Scientific Journal of Impact Factor (SJIF): 5.71

e-ISSN (O): 2348-4470
p-ISSN (P): 2348-6406

International Journal of Advance Engineering and Research Development

Volume 6, Issue 03, March -2019

# TRANSPORT LAYER SECURITY 1.3

Lokesh Kumar[1]

[1]Department of Computer Science & Engineering, College of Engineering and Technology, Bhubaneswar

**Abstract-** *This paper gives a detailed explanation of Transport Layer Security (TLS) protocol version 1.3[1], standardised in 2018. TLS allows client/server applications to communicate over the Internet in such a way that is designed to prevent eavesdropping, tampering, and message forgery. TLS 1.3 comes with improved security and faster speed. Its numerous features and improvements have been detailed out as it is will be standard for the next few decades.*

***Keywords****-TLS, PSK, Handshake, SSL, AEAD, HTTP*

## I. INTRODUCTION

Engineers designed TLS by using tools built by mathematicians. Many of the earliest blueprint decisions were made using heuristics and there was an incomplete understanding of the designing process of robust security protocols. We cannot say its the fault of the protocols designers such as Paul Kocher, Phil Karlton, Alan Freier, Tim Dierks, and others, as the industry was under the learning phase. When TLS was designed, formal papers on the design of secure authentication protocols like Hugo Krawczyk's landmark SIGMA paper were still years away[2].

A secure channel between two communicating peers; the only requirement from the underlying transport is a reliable, in-order data stream. Specifically, the secure channel should provide the following properties:

1-Authentication: Authentication of the server side of the channel is needed; the client side is optionally authenticated. Authentication can happen via asymmetric cryptography (e.g., RSA[3], the Elliptic Curve Digital Signature Algorithm (ECDSA)[4], or the Edwards-Curve Digital Signature Algorithm (EdDSA) [5]) or a symmetric pre-shared key (PSK).

2-Confidentiality: Data thats sent over the channel after establishment is only perceptible to the endpoints. TLS does'nt hide the data length it is transmitting, though endpoints can pad TLS records so as to obscure lengths and improve protection against traffic analysis techniques.

3-Integrity: Data that is sent over the channel after establishment cannot be altered by attackers without detection.

TLS consists of two primary components:

1-A handshake protocol (Section 3) that authenticates the communicating parties, negotiates cryptographic modes and parameters, and establishes shared keying material. The handshake protocol is tailored to resist tampering; an active attacker should'nt be able to force the peers to negotiate different parameters than they would if the connection were'nt under attack.

2-A record protocol (Section 4) that uses the parameters established by the handshake protocol to protect traffic between the communicating peers. The record protocol divides traffic up into a series of records, each of which is independently protected using the traffic keys.

## II. PROTOCOL OVERVIEW

The cryptographic parameters employed by the secure channel are generated by the TLS handshake protocol. This sub-protocol of TLS is utilized by the client and server when they are first communicating with each other. The handshake protocol allows the peers to negotiate a protocol version, select cryptographic algorithms, optionally authenticate each other, and establish shared secret keying material. Once the handshake is complete, the peers use the established keys to protect the application-layer traffic.

TLS supports three basic key exchange modes:

1- (EC)DHE (Diffie-Hellman over either finite fields or elliptic curves)

2- PSK-only

3- PSK with (EC)DHE

In figure 1 below, "+" indicates extensions sent in the previous noted message, "*" indicates optional messages or extensions that are not always sent,"{}" indicates messages protected using keys derived from a sender handshake traffic secret and "[]" indicates messages protected using keys derived from sender application traffic secret N.

 

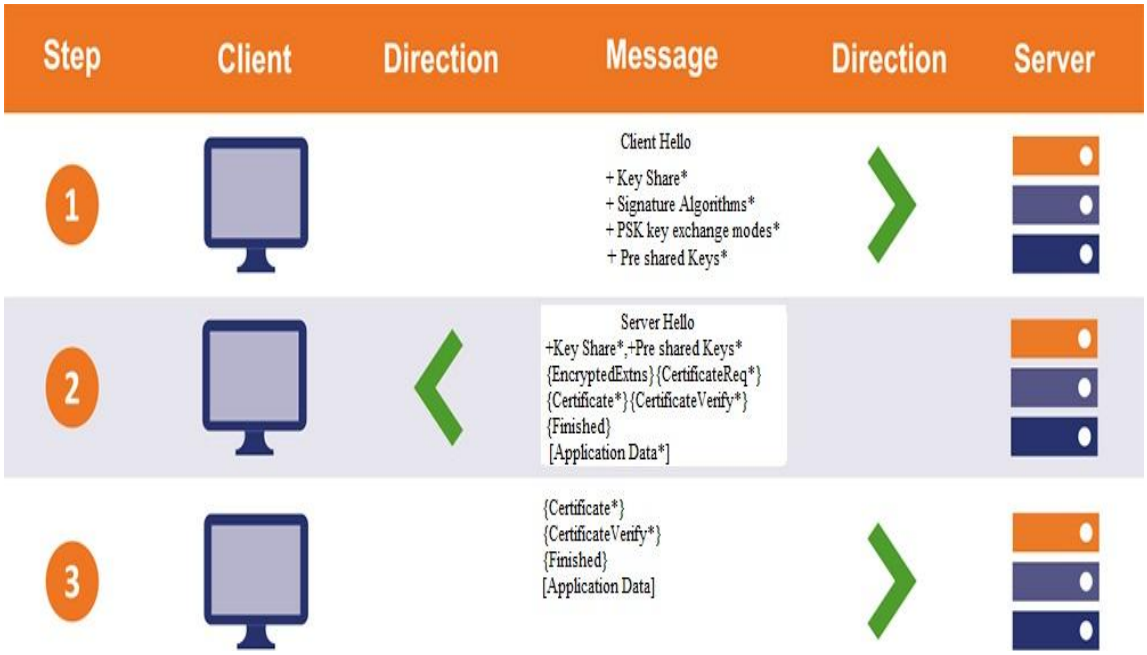


*Figure 1. Message flow for full TLS handshake*

The handshake has three phases-
1-Key Exchange-Here shared key material is formed and cryptographic parameters are chosen.Beyond this phase everything is encrypted.
2-Server Parameters- Here other handshake parameters (whether the client is authenticated, application-layer protocol support, etc.) are established.
3-Authentication: Authenticate the server (and, optionally, the client) and provide key confirmation and handshake integrity.
In the Key Exchange phase, the client sends the Client Hello (Section 3.1.2) message, which contains a random nonce (Client Hello. random).It also sends its offered protocol versions; a list of symmetric cipher/HKDF hash pairs; either a set of Diffie-Hellman key shares (in the "key_share" extension), a set of pre-shared key labels (in the "pre_shared_key" extension), or both; and potentially additional extensions. Additional fields and/or messages may be included for middlebox compatibility.
The server processes the Client Hello and determines the apt cryptographic parameters to be used for the connection. It then responds with its own Server Hello (Section 3.1.3), which indicates the negotiated connection parameters. The fusion of the Client Hello and the Server Hello determines the shared keys. If (EC)DHE key establishment is in use, then the Server Hello contains a "key share" extension with the server's ephemeral Diffie-Hellman share; the server's share must be in the same group as one of the client's shares. If PSK key establishment is in use, then the Server Hello will contain a "pre shared_key" extension indicating which of the client's offered PSKs was selected. Note that implementations can use (EC)DHE and PSK together, in which case both extensions will be supplied.
The server then sends two messages to establish the Server Parameters:
1-EncryptedExtensions: It responds to Client Hello extensions that are not required to determine the cryptographic parameters, other than those that are specific to individual certificates.
2-Certificate Request: if certificate-based client authentication is desired, the desired parameters for that certificate is held here. This message is omitted if client authentication is not desired.
Finally, the client and server exchange Authentication messages. TLS uses the same set of messages every time that certificate-based authentication is needed. (PSK-based authentication happens as a side effect of key exchange.) Specifically:
Certificate: The certificate of the endpoint and any per-certificate extensions. This message is omitted by the server if not authenticating with a certificate and by the client if the server did not send Certificate Request (thus indicating that the client should not authenticate with a certificate). Note that if raw public keys [6] or the cached information extension [7] are in use, then this message will not contain a certificate but rather some other value corresponding to the server's long-term key.
Certificate Verify: A signature over the entire handshake using the private key corresponding to the public key in the Certificate message. This message is omitted if the endpoint is not authenticating via a certificate.
Finished: A MAC (Message Authentication Code) over the entire handshake. This message provides key confirmation, binds the endpoint's identity to the exchanged keys, and in PSK mode also authenticates the handshake.
Upon receiving the server's messages, the client responds with its Authentication messages, namely Certificate and Certificate Verify (if requested), and Finished.

 

At this point, the handshake is complete, and the client and server derive the keying material required by the record layer to exchange application-layer data protected through authenticated encryption. Application Data must not be forwarded prior to sending the Finished message, except as specified in (Section 2.32). While the server may send Application Data prior to receiving the client's Authentication messages, any data sent at that point is, of course, being sent to an unauthenticated peer.

**2.1. Resumption and Pre-Shared Key (PSK)**

Even though TLS PSKs can be initiated out of band, PSKs can also be established in a previous connection and then used to establish a newborn connection ("session resumption" or "resuming" with a PSK). Once a handshake has completed, the server can send the client a PSK identity that corresponds to a unique key derived from the initial handshake. After that the client can then use that PSK identity in future handshakes to negotiate the use of the associated PSK. If the server obtains the PSK, then the security context of the new connection is cryptographically tied to the original connection and the key derived from the initial handshake is employed to bootstrap the cryptographic state instead of a full handshake. In TLS 1.2, this functionality was provided by session IDs and session tickets [8]. Both mechanisms are obsoleted in TLS 1.3.

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


*Figure 2. Message flow for resumption and PSK*

As the server is authenticating via a PSK, it does not send a Certificate or a Certificate Verify message. When a client offers resumption via a PSK, it SHOULD also supply a "key_share" extension to the server to allow the server to decline resumption and fall back to a full handshake, if needed. The server responds with a "pre_shared_key" extension to negotiate the use of PSK key establishment and can respond with a "key_share" extension to do (EC) DHE key establishment, thus providing forward secrecy.

When PSKs are allocated out of band, the PSK identity and the KDF hash algorithm to be used with the PSK must also be provisioned.

**2.2. 0-RTT Data**

A zero round trip time (0-RTT) mode was appended, saving a round trip at connection setup for some application data, at the cost of certain security properties.When clients and servers share a PSK (either obtained externally or via a previous handshake), TLS version 1.3 allows clients to send data on the first flight ("early data"). The client uses the PSK to authenticate the server and encrypts the early data. As shown in Figure 3, the 0-RTT data is just added to the 1-RTT handshake in the first flight. The rest of the handshake uses the identical messages as for a 1-RTT handshake with PSK resumption.

 


*Figure 3. Message Flow for a 0-RTT handshake*

The security properties of 0-RTT data are weak because its not a forwarded secret as its encrypted using keys derived using the PSK.Also there are no guarantees of non replay between connections. If an malicious person captures a 0-RTT packet that was sent to server, they can replay it and there's a chance that the server will accept it as valid. This can have interesting negative consequences.
There are two potential threats to be concerned with:
1-Duplication of 0-RTT data and starting a replay attack.
2- Network attackers taking advantage of client retry behavior to arrange for the server to receive multiple copies of an application message. This threat exists already to some extent because clients that value robustness respond to network errors by attempting to retry requests. Here, 0-RTT adds an additional dimension for any server system which does not maintain globally consistent server state. Specifically, if a server system has diverse zones where tickets from zone A will not be accepted in zone B, then an attacker can duplicate a Client Hello and early data intended for A to both A and B. At A, the data will be accepted in 0-RTT, but at B the server will reject 0-RTT data and instead force a full handshake. If the attacker bars the Server Hello from A, then the client will complete the handshake with B and probably retry the request, leading to duplication on the server system as a whole.
The first category of attack can be prevented by sharing state to guarantee that the 0-RTT data is accepted at most once. Servers should impart that level of replay safety by implementing one of the methods described in this section or by equivalent means. It is understood, however, that due to concerns regarding operations not all deployments will maintain state at that level. Therefore, in normal operation, clients will not know which, if any, of these mechanisms servers actually implement and hence must only send early data which they deem safe to be replayed.
In addition to the direct effects of replays, there is a class of attacks where even operations normally considered idempotent could be exploited by a large number of replays (timing attacks, resource limit exhaustion and others.Those can be mitigated by ensuring that every 0-RTT payload can be replayed only a limited number of times. The "at most once per server instance" guarantee is a base requirement; servers should limit 0-RTT replays further when feasible.
The second class of attack cannot be prevented at the TLS layer and must be dealt with by any application by implementing some sort of anti-replay defense.

## III. HANDSHAKE PROTOCOL

The handshake protocol is used to negotiate the security parameters for a new connection. Handshake messages are dispensed to the TLS record layer, where they are encapsulated within one or more TLS Plaintext or TLS Ciphertext structures which are processed and transmitted as specified by the current active connection state.

### 3.1. Key Exchange Messages

The key exchange messages are used to ascertain the security capabilities of the client and the server and to establish shared secrets, including the traffic keys used to shield the rest of the handshake and the data.

**3.1.1. Cryptographic Negotiation**-In TLS, the cryptographic negotiation proceeds by the client offering the following four sets of options in its Client Hello:
1-A list of cipher suites which indicates the AEAD algorithm/HKDF hash pairs which the client supports.
2-A "supported_groups" extension which indicates the(EC)DHE groups which the client supports for key exchange, ordered from most preferred to least preferred.And a "key_share" extension which contains (EC)DHE shares for some or all of these groups.It contains the endpoint's cryptographic parameters.
3-A "signature_algorithms" extension which points to the signature algorithms which the client can accept.

TLS 1.3 provides two extensions for denoting which signature algorithms may be used in digital signatures. The "signature_algorithms_cert" extension applies to signatures in certificates, and the "signature_algorithms" extension, which originally appeared in TLS 1.2, applies to signatures in Certificate Verify messages. The keys found in certificates must also be of suitable type for the signature algorithms they are used with. This is a distinct issue for RSA keys and PSS signatures, as described below. If no "signature_algorithms_cert" extension is existent, then the "signature_algorithms" extension also applies to signatures appearing in certificates. Clients which fancy the server to authenticate itself via a certificate must send the "signature_algorithms" extension. If a server is authenticating via a certificate and the client has not sent a "signature_algorithms" extension, then the server must terminate the handshake with a "missing_extension" alert. The "signature_algorithms_cert" extension was added to allow implementations which reinforced different sets of algorithms for certificates and in TLS itself to clearly signal their capabilities.
4-A "pre_shared_key" extension contains a list of symmetric key identities known to the client. It is used to negotiate the identity of the pre-shared key to be used with a given handshake in association with PSK key establishment.And a "psk_key_exchange_modes" extension which indicates the key exchange modes that may be used with PSKs. In order to use PSKs, clients must also dispatch a "psk_key_exchange_modes" extension. The semantics of this extension are that the client only supports the use of PSKs with these modes, which restricts both the use of PSKs offered in this Client Hello and those which the server might supply via NewSessionTicket.
If the server does not select a PSK, then the first three of these options are orthogonal: the server independently selects a cipher suite, an (EC)DHE group and key share for key establishment,
and a signature algorithm/certificate pair to authenticate itself to the client. If there is no overlap between the obtained "supported_groups" and the groups supported by the server, then the server must abort the handshake with a "handshake_failure" or an "insufficient_security" alert. If the server selects a PSK, then it must also select a key establishment mode from the set indicated by the client's "psk_key_exchange_modes" extension (at present, PSK alone or with (EC)DHE). If the PSK can be used without (EC)DHE, then non-overlap in the "supported_groups" parameters need not be fatal, as it is in the non-PSK case discussed in the previous paragraph. If the server selects an (EC)DHE group and the client did not offer a compatible "key_share" extension in the opening Client Hello, the server must respond with a HelloRetryRequest message.If the server successfully picks parameters and does not require a HelloRetryRequest, it indicates the selected parameters in the Server Hello as follows-
1-If PSK is being used, then the server will send a "pre_shared_key" extension indicating the selected key.
2-When (EC)DHE is in use, the server will also provide a "key_share" extension. If PSK is not being used, then (EC)DHE and certificate-based authentication are always used.
3-When authenticating via a certificate, the server will send the Certificate (Section 3.4.2) and Certificate Verify (Section 3.4.3) messages. In TLS 1.3, either a PSK or a certificate is always used, but not both.
If the server is unable to broker a supported set of parameters (i.e., there is no overlap between the client and server parameters), it must abort the handshake with either a "handshake_failure" or "insufficient_security" fatal alert (see Section 5).

**3.1.2. Client Hello**-When a client first connects to a server, it is required to send the Client Hello as its first TLS message. The client will also send a Client Hello when the server has responded to its Client Hello with a Hello Retry Request. Because TLS 1.3 forbids renegotiation, if a server has negotiated TLS 1.3 and receives a Client Hello at any other time duration, it must terminate the connection with an "unexpected_message" alert.
If a server sets up a TLS connection with a previous version of TLS and obtains a TLS 1.3 Client Hello in a renegotiation, it must retain the previous protocol version. in particular, it must not negotiate tls 1.3. After sending the Client Hello message, the client waits for a Server Hello or helloretryrequest message. if early data is in use, the client may transmit early application data (section 2.2) while waiting for the next handshake message.

**3.1.3. Server Hello**-The server will send this message in response to a Client Hello message to proceed with the handshake if it is able to negotiate an acceptable set of handshake parameters based on the Client Hello.

**3.2. Extensions**
A number of TLS messages contains a tag-length-value encoded extensions structures. Extensions are structured in a request or response fashion, though some extensions are just indications with no corresponding response. The client sends its extension requests in the Client Hello message, and the server sends its extension responses in the server hello, encrypted extensions, hello retry request, and certificate messages. the server sends extension requests in the certificate request message which a client may respond to with a certificate message. the server may also send unsolicited extensions in the New Session Ticket, though the client does not respond directly to these.

 

*Table 1. Extension type maintained by IANA*

| Extension | TLS 1.3 |
|---|---|
| server_name [RFC6066] | CH, EE |
| max_fragment_length [RFC6066] | CH, EE |
| status_request [RFC6066] | CH, CR, CT |
| supported_groups [RFC7919] | CH, EE |
| signature_algorithms (RFC 8446) | CH, CR |
| use_srtp [RFC5764] | CH, EE |
| heartbeat [RFC6520] | CH, EE |
| application_layer_protocol_negotiation [RFC7301] | CH, EE |
| signed_certificate_timestamp [RFC6962] | CH, CR, CT |
| client_certificate_type [RFC7250] | CH, EE |
| server_certificate_type [RFC7250] | CH, EE |
| padding [RFC7685] | CH |
| key_share (RFC 8446) | CH, SH, HRR |
| pre_shared_key (RFC 8446) | CH, SH |
| psk_key_exchange_modes (RFC 8446) | CH |
| early_data (RFC 8446) | CH, EE, NST |
| cookie (RFC 8446) | CH, HRR |
| supported_versions (RFC 8446) | CH, SH, HRR |
| certificate_authorities (RFC 8446) | CH, CR |
| oid_filters (RFC 8446) | CR |
| post_handshake_auth (RFC 8446) | CH |
| signature_algorithms_cert (RFC 8446) | CH, CR |

The table above indicates the messages where a given extension may appear, using the following notation: CH (Client Hello), SH (Server Hello), EE (Encrypted Extensions), CT (Certificate), CR (Certificate Request), NST (New Session Ticket), and HRR (Hello Retry Request). If an implementation receives an extension which it recollects and which is not specified for the message in which it appears, it must abort the handshake with an "illegal_parameter" alert.
few of the extensions are explained below:
1-Certificate Authorities-The "certificate_authorities" extension is used to indicate the certificate authorities (CAs) which an endpoint supports and which should be used by the receiving endpoint to guide certificate selection.
The body of the "certificate_authorities" extension comprises of a Certificate Authorities Extension structure.
The client may send the "certificate_authorities" extension in the Client Hello message. The server may send it in the Certificate Request message.
2-Post-Handshake Client Authentication-The "post_handshake_auth" extension is used to indicate that a client is willing to execute post-handshake authentication. Servers must not send a post-handshake certificate request to clients which do not present this extension. servers must not send this extension. The "extension_data" field of the "post_handshake_auth" extension is zero length.
3-Supported Groups-When sent by the client, the "supported_groups" extension indicates the groups which the client supports for key exchange, ordered from most preferred to least preferred. Elliptic Curve Groups (ECDHE): Indicates support for the relative named curve, defined in either FIPS 186-4 [9] or [10]. Values 0xFE00 through 0xFEFF are bespoken for Private Use [11]. Finite Field Groups (DHE): It indicates support for the corresponding finite field group, defined in [12]. Values 0x01FC through 0x01FF are reserved for private use.

### 3.3. Server Parameters

The next two messages from the server, Encrypted Extensions and Certificate Request, holds information from the server that determines the rest of the handshake. These messages are encrypted with keys derived from the server handshake traffic secret.

**3.3.1. Encrypted Extensions-**In all handshakes, the server must send the Encrypted Extensions message promptly after the Server Hello message. This is the opening message that is encrypted under keys derived from the server handshake traffic secret. The Encrypted Extensions message contains extensions that can be protected, i.e., any which are not needed to establish the cryptographic context but which are not associated with individual certificates. The client must

 

check Encrypted Extensions for the presence of any prohibited or forbidden extensions and if any are found must abort the handshake with an "illegal_parameter" alert.

**3.3.2. Certificate Request**-A server which is authenticating with a certificate may optionally request a certificate from the client. This message, if sent, must follow after the Encrypted Extensions.

**3.4. Authentication Messages**

TLS uses a routine set of messages for authentication, key confirmation, and handshake integrity: Certificate, Certificate Verify, and Finished. These three messages are always sent as the last messages in their handshake flight. The Certificate and Certificate Verify messages are only sent under certain circumstances. The Finished message is always sent as part of the Authentication Block. These messages are encrypted under keys derived from the [sender]_handshake_traffic_secret.

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

1-The certificate and signing key to be used.

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

3-A Base Key to be used to compute a MAC key.

Based on these inputs, the messages then contain:

Certificate: The certificate to be used for authentication, and any supporting certificates in the chain. Note that certificate-based client authentication is not available in PSK handshake flows (including 0-RTT).

2-Certificate Verify: A signature over the value Transcript-Hash (Handshake Context, Certificate).

3-Finished: A MAC over the value Transcript-Hash (Handshake Context, Certificate, Certificate Verify) using a MAC key derived from the Base Key.

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

For concreteness, the transcript hash is always taken from the following sequence of handshake messages, starting at the first Client Hello and including only those messages that were sent: Client Hello, Hello Retry Request, Client Hello, Server Hello, Encrypted Extensions, server Certificate Request, server Certificate, server Certificate Verify, server Finished, End Of Early Data, client Certificate, client Certificate Verify, client Finished.

**3.4.2. Certificate-** This message conveys the endpoint's certificate chain to the peer. The server must send a Certificate message whenever the agreed-upon key exchange method uses certificates for authentication. The client must send a Certificate message if and only if the server has requested client authentication via a Certificate Request message. If the server requests client authentication but no suitable certificate is available, the client must send a Certificate message containing no certificates (i.e., with the certificate_list" field having length 0). A Finished message must be sent regardless of whether the Certificate message is empty.

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

The digital signature is then computed over the concatenation of:

1-A string that consists of octet 32 (0x20) repeated 64 times

2-The context string

3-A single 0 byte which serves as the separator

4-The content to be signed

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

On the sender side, the process for computing the signature field of the Certificate Verify message takes as input:

1-The content covered by the digital signature

2-The private signing key corresponding to the certificate sent in the previous message

The receiver of a Certificate Verify message must verify the signature field. The verification process takes as input:

1-The content covered by the digital signature

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

3-The digital signature received in the signature field of the Certificate Verify message

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

 

**3.4.4. Finished-** The Finished message is the final message in the Authentication Block. It is essential for providing authentication of the handshake and of the computed keys.
Recipients of Finished messages must verify that the contents are correct and if incorrect must terminate the connection with a "decrypt_error" alert.
Once a side has sent its Finished message and has received and validated the Finished message from its peer, it may begin to send and receive Application Data over the connection.

## IV. RECORD PROTOCOL

The TLS record protocol takes messages to be transmitted, breaks the data into manageable blocks, protects the records, and transmits the result. Received data is verified, decrypted, reassembled, and then delivered to higher-level clients. TLS records are typed, which allows various higher-level protocols to be multiplexed over the same record layer. This document defines four content types: handshake, application_data, alert, and change_cipher_spec. The change_cipher_spec record is used only for compatibility purposes.
Implementations must not send record types not defined unless negotiated by some extension. If a TLS implementation receives an unexpected record type, it must terminate the connection with an "unexpected_message" alert.

### 4.1. Record Layer

The record layer fragments information blocks into TLS Plaintext records carrying data in chunks of 2^14 bytes or less. Message boundaries are handled differently depending on the underlying Content Type. Any future content types must specify appropriate rules. These rules are stringent than what was enforced in TLS 1.2. Handshake messages may be fused into a single TLS Plaintext record or fragmented across several records, provided that:
1-Handshake messages must not be interleaved with other record types. That is, if a handshake message is divided over two or more records, there must not be any other records between them.
2-Handshake messages must not span key changes. Implementations must verify that all messages instantly preceding a key change align with a record boundary; if not, then they must terminate the connection with an "unexpected_message" alert. Because the Client Hello, End Of Early Data, Server Hello, Finished, and Key Update messages can immediately precede a key change, implementations must send these messages in alignment with a record boundary.
Implementations must not send zero-length fragments of hndshake types, even if those fragments contain padding. Alert messages (Section 5) must not be fragmented across records, and multiple alert messages must not be combined into a single TLS Plaintext record. In other words, a record with an Alert type must include exactly one message.
Application data messages contain data that is obscure to TLS. Application Data messages are always protected. Zero-length fragments of Application Data may be sent, as they are potentially useful as a traffic analysis countermeasure. Application data fragments may be splited across multiple records or merged into a single record. When record protection has not yet been engaged, TLS Plaintext structures are written directly onto the wire. Once record protection has started, TLS Plaintext records are protected and sent. Also application data records must not be written to the wire unprotected

### 4.2. Record Payload Protection

The record protection functions translate a TLS Plaintext structure into a TLS Ciphertext structure. 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 Associated Data" (AEAD) [13]. AEAD functions provide a unified encryption and authentication operation which turns plaintext into authenticated ciphertext and back again. Each encrypted record consists of a plaintext header followed by an encrypted body, which itself contains a type and optional padding.
EAD algorithms take as input a single key, a nonce, a plaintext, and "additional data" to be included in the authentication check, as described in Section 2.1 of [13]. The key is either the client_write_key or the server_write_key, the nonce is derived from the sequence number and the client_write_iv or server_write_iv, and the additional data input is the record header. The plaintext input to the AEAD algorithm is the encoded TLS Inner Plaintext structure.
The AEAD output consists of the ciphertext output from the AEAD encryption operation. The length of the plaintext is greater than the relative TLS Plaintext.length due to the inclusion of TLS Inner Plaintext. type and any padding supplied by the sender. The length of the AEAD output will usually be larger than the plaintext, but by an amount that varies with the AEAD algorithm.
Since the ciphers might incorporate padding, the amount of overhead could vary with different lengths of plaintext. The encrypted_record field of TLS Ciphertext is set to AEAD Encrypted.
In order to decrypt and verify, the cipher takes as input the key, nonce, additional data, and the AEAD Encrypted value. The output is either the plaintext or an error indicating that the decryption failed. There is no separate integrity check. If the decryption fails, the receiver must terminate the connection with a "bad_record_mac" alert.
An AEAD algorithm used in TLS 1.3 MUST NOT produce an expansion greater than 255 octets. An endpoint that receives a record from its peer with TLS Ciphertext. length larger than 2^14 + 256 octets MUST terminate the connection with a "record_overflow" alert. This limit is derived from the maximum TLS Inner Plaintext length of 2^14 octets + 1 octet for Content Type + the maximum AEAD expansion of 255 octets. Per-Record Nonce

 

A 64-bit sequence number is maintained separately for reading and writing records. The appropriate sequence number is incremented by one after reading or writing each record. Each sequence number is set to zero at the beginning of a connection and whenever the key is changed; the first record transmitted under a particular traffic key MUST use sequence number 0.
Because the size of sequence numbers is 64-bit, they should not wrap. If a TLS implementation would need to wrap a sequence number, it must either rekey or terminate the connection. The per-record nonce for the AEAD construction is formed as follows:
1. The 64-bit record sequence number is encoded in network byte order and padded to the left with zeros to iv_length.
2. The padded sequence number is XORed with either 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.

**4.3. Record Padding**
All encrypted TLS records can be padded to inflate the size of the TLS Ciphertext. This allows the sender to cloud the size of the traffic from an observer.
When generating a TLS Ciphertext record, implementations may choose to pad. An unpadded record is just a record with a padding length of zero. Padding is a string of zero-valued bytes appended to the Content Type field before encryption. Implementations must set the padding octets to all zeros before encrypting.
Application data records may contain a zero-length TLS Inner Plaintext. content if the sender desires. This permits generation of plausibly sized cover traffic in contexts where the presence or absence of activity may be sensitive. Implementations must not forward handshake and Alert records that have a zero-length TLS Inner Plaintext. content; if such a message is received, the receiving implementation must abort the connection with an "unexpected_message" alert. The padding sent is automatically verified by the record protection mechanism; upon successful decryption of a TLS Ciphertext. encrypted_record, the receiving implementation scans the field from the end toward the beginning until it finds a non-zero octet. This non-zero octet is the content type of the message. This padding scheme was selected because it allows padding of any encrypted TLS record by an arbitrary size (from zero up to TLS record size limits) without introducing new content types. The design also enforces all-zero padding octets, which allows for quick detection of padding errors.
Implementations must limit their scanning to the cleartext returned from the AEAD decryption. If a receiving implementation does not find a non-zero octet in the cleartext, it MUST terminate the connection with an "unexpected_message" alert.
The presence of padding does not change the overall record size limitations: the full encoded TLS Inner Plaintext MUST NOT exceed 2^14 + 1 octets. If the maximum fragment length is reduced as, for example, by the record_size_limit extension from [14] then the reduced limit applies to the full plaintext, including the content type and padding. Selecting a padding policy that suggests when and how much to pad is a complex topic and is beyond the scope of this specification. If the application-layer protocol on top of TLS has its own padding, it may be preferable to pad Application Data TLS records within the application layer. Padding for encrypted Handshake or Alert records must still be handled at the TLS layer, though. Later documents may define padding selection algorithms or define a padding policy request mechanism through TLS extensions or some other means.

## V. ALERT PROTOCOL

TLS provides an Alert content type to indicate closure information and errors. Like other messages, alert messages are encrypted as specified by the current connection state.
Alert messages convey a description of the alert and a legacy field that conveyed the severity level of the message in previous versions of TLS. Alerts are divided into two classes: closure alerts and error alerts. In TLS 1.3, the severity is implicit in the type of alert being sent, and the "level" field can safely be ignored. The
"close_notify" alert is used to indicate orderly closure of one direction of the connection. Upon receiving such an alert, the TLS implementation should indicate end-of-data to the application.
Error alerts indicate abortive closure of the connection (see Section 5.2). Upon receiving an error alert, the TLS implementation should indicate an error to the application and must not allow any further data to be sent or received on the connection. Servers and clients must forget the secret values and keys established in failed connections, with the exception of the PSKs associated with session tickets, which should be discarded if possible. All the alerts listed in Section 5.2 must be sent with AlertLevel=fatal and must be treated as error alerts when received regardless of the AlertLevel in the message. Unknown Alert types must be treated as error alerts.

**5.1. Closure Alerts**
The client and the server must share knowledge that the connection is ending in order to avoid a truncation attack.
close_notify: This alert notifies the recipient that the sender will not send any more messages on this connection. Any data received after a closure alert has been received must be ignored.
user_canceled: This alert notifies the recipient that the sender is canceling the handshake for some reason unrelated to a protocol failure. If a user cancels an operation after the handshake is complete, just closing the connection by sending a

"close_notify" is more appropriate. This alert should be followed by a "close_notify". This alert generally has AlertLevel=warning. Either party may initiate a close of its write side of the connection by sending a "close_notify" alert. Any data received after receiving a closure alert must be ignored. If a transport-level close is received prior to a "close_notify", the receiver cannot know that all the data that was sent has been received.
Each party MUST send a "close_notify" alert before closing its write side of the connection, unless it has already sent some error alert. This does not have any effect on its read side of the connection. Note that this is a change from versions of TLS prior to TLS 1.3 in which implementations were required to react to a "close_notify" by discarding pending writes and sending an immediate "close_notify" alert of their own. That previous requirement could cause truncation in the read side. Both parties need not wait to receive a "close_notify" alert before closing their read side of the connection, though doing so would introduce the possibility of truncation.
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 a "close_notify" alert before indicating end-of-data to the application layer. 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.

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

## VI. IMPROVEMENTS

TLS 1.3 contains improved security and speed. The major differences include:

- The list of supported symmetric algorithms has been pruned of all legacy algorithms. The remaining algorithms all employ Authenticated Encryption with Associated Data (AEAD) algorithms.
- A zero-RTT (0-RTT) mode was added, saving a round-trip at connection setup for some application data at the cost of some security properties.
- Static RSA and Diffie-Hellman cipher suites have been axed; all public-key based key exchange mechanisms now provide forward secrecy.
- All handshake messages after the Server Hello are now encrypted.
- Key derivation functions have been re-designed, with the HMAC-based Extract-and-Expand Key Derivation Function (HKDF) being used as a primitive.
- The handshake state machine has been restructured to be more consistent and remove superfluous messages.
- ECC is in the base spec and includes new signature algorithms. Point format negotiation has been removed in support of single point format for each curve.
- Compression, custom DHE groups, and DSA have been removed, RSA padding now uses PSS.
- TLS 1.2 version negotiation verification mechanism was deplored in favor of a version list in an extension.
- Session resumption with and without server-side state and the PSK-based ciphersuites of earlier versions of TLS have been replaced by a single new PSK exchange.

## VII. DEPLOYABILITY

TLS 1.3 is a radical departure from TLS 1.2 and in order to be deployed widely, it has to be backwards compatible with existing software. One of the reasons TLS 1.3 has taken so long to go from draft to final publication was the fact that some existing software; namely middleboxes network appliances designed to monitor and sometimes intercept HTTPS traffic inside corporate environments and mobile networks, wasn't playing nicely with the new changes. Even small changes to the TLS 1.3 protocol that were visible on the wire (such as eliminating the redundant Change CipherSpec message, bumping the version from 0x0303 to 0x0304) ended up causing connection issues for some people. Despite the fact that future flexibility has been built into the TLS spec, some implementations made incorrect assumptions about how to handle future TLS versions. The phenomenon responsible for this change is called ossification, a protocol design antipattern. If a protocol is designed with a flexible structure, but that flexibility is never used in practice, some implementation is going to assume it is fixed. To accommodate these changes, TLS 1.3 was altered to look a lot like TLS 1.2 session.

## VIII. CONCLUSION

TLS 1.3 is a modern security protocol built with modern tools like formal analysis[15] that retains its backwards compatibility. It has been tested widely and iterated upon by using real world deployment data. It's a simpler, faster, and more secure protocol ready to become the two-party encryption protocol in practice online. Standardising TLS 1.3 is a huge accomplishment. It is one the best illustration of how it is possible to take 20 years of deployed legacy code and change it on the fly, resulting in a better internet for everyone. TLS 1.3 has been debated and analyzed for the last few years and it's now ready for the limeight.

 

**REFERENCES**

[1] E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.3", Internet Engineering Task Force: RFC 8446, August 2018
[2] Hugo Krawczyk, "SIGMA: the `SIGn-and-MAc' Approach to Authenticated Diffie-Hellman and its Use in the IKE Protocols", http://www.ee.technion.ac.il/~hugo/sigma.html, 2003
[3] Rivest, R., Shamir, A., and L. Adleman, "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems", Communications of the ACM, Vol. 21 No. 2, pp. 120-126, , February 1978.
[4] American National Standards Institute, "Public Key Cryptography for the Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA)", November 2005.
[5] S. Josefsson, I. Liusvaara, "Edwards-Curve Digital Signature Algorithm (EdDSA)", Internet Research Task Force: RFC 8032, January 2017
[6] P. Wouters, H. Tschofenig, J. Gilmore, S. Weiler, T. Kivinen,"Using Raw Public Keys in Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)", Internet Research Task Force: RFC 7250, June 2014
[7] S Santesson, H. Tschofenig"Transport Layer Security (TLS) Cached Information Extension", Internet Research Task Force: RFC 7924, July 2016
[8] J. Salowey, H. Zhou, P. Eronen, H. Tschofenig,"Transport Layer Security (TLS) Session Resumption without Server-Side State", Network Working Group:RFC 5077, January 2008
[9] National Institute of Standards and Technology, U.S.Department of Commerce, "Digital Signature Standard (DSS)", NIST FIPS PUB 186-4, July 2013
[10] A. Langley, M. Hamburg, S. Turner, "Elliptic Curves for Security", Internet Research Task Force: RFC 7748, January 2016
[11] M. Cotton, B. Leiba, T. Narten, "Guidelines for Writing an IANA Considerations Section in RFCs", Internet Research Task Force: RFC 8126, June 2017
[12] D. Gillmor, "Negotiated Finite Field Diffie-Hellman Ephemeral Parameters for Transport Layer Security (TLS)", Internet Research Task Force: RFC 7919, August 2016
[13] D. McGrew, "An Interface and Algorithms for Authenticated Encryption", Network Working Group: RFC 5116, January 2008
[14] M. Thomson, "Record Size Limit Extension for TLS", Internet Research Task Force: RFC 8449, August 2018
[15] Cas Cremers, Marko Horvat, Jonathan Hoyland, Sam Scott, Thyla van der Merwe, "A Comprehensive Symbolic Analysis of TLS 1.3", CCS '17, 2017

 