

# Transport Layer Security

From Wikipedia, the free encyclopedia

**Transport Layer Security (TLS)** and its predecessor, **Secure Sockets Layer (SSL)**, are cryptographic protocols that provide security and data integrity for communications over networks such as the Internet. TLS and SSL encrypt the segments of network connections at the Transport Layer end-to-end.

Several versions of the protocols are in wide-spread use in applications like web browsing, electronic mail, Internet faxing, instant messaging and voice-over-IP (VoIP).

TLS is an IETF standards track protocol, last updated in RFC 5246, that was based on the earlier SSL specifications developed by Netscape Corporation.<sup>[1]</sup>

## Contents

- 1 Description
- 2 History and development
- 3 Applications
- 4 Security
- 5 How it works
  - 5.1 TLS handshake in detail
    - 5.1.1 Simple TLS handshake
    - 5.1.2 Client-authenticated TLS handshake
    - 5.1.3 Resumed TLS handshake
  - 5.2 TLS record protocol
  - 5.3 Handshake protocol
  - 5.4 Alert protocol
  - 5.5 ChangeCipherSpec protocol
  - 5.6 Application protocol
- 6 Support for name-based virtual servers
- 7 Government-imposed protocol limitations
- 8 Implementations
  - 8.1 Browser implementations
- 9 Standards
- 10 See also
  - 10.1 Software
- 11 References and footnotes
- 12 Further reading
- 13 External links

## The Internet Protocol Suite

### Application Layer

BGP · DHCP · DNS · FTP · GTP · HTTP ·  
IMAP · IRC · Megaco · MGCP · NNTP ·  
NTP · POP · RIP · RPC · RTP · RTSP · SDP ·  
SIP · SMTP · SNMP · SOAP · SSH · Telnet ·  
**TLS/SSL** · XMPP · (more)

### Transport Layer

TCP · UDP · DCCP · SCTP · RSVP · ECN ·  
(more)

### Internet Layer

IP (IPv4, IPv6) · ICMP · ICMPv6 · IGMP ·  
IPsec · (more)

### Link Layer

ARP · RARP · NDP · OSPF ·  
Tunnels (L2TP) · PPP · Media Access  
Control (Ethernet, MPLS, DSL, ISDN,  
FDDI) · Device Drivers · (more)

## Description

The TLS protocol allows client/server applications to communicate across a network in a way designed to prevent eavesdropping, tampering, and message forgery. TLS provides endpoint authentication and communications confidentiality over the Internet using cryptography.

In typical end-user/browser usage, TLS authentication is *unilateral*: only the server is *authenticated* (the client knows the server's identity), but not *vice versa* (the client remains unauthenticated or anonymous). More strictly speaking, *server authentication* means different things to the browser (software) and to the end-user (human). At the browser level, it only means that the browser has *validated* the server's certificate, i.e. checked the digital signatures of the server certificate's issuing CA-chain (chain of Certification Authorities that guarantee bindings of identification information to public keys; see public key infrastructure). Once validated, the browser is justified in displaying a security icon (such as "closed padlock"). But mere validation does NOT "identify" the server to the end-user. For true *identification*, it is incumbent on the end-user to be diligent in scrutinizing the identification information contained in the server's certificate (and indeed its whole issuing CA-chain). This is the only way for the end-user to know the "identity" of the server. In particular: the "locked padlock" icon has no relationship to the URL, DNS name or IP address of the server. This is a common misconception. Such a binding can only be securely established if the URL, name or address is specified in the server's certificate itself. Malicious websites can't use the valid certificate of another website because they have no means to encrypt the transmission such that it can be decrypted with the valid certificate. Since only a trusted CA can embed a URL in the certificate, this ensures that checking the apparent URL with the URL specified in the certificate is a valid way of identifying the true site. Understanding this subtlety makes it very difficult for end-users to properly assess the security of web browsing (though this is not a shortcoming of the TLS protocol itself — it's a shortcoming of PKI).

TLS also supports the more secure *bilateral* connection mode (typically used in enterprise applications), in which both ends of the "conversation" can be ensured with whom they are communicating (provided they diligently scrutinize the identity information in the other party's certificate). This is known as mutual authentication. Mutual authentication requires that the TLS client-side also hold a certificate (which is not usually the case in the end-user/browser scenario). Unless, that is, TLS-PSK or the Secure Remote Password (SRP) protocol or some other protocol is used that can provide strong mutual authentication in the absence of certificates.

TLS involves three basic phases:

1. Peer negotiation for **algorithm support**
2. Key exchange and authentication
3. Symmetric cipher encryption and message authentication

During the first phase, the client and server negotiate *cipher suites*, which determine the ciphers to be used, the key exchange and authentication algorithms, as well as the message authentication codes (MACs). The key exchange and authentication algorithms are typically public key algorithms, or as in TLS-PSK preshared keys could be used. The message authentication codes are made up from cryptographic hash functions using the HMAC construction for TLS, and a non-standard pseudorandom function for SSL.

Typical algorithms are:

- For key exchange: RSA, Diffie-Hellman, ECDH, SRP, PSK

- For authentication: RSA, DSA, ECDSA
- Symmetric ciphers: RC4, Triple DES, AES, IDEA, DES, or Camellia. In older versions of SSL, RC2 was also used.
- For cryptographic hash function: HMAC-MD5 or HMAC-SHA are used for TLS, MD5 and SHA for SSL, while older versions of SSL also used MD2 and MD4.

## History and development

Early research efforts toward transport layer security included the Secure Network Programming (SNP) API, which in 1993 explored the approach of having a secure transport layer API closely resembling sockets, to facilitate retrofitting preexisting network applications with security measures.<sup>[2]</sup> The SNP project received the 2004 ACM Software System Award.<sup>[3]</sup>

The SSL protocol was originally developed by Netscape. Version 1.0 was never publicly released; version 2.0 was released in February 1995 but "contained a number of security flaws which ultimately led to the design of SSL version 3.0", which was released in 1996 (Rescorla 2001). This later served as the basis for TLS version 1.0, an Internet Engineering Task Force (IETF) standard protocol first defined in RFC 2246 in January 1999. Visa<sup>[4]</sup>, MasterCard<sup>[5][6]</sup>, American Express<sup>[7]</sup> and many leading financial institutions have endorsed SSL for commerce over the Internet.

SSL operates in modular fashion. It is extensible by design, with support for forward and backward compatibility and negotiation between peers.

## Applications

TLS runs on layers beneath application protocols such as HTTP, FTP, SMTP, NNTP, and XMPP and above a reliable transport protocol, TCP for example. While it can add security to any protocol that uses reliable connections (such as TCP), it is most commonly used with HTTP to form HTTPS. HTTPS is used to secure World Wide Web pages for applications such as electronic commerce and asset management. SMTP is also an area in which TLS has been growing and is specified in RFC 3207. These applications use public key certificates to verify the identity of endpoints.

An increasing number of client and server products support TLS natively, but many still lack support. As an alternative, users may wish to use standalone TLS products like Stunnel. Wrappers such as Stunnel rely on being able to obtain a TLS connection immediately, by simply connecting to a separate port reserved for the purpose. For example, by default the TCP port for HTTPS is 443, to distinguish it from HTTP on port 80.

TLS can also be used to tunnel an entire network stack to create a VPN, as is the case with OpenVPN. Many vendors now marry TLS's encryption and authentication capabilities with authorization. There has also been substantial development since the late 1990s in creating client technology outside of the browser to enable support for client/server applications. When compared against traditional IPsec VPN technologies, TLS has some inherent advantages in firewall and NAT traversal that make it easier to administer for large remote-access populations.

TLS is also increasingly being used as the standard method for protecting SIP application signaling. TLS can be used to provide authentication and encryption of the SIP signalling associated with VoIP and other

SIP-based applications.

## Security

TLS/SSL have a variety of security measures:

- The client may use the certificate authority's (CA's) *public* key to validate the CA's digital signature on the server certificate. If the digital signature can be verified, the client accepts the server certificate as a valid certificate issued by a trusted CA.
- The client verifies that the issuing CA is on its list of trusted CAs.
- The client checks the server's certificate validity period. The authentication process stops if the current date and time fall outside of the validity period.
- Protection against a downgrade of the protocol to a previous (less secure) version or a weaker cipher suite.
- Numbering all the Application records with a sequence number, and using this sequence number in the message authentication codes (MACs).
- Using a message digest enhanced with a key (so only a key-holder can check the MAC). This is specified in RFC 2104. *TLS only*.
- The message that ends the handshake ("Finished") sends a hash of all the exchanged handshake messages seen by both parties.
- The pseudorandom function splits the input data in half and processes each one with a different hashing algorithm (MD5 and SHA-1), then XORs them together to create the MAC. This provides protection even if one of these algorithms is found to be vulnerable. *TLS only*.
- SSL v3 improved upon SSL v2 by adding SHA-1 based ciphers, and support for certificate authentication. Additional improvements in SSL v3 include better handshake protocol flow and increased resistance to man-in-the-middle attacks.

SSL v2 is flawed in a variety (<http://www.eucybervote.org/Reports/MSI-WP2-D7V1-V1.0-02.htm>) of ways:

- Identical cryptographic keys are used for message authentication and encryption.
- MACs are unnecessarily weakened in the "export mode" required by U.S. export restrictions (symmetric key length was limited to 40 bits in Netscape and Internet Explorer).
- SSL v2 has a weak MAC construction and relies solely on the MD5 hash function.
- SSL v2 does not have any protection for the handshake, meaning a man-in-the-middle downgrade attack can go undetected.
- SSL v2 uses the TCP connection close to indicate the end of data. This means that truncation attacks are possible: the attacker simply forges a TCP FIN, leaving the recipient unaware of an illegitimate end of data message (SSL v3 fixes this problem by having an explicit closure alert).
- SSL v2 assumes a single service, and a fixed domain certificate, which clashes with the standard feature of virtual hosting in web servers. This means that most websites are practically impaired from using SSL. TLS/SNI fixes this but is not deployed in web servers as yet.

SSL v2 is disabled by default in Internet Explorer 7,<sup>[8]</sup> Mozilla Firefox 2 and Mozilla Firefox 3,<sup>[9]</sup> and Safari. After it sends a TLS **ClientHello**, if Mozilla Firefox finds that the server is unable to complete the handshake, it will attempt to *fall back* to using SSL 3.0 with an SSL 3.0 **ClientHello** in SSL v2 format to maximize the likelihood of successfully handshaking with older servers.<sup>[10]</sup> Support for SSL v2 (and weak 40-bit and 56-bit ciphers) has been removed completely from Opera as of version 9.5.<sup>[11]</sup>

## How it works

<sup>[12]</sup> A TLS client and server negotiate a stateful connection by using a handshaking procedure. During this handshake, the client and server agree on various parameters used to establish the connection's security.

- The handshake begins when a client connects to a TLS-enabled server requesting a secure connection, and presents a list of supported ciphers and hash functions.
- From this list, the server picks the strongest cipher and hash function that it also supports and notifies the client of the decision.
- The server sends back its identification in the form of a digital certificate. The certificate usually contains the server name, the trusted certificate authority (CA), and the server's public encryption key.

The client may contact the server that issued the certificate (the trusted CA as above) and confirm that the certificate is authentic before proceeding.

- In order to generate the session keys used for the secure connection, the client encrypts a random number (RN) with the server's public key (PbK), and sends the result to the server. Only the server can decrypt it (with its private key (PvK)): this is the one fact that makes the keys hidden from third parties, since only the server and the client have access to this data. The client knows PbK and RN, and the server knows PvK and (after decryption of the client's message) RN. A third party may only know PbK, unless PvK has been compromised.
- From the random number, both parties generate key material for encryption and decryption.

This concludes the handshake and begins the secured connection, which is encrypted and decrypted with the key material until the connection closes.

*If any one of the above steps fails, the TLS handshake fails, and the connection is not created.*

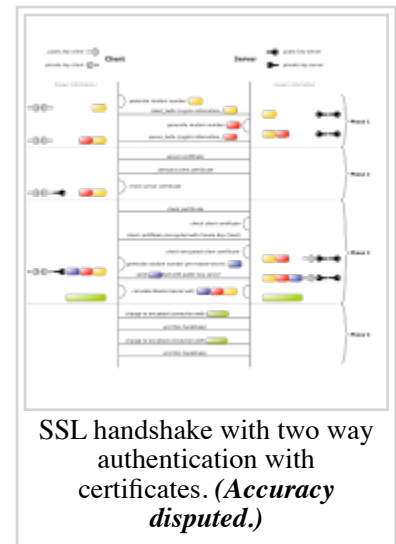
### TLS handshake in detail

The TLS protocol exchanges *records*, which encapsulate the data to be exchanged. Each record can be compressed, padded, appended with a message authentication code (MAC), or encrypted, all depending on the state of the connection. Each record has a *content type* field that specifies the record, a length field, and a TLS version field.

When the connection starts, the record encapsulates another protocol - the handshake messaging protocol - which has *content type 22*.

### Simple TLS handshake

A simple connection example, illustrating a handshake where the server is authenticated by its certificate (but not the client), follows:



### 1. Negotiation phase:

- A client sends a **ClientHello** message specifying the highest TLS protocol version it supports, a random number, a list of suggested cipher suites and compression methods.
  - The server responds with a **ServerHello** message, containing the chosen protocol version, a random number, cipher suite, and compression method from the choices offered by the client. The server may also send a *session id* as part of the message to perform a resumed handshake.
  - The server sends its **Certificate** message (depending on the selected cipher suite, this may be omitted by the server).<sup>[13]</sup>
  - The server sends a **ServerHelloDone** message, indicating it is done with handshake negotiation.
  - The client responds with a **ClientKeyExchange** message, which may contain a *PreMasterSecret*, public key, or nothing. (Again, this depends on the selected cipher.)
  - The client and server then use the random numbers and *PreMasterSecret* to compute a common secret, called the "master secret". All other key data for this connection is derived from this master secret (and the client- and server-generated random values), which is passed through a carefully designed "pseudorandom function".
2. The client now sends a **ChangeCipherSpec** record, essentially telling the server, "Everything I tell you from now on will be authenticated (and encrypted if encryption parameters were present in the server certificate)." The **ChangeCipherSpec** is itself a record-level protocol, and has type 20, and not 22.
- Finally, the client sends an authenticated and encrypted **Finished** message, containing a hash and MAC over the previous handshake messages.
  - The server will attempt to decrypt the client's *Finished* message, and verify the hash and MAC. If the decryption or verification fails, the handshake is considered to have failed and the connection should be torn down.
3. Finally, the server sends a **ChangeCipherSpec**, telling the server, "Everything I tell you from now on will be authenticated (and encrypted with the server private key associated to the public key in the server certificate, if encryption was negotiated)."
- The server sends its authenticated and encrypted **Finished** message.
  - The client performs the same decryption and verification.
4. Application phase: at this point, the "handshake" is complete and the application protocol is enabled, with content type of 23. Application messages exchanged between client and server will also be authenticated and optionally encrypted exactly like in their *Finished* message.

## Client-authenticated TLS handshake

The following *full* example shows a client being authenticated (in addition to the server like above) via TLS using certificates exchanged between both peers.

### 1. Negotiation phase:

- A client sends a **ClientHello** message specifying the highest TLS protocol version it supports, a random number, a list of suggested cipher suites and compression methods.
- The server responds with a **ServerHello** message, containing the chosen protocol version, a random number, cipher suite, and compression method from the choices offered by the client. The server may also send a *session id* as part of the message to perform a resumed handshake.
- The server sends its **Certificate** message (depending on the selected cipher suite, this may be omitted by the server).<sup>[13]</sup>

- The server requests a certificate from the client, so that the connection can be mutually authenticated, using a **CertificateRequest** message.
  - The server sends a **ServerHelloDone** message, indicating it is done with handshake negotiation.
  - The client responds with a **Certificate** message, which contains the client's certificate.
  - The client sends a **ClientKeyExchange** message, which may contain a *PreMasterSecret*, public key, or nothing. (Again, this depends on the selected cipher.) This *PreMasterSecret* is encrypted using the public key of the server certificate.
  - The client sends a **CertificateVerify** message, which is a signature over the previous handshake messages using the client's certificate's private key. This signature can be verified by using the client's certificate's public key. This lets the server know that the client has access to the private key of the certificate and thus owns the certificate.
  - The client and server then use the random numbers and *PreMasterSecret* to compute a common secret, called the "master secret". All other key data for this connection is derived from this master secret (and the client- and server-generated random values), which is passed through a carefully designed "pseudorandom function".
2. The client now sends a **ChangeCipherSpec** record, essentially telling the server, "Everything I tell you from now on will be authenticated (and encrypted if encryption was negotiated)." The **ChangeCipherSpec** is itself a record-level protocol, and has type 20, and not 22.
    - Finally, the client sends an encrypted **Finished** message, containing a hash and MAC over the previous handshake messages.
    - The server will attempt to decrypt the client's *Finished* message, and verify the hash and MAC. If the decryption or verification fails, the handshake is considered to have failed and the connection should be torn down.
  3. Finally, the server sends a **ChangeCipherSpec**, telling the server, "Everything I tell you from now on will be authenticated (and encrypted if encryption was negotiated)."
    - The server sends its own encrypted **Finished** message.
    - The client performs the same decryption and verification.
  4. Application phase: at this point, the "handshake" is complete and the application protocol is enabled, with content type of 23. Application messages exchanged between client and server will also be encrypted exactly like in their *Finished* message.

## Resumed TLS handshake

Public key operations (e.g., RSA) are relatively expensive in terms of computational power. TLS provides a secure shortcut in the handshake mechanism to avoid these operations. In an ordinary *full* handshake, the server sends a *session id* as part of the **ServerHello** message. The client associates this *session id* with the server's IP address and TCP port, so that when the client connects again to that server, it can use the *session id* to shortcut the handshake. In the server, the *session id* maps to the cryptographic parameters previously negotiated, specifically the "master secret". Both sides must have the same "master secret" or the resumed handshake will fail (this prevents an eavesdropper from using a *session id*). The random data in the **ClientHello** and **ServerHello** messages virtually guarantee that the generated connection keys will be different than in the previous connection. In the RFCs, this type of handshake is called an *abbreviated* handshake. It is also described in the literature as a *restart* handshake.

1. Negotiation phase:
  - A client sends a **ClientHello** message specifying the highest TLS protocol version it supports, a random number, a list of suggested cipher suites and compression methods. Included in the

message is the *session id* from the previous TLS connection.

- The server responds with a **ServerHello** message, containing the chosen protocol version, a random number, cipher suite, and compression method from the choices offered by the client. If the server recognizes the *session id* sent by the client, it responds with the same *session id*. The client uses this to recognize that a resumed handshake is being performed. If the server does not recognize the *session id* sent by the client, it sends a different value for its *session id*. This tells the client that a resumed handshake will not be performed. At this point, both the client and server have the "master secret" and random data to generate the key data to be used for this connection.
- 2. The client now sends a **ChangeCipherSpec** record, essentially telling the server, "Everything I tell you from now on will be encrypted." The ChangeCipherSpec is itself a record-level protocol, and has type 20, and not 22.
  - Finally, the client sends an encrypted **Finished** message, containing a hash and MAC over the previous handshake messages.
  - The server will attempt to decrypt the client's *Finished* message, and verify the hash and MAC. If the decryption or verification fails, the handshake is considered to have failed and the connection should be torn down.
- 3. Finally, the server sends a **ChangeCipherSpec**, telling the server, "Everything I tell you from now on will be encrypted."
  - The server sends its own encrypted **Finished** message.
  - The client performs the same decryption and verification.
- 4. Application phase: at this point, the "handshake" is complete and the application protocol is enabled, with content type of 23. Application messages exchanged between client and server will also be encrypted exactly like in their *Finished* message.

Apart from the performance benefit, resumed sessions can also be used for single sign-on as it is guaranteed that both the original session as well as any resumed session originate from the same client. This is of particular importance for the FTP over TLS/SSL protocol which would otherwise suffer from a man in the middle attack in which an attacker could intercept the contents of the secondary data connections.<sup>[14]</sup>

## TLS record protocol

This is the general format of all TLS records.

+	Byte +0	Byte +1	Byte +2	Byte +3
<b>Byte 0</b>	Content type			
<b>Bytes 1..4</b>	Version		Length	
	<i>(Major)</i>	<i>(Minor)</i>	<i>(bits 15..8)</i>	<i>(bits 7..0)</i>
<b>Bytes 5..(m-1)</b>	Protocol message(s)			
<b>Bytes m..(p-1)</b>	MAC (optional)			

<b>Bytes</b> <i>p..(q-1)</i>	Padding (block ciphers only)
---------------------------------	------------------------------

### Content type

This field identifies the Record Layer Protocol Type contained in this Record.

#### Content types

Hex	Dec	Type
0x14	20	ChangeCipherSpec
0x15	21	Alert
0x16	22	Handshake
0x17	23	Application

### Version

This field identifies the major and minor version of TLS for the contained message. For a ClientHello message, this need not be the *highest* version supported by the client.

#### Versions

Major Version	Minor Version	Version Type
3	0	SSLv3
3	1	TLS 1.0
3	2	TLS 1.1
3	3	TLS 1.2

### Length

The length of Protocol message(s), not to exceed  $2^{14}$  bytes (16 KiB).

#### Protocol message(s)

One or more messages identified by the Protocol field. Note that this field may be encrypted depending on the state of the connection.

#### MAC and Padding

A message authentication code computed over the Protocol message, with additional key material included. Note that this field may be encrypted, or not included entirely, depending on the state of the connection.

No MAC or Padding can be present at end of TLS records before all cipher algorithms and parameters have been negotiated and handshaked, and then confirmed by sending a CipherStateChange record (see below) for signaling that these parameters will take effect in all further records sent by the same peer.

## Handshake protocol

Most messages exchanged during the setup of the TLS session are based on this record, unless an error or warning occurs and needs to be signaled by an Alert protocol record (see below), or the encryption mode of the session is modified by another record (see ChangeCipherSpec protocol below).

<b>+</b>	<b>Byte +0</b>	<b>Byte +1</b>	<b>Byte +2</b>	<b>Byte +3</b>
<b>Byte 0</b>	22			
<b>Bytes 1..4</b>	Version		Length	
	<i>(Major)</i>	<i>(Minor)</i>	<i>(bits 15..8)</i>	<i>(bits 7..0)</i>
<b>Bytes 5..8</b>	Message type	Handshake message data length		
		<i>(bits 23..16)</i>	<i>(bits 15..8)</i>	<i>(bits 7..0)</i>
<b>Bytes 9..(n-1)</b>	Handshake message data			
<b>Bytes n..(n+3)</b>	Message type	Handshake message data length		
		<i>(bits 23..16)</i>	<i>(bits 15..8)</i>	<i>(bits 7..0)</i>
<b>Bytes (n+4)..</b>	Handshake message data			

### Message type

This field identifies the Handshake message type.

<b>Message Types</b>	
<b>Code</b>	<b>Description</b>
0	HelloRequest
1	ClientHello
2	ServerHello
11	Certificate
12	ServerKeyExchange
13	CertificateRequest
14	ServerHelloDone
15	CertificateVerify
16	ClientKeyExchange
20	Finished

### Handshake message data length

This is a 3-byte field indicating the length of the handshake data, not including the header.

Note that multiple Handshake messages may be combined within one record.

## Alert protocol

This record should normally not be sent during normal handshaking or application exchanges. However, this message can be sent at any time during the handshake and up to the closure of the session. If this is used to signal a fatal error, the session will be closed immediately after sending this record, so this record is used to give a reason for this closure. If the alert level is flagged as a warning, the remote can decide to close the session if it decides that the session is not reliable enough for its needs (before doing so, the remote may also send its own signal).

+	Byte +0	Byte +1	Byte +2	Byte +3
<b>Byte 0</b>	21			
<b>Bytes 1..4</b>	Version		Length	
	<i>(Major)</i>	<i>(Minor)</i>	0	2
<b>Bytes 5..6</b>	Level	Description		
<b>Bytes 7..(p-1)</b>	MAC (optional)			
<b>Bytes p..(q-1)</b>	Padding (block ciphers only)			

### Level

This field identifies the level of alert. If the level is fatal, the sender should close the session immediately. Otherwise, the recipient may decide to terminate the session itself, by sending its own fatal alert and closing the session itself immediately after sending it. The use of Alert records is optional, however if it is missing before the session closure, the session may be resumed automatically (with its handshakes).

Normal closure of a session after termination of the transported application should preferably be alerted with at least the *Close notify* Alert type (with a simple warning level) to prevent such automatic resume of a new session. Signaling explicitly the normal closure of a secure session before effectively closing its transport layer is useful to prevent or detect attacks (like attempts to truncate the securely transported data, if it intrinsically does not have a predetermined length or duration that the recipient of the secured data may expect).

### Alert level types

Code	Level type	Connection state
1	<b>warning</b>	connection or security may be unstable.

2	<b>fatal</b>	connection or security may be compromised, or an unrecoverable error has occurred.
---	--------------	--

## Description

This field identifies which type of alert is being sent.

### Alert description types

Code	Description	Level types	Note
0	Close notify	<b>warning</b> or <b>fatal</b>	
10	Unexpected message	<b>fatal</b>	
20	Bad record MAC	<b>fatal</b>	
21	Decryption failed	<b>fatal</b>	TLS only, reserved
22	Record overflow	<b>fatal</b>	TLS only
30	Decompression failure	<b>fatal</b>	
40	Handshake failure	<b>fatal</b>	
41	No certificate	<b>warning</b> or <b>fatal</b>	SSL v3 only, reserved
42	Bad certificate	<b>warning</b> or <b>fatal</b>	
43	Unsupported certificate	<b>warning</b> or <b>fatal</b>	
44	Certificate revoked	<b>warning</b> or <b>fatal</b>	
45	Certificate expired	<b>warning</b> or <b>fatal</b>	
46	Certificate unknown	<b>warning</b> or <b>fatal</b>	
47	Illegal parameter	<b>fatal</b>	
48	Unknown CA (Certificate authority)	<b>fatal</b>	TLS only
49	Access denied	<b>fatal</b>	TLS only
50	Decode error	<b>fatal</b>	TLS only
51	Decrypt error	<b>warning</b> or <b>fatal</b>	TLS only
60	Export restriction	<b>fatal</b>	TLS only, reserved
70	Protocol version	<b>fatal</b>	TLS only
71	Insufficient security	<b>fatal</b>	TLS only
80	Internal error	<b>fatal</b>	TLS only
90	User cancelled	<b>fatal</b>	TLS only
100	No renegotiation	<b>warning</b>	TLS only
110	Unsupported extension	<b>warning</b>	TLS only

## ChangeCipherSpec protocol

+	Byte +0	Byte +1	Byte +2	Byte +3
Byte 0	20			
Bytes 1..4	Version		Length	
	(Major)	(Minor)	0	1
Byte 5	CCS protocol type			

CCS protocol type  
 Currently only 1.

## Application protocol

+	Byte +0	Byte +1	Byte +2	Byte +3
Byte 0	23			
Bytes 1..4	Version		Length	
	(Major)	(Minor)	(bits 15..8)	(bits 7..0)
Bytes 5..(m-1)	Application data			
Bytes m..(p-1)	MAC (optional)			
Bytes p..(q-1)	Padding (block ciphers only)			

Length  
 Length of Application data (excluding the protocol header, and the MAC and padding trailers)

MAC  
 20 bytes for the SHA-1-based HMAC, 16 bytes for the MD5-based HMAC.

Padding  
 Variable length ; last byte contains the padding length.

## Support for name-based virtual servers

From the application protocol point of view, TLS belongs to a lower layer, although the TCP/IP model is too coarse to show it. This means that the TLS handshake is usually (except in the STARTTLS case) performed before the application protocol can start. The name-based virtual server feature being provided by the application layer, all co-hosted virtual servers share the same certificate because the server has to

select and send a certificate immediately after the ClientHello message. This is a big problem in hosting environments because it means either sharing the same certificate among all customers or using a different IP address for each of them.

There are two known workarounds provided by X.509:

- If all virtual servers belongs to the same domain, you can use a wildcard certificate (<http://wiki.cacert.org/wiki/WildcardCertificates>) . Besides the loose host name selection that might be a problem or not, there is no common agreement about how to match wildcard certificates. Different rules are applied depending on the application protocol or software used.<sup>[15]</sup>
- Add every virtual host name in the subjectAltName extension. The major problem being that you need to reissue a certificate whenever you declare a new virtual server.

In order to provide the server name, RFC 4366 Transport Layer Security (TLS) Extensions allow clients to include a *Server Name Indication* extension (SNI) in the extended ClientHello message. This extension hints the server immediately which name the client wishes to connect to, so the server can select the appropriate certificate to send to the client.

## Government-imposed protocol limitations

Some early implementations of SSL used 40-bit symmetric keys because of US government restrictions on the export of cryptographic technology. After several years of public controversy, a series of lawsuits, and eventual US government recognition of cryptographic products with longer key sizes produced outside the US, the authorities relaxed some aspects of the export restrictions.

## Implementations

SSL and TLS have been widely implemented in several open source software projects. Programmers may use the OpenSSL, NSS, or GnuTLS libraries for SSL/TLS functionality. Microsoft Windows includes an implementation of SSL and TLS as part of its Secure Channel package. Delphi programmers may use a library called Indy.

### Browser implementations

Mozilla Firefox supports TLS 1.0 since version 2.<sup>[16]</sup>

The Internet Explorer 8 in Windows 7 and Windows Server 2008 R2 supports TLS 1.2.<sup>[17]</sup>

As of Presto 2.2, featured in Opera 10, Opera supports TLS 1.2.<sup>[18]</sup>

## Standards

The current approved version is 1.2, which is specified in:

- RFC 5246: “The Transport Layer Security (TLS) Protocol Version 1.2”.

The current standard obsoletes these former versions:

- RFC 2246: “The TLS Protocol Version 1.0”.
- RFC 4346: “The Transport Layer Security (TLS) Protocol Version 1.1”.

Other RFCs subsequently extended TLS, including:

- RFC 2595: “Using TLS with IMAP, POP3 and ACAP”. Specifies an extension to the IMAP, POP3 and ACAP services that allow the server and client to use transport-layer security to provide private, authenticated communication over the Internet.
- RFC 2712: “Addition of Kerberos Cipher Suites to Transport Layer Security (TLS)”. The 40-bit ciphersuites defined in this memo appear only for the purpose of documenting the fact that those ciphersuite codes have already been assigned.
- RFC 2817: “Upgrading to TLS Within HTTP/1.1”, explains how to use the Upgrade mechanism in HTTP/1.1 to initiate Transport Layer Security (TLS) over an existing TCP connection. This allows unsecured and secured HTTP traffic to share the same *well known* port (in this case, http: at 80 rather than https: at 443).
- RFC 2818: “HTTP Over TLS”, distinguishes secured traffic from insecure traffic by the use of a different 'server port'.
- RFC 3207: “SMTP Service Extension for Secure SMTP over Transport Layer Security”. Specifies an extension to the SMTP service that allows an SMTP server and client to use transport-layer security to provide private, authenticated communication over the Internet.
- RFC 3268: “AES Ciphersuites for TLS”. Adds Advanced Encryption Standard (AES) ciphersuites to the previously existing symmetric ciphers.
- RFC 3546: “Transport Layer Security (TLS) Extensions”, adds a mechanism for negotiating protocol extensions during session initialisation and defines some extensions. Made obsolete by RFC 4366.
- RFC 3749: “Transport Layer Security Protocol Compression Methods”, specifies the framework for compression methods and the DEFLATE compression method.
- RFC 3943: “Transport Layer Security (TLS) Protocol Compression Using Lempel-Ziv-Stac (LZS)”.
- RFC 4132: “Addition of Camellia Cipher Suites to Transport Layer Security (TLS)”.
- RFC 4162: “Addition of SEED Cipher Suites to Transport Layer Security (TLS)”.
- RFC 4217: “Securing FTP with TLS”.
- RFC 4279: “Pre-Shared Key Cipher Suites for Transport Layer Security (TLS)”, adds three sets of new ciphersuites for the TLS protocol to support authentication based on pre-shared keys.
- RFC 4347: “Datagram Transport Layer Security” specifies a TLS variant that works over datagram protocols (such as UDP).
- RFC 4366: “Transport Layer Security (TLS) Extensions” describes both a set of specific extensions, and a generic extension mechanism.
- RFC 4492: “Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS)”.
- RFC 4507: “Transport Layer Security (TLS) Session Resumption without Server-Side State”.
- RFC 4680: “TLS Handshake Message for Supplemental Data”.
- RFC 4681: “TLS User Mapping Extension”.
- RFC 4785: “Pre-Shared Key (PSK) Cipher Suites with NULL Encryption for Transport Layer Security (TLS)”.

## See also

- Certificate authority
- Public key certificate
- Extended Validation Certificate

- SSL acceleration
- Datagram Transport Layer Security
- Multiplexed Transport Layer Security
- X.509
- Virtual private network
- SEED
- Server gated cryptography

## Software

- OpenSSL: a free implementation (BSD license with some extensions)
- GnuTLS: a free implementation (LGPL licensed)
- JSSE: a Java implementation included in the Java Runtime Environment
- Network Security Services (NSS): FIPS 140 validated open source library

## References and footnotes

- <sup>1</sup> <sup>^</sup> The SSL Protocol: Version 3.0 (<http://www.mozilla.org/projects/security/pki/nss/ssl/draft302.txt>) Netscape's final SSL 3.0 draft (November 18, 1996)
- <sup>2</sup> <sup>^</sup> Woo, Thomas Y. C. and Bindignavle, Raghuram and Su, Shaowen and Lam, Simon S. 1994. *SNP: An interface for secure network programming* In Usenix Summer Technical Conference
- <sup>3</sup> <sup>^</sup> Association for Computing Machinery, "ACM: Press Release, March 15, 2005" ([http://campus.acm.org/public/pressroom/press\\_releases/3\\_2005/ss\\_award\\_3\\_15\\_2005.cfm](http://campus.acm.org/public/pressroom/press_releases/3_2005/ss_award_3_15_2005.cfm)) , *campus.acm.org*, accessed December 26, 2007. (English version).
- <sup>4</sup> <sup>^</sup> Visa USA, Inc., *Operating Regulations: Volume 1--General Rules*. 15 November 2008. p. 166 (section 4.2.C.6.c). <http://corporate.visa.com/pd/rules/pdf/visa-usa-operating-regulations1.pdf>. Retrieved on 16 January 2009.
- <sup>5</sup> <sup>^</sup> MasterCard Worldwide (November 2007). *POS Terminal Security Program -- Program Manual*. p. 2-2 (mandates use of PCI standard in next reference). [http://www.mastercard.com/us/wce/PDF/PTS\\_Program\\_Manual\\_NOV\\_07.pdf](http://www.mastercard.com/us/wce/PDF/PTS_Program_Manual_NOV_07.pdf). Retrieved on 19 January 2009.
- <sup>6</sup> <sup>^</sup> PCI Security Standards Council (October 2008). *Payment Card Industry Data Security Standard Requirements and Security Assessment Procedures, version 1.2*. p. 19 (requirement 2.3), p. 26 (requirement 4.1). [https://www.pcisecuritystandards.org/security\\_standards/download.html?id=pci\\_dss\\_v1-2.pdf](https://www.pcisecuritystandards.org/security_standards/download.html?id=pci_dss_v1-2.pdf). Retrieved on 19 January 2009.
- <sup>7</sup> <sup>^</sup> "Accept the Card, FAQs: How safe is it to accept the Card online?". American Express. [https://www209.americanexpress.com/merchant/singlevoice/USEng/FrontServlet?request\\_type=navigate&page=acceptCardFAQ#q2](https://www209.americanexpress.com/merchant/singlevoice/USEng/FrontServlet?request_type=navigate&page=acceptCardFAQ#q2). Retrieved on 19 January 2009.
- <sup>8</sup> <sup>^</sup> Lawrence, Eric (2005-10-22). "IEBlog : Upcoming HTTPS Improvements in Internet Explorer 7 Beta 2". MSDN Blogs. <http://blogs.msdn.com/ie/archive/2005/10/22/483795.aspx>. Retrieved on 2007-11-25.
- <sup>9</sup> <sup>^</sup> "Bugzilla@Mozilla - Bug 236933 - Disable SSL2 and other weak ciphers". Mozilla Corporation. [https://bugzilla.mozilla.org/show\\_bug.cgi?id=236933](https://bugzilla.mozilla.org/show_bug.cgi?id=236933). Retrieved on 2007-11-25.
- <sup>10</sup> <sup>^</sup> "Firefox still sends SSLv2 handshake even though the protocol is disabled". 2008-09-11. [https://bugzilla.mozilla.org/show\\_bug.cgi?id=454759](https://bugzilla.mozilla.org/show_bug.cgi?id=454759).
- <sup>11</sup> <sup>^</sup> Pettersen, Yngve (2007-04-30). "10 years of SSL in Opera - Implementer's notes". Opera Software. <http://my.opera.com/yngve/blog/2007/04/30/10-years-of-ssl-in-opera>. Retrieved on 2007-11-25.
- <sup>12</sup> <sup>^</sup> "SSL/TLS in Detail (<http://technet.microsoft.com/en-us/library/cc785811.aspx>) ". *Microsoft TechNet*. Updated July 31, 2003.
- <sup>13</sup> <sup>^</sup> <sup>*a*</sup> <sup>*b*</sup> These certificates are currently X.509, but there is also a draft specifying the use of OpenPGP based certificates.
- <sup>14</sup> <sup>^</sup> vsftpd-2.1.0 released (<http://scarybeastsecurity.blogspot.com/2009/02/vsftpd-210-released.html>) Using TLS session resume for FTPS data connection authentication. Retrieved on 2009-04-28.
- <sup>15</sup> <sup>^</sup> Named-based SSL virtual hosts: how to tackle the problem ([https://www.switch.ch/pki/meetings/2007-01/namebased\\_ssl\\_virtualhosts.pdf](https://www.switch.ch/pki/meetings/2007-01/namebased_ssl_virtualhosts.pdf)) , SWITCH.

16. ^ Mozilla (2008-08-06/). "Security in Firefox 2". [https://developer.mozilla.org/en/Security\\_in\\_Firefox\\_2](https://developer.mozilla.org/en/Security_in_Firefox_2). Retrieved on 2009-03-31/.
17. ^ Microsoft (2009-02-27/). "MS-TLSP Appendix A". <http://msdn.microsoft.com/en-us/library/dd208005%28PROT.13%29.aspx>. Retrieved on 2009-03-19/.
18. ^ Yngve Nysæter Pettersen (2009-02-25/). "New in Opera Presto 2.2: TLS 1.2 Support". <http://my.opera.com/core/blog/2009/02/25/new-in-opera-presto-2-2-tls-1-2-support>. Retrieved on 2009-02-25/.

## Further reading

- Wagner, David; Schneier, Bruce (November 1996). "Analysis of the SSL 3.0 Protocol". *The Second USENIX Workshop on Electronic Commerce Proceedings*, USENIX Press.
- Eric Rescorla (2001). *SSL and TLS: Designing and Building Secure Systems*. United States: Addison-Wesley Pub Co. ISBN 0-201-61598-3.
- Stephen A. Thomas (2000). *SSL and TLS essentials securing the Web*. New York: Wiley. ISBN 0-471-38354-6.
- Bard, Gregory (2006). "A Challenging But Feasible Blockwise-Adaptive Chosen-Plaintext Attack On Ssl". *International Association for Cryptologic Research* (136). <http://citeseer.ist.psu.edu/bard04vulnerability.html>. Retrieved on 2007-04-20.
- Canel, Brice. "Password Interception in a SSL/TLS Channel". [http://lasecwww.epfl.ch/memo/memo\\_ssl.shtml](http://lasecwww.epfl.ch/memo/memo_ssl.shtml). Retrieved on 2007-04-20.

## External links

- SSL 2 specification (<http://www.mozilla.org/projects/security/pki/nss/ssl/draft02.html>) (published 1994)
- SSL 3.0 specification (<http://www.freesoft.org/CIE/Topics/ssl-draft/3-SPEC.HTM>) (published 1996)
- Netscape's final SSL 3.0 draft (1996) (<http://www.mozilla.org/projects/security/pki/nss/ssl/draft302.txt>)
- The IETF (Internet Engineering Task Force) TLS Workgroup (<http://www.ietf.org/html.charters/tls-charter.html>)
- SSL tutorial (<http://www2.rad.com/networks/2001/ssl/index.htm>)
- OpenSSL thread safe connections tutorial with example source code (<http://ardoino.com/40-openssl-thread-safe-secure-connections/>)
- ECMAScript Secure Transform (Web 2 Secure Transform Method) (<http://www.semnanweb.com/ecmast-ecmascript-secure-transform/>)

*This article was originally based on material from the Free On-line Dictionary of Computing, which is licensed under the GFDL.*

Retrieved from "http://en.wikipedia.org/wiki/Transport\_Layer\_Security"

Categories: Cryptographic protocols | Internet standards | Internet protocols | Electronic commerce | Secure communication | Application layer protocols

Hidden categories: Articles needing additional references from December 2007 | All articles with unsourced statements | Articles with unsourced statements since December 2007 | Articles with unsourced statements since April 2009 | Articles with unsourced statements since February 2009 | Wikipedia articles incorporating text from FOLDOC

- 
- This page was last modified on 29 April 2009, at 19:49 (UTC).

- All text is available under the terms of the GNU Free Documentation License. (See **Copyrights** for details.)  
Wikipedia® is a registered trademark of the Wikimedia Foundation, Inc., a U.S. registered 501(c)(3) tax-deductible nonprofit charity.