9.   It is the regular practice of the IETF to make and keep the records in the online repository.

10.   Exhibit 1 is a true and correct copy of an announcement of the publication of draft-ietf-tls-tls13-07, titled "The Transport Layer Security (TLS) Protocol Version 1.3." I have determined that an announcement of the publication of this Internet-Draft was made on July 8, 2015. Therefore, based on the normal practice of the IETF, that Internet-Draft was reasonably available to the public within 24 hours of that announcement. At that time, the Internet-Draft would have been disseminated or otherwise available to the extent that persons interested and ordinarily skilled in the subject matter or art, exercising reasonable diligence, could have located it.

2

Pursuant to Section 1746 of Title 28 of United States Code, I declare under penalty of perjury under the laws of the United States of America that the foregoing is true and correct and that the foregoing is based upon personal knowledge and information and is believed to be true.

Date: _March 27, 2023_

By: _____
Alexa Morris

4884-3111-2793

           The Transport Layer Security (TLS) Protocol Version 1.3
                         draft-ietf-tls-tls13-07

Abstract

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

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

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

RFC EDITOR: PLEASE REMOVE THE FOLLOWING PARAGRAPH The source for this
   draft is maintained in GitHub. Suggested changes should be submitted
   as pull requests at https://github.com/tlswg/tls13-spec.
   Instructions are on that page as well. Editorial changes can be
   managed in GitHub, but any substantive change should be discussed on
   the TLS mailing list.

   The primary goal of the TLS protocol is to provide privacy and data
   integrity between two communicating applications. The 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], etc.). 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 Record Protocol can also be used
      without encryption, i.e., in integrity-only modes.

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

   - The Record Protocol can operate in an insecure mode but is
      generally only used in this mode while another protocol is using
      the 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], DSA [DSS], etc.).
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-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.

- Reworked handshake to provide 1-RTT mode.

- Remove custom DHE groups.

- Removed support for compression.

- Removed support for static RSA and DH key exchange.

- Removed 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 protocol 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 6.3.1.4.1 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–PKCS1–v1_5 signature scheme defined in [RFC3447].
   As discussed in [RFC3447], the DigestInfo MUST be DER–encoded
[X680]
   [X690]. For hash algorithms without parameters (which includes
SHA–
   1), the DigestInfo.AlgorithmIdentifier.parameters field MUST be
NULL,
   but implementations MUST accept both without parameters and with
NULL

   parameters. Note that earlier versions of TLS used a different RSA
   signature scheme that did not include a DigestInfo encoding.

   In DSA, the 20 bytes of the SHA–1 hash are run directly through the
   Digital Signing Algorithm with no additional hashing. This
produces

two values, r and s. The DSA signature is an opaque vector, as
above, the contents of which are the DER encoding of:

```
Dss-Sig-Value ::= SEQUENCE {
    r INTEGER,
    s INTEGER
}
```

Note: In current terminology, DSA refers to the Digital Signature
Algorithm and DSS refers to the NIST standard. In the original SSL
and TLS specs, "DSS" was used universally. This document uses
"DSA"
to refer to the algorithm, "DSS" to refer to the standard, and it
uses "DSS" in the code point definitions for historical continuity.

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 default hash function is SHA-1 [SHS].
However, an alternative hash function, such as one of the new SHA
hash functions specified in FIPS 180-2 may be used instead if the
certificate containing the EC public key explicitly requires use of
another hash function. (The mechanism for specifying the required
hash function has not been standardized, but this provision
anticipates such standardization and obviates the need to update
this
document in response. Future PKIX RFCs may choose, for example, to
specify the hash function to be used with a public key in the
parameters field of subjectPublicKeyInfo.) [[OPEN ISSUE: This
needs
updating per 4492-bis https://github.com/tlswg/tls13-spec/issues/
59]]

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.

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:

```
      202020202020202020202020202020202020202020202020202020202020202020
      202020202020202020202020202020202020202020202020202020202020202020
      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.

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 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 record protocol are described in this
   document: the 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
   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 type and length of a record are not
protected
   by encryption. If this information is itself sensitive,
application
   designers may wish to take steps (padding, cover traffic) to
minimize
   information leakage.

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. It is possible to have
      AEAD algorithms which do not provide any confidentiality and
      Section 5.2.2 defines a special NULL_NULL AEAD algorithm for use
      in the initial handshake). 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).

        struct {
            uint8 major;
            uint8 minor;
        } ProtocolVersion;

        enum {
            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;
```

   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 D.

   Implementations MUST NOT send zero-length fragments of Handshake or
   Alert types. Zero-length fragments of Application data MAY be sent
   as they are potentially useful as a traffic analysis
countermeasure.

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

   type
      The type field is identical to TLSPlaintext.type.

   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 + 2048.

fragment
    The AEAD encrypted form of TLSPlaintext.fragment.

    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 TLSPlaintext.fragment.

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

    additional_data = seq_num + TLSPlaintext.type +
                      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 will generally be larger than
   TLSPlaintext.length, but by an amount that varies with the AEAD
   cipher. Since the ciphers might incorporate padding, the amount of
   overhead could vary with different TLSPlaintext.length values.
Each
   AEAD cipher MUST NOT produce an expansion of greater than 1024
bytes.
   Symbolically,

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

   [[OPEN ISSUE: Reduce these values? https://github.com/tlswg/tls13-
   spec/issues/55]]

   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:

      TLSPlaintext.fragment = AEAD-Decrypt(write_key, nonce,
                                           AEADEncrypted,
                                           additional_data)

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

   As a special case, we define the NULL_NULL AEAD cipher which is
   simply the identity operation and thus provides no security. This
   cipher MUST ONLY be used with the initial TLS_NULL_WITH_NULL_NULL
   cipher suite.

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 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 */
         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 */
         user_canceled(90),
         no_renegotiation(100),               /* fatal */
```

```
            unsupported_extension(110),        /* fatal */
            (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. Either party may
   initiate the exchange of closing messages.

   close_notify
      This message notifies the recipient that the sender will not
send
      any more messages on this connection. Note that as of TLS 1.1,
      failure to properly close a connection no longer requires that a
      session not be resumed. This is a change from TLS 1.0 to
conform
      with widespread implementation practice.

   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.

   Unless some other fatal alert has been transmitted, each party is
   required to send a "close_notify" alert before closing the write
side
   of the connection. The other party MUST respond with a
   "close_notify" alert of its own and close down the connection
   immediately, discarding any pending writes. It is not required for
   the initiator of the close to 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 the other
   party. 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
   "no_renegotiation" alert that it is not willing to accept), it
SHOULD
   send a fatal alert to terminate the connection. Given this, the
   sending party 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 peer 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 message is used for all deprotection
failures.
      This message is always fatal and should never be observed in
      communication between proper implementations (except when
messages
      were corrupted in the network).

   decryption_failed_RESERVED
      This alert was used in some earlier versions of TLS, and may
have
      permitted certain attacks against the CBC mode [CBCATT]. It
MUST
      NOT be sent by compliant implementations. This message is
always
      fatal.

   record_overflow
      A TLSCiphertext record was received that had a length more than
      2^14+2048 bytes, or a record decrypted to a TLSPlaintext record

with more than 2^14 bytes. This message is always fatal and
should never be observed in communication between proper
implementations (except when messages were corrupted in the
network).

   decompression_failure_RESERVED
      This alert was used in previous versions of TLS. TLS 1.3 does
not
      include compression and TLS 1.3 implementations MUST NOT send
this
      alert when in TLS 1.3 mode. This message is always fatal.

   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 message is always
      fatal.

   no_certificate_RESERVED
      This alert was used in SSL 3.0 but not any version of TLS. It
      MUST NOT be sent by compliant implementations. This message 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 message 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
      message is always fatal.

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

   export_restriction_RESERVED
      This alert was used in some earlier versions of TLS. It MUST
NOT
      be sent by compliant implementations. This message 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 message 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 message 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 message is always fatal.

   user_canceled
      This handshake is being canceled for some reason unrelated to a
      protocol failure. If the 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 message is generally a warning.

   no_renegotiation
      Sent by the client in response to a HelloRequest or by the
server
      in response to a ClientHello after initial handshaking.
Versions
      of TLS prior to TLS 1.3 supported renegotiation of a previously
      established connection; TLS 1.3 removes this feature. This
      message is always fatal.

   unsupported_extension
      sent by clients that receive an extended ServerHello containing
an
      extension that they did not put in the corresponding
ClientHello.
      This message is always fatal.

   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:

```
   Client                                            Server

   ClientHello
     + ClientKeyShare        -------->
                                                     ServerHello
                                                  ServerKeyShare*
                                             {EncryptedExtensions}
                                            {ServerConfiguration*}
                                                     {Certificate*}
                                             {CertificateRequest*}
                                              {CertificateVerify*}
                             <--------                 {Finished}
   {Certificate*}
   {CertificateVerify*}
   {Finished}                -------->
   [Application Data]        <------->         [Application Data]
```
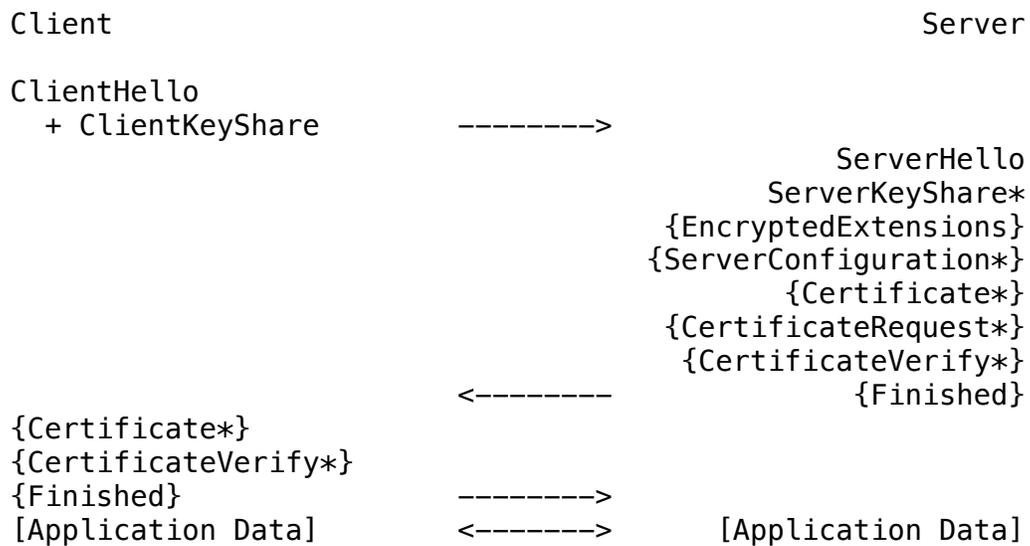
* Indicates optional or situation-dependent messages that are not always sent.

{} 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 ClientKeyShare extension
Section 6.3.1.5.

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]

ServerKeyShare
    the server's ephemeral Diffie-Hellman Share which must be in the
    same group as one of the shares offered by the client. This
    message will be omitted if DH is not in use (i.e., a pure PSK
    cipher suite is selected). The ClientKeyShare and
ServerKeyShare
    are used together to derive the Static Secret and Ephemeral
Secret
    (in this mode they are the same). [Section 6.3.2]

ServerConfiguration

    supplies a configuration for a future handshake (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 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 peers. 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. The 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 ClientKeyShare (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
   ClientKeyShare, as shown in Figure 2:

```
           Client                                          Server

           ClientHello
             + ClientKeyShare        -------->
                                      <--------      HelloRetryRequest

           ClientHello
             + ClientKeyShare        -------->
                                                            ServerHello
                                                         ServerKeyShare
                                                  {EncryptedExtensions*}
                                                  {ServerConfiguration*}
                                                          {Certificate*}
                                                   {CertificateRequest*}
                                                    {CertificateVerify*}
                                      <--------             {Finished}
           {Certificate*}
           {CertificateVerify*}
           {Finished}                -------->
           [Application Data]        <------->      [Application Data]
```
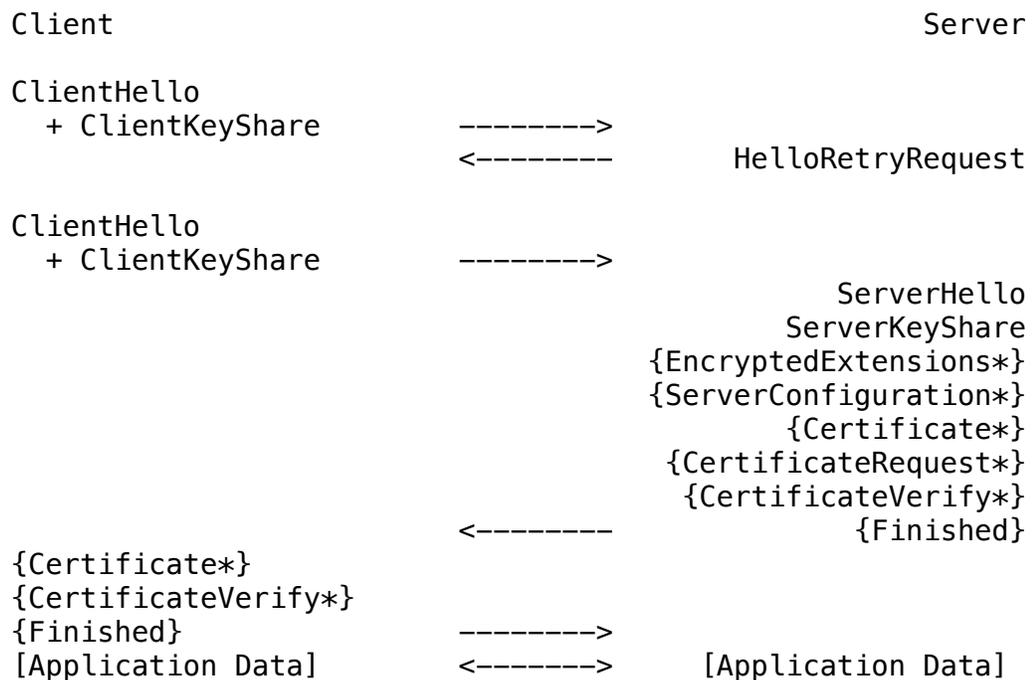
Figure 2: Message flow for a full handshake with mismatched
parameters

[[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 fatal alert.

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

6.2.2. Cached Server Configuration

   During an initial handshake, the server can provide a
   ServerConfiguration message containing a long-term (EC)DH share.
On
   future connections, the client can indicate to the server that it
   knows the server's configuration and if that configuration is valid
   the server can omit both the Certificate or CertificateVerify
message
   (provided that a new configuration is not supplied in this
   handshake).

   When a known configuration is used, the server's long-term DHE key
is
   combined with the client's ClientKeyShare to produce SS. ES is
   computed as above. This optimization allows the server to amortize
   the transmission of these messages and the server's signature over
   multiple handshakes, thus reducing the server's computational cost
   for cipher suites where signatures are slower than key agreement,
   principally RSA signatures paired with ECDHE.

6.2.3. Zero-RTT Exchange

When a cached ServerConfiguration is used, the client can also send
application data as well as its Certificate and CertificateVerify
(if
client authentication is requested) on its first flight, thus
reducing handshake latency, as shown below.

```
    Client                                              Server

    ClientHello
      + ClientKeyShare
    (Certificate*)
    (CertificateVerify*)
    (Application Data)          -------->
                                                        ServerHello
                                                      ServerKeyShare
                                <--------               {Finished}
    {Finished}                  -------->

    [Application Data]          <------->         [Application Data]
```
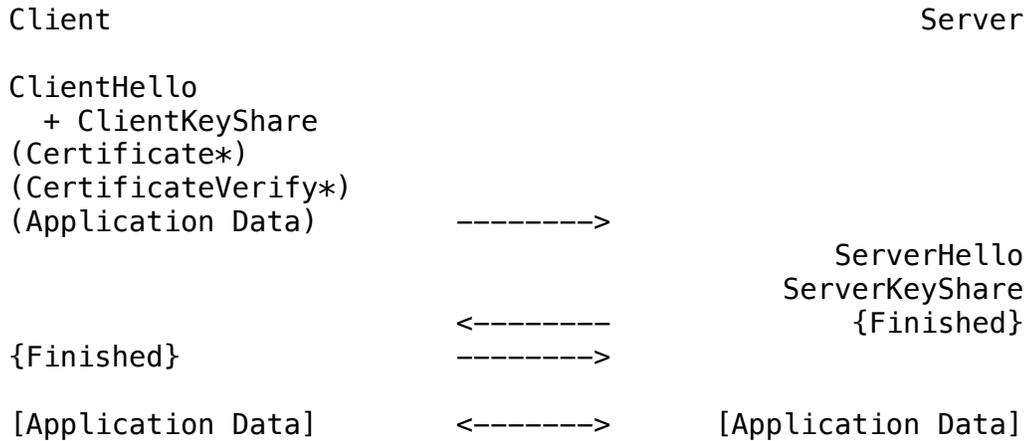
() Indicates messages protected using keys derived from the static
secret.

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 complete, 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 6.3.1.5.5.1), 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).

6.2.4. 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 ciphersuites 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:

```
        Client                                              Server

    Initial Handshake:

        ClientHello
          + ClientKeyShare        -------->
                                                          ServerHello
                                                        ServerKeyShare
                                                   {EncryptedExtensions}
                                                   {ServerConfiguration*}
                                                         {Certificate*}
                                                   {CertificateRequest*}
                                                    {CertificateVerify*}
                                  <--------                  {Finished}
        {Certificate*}
        {CertificateVerify*}
        {Finished}               -------->
                                  <--------          [NewSessionTicket]
        [Application Data]        <------->          [Application Data]


    Subsequent Handshake:
        ClientHello
          + ClientKeyShare,
            PreSharedKeyExtension -------->
                                                          ServerHello
                                                   +PreSharedKeyExtension
                                  <--------                  {Finished}
```
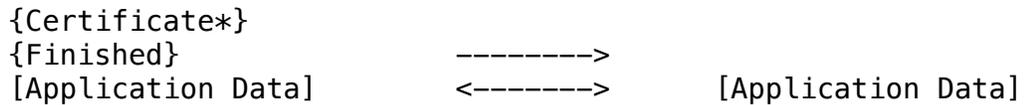
```
        {Certificate*}
        {Finished}                 -------->
        [Application Data]         <------->          [Application Data]

              Figure 4: Message flow for resumption and PSK
```

   Note that the client supplies a ClientKeyShare to the server as
well,
   which allows the server to decline resumption and fall back to a
full
   handshake. However, because 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.

   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 {
          reserved(0), client_hello(1), server_hello(2),
          session_ticket(4), hello_retry_request(6),
          server_key_share(7), certificate(11), reserved(12),
          certificate_request(13), server_configuration(14),
          certificate_verify(15), reserved(16), 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 server_key_share:   ServerKeyShare;
             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 handshake protocol messages are presented below in the order they
MUST be sent; sending handshake messages in an unexpected order
results in a 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 ClientKeyShare. In that case, the client
MUST
send the same ClientHello (without modification) except
including
a new ClientKeyShare. [[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
"no_renegotiation" alert.

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.

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>;
         select (extensions_present) {
             case false:
                 struct {};
             case true:
                 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 D 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 6.2.4). 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 6.3.1.4.

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 ClientKeyShare extension was acceptable. If
the
        client proposed groups are not acceptable by the server, it will
        respond with an "insufficient_security" fatal alert.

    Structure of this message:

        struct {
            ProtocolVersion server_version;
            Random random;
            uint8 session_id_len; // Must be 0.
            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 D for details about backward compatibility.)

random
    This structure is generated by the server and MUST be generated
    independently of the ClientHello.random.

session_id_len
    A single 0 value for backward compatible formatting. [[OPEN
    ISSUE: Should we remove?]]

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:

      The server will 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
      ClientKeyShare 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 does not identify a group which was
   not in the original ClientHello. If it was present, 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
   ClientKeyShare extension to the server. The ClientKeyShare MUST
   append a new ClientKeyShareOffer which is consistent with the
   "selected_group" field to the groups in the original
ClientKeyShare.

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

6.3.1.4. Hello Extensions

   The extension format is:

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

       enum {
           signature_algorithms(13),
           early_data(TBD),
           supported_groups(TBD),
           known_configuration(TBD),
           pre_shared_key(TBD)
           client_key_shares(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.

6.3.1.4.1. 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. The "extension_data" field of this extension
contains a "supported_signature_algorithms" value.

```
enum {
    none(0), md5(1), sha1(2), sha224(3), sha256(4), sha384(5),
    sha512(6), (255)
} HashAlgorithm;

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

struct {
      HashAlgorithm hash;
      SignatureAlgorithm signature;
} SignatureAndHashAlgorithm;

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

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., DSA with SHA-1, but not SHA-
256), algorithms here are listed in pairs.

hash
    This field indicates the hash algorithm which may be used. The
    values indicate support for unhashed data, MD5 [RFC1321], SHA-1,
    SHA-224, 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.

signature
    This field indicates the signature algorithm that may be used.
    The values indicate anonymous signatures, RSASSA-PKCS1-v1_5
    [RFC3447] and DSA [DSS], and ECDSA [ECDSA], respectively. The
    "anonymous" value is meaningless in this context but used in
    Section 6.3.2. It MUST NOT appear in this extension.

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 describe the appropriate
   rules.

   If the client supports only the default hash and signature algorithms
   (listed in this section), it MAY omit the signature_algorithms
   extension. If the client does not support the default algorithms, or
   supports other hash and signature algorithms (and it is willing to
   use them for verifying messages sent by the server, i.e., server
   certificates and server key share), it MUST send the
   signature_algorithms extension, listing the algorithms it is willing
   to accept.

   If the client does not send the signature_algorithms extension, the
   server MUST do the following:

   - If the negotiated key exchange algorithm is one of (DHE_RSA,
      ECDHE_RSA), behave as if client had sent the value {sha1,rsa}.

   - If the negotiated key exchange algorithm is DHE_DSS, behave as if
      the client had sent the value {sha1,dsa}.

   - If the negotiated key exchange algorithm is ECDHE_ECDSA, behave as
      if the client had sent value {sha1,ecdsa}.

   Note: This extension is not meaningful for TLS versions prior to 1.2.
   Clients MUST NOT offer it if they are offering prior versions.
   However, even if clients do offer it, the rules specified in
   [RFC6066] require servers to ignore extensions they do not
   understand.

   Servers MUST NOT send this extension. TLS servers MUST support

receiving this extension.

6.3.1.4.2. 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].

The "extension_data" field of this extension SHALL contain a
"NamedGroupList" value:

```
enum {
    // Elliptic Curve Groups.
    sect163k1 (1), sect163r1 (2), sect163r2 (3),
    sect193r1 (4), sect193r2 (5), sect233k1 (6),
    sect233r1 (7), sect239k1 (8), sect283k1 (9),
    sect283r1 (10), sect409k1 (11), sect409r1 (12),
    sect571k1 (13), sect571r1 (14), secp160k1 (15),
    secp160r1 (16), secp160r2 (17), secp192k1 (18),
    secp192r1 (19), secp224k1 (20), secp224r1 (21),
    secp256k1 (22), secp256r1 (23), secp384r1 (24),
    secp521r1 (25),

    // Finite Field Groups.
    ffdhe2048 (256), ffdhe3072 (257), ffdhe4096 (258),
    ffdhe6144 (259), ffdhe8192 (260),
    ffdhe_private_use (0x01FC..0x01FF),

    // Reserved Code Points.
    reserved (0xFE00..0xFEFF),
    reserved(0xFF01),
    reserved(0xFF02),
    (0xFFFF)
```

```
      } NamedGroup;

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

   sect163k1, etc
      Indicates support of the corresponding named curve The named
      curves defined here are those specified in SEC 2 [13]. Note
that
      many of these curves are also recommended in ANSI X9.62 [X962]
and
      FIPS 186-2 [DSS]. Values 0xFE00 through 0xFEFF are reserved for
      private use. Values 0xFF01 and 0xFF02 were used in previous
      versions of TLS but MUST NOT be offered by TLS 1.3
      implementations. [[OPEN ISSUE: Triage curve list.]]

   ffdhe2432, etc
      Indicates support of the corresponding finite field group,
defined
      in [I-D.ietf-tls-negotiated-ff-dhe]

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

   As an example, a client that only supports secp192r1 (aka NIST
P-192;
   value 19 = 0x0013) and secp224r1 (aka NIST P-224; value 21 =
0x0015)
   and prefers to use secp192r1 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 13 00 15

   The client MUST supply a "named_groups" extension containing at

least
   one group for each key exchange algorithm (currently DHE and ECDHE)
   for which it offers a cipher suite. If the client does not supply
a
   "named_groups" extension with a compatible group, 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.

   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 the ServerKeyExchange message. The
server
   must consider the supported groups in both cases.

   [[TODO: IANA Considerations.]]

6.3.1.5. Client Key Share

   The client_key_share extension MUST be provided by the client if it
   offers any cipher suites that involve non-PSK (currently DHE or
   ECDHE) key exchange. It contains the client's cryptographic
   parameters for zero or more key establishment methods. [[OPEN
ISSUE:
   Would it be better to omit it if it's empty?.
   https://github.com/tlswg/tls13-spec/issues/190]]

   Meaning of this message:

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

   group
      The named group for the key share offer. This identifies the
      specific key exchange method that the ClientKeyShareOffer
      describes. Finite Field Diffie-Hellman [DH] parameters are
      described in Section 6.3.1.5.1; Elliptic Curve Diffie-Hellman
      parameters are described in Section 6.3.1.5.2.

   key_exchange

Key exchange information. The contents of this field are
determined by the value of NamedGroup entry and its corresponding
definition.

```
struct {
    ClientKeyShareOffer offers<0..2^16-1>;
} ClientKeyShare;
```

offers
    A list of ClientKeyShareOffer values in descending order of client
    preference.

Clients may offer an arbitrary number of ClientKeyShareOffer values,
each representing a single set of key agreement parameters; for
instance a client might offer shares for several elliptic curves or
multiple integer DH groups. The shares for each ClientKeyShareOffer
MUST by generated independently. Clients MUST NOT offer multiple
ClientKeyShareOffers for the same parameters. It is explicitly
permitted to send an empty client_key_share extension as this is used
to elicit the server's parameters if the client has no useful
information. [TODO: Recommendation about what the client offers.
Presumably which integer DH groups and which curves.]

6.3.1.5.1. Diffie-Hellman Parameters

Diffie-Hellman [DH] parameters for both clients and servers are
encoded in the opaque key_exchange field of the ClientKeyShareOffer
or ServerKeyShare structures. 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>;
```

6.3.1.5.2. ECDHE Parameters

ECDHE parameters for both clients and servers are encoded in the
opaque key_exchange field of the ClientKeyShareOffer or
ServerKeyShare structures. 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.]]

6.3.1.5.3. Known Configuration Extension

   The known_configuration extension allows the client to indicate
that
   it wishes to reuse the server's known configuration and semi-static
   (EC)DHE key (see Section 6.3.6 for how to establish these

configurations. This extension allows the omission of the server
certificate and signature, with three potential benefits:

- Shortening the handshake because the certificate may be large.

- Reducing cryptographic burden on the server if the server has an
   RSA certificate, as well as on the client if the server has an
   ECDSA certificate.

- Allowing the client and server to do a 0-RTT exchange (See
   Section 6.2.3)

The extension is defined as:

```
    struct {
      select (Role) {
        case client:
          opaque identifier<0..2^16-1>;

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

identifier
   An opaque label for the configuration in question.

A client which wishes to reuse a known configuration MAY supply a
single KnownConfigurationExtension value which indicates the known
configuration it desires to use. It is a fatal error to supply
more
than one extension. A server which wishes to use the key replies
with an empty extension (i.e., with a length field of 0) in its
ServerHello.

When the client and server mutually agree upon a known
configuration
via this mechanism, then the Static Secret (SS) is computed based

on
   the server's (EC)DHE key from the identified configuration and the
   client's key found in the ClientKeyShare. If no key from an
   acceptable group is in the ClientKeyShare, the server MUST ignore
the
   known_configuration extension. When this mechanism is used, the
   server MUST NOT send a Certificate/CertificateVerify message unless
   the ServerConfiguration message is also sent.

   When the known_configuration data extension is in use, the
handshake
   hash is extended to include the server's configuration data and
   certificate (see Section 7.2.1) so as to tightly bind them
together.

6.3.1.5.4. 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).

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

        struct {
          select (Role) {
            case client:
              psk_identity identities<0..2^16-1>;

            case server:
              psk_identity identity;

        } PreSharedKeyExtension;
```

   identifier
      An opaque label for the pre-shared key.

   When the client offers a PSK cipher suite, it MUST also supply a
   PreSharedKeyExtension to indicate the PSK(s) to be used. If no
such
   extension is present, the server MUST NOT negotiate a PSK cipher
   suite. If no suitable identity is present, the server MUST NOT
   negotiate a PSK cipher suite.

   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 "handshake_failure" alert.

6.3.1.5.5. Early Data Indication

   In cases where TLS clients have previously interacted with the
server
   and the server has supplied a known configuration, 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. This technique
   MUST only be used along with the "known_configuration" extension.

```
        enum { early_handshake(1), early_data(2),
               early_handshake_and_data(3), (255) } EarlyDataType;

        struct {
          select (Role) {
            case client:
              opaque context<0..255>;
              EarlyDataType type;
            case server:
              struct {};
          }
        } EarlyDataIndication;
```

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

   type
      The type of early data that is being sent. "early_handshake"
means
      that only handshake data is being sent. "early_data" means that
      only data is being sent. "early_handshake_and_data" means that
      both are being sent.

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 relies on content types not
being encrypted. If we had content types that were encrypted, this
would basically require trial decryption.]]

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.

The server MUST first validate that the client's
"known_configuration" extension is valid and that the client has
suppled a valid key share in the "client_key_shares" extension. If
not, it MUST ignore the extension and discard the early handshake
data and early data.

[[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.]]

[[TODO: We should really allow early_data to be used with PSKs. In
order to make this work, we need to either:

(a) explicitly signal the entire cryptographic parameter set (b)
tie
it to the PSK identifier (as is presently done in the
known_configuration extension).

These two should match.]]

6.3.1.5.5.1. Replay Properties

As noted in Section 6.2.3, 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.

6.3.2. Server Key Share

   When this message will be sent:

      This message will be sent immediately after the ServerHello
      message if the client has provided a ClientKeyShare extension
      which is compatible with the selected cipher suite and group
      parameters.

   Meaning of this message:

      This message conveys cryptographic information to allow the
client
      to compute a shared secret secret: a Diffie-Hellman public key
      with which the client can complete a key exchange (with the
result
      being the shared secret) or a public key for some other
algorithm.

   Structure of this message:

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

   group
      The named group for the key share offer. This identifies the
      selected key exchange method from the ClientKeyShare
      (Section 6.3.1.5), identifying which value from the
      ClientKeyShareOffer the server has accepted as is responding to.

   key_exchange

Key exchange information. The contents of this field are
determined by the value of NamedGroup entry and its
corresponding
definition.

6.3.3. Encrypted Extensions

When this message will be sent:

If this message is sent, it MUST be sent immediately after the
server's ServerKeyShare. 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 DH_anon and PSK), unless the KnownKeyExtension is used.
This message will always immediately follow either the

EncryptedExtensions message if one is sent or the ServerKeyShare
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 MUST directly certify the one preceding it. Because
    certificate validation requires that root keys be distributed
    independently, the self-signed certificate that specifies the
root
    certificate authority MAY be omitted from the chain, under the
    assumption that the remote end must already possess it in order
to
    validate it in any case.

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 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   RSA public key; the certificate MUST allow the
      ECDHE_RSA   key to be used for signing (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 key exchange message.
                         Note: ECDHE_RSA is defined in [RFC4492].

      DHE_DSS   DSA public key; the certificate MUST allow the
                         key to be used for signing with the hash
                         algorithm that will be employed in the server
                         key exchange message.

      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 key exchange message. 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.

   If the client provided a "signature_algorithms" extension, then all
   certificates provided by the server MUST be signed by a hash/
   signature algorithm pair that appears in that extension. Note that
   this implies 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 DSA 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,
   etc.). If the server has a single certificate, it SHOULD attempt
to
   validate that it meets these criteria.

   Note that there are certificates that use algorithms and/or
algorithm
   combinations that cannot be currently used with TLS. For example,
a
   certificate with RSASSA-PSS signature key (id-RSASSA-PSS OID in
   SubjectPublicKeyInfo) cannot be used because TLS defines no
   corresponding signature algorithm.

   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:

      enum {
          rsa_sign(1), dss_sign(2), rsa_fixed_dh(3), dss_fixed_dh(4),
          rsa_ephemeral_dh_RESERVED(5), dss_ephemeral_dh_RESERVED(6),
          fortezza_dms_RESERVED(20), (255)
      } ClientCertificateType;

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

      struct {
          ClientCertificateType certificate_types<1..2^8-1>;
          SignatureAndHashAlgorithm
            supported_signature_algorithms<2..2^16-2>;
          DistinguishedName certificate_authorities<0..2^16-1>;
      } CertificateRequest;

   certificate_types
      A list of the types of certificate types that the client may
      offer.

       rsa_sign   a certificate containing an RSA key
       dss_sign   a certificate containing a DSA key
       rsa_fixed_dh   a certificate containing a static DH key.
       dss_fixed_dh   a certificate containing a static DH key

   supported_signature_algorithms
      A list of the hash/signature algorithm pairs that the server is
      able to verify, listed in descending order of preference.

   certificate_authorities
      A list of the distinguished names [X501] of acceptable
      certificate_authorities, represented in DER-encoded 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 of the appropriate ClientCertificateType,
     unless there is some external arrangement to the contrary.

  The interaction of the certificate_types and
  supported_signature_algorithms fields is somewhat complicated.
  certificate_types has been present in TLS since SSL 3.0, but was
  somewhat underspecified. Much of its functionality is superseded
by
  supported_signature_algorithms. The following rules apply:

  - Any certificates provided by the client MUST be signed using a
    hash/signature algorithm pair found in
    supported_signature_algorithms.

  - The end-entity certificate provided by the client MUST contain a
    key that is compatible with certificate_types. If the key is a
    signature key, it MUST be usable with some hash/signature
    algorithm pair in supported_signature_algorithms.

  - For historical reasons, the names of some client certificate
types
    include the algorithm used to sign the certificate. For
example,
    in earlier versions of TLS, rsa_fixed_dh meant a certificate
    signed with RSA and containing a static DH key. In TLS 1.2,
this
    functionality has been obsoleted by the
    supported_signature_algorithms, and the certificate type no
longer
    restricts the algorithm used to sign the certificate. For
    example, if the server sends dss_fixed_dh certificate type and
    {{sha1, dsa}, {sha1, rsa}} signature types, the client MAY reply
    with a certificate containing a static DH key, signed with RSA-

SHA1.

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

Note: Values listed as RESERVED MUST NOT be used. They were used in
SSL 3.0.

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:

```
struct {
    opaque configuration_id<1..2^16-1>;
    uint32 expiration_date;
    NamedGroup group;
    opaque server_key<1..2^16-1>;
    Boolean early_data_allowed;
} ServerConfiguration;
```

configuration_id
    The configuration identifier to be used with the known
    configuration extension Section 6.3.1.5.3.

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_allowed
         Whether the client may send data in its first flight (see
         Section 6.3.1.5.5).

      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. [[OPEN ISSUE: Should this
      allow some sort of parameter negotiation?]]

      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.

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 only 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 session_hash is as described in {{the-handshake-hash} 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.

If the client has offered the "signature_algorithms" extension, the signature algorithm and hash algorithm MUST be a pair listed in that extension. Note that there is a possibility for inconsistencies here. For instance, the client might offer DHE_DSS key exchange but omit any DSA 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.

Because DSA signatures do not contain any secure indication of
hash algorithm, there is a risk of hash substitution if multiple
hashes may be used with any key. Currently, DSA [DSS] may only
be
used with SHA-1. Future revisions of DSS [DSS-3] are expected
to
allow the use of other digest algorithms with DSA, as well as
guidance as to which digest algorithms should be used with each
key size. In addition, future revisions of [RFC5280] may
specify
mechanisms for certificates to indicate which digest algorithms
are to be used with DSA. [[TODO: Update this to deal with DSS-3
and DSS-4. https://github.com/tlswg/tls13-spec/issues/59]]

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 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.
   The input to the client and server Finished messages may not be the
   same because the server's Finished does not include the client's
   Certificate and CertificateVerify message.

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]).

   - The end-entity certificate's public key (and associated
      restrictions) has to be compatible with the certificate types
      listed in CertificateRequest:

```
     Client Cert. Type    Certificate Key Type

     rsa_sign             RSA public key; the certificate MUST allow the
                          key to be used for signing with the signature
                          scheme and hash algorithm that will be
                          employed in the certificate verify message.

     dss_sign             DSA public key; the certificate MUST allow the
                          key to be used for signing with the hash
                          algorithm that will be employed in the
                          certificate verify message.

     ecdsa_sign           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
                          certificate verify message; the public key
                          MUST use a curve and point format supported by
                          the server.

     rsa_fixed_dh         Diffie-Hellman public key; MUST use the same
     dss_fixed_dh         parameters as server's key.

     rsa_fixed_ecdh       ECDH-capable public key; MUST use the
     ecdsa_fixed_ecdh     same curve as the server's key, and MUST use a
                          point format supported by the server.
```

   - 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.

   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.

Because DSA signatures do not contain any secure indication of
hash algorithm, there is a risk of hash substitution if multiple
hashes may be used with any key. Currently, DSA [DSS] may only be
used with SHA-1. Future revisions of DSS [DSS-3] are expected to
allow the use of other digest algorithms with DSA, as well as
guidance as to which digest algorithms should be used with each
key size. In addition, future revisions of [RFC5280] may specify
mechanisms for certificates to indicate which digest algorithms
are to be used with DSA.

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 6.3.1.5.4) 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. 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
```

agreement, 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/ known_configuration) | Client ephemeral w/ Known Key | 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.

```
HKDF-Expand-Label(Secret, Label, HashValue, Length) =
     HKDF-Expand(Secret, Label + '\0' + HashValue, Length)
```

1. xSS = HKDF(0, SS, "extractedSS", L)

2. xES = HKDF(0, ES, "extractedES", L)

3. master_secret= HKDF(xSS, xES, "master secret", L)

4. 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.

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...

ServerKeyShare

   Application  master "application data key expansion"    All
handshake
                 secret                                     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 the known_configuration extension is in use
    (Section 6.3.1.5.3, this contains the concatenation of the
    ServerConfiguration and Certificate messages from the handshake
    where the configuration was established. 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 Cipher Suites

    In the absence of an application profile standard specifying
    otherwise, a TLS-compliant application MUST implement the cipher
    suite TODO:Needs to be selected [1]. (See Appendix A.4 for the
    definition.)

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 C, D, and E.

11. IANA Considerations

    [[TODO: Update https://github.com/tlswg/tls13-spec/issues/62]]

    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 ClientCertificateType Identifiers Registry: 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 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 6.3.1.4).

   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 6.3.1.4.1.
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 6.3.1.4.1.
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.

   [DSS]   National Institute of Standards and Technology, U.S.
              Department of Commerce, "Digital Signature Standard",
NIST
              FIPS PUB 186-2, 2000.

   [RFC1321] Rivest, R., "The MD5 Message-Digest Algorithm", RFC
1321,
              April 1992.

   [RFC2104] Krawczyk, H., Bellare, M., and R. Canetti, "HMAC: Keyed-

              Hashing for Message Authentication", RFC 2104, February
              1997.

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

   [RFC2434] Narten, T. and H. Alvestrand, "Guidelines for Writing an
              IANA Considerations Section in RFCs", BCP 26, RFC 2434,
              October 1998.

   [RFC3447] Jonsson, J. and B. Kaliski, "Public-Key Cryptography
              Standards (PKCS) #1: RSA Cryptography Specifications
              Version 2.1", RFC 3447, February 2003.

   [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, May 2008.

   [RFC5288] Salowey, J., Choudhury, A., and D. McGrew, "AES Galois
              Counter Mode (GCM) Cipher Suites for TLS", RFC 5288,
              August 2008.

   [RFC5289] Rescorla, E., "TLS Elliptic Curve Cipher Suites with
SHA-
              256/384 and AES Galois Counter Mode (GCM)", RFC 5289,
              August 2008.

   [RFC5869] Krawczyk, H. and P. Eronen, "HMAC-based Extract-and-
Expand
              Key Derivation Function (HKDF)", RFC 5869, May 2010.

   [SHS]   National Institute of Standards and Technology, U.S.
              Department of Commerce, "Secure Hash Standard", NIST
FIPS
              PUB 180-2, August 2002.

   [X680]   ITU-T, "Information technology - Abstract Syntax

```
Notation
                One (ASN.1): Specification of basic notation", ISO/IEC
                8824-1:2002, 2002.

    [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

```
    [CBCATT]    Moeller, B., "Security of CBC Ciphersuites in SSL/TLS:
                Problems and Countermeasures", May 2004,
                <https://www.openssl.org/~bodo/tls-cbc.txt>.

    [DSS-3]     National Institute of Standards and Technology, U.S.,
                "Digital Signature Standard", NIST FIPS PUB 186-3 Draft,
                2006.

    [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.
```

   [I-D.ietf-tls-session-hash]
              Bhargavan, K., Delignat-Lavaud, A., Pironti, A., Langley,
              A., and M. Ray, "Transport Layer Security (TLS) Session
              Hash and Extended Master Secret Extension", draft-ietf-
              tls-session-hash-05 (work in progress), April 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, September 1981.

   [RFC1948] Bellovin, S., "Defending Against Sequence Number Attacks",
              RFC 1948, May 1996.

   [RFC2246] Dierks, T. and C. Allen, "The TLS Protocol Version 1.0",
              RFC 2246, January 1999.

   [RFC3268] Chown, P., "Advanced Encryption Standard (AES)
              Ciphersuites for Transport Layer Security (TLS)", RFC
              3268, June 2002.

   [RFC4086] Eastlake, D., Schiller, J., and S. Crocker, "Randomness
              Requirements for Security", BCP 106, RFC 4086, June 2005.

   [RFC4279] Eronen, P. and H. Tschofenig, "Pre-Shared Key Ciphersuites
              for Transport Layer Security (TLS)", RFC 4279, December
              2005.

   [RFC4302] Kent, S., "IP Authentication Header", RFC 4302, December
              2005.

   [RFC4303] Kent, S., "IP Encapsulating Security Payload (ESP)", RFC
              4303, December 2005.

   [RFC4346] Dierks, T. and E. Rescorla, "The Transport Layer
Security

              (TLS) Protocol Version 1.1", RFC 4346, April 2006.

   [RFC4366] Blake-Wilson, S., Nystrom, M., Hopwood, D., Mikkelsen,
J.,

              and T. Wright, "Transport Layer Security (TLS)
              Extensions", RFC 4366, April 2006.

   [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, May 2006.

   [RFC4506] Eisler, M., "XDR: External Data Representation
Standard",

              STD 67, RFC 4506, May 2006.

   [RFC5077] Salowey, J., Zhou, H., Eronen, P., and H. Tschofenig,
              "Transport Layer Security (TLS) Session Resumption
without

              Server-Side State", RFC 5077, January 2008.

   [RFC5081] Mavrogiannopoulos, N., "Using OpenPGP Keys for Transport
              Layer Security (TLS) Authentication", RFC 5081, November
              2007.

   [RFC5116] McGrew, D., "An Interface and Algorithms for
Authenticated

              Encryption", RFC 5116, January 2008.

   [RFC5705] Rescorla, E., "Keying Material Exporters for Transport
              Layer Security (TLS)", RFC 5705, March 2010.

   [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, May 2010.

   [RFC6066]   Eastlake, D., "Transport Layer Security (TLS)
Extensions:

               Extension Definitions", RFC 6066, January 2011.

   [RFC6176]   Turner, S. and T. Polk, "Prohibiting Secure Sockets
Layer

               (SSL) Version 2.0", RFC 6176, March 2011.

   [RFC7465]   Popov, A., "Prohibiting RC4 Cipher Suites", RFC 7465,
               February 2015.

   [RFC7568]   Barnes, R., Thomson, M., Pironti, A., and A. Langley,
               "Deprecating Secure Sockets Layer Version 3.0", RFC
7568,

               June 2015.

   [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] https://github.com/tlswg/tls13-spec/issues/32

   [2] mailto:tls@ietf.org

Appendix A. Protocol Data Structures and Constant Values

   This section describes protocol types and constants.

A.1. Record Layer

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

enum {
    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 type;
    ProtocolVersion record_version = { 3, 1 };    /* TLS v1.x */
    uint16 length;
    aead-ciphered struct {
       opaque content[TLSPlaintext.length];
    } 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 */
          user_canceled(90),
          no_renegotiation(100),               /* fatal */
          unsupported_extension(110),          /* fatal */
          (255)
      } AlertDescription;

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

A.3. Handshake Protocol

```
    enum {
        reserved(0), client_hello(1), server_hello(2),
        session_ticket(4), hello_retry_request(6),
        server_key_share(7), certificate(11), reserved(12),
        certificate_request(13), server_configuration(14),
        certificate_verify(15), reserved(16), 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 server_key_share:    ServerKeyShare;
            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>;
        select (extensions_present) {
            case false:
                struct {};
            case true:
                Extension extensions<0..2^16-1>;
        };
    } ClientHello;

    struct {
        ProtocolVersion server_version;
        Random random;
```

```
        uint8 session_id_len;  // Must be 0.
        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 {
```

```
        signature_algorithms(13),
        early_data(TBD),
        supported_groups(TBD),
        known_configuration(TBD),
        pre_shared_key(TBD)
        client_key_shares(TBD)
        (65535)
    } ExtensionType;

    struct {
      select (Role) {
        case client:
          opaque identifier<0..2^16-1>;

        case server:
          struct {};
      }
    } KnownConfigurationExtension

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

    struct {
      select (Role) {
        case client:
          psk_identity identities<0..2^16-1>;
```

```
        case server:
          psk_identity identity;

    } PreSharedKeyExtension;

    enum { early_handshake(1), early_data(2),
           early_handshake_and_data(3), (255) } EarlyDataType;

    struct {
      select (Role) {
        case client:
          opaque context<0..255>;
          EarlyDataType type;
        case server:
```

```
            struct {};
          }
        } EarlyDataIndication;

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

      struct {
          opaque configuration_id<1..2^16-1>;
          uint32 expiration_date;
          NamedGroup group;
          opaque server_key<1..2^16-1>;
          Boolean early_data_allowed;
      } ServerConfiguration;
```

A.3.1.1. Signature Algorithm Extension

```
    enum {
        none(0), md5(1), sha1(2), sha224(3), sha256(4), sha384(5),
        sha512(6), (255)
    } HashAlgorithm;

    enum { anonymous(0), rsa(1), dsa(2), ecdsa(3), (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.
        sect163k1 (1), sect163r1 (2), sect163r2 (3),
```

```
          sect193r1 (4), sect193r2 (5), sect233k1 (6),
          sect233r1 (7), sect239k1 (8), sect283k1 (9),
          sect283r1 (10), sect409k1 (11), sect409r1 (12),
          sect571k1 (13), sect571r1 (14), secp160k1 (15),
          secp160r1 (16), secp160r2 (17), secp192k1 (18),
          secp192r1 (19), secp224k1 (20), secp224r1 (21),
          secp256k1 (22), secp256r1 (23), secp384r1 (24),
          secp521r1 (25),

          // Finite Field Groups.
          ffdhe2048 (256), ffdhe3072 (257), ffdhe4096 (258),
          ffdhe6144 (259), ffdhe8192 (260),
          ffdhe_private_use (0x01FC..0x01FF),

          // Reserved Code Points.
          reserved (0xFE00..0xFEFF),
          reserved(0xFF01),
          reserved(0xFF02),
          (0xFFFF)
      } NamedGroup;

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

A.3.2. Key Exchange Messages

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

        struct {
            ClientKeyShareOffer offers<0..2^16-1>;
        } ClientKeyShare;

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

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

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

A.3.3.   Authentication Messages

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

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

        enum {
            rsa_sign(1), dss_sign(2), rsa_fixed_dh(3), dss_fixed_dh(4),
            rsa_ephemeral_dh_RESERVED(5), dss_ephemeral_dh_RESERVED(6),
            fortezza_dms_RESERVED(20), (255)
        } ClientCertificateType;

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

        struct {
            ClientCertificateType certificate_types<1..2^8-1>;
            SignatureAndHashAlgorithm
              supported_signature_algorithms<2..2^16-2>;
            DistinguishedName certificate_authorities<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. The Cipher Suite

   The following values define the cipher suite codes used in the
   ClientHello and ServerHello messages. A cipher suite defines a
   cipher specification supported in TLS.

   TLS_NULL_WITH_NULL_NULL is specified and is the initial state of a
   TLS connection during the first handshake on that channel, but MUST
   NOT be negotiated, as it provides no more protection than an
   unsecured connection.

      CipherSuite TLS_NULL_WITH_NULL_NULL = {0x00,0x00};

   The following cipher suite definitions, defined in [RFC5288], are
   used for server-authenticated (and optionally client-authenticated)
   Diffie-Hellman. DHE denotes ephemeral Diffie-Hellman, where the
   Diffie-Hellman parameters are signed by a signature-capable
   certificate, which has been signed by the CA. The signing algorithm
   used by the server is specified after the DHE component of the
   CipherSuite name. The server can request any signature-capable
   certificate from the client for client authentication.

      CipherSuite TLS_DHE_RSA_WITH_AES_128_GCM_SHA256 = {0x00,0x9E};

```
      CipherSuite TLS_DHE_RSA_WITH_AES_256_GCM_SHA384 = {0x00,0x9F};
      CipherSuite TLS_DHE_DSS_WITH_AES_128_GCM_SHA256 = {0x00,0xA2};
      CipherSuite TLS_DHE_DSS_WITH_AES_256_GCM_SHA384 = {0x00,0xA3};
```

   The following cipher suite definitions, defined in [RFC5289], are
   used for server-authenticated (and optionally client-authenticated)
   Elliptic Curve Diffie-Hellman. ECDHE denotes ephemeral Diffie-
   Hellman, where the Diffie-Hellman parameters are signed by a
   signature-capable certificate, which has been signed by the CA.
The
   signing algorithm used by the server is specified after the DHE
   component of the CipherSuite name. The server can request any

   signature-capable certificate from the client for client
   authentication.

```
      CipherSuite TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256 =
{0xC0,0x2B};
      CipherSuite TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384 =
{0xC0,0x2C};
      CipherSuite TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256   =
{0xC0,0x2F};
      CipherSuite TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384   =
{0xC0,0x30};
```

   The following ciphers, defined in [RFC5288], are used for
completely
   anonymous Diffie-Hellman communications in which neither party is
   authenticated. Note that this mode is vulnerable to man-in-the-
   middle attacks. Using this mode therefore is of limited use: These
   cipher suites MUST NOT be used by TLS implementations unless the
   application layer has specifically requested to allow anonymous key
   exchange. (Anonymous key exchange may sometimes be acceptable, for
   example, to support opportunistic encryption when no set-up for
   authentication is in place, or when TLS is used as part of more
   complex security protocols that have other means to ensure
   authentication.)

```
      CipherSuite TLS_DH_anon_WITH_AES_128_GCM_SHA256 = {0x00,0xA6};
```

```
      CipherSuite TLS_DH_anon_WITH_AES_256_GCM_SHA384 = {0x00,0xA7};
```

   [[TODO: Add all the defined AEAD ciphers. This currently only
lists
   GCM. https://github.com/tlswg/tls13-spec/issues/53]] Note that
using
   non-anonymous key exchange without actually verifying the key
   exchange is essentially equivalent to anonymous key exchange, and
the
   same precautions apply. While non-anonymous key exchange will
   generally involve a higher computational and communicational cost
   than anonymous key exchange, it may be in the interest of
   interoperability not to disable non-anonymous key exchange when the
   application layer is allowing anonymous key exchange.

   o For cipher suites ending with _SHA256, HKDF is used with SHA-256
as
   the hash function.

   o For cipher suites ending with _SHA384, HKDF is used with SHA-384
as
   the hash function.

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

   Note: The cipher suite values { 0x00, 0x1C } and { 0x00, 0x1D } are
    reserved to avoid collision with Fortezza-based cipher suites in
SSL
   3.0.

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 and Section 6.3.9, the restrictions
on
   the signature algorithms used to sign certificates are no longer
tied
   to the cipher suite (when used by the server) or the
   ClientCertificateType (when used by the client). 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. Cipher Suite Definitions

   Cipher Suite   Key   Record
                                               Exchange   Protection   Hash

   TLS_NULL_WITH_NULL_NULL   NULL   NULL_NULL   N/A
   TLS_DHE_RSA_WITH_AES_128_GCM_SHA256   DHE_RSA   AES_128_GCM
SHA256
   TLS_DHE_RSA_WITH_AES_256_GCM_SHA384   DHE_RSA   AES_256_GCM
SHA384
   TLS_DHE_DSS_WITH_AES_128_GCM_SHA256   DHE_DSS   AES_128_GCM
SHA256
   TLS_DHE_DSS_WITH_AES_256_GCM_SHA384   DHE_DSS   AES_256_GCM
SHA384
   TLS_DH_anon_WITH_AES_128_GCM_SHA256   DH_anon   AES_128_GCM
SHA256
   TLS_DH_anon_WITH_AES_256_GCM_SHA384   DH_anon   AES_128_GCM
SHA384

Appendix C. Implementation Notes

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

C.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-1, 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.

C.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.

C.3. Cipher Suites

   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. For instance, anonymous
   Diffie-Hellman is strongly discouraged because it cannot prevent man-
   in-the-middle attacks. 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.

C.4. Implementation Pitfalls

   Implementation experience has shown that certain parts of earlier TLS
   specifications are not easy to understand, and have been a source of
   interoperability and security problems. Many of these areas have

been clarified in this document, but this appendix contains a short
list of the most important things that require special attention
from
implementors.

TLS protocol issues:

- Do you correctly handle handshake messages that are fragmented
to
multiple TLS records (see Section 5.2.1)? 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 D)

- Have you ensured that all support for SSL, RC4, and EXPORT
ciphers
is completely removed from all possible configurations that
support TLS 1.3 or later, and that attempts to use these
obsolete
capabilities fail correctly? (see Appendix D)

- Do you handle TLS extensions in ClientHello correctly, including
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)?

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 E.1.1.2)?

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

Appendix D. Backward Compatibility

   The TLS protocol provides a built-in mechanism for version
   negotiation between endpoints potentially supporting different
   versions of TLS.

   TLS 1.x and SSL 3.0 use compatible ClientHello messages. Servers
can
   also handle clients trying to use future versions of TLS as long as
   the ClientHello format remains compatible and the client supports
the
   highest protocol version available in the server.

   Prior versions of TLS used the record layer version number for
   various purposes. (TLSPlaintext.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 usage of TLS 1.0-1.2 SHOULD set the record layer
   version number to the negotiated version for the ServerHello and
all
   records thereafter.

D.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.

D.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.

D.3. Backwards Compatibility Security Restrictions

If an implementation negotiates usage 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 usage 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.

Appendix E. 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.

E.1. Handshake Protocol

The 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
handshake protocol can also optionally authenticate parties who
have
certificates signed by a trusted certificate authority.

E.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.

E.1.1.1. Anonymous Key Exchange

   Completely anonymous sessions can be established using Diffie-Hellman
   for key exchange. The server's public parameters are contained in
   the server key share message, and the client's are sent in the client
   key share message. Eavesdroppers who do not know the private values
   should not be able to find the Diffie-Hellman result.

Warning: Completely anonymous connections only provide protection
against passive eavesdropping. Unless an independent tamper-proof
channel is used to verify that the Finished messages were not
replaced by an attacker, server authentication is required in
environments where active man-in-the-middle attacks are a concern.

E.1.1.2. Diffie-Hellman Key Exchange with Authentication

When Diffie-Hellman key exchange is used, the client and server use
the client key exchange and server key exchange messages 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.

E.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.

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.

E.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.

E.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.

E.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].

E.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 F. Working Group Information

   The discussion list for the IETF TLS working group is located at
the
   e-mail address tls@ietf.org [2]. 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 G. Contributors

   Martin Abadi
   University of California, Santa Cruz
   abadi@cs.ucsc.edu

   Christopher Allen (co-editor of TLS 1.0)
   Alacrity Ventures
   ChristopherA@AlacrityManagement.com

   Steven M. Bellovin
   Columbia University
   smb@cs.columbia.edu

   Benjamin Beurdouche

   Karthikeyan Bhargavan (co-author of [I-D.ietf-tls-session-hash])
   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 [I-D.ietf-tls-session-hash])
   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 [I-D.ietf-tls-session-hash])
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 [I-D.ietf-tls-session-hash])
INRIA
alfredo.pironti@inria.fr

Marsh Ray (co-author of [I-D.ietf-tls-session-hash])
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

Martin Thomson
Mozilla
mt@mozilla.com

Tom Weinstein

Tim Wright
Vodafone
timothy.wright@vodafone.com

Author's Address

Eric Rescorla
RTFM, Inc.

EMail: ekr@rtfm.com