7. 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:

session identifier An arbitrary byte sequence chosen by the server to identify an active or resumable session state.

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

compression method The algorithm used to compress data prior to encryption.

cipher spec Specifies the pseudorandom function (PRF) used to generate keying material, the bulk data encryption algorithm (such as null, AES, etc.) and the MAC algorithm (such as HMAC-SHA1). It also defines cryptographic attributes such as the mac_length. (See Appendix A.6 for formal definition.)

master secret 48-byte secret shared between the client and server.

is resumable A flag indicating whether the session can be used to initiate new 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 through the resumption feature of the TLS Handshake Protocol.

7.1. Change Cipher Spec Protocol

The change cipher spec protocol exists to signal transitions in ciphering strategies. The protocol consists of a single message, which is encrypted and compressed under the current (not the pending) connection state. The message consists of a single byte of value 1.

```
struct {
    enum { change_cipher_spec(1), (255) } type;
} ChangeCipherSpec;
```

The ChangeCipherSpec message is sent by both the client and the server to notify the receiving party that subsequent records will be protected under the newly negotiated CipherSpec and keys. Reception of this message causes the receiver to instruct the record layer to immediately copy the read pending state

into the read current state. Immediately after sending this message, the sender MUST instruct the record layer to make the write pending state the write active state.

(See Section 6.1.) The ChangeCipherSpec message is sent during the handshake after the security parameters have been agreed upon, but before the verifying Finished message is sent.

Note: If a rehandshake occurs while data is flowing on a connection, the communicating parties may continue to send data using the old CipherSpec. However, once the ChangeCipherSpec has been sent, the new CipherSpec MUST be used. The first side to send the ChangeCipherSpec does not know that the other side has finished computing the new keying material (e.g., if it has to perform a time-consuming public key operation). Thus, a small window of time, during which the recipient must buffer the data, MAY exist. In practice, with modern machines this interval is likely to be fairly short.

7.2. 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 and compressed, as specified by the current connection state.

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


enum {
    close_notify(0),
    unexpected_message(10),
    bad_record_mac(20),
    decryption_failed_RESERVED(21),
    record_overflow(22),
    decompression_failure(30),
    handshake_failure(40),
    no_certificate_RESERVED(41),
    bad_certificate(42),
    unsupported_certificate(43),
    certificate_revoked(44),
    certificate_expired(45),
    certificate_unknown(46),
    illegal_parameter(47),
```

```
        unknown_ca(48),
        access_denied(49),
        decode_error(50),
        decrypt_error(51),

        export_restriction_RESERVED(60),
        protocol_version(70),
        insufficient_security(71),
        internal_error(80),
        user_canceled(90),
        no_renegotiation(100),
        unsupported_extension(110),
        (255)
    } AlertDescription;


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

7.2.1. 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.

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.

7.2.2. 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 with an incorrect MAC. This alert also MUST be returned if an alert is sent because a TLSCiphertext decrypted in an invalid way: either it wasn't an even multiple of the block length, or its padding values, when checked, weren't correct. 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.

record_overflow A TLSCiphertext record was received that had a length more than 2^14+2048 bytes, or a record decrypted to a TLSCompressed record with more than 2^14+1024 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 The decompression function received improper input (e.g., data that would expand to excessive length). This message is always fatal and should never be observed in communication between proper implementations.

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 is a fatal error.

no_certificate_RESERVED This alert was used in SSLv3 but not any version of TLS. It MUST NOT be sent by compliant implementations.

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.

protocol_version The protocol version the client 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 hello request or by the server in response to a client hello after initial handshaking. Either of these would normally lead to renegotiation; when that is not appropriate, the recipient should respond with this alert. At that point, the original requester can decide whether to proceed with the connection. One case where this would be appropriate is where a server has spawned a process to satisfy a request; the process might receive security parameters (key length, authentication, etc.) at startup, and it might be difficult to communicate changes to these parameters after that point. This message is always a warning.

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

New Alert values are assigned by IANA as described in Section 12.

7.3. 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 use public-key encryption techniques to generate shared secrets.

The TLS Handshake Protocol involves the following steps:

- Exchange hello messages to agree on algorithms, exchange random values, and check for session resumption.
- Exchange the necessary cryptographic parameters to allow the client and server to agree on a premaster secret.
- Exchange certificates and cryptographic information to allow the client and server to authenticate themselves.
- Generate a master secret from the premaster secret and exchanged random values.
- Provide security parameters to the record layer.
- Allow the client and server to verify that their peer has calculated the same security parameters and that the handshake occurred without tampering by an attacker.

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 3DES with a 1024-bit RSA key exchange with a host whose certificate you have verified, you can expect to be that secure.

These goals are achieved by the handshake protocol, which can be summarized as follows: The client sends a ClientHello message to which the server must respond with a ServerHello message, or else a fatal error will occur and the connection will fail. The ClientHello and ServerHello are used to establish security enhancement capabilities between client and server. The ClientHello and ServerHello establish the following attributes: Protocol Version, Session ID, Cipher Suite, and Compression Method. Additionally, two random values are generated and exchanged: ClientHello.random and ServerHello.random.

The actual key exchange uses up to four messages: the server Certificate, the ServerKeyExchange, the client Certificate, and the ClientKeyExchange. New key exchange methods can be created by specifying a format for these messages and by defining the use of the messages to allow the client and server to agree

upon a shared secret. This secret MUST be quite long; currently defined key exchange methods exchange secrets that range from 46 bytes upwards.

Following the hello messages, the server will send its certificate in a Certificate message if it is to be authenticated. Additionally, a ServerKeyExchange message may be sent, if it is required (e.g., if the server has no certificate, or if its certificate is for signing only). If the server is authenticated, it may request a certificate from the client, if that is appropriate to the cipher suite selected. Next, the server will send the ServerHelloDone message, indicating that the hello-message phase of the handshake is complete. The server will then wait for a client response. If the server has sent a CertificateRequest message, the client MUST send the Certificate message. The ClientKeyExchange message is now sent, and the content of that message will depend on the public key algorithm selected between the ClientHello and the ServerHello. If the client has sent a certificate with signing ability, a digitally-signed CertificateVerify message is sent to explicitly verify possession of the private key in the certificate.

At this point, a ChangeCipherSpec message is sent by the client, and the client copies the pending Cipher Spec into the current Cipher Spec. The client then immediately sends the Finished message under the new algorithms, keys, and secrets. In response, the server will send its own ChangeCipherSpec message, transfer the pending to the current Cipher Spec, and send its Finished message under the new Cipher Spec. At this point, the handshake is complete, and the client and server may begin to exchange application layer data. (See flow chart below.) Application data MUST NOT be sent prior to the completion of the first handshake (before a cipher suite other than TLS_NULL_WITH_NULL_NULL is established).

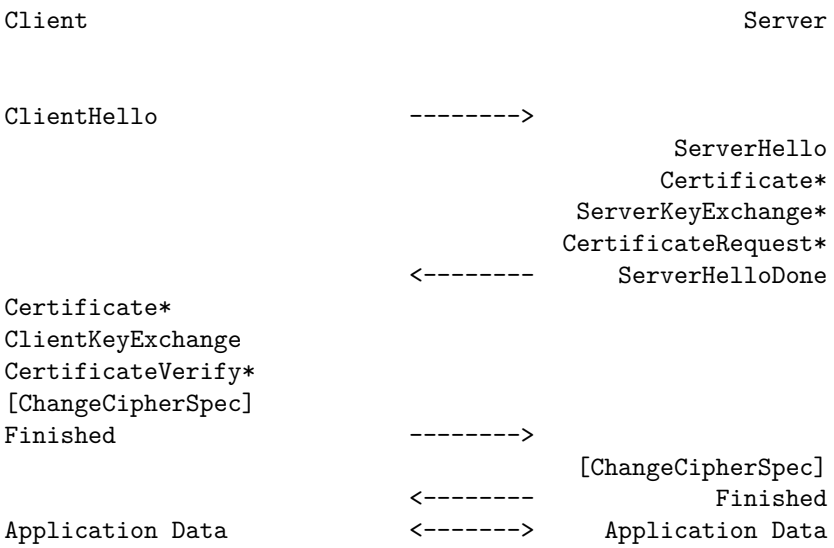

Figure 1. Message flow for a full handshake

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

Note: To help avoid pipeline stalls, ChangeCipherSpec is an independent TLS protocol content type, and is not actually a TLS handshake message.

When the client and server decide to resume a previous session or duplicate an existing session (instead of negotiating new security parameters), the message flow is as follows:

The client sends a ClientHello using the Session ID of the session to be resumed. The server then checks its session cache for a match. If a match is found, and the server is willing to re-establish the connection under the specified session state, it will send a ServerHello with the same Session ID value. At this point, both client and server MUST send ChangeCipherSpec messages and proceed directly to Finished messages. Once the re-establishment is complete, the client and server MAY begin to exchange application layer data. (See flow chart below.) If a Session ID match is not found, the server generates a new session ID, and the TLS client and server perform a full handshake.

```
Client                                               Server


ClientHello                  -------->
                                                ServerHello
                                              [ChangeCipherSpec]
                             <--------             Finished
[ChangeCipherSpec]
Finished                     -------->
Application Data             <------->     Application Data
```

Figure 2. Message flow for an abbreviated handshake

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

7.4. 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 structures, which are processed and transmitted as specified by the current active session state.

```
enum {
```

```
    hello_request(0), client_hello(1), server_hello(2),
    certificate(11), server_key_exchange (12),
    certificate_request(13), server_hello_done(14),
    certificate_verify(15), client_key_exchange(16),
    finished(20), (255)
} HandshakeType;


struct {
    HandshakeType msg_type;    /* handshake type */
    uint24 length;             /* bytes in message */
    select (HandshakeType) {
        case hello_request:       HelloRequest;
        case client_hello:        ClientHello;
        case server_hello:        ServerHello;
        case certificate:         Certificate;
        case server_key_exchange: ServerKeyExchange;
        case certificate_request: CertificateRequest;
        case server_hello_done:   ServerHelloDone;
        case certificate_verify:  CertificateVerify;
        case client_key_exchange: ClientKeyExchange;
        case finished:            Finished;
    } 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. Note one exception to the ordering: the Certificate message is used twice in the handshake (from server to client, then from client to server), but described only in its first position. The one message that is not bound by these ordering rules is the HelloRequest message, which can be sent at any time, but which SHOULD be ignored by the client if it arrives in the middle of a handshake.

New handshake message types are assigned by IANA as described in Section 12.

7.4.1. 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 encryption, hash, and compression algorithms are initialized to null. The current connection state is used for renegotiation messages.

7.4.1.1. Hello Request

When this message will be sent:

```
The HelloRequest message MAY be sent by the server at any time.
```

Meaning of this message:

```
HelloRequest is a simple notification that the client should begin
the negotiation process anew.  In response, the client should send
a ClientHello message when convenient.  This message is not
intended to establish which side is the client or server but
merely to initiate a new negotiation.  Servers SHOULD NOT send a
HelloRequest immediately upon the client's initial connection.  It
is the client's job to send a ClientHello at that time.


This message will be ignored by the client if the client is
currently negotiating a session.  This message MAY be ignored by
the client if it does not wish to renegotiate a session, or the
client may, if it wishes, respond with a no_renegotiation alert.
Since handshake messages are intended to have transmission
precedence over application data, it is expected that the
negotiation will begin before no more than a few records are
received from the client.  If the server sends a HelloRequest but
does not receive a ClientHello in response, it may close the
connection with a fatal alert.


After sending a HelloRequest, servers SHOULD NOT repeat the
request until the subsequent handshake negotiation is complete.
```

Structure of this message:

```
struct { } HelloRequest;
```

This message MUST NOT be included in the message hashes that are maintained throughout the handshake and used in the Finished messages and the certificate verify message.

7.4.1.2. 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 can also send a
ClientHello in response to a HelloRequest or on its own initiative
in order to renegotiate the security parameters in an existing
```

```
connection.
```

Structure of this message:

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

```
   struct {
       uint32 gmt_unix_time;
       opaque random_bytes[28];
   } Random;
```

```
gmt_unix_time
   The current time and date in standard UNIX 32-bit format
   (seconds since the midnight starting Jan 1, 1970, UTC, ignoring
   leap seconds) according to the sender's internal clock.  Clocks
   are not required to be set correctly by the basic TLS protocol;
   higher-level or application protocols may define additional
   requirements.  Note that, for historical reasons, the data
   element is named using GMT, the predecessor of the current
   worldwide time base, UTC.
```

```
random_bytes
   28 bytes generated by a secure random number generator.
```

The ClientHello message includes a variable-length session identifier. If not empty, the value identifies a session between the same client and server whose security parameters the client wishes to reuse. The session identifier MAY be from an earlier connection,

this connection, or from another currently active connection. The second option is useful if the client only wishes to update the random structures and derived values of a connection, and the third option makes it possible to establish several independent secure connections without repeating the full handshake protocol. These independent connections may occur sequentially or simultaneously; a SessionID becomes valid when the handshake negotiating it completes with the exchange of Finished messages and persists until it is removed due to aging or because a fatal error was encountered on a connection associated with the session. The actual contents of the SessionID are defined by the server.

```
opaque SessionID<0..32>;
```

Warning: Because the SessionID is transmitted without encryption or immediate MAC protection, servers MUST NOT place confidential information in session identifiers or let the contents of fake session identifiers cause any breach of security. (Note that the content of the handshake as a whole, including the

SessionID, is protected by the Finished messages exchanged at the end of the handshake.)

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 bulk encryption algorithm (including secret key length), a MAC algorithm, and a PRF. 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 */
```

The ClientHello includes a list of compression algorithms supported by the client, ordered according to the client's preference.

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

```
struct {
    ProtocolVersion client_version;
    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 3.3 (see Appendix E for details about backward compatibility).

random A client-generated random structure.

session_id The ID of a session the client wishes to use for this connection. This field is empty if no session_id is available, or if the client wishes to generate new security parameters.

cipher_suites This is a list of the cryptographic options supported by the client, with the client's first preference first. If the session_id field is not empty (implying a session resumption request), this vector MUST include at least the cipher_suite from that session. Values are defined in Appendix A.5.

compression_methods This is a list of the compression methods supported by the client, sorted by client preference. If the session_id field is not empty (implying a session resumption request), it MUST include the

```
compression_method from that session.  This vector MUST contain,
and all implementations MUST support, CompressionMethod.null.
Thus, a client and server will always be able to agree on a
compression method.
```

extensions Clients MAY request extended functionality from servers by sending data in the extensions field. The actual "Extension" format is defined in Section 7.4.1.4.

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 message. Any handshake message returned by the server, except for a HelloRequest, is treated as a fatal error.

7.4.1.3. 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.
If it cannot find such a match, it will respond with a handshake
failure alert.
```

Structure of this message:

```
struct {
    ProtocolVersion server_version;
    Random random;
    SessionID session_id;
    CipherSuite cipher_suite;
    CompressionMethod compression_method;
    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 compression_method field at the end of the ServerHello.

server_version This field will contain the lower of that suggested by the client in the client hello and the highest supported by the server. For this version of the specification, the version is 3.3. (See Appendix E for details about backward compatibility.)

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

session_id This is the identity of the session corresponding to this connection. If the ClientHello.session_id was non-empty, the server will look in its session cache for a match. If a match is found and the server is willing to establish the new connection using the specified session state, the server will respond with the same value as was supplied by the client. This indicates a resumed session and dictates that the parties must proceed directly to the Finished messages. Otherwise, this field will contain a different value identifying the new session.

The server may return an empty session_id to indicate that the session will not be cached and therefore cannot be resumed. If a session is resumed, it must be resumed using the same cipher suite it was originally negotiated with. Note that there is no requirement that the server resume any session even if it had formerly provided a session_id. Clients MUST be prepared to do a full negotiation – including negotiating new cipher suites – during any handshake.

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.

compression_method The single compression algorithm selected by the server from the list in ClientHello.compression_methods. For resumed sessions, this field is the value from the resumed session state.

extensions A list of extensions. Note that only extensions offered by the client can appear in the server's list.

7.4.1.4. Hello Extensions

The extension format is:

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


enum {
    signature_algorithms(13), (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 a companion document [TLSEXT]. The list of extension types is maintained by IANA as described in Section 12.

An extension type MUST NOT appear in the ServerHello unless the same extension type appeared in the corresponding ClientHello. If a client receives an extension type in ServerHello 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. 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 Client Hello, and does not include them in Server Hello. However, some extensions may specify different behavior during session 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.

7.4.1.4.1. 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 [MD5], 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 [PKCS1] and DSA [DSS], and ECDSA [ECDSA], respectively. The "anonymous" value is meaningless in this context but used in Section 7.4.3. 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. Sections 7.4.2 and 7.4.3 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 exchange), 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 (RSA, DHE_RSA, DH_RSA, RSA_PSK, ECDH_RSA, ECDHE_RSA), behave as if client had sent the value {sha1,rsa}.
- If the negotiated key exchange algorithm is one of (DHE_DSS, DH_DSS), behave as if the client had sent the value {sha1,dsa}.
- If the negotiated key exchange algorithm is one of (ECDH_ECDSA, ECDHE_ECDSA), behave as if the client had sent value {sha1,ecdsa}.

Note: this is a change from TLS 1.1 where there are no explicit rules, but as a practical matter one can assume that the peer supports MD5 and SHA-1.

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 [TLSEXT] require servers to ignore extensions they do not understand.

Servers MUST NOT send this extension. TLS servers MUST support receiving this extension.

When performing session resumption, this extension is not included in Server Hello, and the server ignores the extension in Client Hello (if present).

7.4.2. 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).  This message will always immediately follow the
ServerHello 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 ASN.1Cert<1..2^24-1>;


struct {
    ASN.1Cert 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, unless explicitly negotiated otherwise (e.g., [TLSPGP]).
- 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

  RSA RSA public key; the certificate MUST allow the RSA_PSK key to be used for encryption (the keyEncipherment bit MUST be set if the key usage extension is present). Note: RSA_PSK is defined in [TLSPSK].

  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 [TLSECC].

  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.

  DH_DSS Diffie-Hellman public key; the keyAgreement bit DH_RSA MUST be set if the key usage extension is present.

ECDH_ECDSA ECDH-capable public key; the public key MUST
ECDH_RSA use a curve and point format supported by the client, as described in [TLSECC].

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 [TLSECC].

- The "server_name" and "trusted_ca_keys" extensions [TLSEXT] are used to guide certificate selection.

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). This is a departure from TLS 1.1, which required that the algorithms be the same. Note that this also implies that the DH_DSS, DH_RSA, ECDH_ECDSA, and ECDH_RSA key exchange algorithms do not restrict the algorithm used to sign the certificate. Fixed DH certificates MAY be signed with any hash/signature algorithm pair appearing in the extension. The names DH_DSS, DH_RSA, ECDH_ECDSA, and ECDH_RSA are historical.

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.

7.4.3. Server Key Exchange Message

When this message will be sent:

```
This message will be sent immediately after the server Certificate
message (or the ServerHello message, if this is an anonymous
negotiation).


The ServerKeyExchange message is sent by the server only when the
server Certificate message (if sent) does not contain enough data
```

to allow the client to exchange a premaster secret. This is true for the following key exchange methods:

DHE_DSS
DHE_RSA
DH_anon

It is not legal to send the ServerKeyExchange message for the following key exchange methods:

RSA
DH_DSS
DH_RSA

Other key exchange algorithms, such as those defined in [TLSECC], MUST specify whether the ServerKeyExchange message is sent or not; and if the message is sent, its contents.

Meaning of this message:

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

Structure of this message:

```
enum { dhe_dss, dhe_rsa, dh_anon, rsa, dh_dss, dh_rsa
      /* may be extended, e.g., for ECDH -- see [TLSECC] */
     } KeyExchangeAlgorithm;


struct {
    opaque dh_p<1..2^16-1>;
    opaque dh_g<1..2^16-1>;
    opaque dh_Ys<1..2^16-1>;
} ServerDHParams;     /* Ephemeral DH parameters */
```

dh_p
The prime modulus used for the Diffie-Hellman operation.

dh_g
   The generator used for the Diffie-Hellman operation.

dh_Ys
   The server's Diffie-Hellman public value (g^X mod p).

```
struct {
    select (KeyExchangeAlgorithm) {
        case dh_anon:
            ServerDHParams params;
        case dhe_dss:
        case dhe_rsa:
            ServerDHParams params;
            digitally-signed struct {
                opaque client_random[32];
                opaque server_random[32];
                ServerDHParams params;
            } signed_params;
        case rsa:
        case dh_dss:
        case dh_rsa:
            struct {} ;
           /* message is omitted for rsa, dh_dss, and dh_rsa */
        /* may be extended, e.g., for ECDH -- see [TLSECC] */
    };
} ServerKeyExchange;
```

```
params
   The server's key exchange parameters.
```

```
signed_params
   For non-anonymous key exchanges, a signature over the server's
   key exchange parameters.
```

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 [PKIX] may specify mechanisms for certificates to indicate which digest algorithms are to be used with DSA.

As additional cipher suites are defined for TLS that include new key exchange algorithms, the server key exchange message will be sent if and only if the certificate type associated with the key exchange algorithm does not provide enough information for the client to exchange a premaster secret.

7.4.4. 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 ServerKeyExchange
message (if it is sent; otherwise, this message follows 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<2^16-1>;
    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 SSLv3, 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 12.

Note: Values listed as RESERVED may not be used. They were used in SSLv3.

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

7.4.5. Server Hello Done

When this message will be sent:

The ServerHelloDone message is sent by the server to indicate the end of the ServerHello and associated messages. After sending this message, the server will wait for a client response.

Meaning of this message:

This message means that the server is done sending messages to support the key exchange, and the client can proceed with its phase of the key exchange.

Upon receipt of the ServerHelloDone message, the client SHOULD verify that the server provided a valid certificate, if required, and check that the server hello parameters are acceptable.

Structure of this message:

```
struct { } ServerHelloDone;
```

7.4.6. Client Certificate

When this message will be sent:

This is the first message the client can send after receiving a ServerHelloDone message. 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 7.4.2.

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) or calculating the premaster secret (for non-ephemeral Diffie-Hellman). 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, unless explicitly negotiated otherwise (e.g., [TLSPGP]).
- 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 7.4.4. 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.

7.4.7. Client Key Exchange Message

When this message will be sent:

```
This message is always sent by the client.  It MUST immediately
follow the client certificate message, if it is sent.  Otherwise,
it MUST be the first message sent by the client after it receives
the ServerHelloDone message.
```

Meaning of this message:

With this message, the premaster secret is set, either by direct transmission of the RSA-encrypted secret or by the transmission of Diffie-Hellman parameters that will allow each side to agree upon the same premaster secret.

When the client is using an ephemeral Diffie-Hellman exponent, then this message contains the client's Diffie-Hellman public value. If the client is sending a certificate containing a static DH exponent (i.e., it is doing fixed_dh client authentication), then this message MUST be sent but MUST be empty.

Structure of this message:

The choice of messages depends on which key exchange method has been selected. See Section 7.4.3 for the KeyExchangeAlgorithm definition.

```
struct {
    select (KeyExchangeAlgorithm) {
        case rsa:
            EncryptedPreMasterSecret;
        case dhe_dss:
        case dhe_rsa:
        case dh_dss:
        case dh_rsa:
        case dh_anon:
            ClientDiffieHellmanPublic;
    } exchange_keys;
} ClientKeyExchange;
```

7.4.7.1. RSA-Encrypted Premaster Secret Message

Meaning of this message:

```
If RSA is being used for key agreement and authentication, the
client generates a 48-byte premaster secret, encrypts it using the
public key from the server's certificate, and sends the result in
an encrypted premaster secret message.  This structure is a
variant of the ClientKeyExchange message and is not a message in
itself.
```

Structure of this message:

```
struct {
    ProtocolVersion client_version;
    opaque random[46];
} PreMasterSecret;


client_version
   The latest (newest) version supported by the client.  This is
   used to detect version rollback attacks.


random
   46 securely-generated random bytes.


struct {
    public-key-encrypted PreMasterSecret pre_master_secret;
} EncryptedPreMasterSecret;
```

```
pre_master_secret
   This random value is generated by the client and is used to
   generate the master secret, as specified in Section 8.1.
```

Note: The version number in the PreMasterSecret is the version offered by the client in the ClientHello.client_version, not the version negotiated for the connection. This feature is designed to prevent rollback attacks. Unfortunately, some old implementations use the negotiated version instead, and therefore checking the version number may lead to failure to interoperate with such incorrect client implementations.

Client implementations MUST always send the correct version number in PreMasterSecret. If ClientHello.client_version is TLS 1.1 or higher, server implementations MUST check the version number as described in the note below. If the version number is TLS 1.0 or earlier, server implementations SHOULD check the version number, but MAY have a configuration option to disable the check. Note that if the check fails, the PreMasterSecret SHOULD be randomized as described below.

Note: Attacks discovered by Bleichenbacher [BLEI] and Klima et al. [KPR03] can be used to attack a TLS server that reveals whether a particular message, when decrypted, is properly PKCS#1 formatted, contains a valid PreMasterSecret structure, or has the correct version number.

As described by Klima [KPR03], these vulnerabilities can be avoided by treating incorrectly formatted message blocks and/or mismatched version numbers in a manner indistinguishable from correctly formatted RSA blocks. In other words:

```
1. Generate a string R of 46 random bytes


2. Decrypt the message to recover the plaintext M


3. If the PKCS#1 padding is not correct, or the length of message
   M is not exactly 48 bytes:
      pre_master_secret = ClientHello.client_version || R
   else If ClientHello.client_version <= TLS 1.0, and version
   number check is explicitly disabled:
      pre_master_secret = M
   else:
      pre_master_secret = ClientHello.client_version || M[2..47]
```

Note that explicitly constructing the pre_master_secret with the ClientHello.client_version produces an invalid master_secret if the client has sent the wrong version in the original pre_master_secret.

An alternative approach is to treat a version number mismatch as a PKCS-1

formatting error and randomize the premaster secret completely:

```
1. Generate a string R of 48 random bytes


2. Decrypt the message to recover the plaintext M


3. If the PKCS#1 padding is not correct, or the length of message
   M is not exactly 48 bytes:
      pre_master_secret = R
   else If ClientHello.client_version <= TLS 1.0, and version
   number check is explicitly disabled:
      premaster secret = M
   else If M[0..1] != ClientHello.client_version:
      premaster secret = R
   else:
      premaster secret = M
```

Although no practical attacks against this construction are known, Klima et al. [KPR03] describe some theoretical attacks, and therefore the first construction described is RECOMMENDED.

In any case, a TLS server MUST NOT generate an alert if processing an RSA-encrypted premaster secret message fails, or the version number is not as expected. Instead, it MUST continue the handshake with a randomly generated premaster secret. It may be useful to log the real cause of failure for troubleshooting purposes; however, care must be taken to avoid leaking the information to an attacker (through, e.g., timing, log files, or other channels.)

The RSAES-OAEP encryption scheme defined in [PKCS1] is more secure against the Bleichenbacher attack. However, for maximal compatibility with earlier versions of TLS, this specification uses the RSAES-PKCS1-v1_5 scheme. No variants of the Bleichenbacher attack are known to exist provided that the above recommendations are followed.

Implementation note: Public-key-encrypted data is represented as an opaque vector <0..2^16-1> (see Section 4.7). Thus, the RSA-encrypted PreMasterSecret in a ClientKeyExchange is preceded by two length bytes. These bytes are redundant in the case of RSA because the EncryptedPreMasterSecret is the only data in the ClientKeyExchange and its length can therefore be unambiguously determined. The SSLv3 specification was not clear about the encoding of public-key- encrypted data, and therefore many SSLv3 implementations do not include the length bytes – they encode the RSA-encrypted data directly in the ClientKeyExchange message.

This specification requires correct encoding of the EncryptedPreMasterSecret complete with length bytes. The resulting PDU is incompatible with many SSLv3 implementations. Implementors

upgrading from SSLv3 MUST modify their implementations to generate and accept the correct encoding. Implementors who wish to be compatible with both SSLv3 and TLS should make their implementation's behavior dependent on the protocol version.

Implementation note: It is now known that remote timing-based attacks on TLS are possible, at least when the client and server are on the same LAN. Accordingly, implementations that use static RSA keys MUST use RSA blinding or some other anti-timing technique, as described in [TIMING].

7.4.7.2. Client Diffie-Hellman Public Value

Meaning of this message:

```
This structure conveys the client's Diffie-Hellman public value
(Yc) if it was not already included in the client's certificate.
The encoding used for Yc is determined by the enumerated
PublicValueEncoding.  This structure is a variant of the client
key exchange message, and not a message in itself.
```

Structure of this message:

```
enum { implicit, explicit } PublicValueEncoding;


implicit
   If the client has sent a certificate which contains a suitable
   Diffie-Hellman key (for fixed_dh client authentication), then
   Yc is implicit and does not need to be sent again.  In this
   case, the client key exchange message will be sent, but it MUST
   be empty.


explicit
   Yc needs to be sent.


struct {
    select (PublicValueEncoding) {
        case implicit: struct { };
        case explicit: opaque dh_Yc<1..2^16-1>;
    } dh_public;
} ClientDiffieHellmanPublic;


dh_Yc
   The client's Diffie-Hellman public value (Yc).
```

7.4.8. 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 key exchange message.
```

Structure of this message:

```
struct {
     digitally-signed struct {
         opaque handshake_messages[handshake_messages_length];
     }
} CertificateVerify;
```

```
Here handshake_messages refers to all handshake messages sent or
received, starting at client hello and up to, but not including,
this message, including the type and length fields of the
handshake messages.  This is the concatenation of all the
Handshake structures (as defined in Section 7.4) exchanged thus
far.  Note that this requires both sides to either buffer the
messages or compute running hashes for all potential hash
algorithms up to the time of the CertificateVerify computation.
Servers can minimize this computation cost by offering a
restricted set of digest algorithms in the CertificateRequest
message.
```

```
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 [PKIX] may specify mechanisms for certificates to indicate which digest algorithms are to be used with DSA.

7.4.9. Finished

When this message will be sent:

A Finished message is always sent immediately after a change cipher spec message to verify that the key exchange and authentication processes were successful. It is essential that a change cipher spec message be received between the other handshake messages and the Finished message.

Meaning of this message:

The Finished message is the first one protected with the just negotiated algorithms, keys, and secrets. 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.

Structure of this message:

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


verify_data
   PRF(master_secret, finished_label, Hash(handshake_messages))
      [0..verify_data_length-1];


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


Hash denotes a Hash of the handshake messages. For the PRF
defined in Section 5, the Hash MUST be the Hash used as the basis
for the PRF. Any cipher suite which defines a different PRF MUST
also define the Hash to use in the Finished computation.


In previous versions of TLS, the verify_data was always 12 octets
```

long. In the current version of TLS, it depends on the cipher suite. Any cipher suite which does not explicitly specify verify_data_length has a verify_data_length equal to 12. This includes all existing cipher suites. Note that this representation has the same encoding as with previous versions. Future cipher suites MAY specify other lengths but such length MUST be at least 12 bytes.

handshake_messages
: All of the data from all messages in this handshake (not including any HelloRequest messages) up to, but not including, this message. This is only data visible at the handshake layer and does not include record layer headers. This is the concatenation of all the Handshake structures as defined in Section 7.4, exchanged thus far.

It is a fatal error if a Finished message is not preceded by a ChangeCipherSpec message at the appropriate point in the handshake.

The value handshake_messages includes all handshake messages starting at ClientHello up to, but not including, this Finished message. This may be different from handshake_messages in Section 7.4.8 because it would include the CertificateVerify message (if sent). Also, the handshake_messages for the Finished message sent by the client will be different from that for the Finished message sent by the server, because the one that is sent second will include the prior one.

Note: ChangeCipherSpec messages, alerts, and any other record types are not handshake messages and are not included in the hash computations. Also, HelloRequest messages are omitted from handshake hashes.