Transport Layer Security (**TLS**) and its predecessor, **Secure Sockets Layer** (**SSL**), are cryptographic protocols which are designed to provide communication security over the Internet.[1] They use X.509 certificates and hence asymmetric cryptography to assure the counterparty with whom they are communicating, and to exchange a symmetric key. This session key is then used to encrypt data flowing between the parties. This allows for data/message confidentiality, and message authentication codes for message integrity and as a by-product, message authentication. Several versions of the protocols are in widespread use in applications such as web browsing, electronic mail, Internet faxing, instant messaging, and voice-over-IP (VoIP). An important property in this context is forward secrecy, so the short term session key cannot be derived from the long term asymmetric secret key.[2]

As a consequence of choosing X.509 certificates, certificate authorities and a public key infrastructure are necessary to verify the relation between a certificate and its owner, as well as to generate, sign, and administer the validity of certificates. While this can be more beneficial than verifying the identities via a web of trust, the 2013 mass surveillance disclosures made it more widely known that certificate authorities are a weak point from a security standpoint, allowing man-in-the-middle attacks (MITM).[3][4]

In the TCP/IP model view, TLS and SSL encrypt the data of network connections at a lower sublayer of its application layer. In OSI model equivalences, TLS/SSL is initialized at layer 5 (the session layer) then works at layer 6 (the presentation layer): first the session layer has a handshake using an asymmetric cipher in order to establish cipher settings and a shared key for that session; then the presentation layer encrypts the rest of the communication using a symmetric cipher and that session key. In both models, TLS and SSL work on behalf of the underlying transport layer, whose segments carry encrypted data.

TLS is an IETF standards track protocol, first defined in 1999 and last updated in RFC 5246 (August 2008) and RFC 6176 (March 2011). It is based on the earlier SSL specifications (1994, 1995, 1996) developed by Netscape Communications[5] for adding the HTTPS protocol to their Navigator web browser.

Description

The TLS protocol allows client-server applications to communicate across a network in a way designed to prevent eavesdropping and tampering.

Since protocols can operate either with or without TLS (or SSL), it is necessary for the client to indicate to the server whether it wants to set up a TLS connection or not. There are two main ways of achieving this. One option is to use a different port number for TLS connections (for example port 443 for HTTPS). The other is to use the regular port number and have the client request that the server switch the connection to TLS using a protocol-specific mechanism (for example STARTTLS for mail and news protocols).

Once the client and server have decided to use TLS, they negotiate a stateful connection by using a handshaking procedure.[6] During this handshake, the client and server agree on various parameters used to establish the connection's security:

1. The client sends the server the client's SSL version number, cipher settings, session-

specific data, and other information that the server needs to communicate with the client using SSL.
2. The server sends the client the server's SSL version number, cipher settings, session-specific data, and other information that the client needs to communicate with the server over SSL. The server also sends its own certificate, and if the client is requesting a server resource that requires client authentication, the server requests the client's certificate.
3. The client uses the information sent by the server to authenticate the server—e.g., in the case of a web browser connecting to a web server, the browser checks whether the received certificate's subject name actually matches the name of the server being contacted, whether the issuer of the certificate is a trusted certificate authority, whether the certificate has expired, and, ideally, whether the certificate has been revoked.[7] If the server cannot be authenticated, the user is warned of the problem and informed that an encrypted and authenticated connection cannot be established. If the server can be successfully authenticated, the client proceeds to the next step.
4. Using all data generated in the handshake thus far, the client (with the cooperation of the server, depending on the cipher in use) creates the pre-master secret for the session, encrypts it with the server's public key (obtained from the server's certificate, sent in step 2), and then sends the encrypted pre-master secret to the server.
5. If the server has requested client authentication (an optional step in the handshake), the client also signs another piece of data that is unique to this handshake and known by both the client and server. In this case, the client sends both the signed data and the client's own certificate to the server along with the encrypted pre-master secret.
6. If the server has requested client authentication, the server attempts to authenticate the client. If the client cannot be authenticated, the session ends. If the client can be successfully authenticated, the server uses its private key to decrypt the pre-master secret, and then performs a series of steps (which the client also performs, starting from the same pre-master secret) to generate the master secret.
7. Both the client and the server use the master secret to generate the session keys, which are symmetric keys used to encrypt and decrypt information exchanged during the SSL session and to verify its integrity (that is, to detect any changes in the data between the time it was sent and the time it is received over the SSL connection).
8. The client sends a message to the server informing it that future messages from the client will be encrypted with the session key. It then sends a separate (encrypted) message indicating that the client portion of the handshake is finished.
9. The server sends a message to the client informing it that future messages from the server will be encrypted with the session key. It then sends a separate (encrypted) message indicating that the server portion of the handshake is finished.

The SSL handshake is now complete and the session begins. The client and the server use the session keys to encrypt and decrypt the data they send to each other and to validate its integrity.

This is the normal operation condition of the secure channel. At any time, due to internal or external stimulus (either automation or user intervention), either side may renegotiate the connection, in which case, the process repeats itself.[8]

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.

In step 3, the client must check a chain of "signatures" from a "root of trust" built into, or added to, the client. The client must *also* check that none of these have been revoked; this is not often implemented correctly, but is a requirement of any public-key authentication system. If the particular signer beginning this server's chain is trusted, and all signatures in the chain remain trusted, then the Certificate (thus the server) is trusted.

History and development

Secure Network Programming API

Early research efforts towards transport layer security included the Secure Network Programming (SNP) application programming interface (API), which in 1993 explored the approach of having a secure transport layer API closely resembling Berkeley sockets, to facilitate retrofitting preexisting network applications with security measures.[9]

SSL 1.0, 2.0 and 3.0

The SSL protocol was originally developed by Netscape.[10] 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."[11] SSL version 3.0, released in 1996, was a complete redesign of the protocol produced by Paul Kocher working with Netscape engineers Phil Karlton and Alan Freier. Newer versions of SSL/TLS are based on SSL 3.0. The 1996 draft of SSL 3.0 was published by IETF as a historical document in RFC 6101. The basic algorithm was written by Dr. Taher Elgamal. As the Chief Scientist of Netscape, Taher was recognized as the "inventor of SSL".

TLS 1.0

TLS 1.0 was first defined in RFC 2246 in January 1999 as an upgrade of SSL Version 3.0. As stated in the RFC, "the differences between this protocol and SSL 3.0 are not dramatic, but they are significant to preclude interoperability between TLS 1.0 and SSL 3.0." TLS 1.0 does include a means by which a TLS implementation can downgrade the connection to SSL 3.0, thus weakening security.

TLS 1.1

TLS 1.1 was defined in RFC 4346 in April 2006.[12] It is an update from TLS version 1.0. Significant differences in this version include:

- Added protection against Cipher block chaining (CBC) attacks.
  - The implicit Initialization Vector (IV) was replaced with an explicit IV.
  - Change in handling of padding errors.
- Support for IANA registration of parameters.

TLS 1.2

TLS 1.2 was defined in RFC 5246 in August 2008. It is based on the earlier TLS 1.1 specification. Major differences include:

- The MD5-SHA-1 combination in the pseudorandom function (PRF) was replaced with SHA-256, with an option to use cipher suite specified PRFs.
- The MD5-SHA-1 combination in the Finished message hash was replaced with SHA-256, with an option to use cipher suite specific hash algorithms. However the size of the hash in the finished message is still truncated to 96-bits.
- The MD5-SHA-1 combination in the digitally signed element was replaced with a single hash negotiated during handshake, defaults to SHA-1.
- Enhancement in the client's and server's ability to specify which hash and signature algorithms they will accept.
- Expansion of support for authenticated encryption ciphers, used mainly for Galois/Counter Mode (GCM) and CCM mode of Advanced Encryption Standard encryption.
- TLS Extensions definition and Advanced Encryption Standard cipher suites were added.

All TLS versions were further refined in RFC 6176 in March 2011 removing their backward compatibility with SSL such that TLS sessions will never negotiate the use of Secure Sockets Layer (SSL) version 2.0.

Applications and adoption

In applications design, TLS is usually implemented on top of any of the Transport Layer protocols, encapsulating the application-specific protocols such as HTTP, FTP, SMTP, NNTP and XMPP. Historically it has been used primarily with reliable transport protocols such as the Transmission Control Protocol (TCP). However, it has also been implemented with datagram-oriented transport protocols, such as the User Datagram Protocol (UDP) and the Datagram Congestion Control Protocol (DCCP), usage which has been standardized independently using the term Datagram Transport Layer Security (DTLS).

Websites

A prominent use of TLS is for securing World Wide Web traffic between the website and the browser carried by HTTP to form HTTPS. Notable applications are electronic commerce and asset management.

Website protocol support

Protocol version	Website support[13]	Security[13][14]
SSL 2.0	23.0% (−0.7%)	Insecure
SSL 3.0	99.3% (−0.1%)	Depends on cipher[n 1] and client mitigations[n 2]
TLS 1.0	97.7% (±0.0%)	Depends on cipher[n 1] and client mitigations[n 2]
TLS 1.1	29.6% (+2.0%)	Depends on cipher[n 1] and client mitigations[n 2]
TLS 1.2	32.3% (+2.1%)	Depends on cipher[n 1] and client mitigations[n 2]

Notes

1. see #Cipher table below
2. see #Web browsers and #Attacks against TLS/SSL sections

Key exchange or key agreement

See also: Cipher suite

Before a client and server can begin to exchange information protected by TLS, they must securely exchange or agree upon an encryption key and a cipher to use when encrypting data (see Cipher). Among the methods used for key exchange/agreement are: public and private keys generated with RSA (denoted TLS_RSA in the TLS handshake protocol), Diffie-Hellman (denoted TLS_DH in the TLS handshake protocol), ephemeral Diffie-Hellman (denoted TLS_DHE in the handshake protocol), Elliptic Curve Diffie-Hellman (denoted TLS_ECDH), ephemeral Elliptic Curve Diffie-Hellman (TLS_ECDHE), anonymous Diffie-Hellman (TLS_DH_anon),[15] and PSK (TLS_PSK).[16]

The TLS_DH_anon key agreement method does not authenticate the server or the user and hence is rarely used. Only TLS_DHE and TLS_ECDHE provide forward secrecy.

Public key certificates used during exchange/agreement also vary in the size of the public/private encryption keys used during the exchange and hence the robustness of the security provided. In July 2013, Google announced that it would no longer use 1024 bit public keys and would switch instead to 2048 bit keys to increase the security of the TLS encryption it provides to its users.[17]

Cipher

See also: Cipher suite, Block cipher, and Cipher security summary

Cipher security against publicly known feasible attacks

Cipher	Protocol version				
	SSL 2.0	SSL 3.0 [note 1][note 2][note 3]	TLS 1.0 [note 1][note 3]	TLS 1.1 [note 1]	TLS 1.2 [note 1]
AES CBC[note 4]	N/A	N/A	Depends	Secure	Secure
AES GCM[18][note 5]	N/A	N/A	N/A	N/A	Secure
AES CCM[19][note 5]	N/A	N/A	N/A	N/A	Secure
Camellia CBC[20][note 4]	N/A	N/A	Depends	Secure	Secure
Camellia GCM[21][note 5]	N/A	N/A	N/A	N/A	Secure
SEED CBC[22][note 4]	N/A	N/A	Depends	Secure	Secure
ChaCha20+Poly1305[23][note 5]	N/A	N/A	N/A	N/A	Secure
IDEA CBC[note 4][note 6]	Insecure	Depends	Depends	Secure	N/A
Triple DES CBC[note 4][note	Insecur	Depends	Depends	Depend	Depend

Cipher	Protocol version				
	SSL 2.0	SSL 3.0 [note 1][note 2][note 3]	TLS 1.0 [note 1][note 3]	TLS 1.1 [note 1]	TLS 1.2 [note 1]
7]	e			s	s
DES CBC[note 4][note 6]	Insecure	Insecure	Insecure	Insecure	N/A
RC2 CBC[note 4][note 6]	Insecure	Insecure	Insecure	Insecure	N/A
RC4[note 8]	Insecure	Insecure	Insecure	Insecure	Insecure

Notes

1. RFC 5746 must be implemented in order to fix a renegotiation flaw that would otherwise break this protocol.
2. If libraries implement fixes listed in RFC 5746, this will violate the SSL 3.0 specification, which the IETF cannot change unlike TLS. Fortunately, most current libraries implement the fix and disregard the violation that this causes.
3. the BEAST attack breaks all block ciphers (CBC ciphers) used in SSL 3.0 and TLS 1.0 unless mitigated by the client. As of March 2014, Apple has turned on this mitigation by default only for Safari 7 for Mac OS X 10.9 and for iOS 7, resulting in for Windows, for Mac OS X 10.8 or earlier, and for iOS 6 or earlier still being theoretically vulnerable to the BEAST attack on those platforms – see #Web browsers
4. CBC ciphers can be attacked with the Lucky 13 attack if the library is not written carefully to eliminate timing side channels.
5. AEAD ciphers (such as GCM and CCM) can be used in only TLS 1.2.
6. IDEA, DES, and RC2 CBC have been removed from TLS 1.2.
7. Triple DES provides only 108 or 112 bits of security, which is below the recommended minimum of 128 bits.[24]
8. the RC4 attacks weaken or break RC4 used in SSL/TLS

Web browsers

Further information: Comparison of web browsers

As of February 2014, the latest version of all major web browsers support SSL 3.0, TLS 1.0, 1.1, and 1.2 enabled by default. However, Mozilla Firefox ESR 24 supports TLS 1.1 and 1.2 but disabled by default, and Internet Explorer for Windows Vista or older and Safari for Mac OS X 10.8 or earlier, for iOS 6 or earlier, and for Windows support only SSL 3.0 and TLS 1.0.

Browser support for TLS

Browser	Version	Platforms	TLS 1.0	TLS 1.1	TLS 1.2	Vulnerabilities Fixed [notes 1]
Google Chrome	0–21	Android, iOS,	Yes	No	No	Not latest
	22–29		Yes[32]	Yes	No[32][33]	Not latest

Browser	Version	Platforms	TLS 1.0	TLS 1.1	TLS 1.2	Vulnerabilities Fixed [notes 1]
					[34][35]	
[notes 2] [notes 3]	30–	Linux, Mac OS X,	Yes[32]	Yes[32]	Yes[33][34] [35]	Depends
Mozilla Firefox [notes 3] [notes 4]	1–22 ESR 10, 17	Android, Firefox OS, Linux, Mac OS X, Windows (XP, Vista, 7, 8)	Yes[36]	No[28]	No[30]	Not latest
	23		Yes[36]	Yes, disabled by default[28] [37]	No[30]	Not latest
	24–26 ESR 24		Yes[36]	Yes, disabled by default[28] [37]	Yes, disabled by default[30] [38]	Depends (latest ESR)
	27–		Yes[36]	Yes[28][37] [39]	Yes[30][38] [39]	Depends (latest non-ESR)
Internet Explorer [notes 5]	6	Windows (98, 2000, ME, XP)	Yes, disabled by default	No	No	Not latest
	7–8	Windows XP	Yes	No	No	Depends (latest for Windows XP)
	7–9	Windows Vista	Yes	No	No	Depends (latest for Windows Vista)
	8–10	Windows 7	Yes	Yes, disabled by default	Yes, disabled by default	Not latest
	10	Windows 8	Yes	Yes, disabled by default	Yes, disabled by default	Depends (latest for Windows 8)
	11	Windows 7, 8.1	Yes	Yes[42]	Yes[42]	Depends[43] (latest for Windows 7, 8.1)
Opera [notes 6] [notes 7]	5–7	Android, iOS, Linux, Mac OS X, Windows	Yes[46]	No	No	Not latest
	8–9		Yes	Yes, disabled by default[47]	No	Not latest
	10–12		Yes	Yes,	Yes,	Depends

Browser	Version	Platforms	TLS 1.0	TLS 1.1	TLS 1.2	Vulnerabilities Fixed [notes 1]
Safari [notes 8]				disabled by default	disabled by default	(latest for Linux)
	14–16		Yes	Yes[48]	No[48]	Not latest
	17–		Yes	Yes[49]	Yes[49]	Depends (latest for Windows, OS X)
	1–6	Mac OS X −10.8[notes 9]	Yes	No	No	No[notes 9] (latest for OS X −10.8)
	7	Mac OS X 10.9 [notes 10]	Yes	Yes	Yes	Depends[55] (latest for OS X 10.9)
	3–5	iPhone OS 1–3, iOS 4.0[notes 11] [notes 9]	Yes[56]	No	No	Not latest
	5–6	iOS 5–6[notes 11][notes 9]	Yes	Yes	Yes	No[notes 9] (latest for iOS 5–6)
	7	iOS 7[notes 11] [notes 9]	Yes	Yes	Yes	Depends[60] (Latest for iOS 7)
	3–5	Windows	Yes	No	No	No[notes 12] (latest for Windows)

Notes

1. Does the current browser have mitigations or is not vulnerable for all the known protocol and cipher attacks listed in this page (BEAST, CRIME, BREACH, Lucky Thirteen). Note actual security depends on other factors such as negotiated cipher (such as RC4), encryption strength etc. Non-current browsers will have unfixed security issues so are not considered.
2. Google Chrome (and Chromium) supports TLS 1.0, and TLS 1.1 from version 22 (it was added, then dropped from version 21). TLS 1.2 support has been added, then dropped from Chrome 29.[25][26][27]
3. Uses the TLS implementation provided by NSS. NSS 3.14 supports TLS 1.0 and 1.1 but not 1.2.[28][29] As of 2 July 2013, TLS 1.2 has been implemented in NSS 3.15.1.[30][31]
4. As of Firefox 19, Firefox supports only TLS 1.0 despite the bundled NSS supporting TLS 1.1. Since Firefox 23, TLS 1.1 can be enabled, but was not enabled by default due to issues. Firefox 24 has TLS 1.2 support disabled by default. TLS 1.1 and TLS 1.2 have been enabled by default in Firefox 27 release.
5. IE uses the TLS implementation of the Microsoft Windows operating system provided by the SChannel security support provider. TLS 1.1 and 1.2 are disabled by default until

IE11.[40][41]
6. Opera 10 added support for TLS 1.2 as of Presto 2.2. Previous support was for TLS 1.0 and 1.1.[44] TLS 1.1 and 1.2 are disabled by default (except for version 9[45] that enabled TLS 1.1 by default).
7. TLS support of Opera 14 and above is same as that of Chrome, because Opera has migrated to Chromium backend.
8. Safari uses the operating system implementation on Mac OS X, Windows (XP, Vista, 7) [50] with unknown version,[51] Safari 5 is the last version available for Windows. OS X 10.8 on have SecureTransport support for TLS 1.1 and 1.2[52] Qualys SSL report simulates Safari 5.1.9 connecting with TLS 1.0 not 1.1 or 1.2[53]
9. As of September 2013, Apple has implemented BEAST mitigation in OS X 10.8 (Mountain Lion), but it is not turned on by default resulting in Safari still being theoretically vulnerable to the BEAST attack on that platform[56][55]
10. Simulated connection by Qualys.[54]
11. Mobile Safari and third-party software utilizing the system UIWebView library use the iOS operating system implementation, which supports TLS 1.2 as of iOS 5.0.[57][58][59]
12. Development and support of Safari for Windows has been discontinued.

Libraries

Main article: Comparison of TLS implementations

Most SSL and TLS programming libraries are free and open source software.

- Botan, a BSD-licensed cryptographic library written in C++.
- Microsoft Windows includes an implementation of SSL and TLS as part of its Secure Channel package.
- OS X includes an implementation of SSL and TLS as part of its Secure Transport package.
- Delphi programmers may use a library called Indy which utilizes OpenSSL.
- OpenSSL: a free implementation (BSD license with some extensions)
- GnuTLS: a free implementation (LGPL licensed)
- cryptlib: a portable open source cryptography library (includes TLS/SSL implementation)
- JSSE: a Java implementation included in the Java Runtime Environment supports TLS 1.1 and 1.2 from Java 7, although is disabled by default for client, and enabled by default for server.[61] Java 8 supports TLS 1.1 and 1.2 enabled on both the client and server by default.[62]
- MatrixSSL: a dual licensed implementation
- Network Security Services (NSS): FIPS 140 validated open source library
- PolarSSL: A tiny SSL library implementation for embedded devices that is designed for ease of use
- CyaSSL: Embedded SSL/TLS Library with a strong focus on speed and size.

A paper presented at the 2012 ACM conference on computer and communications security[63] showed that few applications used some of these SSL libraries incorrectly, leading to vulnerabilities. According to the authors

> "the root cause of most of these vulnerabilities is the terrible design of the APIs to the

> underlying SSL libraries. Instead of expressing high-level security properties of network tunnels such as confidentiality and authentication, these APIs expose low-level details of the SSL protocol to application developers. As a consequence, developers often use SSL APIs incorrectly, misinterpreting and misunderstanding their manifold parameters, options, side effects, and return values."

Other uses

The Simple Mail Transfer Protocol (SMTP) can also be protected by TLS. These applications use public key certificates to verify the identity of endpoints.

TLS can also be used to tunnel an entire network stack to create a VPN, as is the case with OpenVPN and OpenConnect. 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 a standard method to protect Session Initiation Protocol (SIP) application signaling. TLS can be used to provide authentication and encryption of the SIP signaling associated with VoIP and other SIP-based applications.

Security

SSL 2.0

SSL 2.0 is flawed in a variety of ways:[64]

- Identical cryptographic keys are used for message authentication and encryption.
- SSL 2.0 has a weak MAC construction that uses the MD5 hash function with a secret prefix, making it vulnerable to length extension attacks.
- SSL 2.0 does not have any protection for the handshake, meaning a man-in-the-middle downgrade attack can go undetected.
- SSL 2.0 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 3.0 fixes this problem by having an explicit closure alert).
- SSL 2.0 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.

SSL 2.0 is disabled by default, beginning with Internet Explorer 7,[65] Mozilla Firefox 2,[66] Opera 9.5,[67] 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 2.0 format to maximize the likelihood of successfully handshaking with older servers.[68] Support for SSL 2.0 (and weak 40-bit and 56-bit ciphers) has been

removed completely from Opera as of version 10.[69][70]

SSL 3.0

SSL 3.0 improved upon SSL 2.0 by adding SHA-1–based ciphers and support for certificate authentication.

From a security standpoint, SSL 3.0 should be considered less desirable than TLS 1.0. The SSL 3.0 cipher suites have a weaker key derivation process; half of the master key that is established is fully dependent on the MD5 hash function, which is not resistant to collisions and is, therefore, not considered secure. Under TLS 1.0, the master key that is established depends on both MD5 and SHA-1 so its derivation process is not currently considered weak. It is for this reason that SSL 3.0 implementations cannot be validated under FIPS 140-2.[71]

TLS

TLS has a variety of security measures:

- Protection against a downgrade of the protocol to a previous (less secure) version or a weaker cipher suite.
- Numbering subsequent 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). The HMAC construction used by most TLS cipher suites is specified in RFC 2104 (SSL 3.0 used a different hash-based MAC).
- 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.

Attacks against TLS/SSL

Significant attacks against TLS/SSL are listed below:

Renegotiation attack

A vulnerability of the renegotiation procedure was discovered in August 2009 that can lead to plaintext injection attacks against SSL 3.0 and all current versions of TLS. For example, it allows an attacker who can hijack an https connection to splice their own requests into the beginning of the conversation the client has with the web server. The attacker can't actually decrypt the client-server communication, so it is different from a typical man-in-the-middle attack. A short-term fix is for web servers to stop allowing renegotiation, which typically will not require other changes unless client certificate authentication is used. To fix the vulnerability, a renegotiation indication extension was proposed for TLS. It will require the client and server to include and verify information about previous handshakes in any renegotiation handshakes.[72] This extension has become a proposed standard and has been assigned the number RFC 5746. The RFC has been implemented by several libraries.[73][74][75]

Version rollback attacks

Modifications to the original protocols, like **False Start**[76] (adopted and enabled by Google Chrome[77]) or Snap Start, have been reported to introduce limited TLS protocol version rollback attacks[78] or to allow modifications to the cipher suite list sent by the client to the server (an attacker may be able to influence the cipher suite selection in an attempt to downgrade the cipher suite strength, to use either a weaker symmetric encryption algorithm or a weaker key exchange[79]). It has been shown in the Association for Computing Machinery (ACM) conference on computer and communications security that the False Start extension is at risk as in certain circumstances it could allow an attacker to recover the encryption keys offline and access the encrypted data.[80]

BEAST attack

On September 23, 2011 researchers Thai Duong and Juliano Rizzo demonstrated a proof of concept called **BEAST** (**Browser Exploit Against SSL/TLS**)[81] using a Java applet to violate same origin policy constraints, for a long-known cipher block chaining (CBC) vulnerability in TLS 1.0.[82][83] Practical exploits had not been previously demonstrated for this vulnerability, which was originally discovered by Phillip Rogaway[84] in 2002. The vulnerability of the attack had been fixed with TLS 1.1 in 2006, but TLS 1.1 had not seen wide adoption prior to this attack demonstration.

Mozilla updated the development versions of their NSS libraries to mitigate BEAST-like attacks. NSS is used by Mozilla Firefox and Google Chrome to implement SSL. Some web servers that have a broken implementation of the SSL specification may stop working as a result.[85]

Microsoft released Security Bulletin MS12-006 on January 10, 2012, which fixed the BEAST vulnerability by changing the way that the Windows Secure Channel (SChannel) component transmits encrypted network packets.[86]

Users of Windows 7, Windows 8 and Windows Server 2008 R2 can enable use of TLS 1.1 and 1.2, but this workaround will fail if it is not supported by the other end of the connection and will result in a fall-back to TLS 1.0.

CRIME and BREACH attacks

Main articles: CRIME (security exploit) and BREACH (security exploit)

The authors of the BEAST attack are also the creators of the later CRIME attack, which can allow an attacker to recover the content of web cookies when data compression is used along with TLS.[87][88] When used to recover the content of secret authentication cookies, it allows an attacker to perform session hijacking on an authenticated web session.

While the CRIME attack was presented as a general attack that could work effectively against a large number of protocols, including but not limited to TLS, and application-layer protocols such as SPDY or HTTP, only exploits against TLS and SPDY were demonstrated and largely mitigated in browsers and servers. The CRIME exploit against HTTP compression has not been mitigated at all, even though the authors of CRIME have warned that this vulnerability might be even more widespread than SPDY and TLS compression combined. In 2013 a new instance of the CRIME attack against HTTP compression, dubbed BREACH, was announced. Built based

on the CRIME attack a BREACH attack can extract login tokens, email addresses or other sensitive information from TLS encrypted web traffic in as little as 30 seconds (depending on the number of bytes to be extracted), provided the attacker tricks the victim into visiting a malicious web link or is able to inject content into valid pages the user is visiting (ex: a wireless network under the control of the attacker).[89] All versions of TLS and SSL are at risk from BREACH regardless of the encryption algorithm or cipher used.[90] Unlike previous instances of CRIME, which can be successfully defended against by turning off TLS compression or SPDY header compression, BREACH exploits HTTP compression which cannot realistically be turned off, as virtually all web servers rely upon it to improve data transmission speeds for users.[89] This is a known limitation of TLS as it is susceptible to chosen-plaintext attack against the application-layer data it was meant to protect.

Padding attacks

Earlier TLS versions were vulnerable against the padding oracle attack discovered in 2002. A novel variant, called the Lucky Thirteen attack, was published in 2013. As of February 2013, TLS implementors were still working on developing fixes to protect against this form of attack.

RC4 attacks

In spite of existing attacks on RC4 that break it, the cipher suites based on RC4 in SSL and TLS were considered secure because of how the cipher was used in these protocols. In 2011 the RC4 suite was actually recommended as a work around for the BEAST attack.[91] In 2013 a vulnerability was discovered in RC4 suggesting it was not a good workaround for BEAST.[14] An attack scenario was proposed by AlFardan, Bernstein, Paterson, Poettering and Schuldt that used newly discovered statistical biases in the RC4 key table[92] to recover parts of the plaintext with a large number of TLS encryptions.[93][94] A double-byte bias attack on RC4 in TLS and SSL that requires 13×2^{20} encryptions to break RC4 was unveiled on 8 July 2013, and it was described as "feasible" in the accompanying presentation at the 22nd USENIX Security Symposium on August 15, 2013.[95][96]

However, many modern browsers have been designed to defeat BEAST attacks (except Safari for Mac OS X 10.8 or earlier, for iOS 6 or earlier, and for Windows; see #Web browsers). As a result, RC4 is not the best choice for TLS 1.0 anymore. The CBC ciphers which were affected by the BEAST attack in the past are becoming a more popular choice for protection.[24]

Microsoft recommends disabling RC4 where possible.[97][98]

Truncation attack

A TLS truncation attack blocks a victim's account logout requests so that the user unknowingly remains logged into a web service. When the request to sign out is sent, the attacker injects an unencrypted TCP FIN message (no more data from sender) to close the connection. The server therefore doesn't receive the logout request and is unaware of the abnormal termination.[99]

Published in July 2013,[100] the attack causes web services such as Gmail and Hotmail to display a page that informs the user that they have successfully signed-out, while ensuring that the user's browser maintains authorization with the service, allowing an attacker with subsequent access to the browser to access and take over control of the user's logged-in

account. The attack does not rely on installing malware on the victim's computer; attackers need only place themselves between the victim and the web server (e.g., by setting up a rogue wireless hotspot).[99]

Survey of websites

As of April 2014, Trustworthy Internet Movement estimate the ratio of websites that are vulnerable to TLS attacks.[13]

Survey of the TLS vulnerabilities of the most popular websites

Attacks	Security			
	Insecure	Depends	Secure	Other
Renegotiation attack	5.7% (−0.3%) support insecure renegotiation	2.3% (+0.9%) support both	84.7% (−0.4%) support secure renegotiation	7.3% (−0.2%) not support
RC4 attacks	33.4% (−2.3%) support RC4 suites used with modern browsers	58.0% (+2.0%) support some RC4 suites	8.7% (+0.3%) not support	N/A
BEAST attack	71.8% (+2.2%) vulnerable	N/A	N/A	N/A
CRIME attack	12.9% (−0.7%) vulnerable	N/A	N/A	N/A

Forward secrecy

Main article: Forward secrecy

Forward secrecy is a property of cryptographic systems which ensures that a session key derived from a set of public and private keys will not be compromised if one of the private keys is compromised in the future.[101] An implementation of TLS can provide forward secrecy by requiring the use of ephemeral Diffie-Hellman key exchange to establish session keys, and some notable TLS implementations do so exclusively: e.g., Gmail and other Google HTTPS services that use OpenSSL.[102] However, many web servers providing TLS are not configured to implement such restrictions.[103][104] Without forward secrecy, if the server's private key is compromised, not only will all future TLS-encrypted sessions using that server certificate be compromised, but also any past sessions that used it as well (provided of course that these past sessions were intercepted and stored at the time of transmission).[105] In practice, unless a web service uses Diffie-Hellman key exchange to implement forward secrecy, all of the encrypted web traffic to and from that service can be decrypted by a third party if it obtains the server's master (private) key; e.g., by means of a court order.[106]

Even where Diffie-Hellman key exchange is implemented, server-side session management mechanisms can impact forward secrecy. The use of TLS session tickets (a TLS extension) causes the session to be protected by AES128-CBC-SHA256 regardless of any other negotiated TLS parameters, including forward secrecy ciphersuites, and the long-lived TLS session ticket keys defeat the attempt to implement forward secrecy.[107][108][109]

Since late 2011, Google has provided forward secrecy with TLS by default to users of its Gmail service, along with Google Docs and encrypted search among other services.[110] Since November 2013, Twitter has provided forward secrecy with TLS to users of its service.[111] As of April 2014, 6.3% of TLS-enabled websites are configured to use cipher suites that provide forward secrecy to web browsers.[13]

Avoiding Triple-DES CBC

Some experts recommend avoiding Triple-DES CBC. Since the last supported ciphers developed to support Internet Explorer on Windows XP are RC4 and Triple-DES, this makes it difficult to support SSL for IE on XP.[24]

Dealing with MITM attacks

Main article: Man-in-the-middle attack

Certificate pinning

One way to detect and block many kinds of MITM attacks is "certificate pinning", sometimes called "SSL pinning".[112] A client that does certificate pinning adds an extra step to the normal TLS protocol or SSL protocol: After obtaining the server's certificate in the standard way, the client checks the server's certificate against trusted validation data. Typically the trusted validation data is bundled with the app, in the form of a trusted copy of that certificate, or a trusted hash or fingerprint of that certificate or the certificate's public key. For example, Chromium and Google Chrome include validation data for the *.google.com certificate that detected fraudulent certificates in 2011. In other systems the client hopes that the first time it obtains a server's certificate it is trustworthy and stores it; during later sessions with that server, the client checks the server's certificate against the stored certificate to guard against later MITM attacks.

Perspectives Project

The Perspectives Project[113] operates network notaries that clients can use to detect if a site's certificate has changed. By their nature, man-in-the-middle attacks place the attacker between the destination and a single specific target. As such, Perspectives would warn the target that the certificate delivered to the web browser does not match the certificate seen from other perspectives - the perspectives of other users in different times and places. Use of network notaries from a multitude of perspectives makes it possible for a target to detect an attack even if a certificate appears to be completely valid.

Protocol details

The TLS protocol exchanges *records* – which encapsulate the data to be exchanged in a specific format (see below). 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 designates the type of data encapsulated, a length field and a TLS version field. The data encapsulated may be control or procedural messages of

the TLS itself, or simply the application data needed to be transferred by TLS. The specifications (cipher suite, keys etc.) required to exchange application data by TLS, are agreed upon in the "TLS handshake" between the client requesting the data and the server responding to requests. The protocol therefore defines both the structure of payloads transferred in TLS and the procedure to establish and monitor the transfer.

TLS handshake

When the connection starts, the record encapsulates a "control" protocol — the handshake messaging protocol (*content type* 22). This protocol is used to exchange all the information required by both sides for the exchange of the actual application data by TLS. It defines the messages formatting or containing this information and the order of their exchange. These may vary according to the demands of the client and server i.e. there are several possible procedures to set up the connection. This initial exchange results in a successful TLS connection (both parties ready to transfer application data with TLS) or an alert message (as specified below).

Basic TLS handshake

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

1. Negotiation phase:
    - A client sends a **ClientHello** message specifying the highest TLS protocol version it supports, a random number, a list of suggested CipherSuites and suggested compression methods. If the client is attempting to perform a resumed handshake, it may send a *session ID*.
    - The server responds with a **ServerHello** message, containing the chosen protocol version, a random number, CipherSuite and compression method from the choices offered by the client. To confirm or allow resumed handshakes the server may send a *session ID*. The chosen protocol version should be the highest that both the client and server support. For example, if the client supports TLS1.1 and the server supports TLS1.2, TLS1.1 should be selected; SSL 3.0 should not be selected.
    - The server sends its **Certificate** message (depending on the selected cipher suite, this may be omitted by the server).[114]
    - 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.) This *PreMasterSecret* is encrypted using the public key of the server 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 parameters were present in the server certificate)." The ChangeCipherSpec is itself a record-level protocol with content type of 20.
    - 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 client, "Everything I tell you from now on will be authenticated (and encrypted, 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. Otherwise, the content type will return 25 and the client will not authenticate.

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).[114]
    - 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 client, "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: resumed sessions. Resumed sessions are implemented using session IDs or session tickets.

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.[115]

Session IDs

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 from 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 server now sends a **ChangeCipherSpec** record, essentially telling the client, "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 server sends an encrypted **Finished** message, containing a hash and MAC over the previous handshake messages.
    - The client will attempt to decrypt the server'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 client sends a **ChangeCipherSpec**, telling the server, "Everything I tell you from now on will be encrypted. "
    - The client sends its own encrypted **Finished** message.
    - The server 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.

Session tickets

RFC 5077 extends TLS via use of session tickets, instead of session IDs. It defines a way to resume a TLS session without requiring that session-specific state is stored at the TLS server.

When using session tickets, the TLS server stores its session-specific state in a session ticket and sends the session ticket to the TLS client for storing. The client resumes a TLS session by sending the session ticket to the server, and the server resumes the TLS session according to the session-specific state in the ticket. The session ticket is encrypted and authenticated by the server, and the server verifies its validity before using its contents.

One particular weakness of this method is that it always limits encryption and authentication security of the transmitted TLS session ticket to AES128-CBC-SHA256, no matter what other TLS parameters were negotiated for the actual TLS session.[108] This means that the state information (the TLS session ticket) is not as well protected as the TLS session itself. Of particular concern is OpenSSL's storage of the keys in an application-wide context (SSL_CTX), i.e. for the life of the application, and not allowing for re-keying of the AES128-CBC-SHA256

TLS session tickets without resetting the application-wide OpenSSL context (which is uncommon, error-prone and often requires manual administrative intervention).[109][107]

TLS record

This is the general format of all TLS records.

+	Byte +0	Byte +1	Byte +2	Byte +3
Byte 0	Content type			
Bytes 1..4	Version *(Major)*	*(Minor)*	Length *(bits 15..8)*	*(bits 7..0)*
Bytes 5..(*m*-1)	Protocol message(s)			
Bytes *m*..(p-*1)*	MAC (optional)			
Bytes *p*..(q-*1)*	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
0x18	24	Heartbeat

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	SSL 3.0
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 signalling 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).

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

Message type

This field identifies the Handshake message type.

Message Types

Code	Description
0	HelloRequest
1	ClientHello
2	ServerHello
4	NewSessionTicket
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
Byte 0	21			
Bytes 1..4	Version *(Major)*	*(Minor)*	Length 0	2
Bytes 5..6	Level	Description		
Bytes *7..(p-1)*	MAC (optional)			
Bytes *p..(q-1)*	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. Signalling 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	Connection state

	type	
1	warning	connection or security may be unstable.
2	fatal	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	warning/fatal	
10	Unexpected message	fatal	
20	Bad record MAC	fatal	Possibly a bad SSL implementation, or payload has been tampered with e.g. FTP firewall rule on FTPS server.
21	Decryption failed	fatal	TLS only, reserved
22	Record overflow	fatal	TLS only
30	Decompression failure	fatal	
40	Handshake failure	fatal	
41	No certificate	warning/fatal	SSL 3.0 only, reserved
42	Bad certificate	warning/fatal	
43	Unsupported certificate	warning/fatal	e.g. certificate has only Server authentication usage enabled and is presented as a client certificate
44	Certificate revoked	warning/fatal	
45	Certificate expired	warning/fatal	Check server certificate expire also check no certificate in the chain presented has expired
46	Certificate unknown	warning/fatal	
47	Illegal parameter	fatal	
48	Unknown CA (Certificate authority)	fatal	TLS only
49	Access denied	fatal	TLS only – e.g. no client certificate has been presented (TLS: Blank certificate message or SSLv3: No Certificate alert), but server is configured to require one.
50	Decode error	fatal	TLS only
51	Decrypt error	warning/fatal	TLS only

60	Export restriction	**fatal**	TLS only, reserved
70	Protocol version	**fatal**	TLS only
71	Insufficient security	**fatal**	TLS only
80	Internal error	**fatal**	TLS only
90	User canceled	**fatal**	TLS only
100	No renegotiation	**warning**	TLS only
110	Unsupported extension	**warning**	TLS only
111	Certificate unobtainable	**warning**	TLS only
112	Unrecognized name	**warning**	TLS only; client's Server Name Indicator specified a hostname not supported by the server
113	Bad certificate status response	**fatal**	TLS only
114	Bad certificate hash value	**fatal**	TLS only
115	Unknown PSK identity (used in TLS-PSK and TLS-SRP)	**fatal**	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	Padding (block ciphers only)			

p..(q-1)

Length
: Length of Application data (excluding the protocol header and including 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 belong to the same domain, a wildcard certificate can be used. 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.[116]
- Add every virtual host name in the subjectAltName extension. The major problem being that the certificate needs to be reissued whenever a new virtual server is added.

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.

Standards

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

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

The current standard replaces these former versions, which are now considered obsolete:

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

as well as the never standardized SSL 2.0 and 3.0:

- Hickman, Kipp E.B. (April 1995). "The SSL Protocol". Retrieved July 31, 2013. This Internet Draft defines the now completely broken SSL 2.0.

- RFC 6101: "The Secure Sockets Layer (SSL) Protocol Version 3.0".

Other RFCs subsequently extended TLS.

Extensions to TLS 1.0 include:

- 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 cipher suites defined in this memo appear only for the purpose of documenting the fact that those cipher suite 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) cipher suites 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 Ciphersuites for Transport Layer Security (TLS)", adds three sets of new cipher suites for the TLS protocol to support authentication based on pre-shared keys.

Extensions to TLS 1.1 include:

- 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 4680: "TLS Handshake Message for Supplemental Data".
- RFC 4681: "TLS User Mapping Extension".
- RFC 4785: "Pre-Shared Key (PSK) Ciphersuites with NULL Encryption for Transport

Layer Security (TLS)".
- RFC 5054: "Using the Secure Remote Password (SRP) Protocol for TLS Authentication". Defines the TLS-SRP ciphersuites.
- RFC 5077: "Transport Layer Security (TLS) Session Resumption without Server-Side State".
- RFC 5081: "Using OpenPGP Keys for Transport Layer Security (TLS) Authentication", obsoleted by RFC 6091.

Extensions to TLS 1.2 include:

- RFC 5288: "AES Galois Counter Mode (GCM) Cipher Suites for TLS".
- RFC 5289: "TLS Elliptic Curve Cipher Suites with SHA-256/384 and AES Galois Counter Mode (GCM)".
- RFC 5746: "Transport Layer Security (TLS) Renegotiation Indication Extension".
- RFC 5878: "Transport Layer Security (TLS) Authorization Extensions".
- RFC 6066: "Transport Layer Security (TLS) Extensions: Extension Definitions", includes Server Name Indication and OCSP stapling.
- RFC 6091: "Using OpenPGP Keys for Transport Layer Security (TLS) Authentication".
- RFC 6176: "Prohibiting Secure Sockets Layer (SSL) Version 2.0".
- RFC 6209: "Addition of the ARIA Cipher Suites to Transport Layer Security (TLS)".
- RFC 6460: "Suite B Profile for Transport Layer Security (TLS)".

Encapsulations of TLS include:

- RFC 5216: "The EAP-TLS Authentication Protocol"

References

1. T. Dierks, E. Rescorla (August 2008). "The Transport Layer Security (TLS) Protocol, Version 1.2".
2. SSL: Intercepted today, decrypted tomorrow, Netcraft, 2013-06-25.
3. Law Enforcement Appliance Subverts SSL, Wired, 2010-04-03.
4. New Research Suggests That Governments May Fake SSL Certificates, EFF, 2010-03-24.
5. A. Freier, P. Karlton, P. Kocher (August 2011). "The Secure Sockets Layer (SSL) Protocol Version 3.0".
6. "SSL/TLS in Detail". *Microsoft TechNet*. Updated July 31, 2003.
7. Checking for certificate revocation can slow down browsing, so browsers generally don't perform this check unless so configured.
8. "Description of the Secure Sockets Layer (SSL) Handshake". Support. microsoft.com. 2008-07-07. Retrieved 2012-05-17.
9. Thomas Y. C. Woo, Raghuram Bindignavle, Shaowen Su and Simon S. Lam, *SNP: An interface for secure network programming* Proceedings USENIX Summer Technical Conference, June **1994**
10. "THE SSL PROTOCOL". Netscape Corporation. 2007. Archived from the original on 14 June 1997.
11. Rescorla 2001

12. Dierks, T. and E. Rescorla (April 2006). "The Transport Layer Security (TLS) Protocol Version 1.1, RFC 4346".
13. As of April 6, 2014. "SSL Pulse: Survey of the SSL Implementation of the Most Popular Web Sites". Retrieved 2014-04-08.
14. ivanr. "RC4 in TLS is Broken: Now What?". Qualsys Security Labs. Retrieved 2013-07-30.
15. "RFC 5246: The Transport Layer Security (TLS) Protocol Version 1.2". Internet Engineering Task Force. Retrieved 9 September 2013.
16. P. Eronen, Ed. "RFC 4279: Pre-Shared Key Ciphersuites for Transport Layer Security (TLS)". Internet Engineering Task Force. Retrieved 9 September 2013.
17. Gothard, Peter. "Google updates SSL certificates to 2048-bit encryption". *Computing*. Incisive Media. Retrieved 9 September 2013.
18. RFC 5288
19. RFC 6655
20. RFC 5932
21. RFC 6367
22. RFC 4162
23. draft-agl-tls-chacha20poly1305-04
24. Qualys SSL Labs. "SSL/TLS Deployment Best Practices". Retrieved 19 November 2013.
25. Google (2012-05-29). "Dev Channel Update". Retrieved 2011-06-01.
26. Google (2012-08-21). "Stable Channel Update". Retrieved 2012-08-22.
27. Chromium Project (2013-05-30). "Chromium TLS 1.2 Implementation".
28. "Bug 565047 – (RFC4346) Implement TLS 1.1 (RFC 4346)". Retrieved 2013-10-29.
29. "NSS 3.14 release notes". Retrieved 29 October 2013.
30. "Bug 480514 – Implement support for TLS 1.2 (RFC 5246)". Retrieved 2013-10-29.
31. "NSS 3.15.1 release notes". Retrieved 6 July 2013.
32. "SSL/TLS Overview". 2008-08-06. Retrieved 2013-03-29.
33. "Chromium Issue 90392". 2008-08-06. Retrieved 2013-06-28.
34. "Issue 23503030 Merge 219882". 2013-09-03. Retrieved 2013-09-19.
35. "Issue 278370: Unable to submit client certificates over TLS 1.2 from Windows". 2013-08-23. Retrieved 2013-10-03.
36. "Security in Firefox 2". 2008-08-06. Retrieved 2009-03-31.
37. "Bug 733647 – Implement TLS 1.1 (RFC 4346) in Gecko (Firefox, Thunderbird), on by default". Retrieved 2013-12-04.
38. "Bug 861266 – Implement TLS 1.2 (RFC 5246) in Gecko (Firefox, Thunderbird), on by default". Retrieved 2013-11-18.
39. "Firefox Notes – Desktop". 2014-02-04. Retrieved 2014-02-04.
40. Microsoft (2012-09-05). "Secure Channel". Retrieved 2012-10-18.
41. Microsoft (2009-02-27). "MS-TLSP Appendix A". Retrieved 2009-03-19.
42. Microsoft (2013-09-24). "IE11 Changes". Retrieved 2013-11-01.
43. Microsoft (2013-06-24). "Release Notes: Important Issues in Windows 8.1 Preview". Retrieved 2013-10-29.
44. Yngve Nysæter Pettersen (2009-02-25). "New in Opera Presto 2.2: TLS 1.2 Support". Retrieved 2009-02-25.
45. "Changelog for Opera 9.0 for Windows" at Opera.com
46. "Changelog for Opera 5.x for Windows" at Opera.com

47."Changelog for Opera [8] Beta 2 for Windows" at Opera.com
48.same as Chrome 27–29
49.same as Chrome 30-current
50.Adrian, Dimcev. "Common browsers/libraries/servers and the associated cipher suites implemented". *TLS Cipher Suites Project*.
51.Apple (2009-06-10). "Features". Retrieved 2009-06-10.
52.Curl: Patch to add TLS 1.1 and 1.2 support & replace deprecated functions in SecureTransport
53.Qualys SSL Report: google.co.uk (simulation Safari 5.1.9 TLS 1.0)
54.Qualys (2013-09-28). "SSL Report: facebook.com (173.252.110.27)". Retrieved 2013-09-28.
55.Ivan Ristić (2013-10-31). "Apple enabled BEAST mitigations in OS X 10.9 Mavericks". Retrieved 2013-11-07.
56.Ristic, Ivan. "Is BEAST Still a Threat?". *qualys.com*.
57.Apple (2011-10-14). "Technical Note TN2287 – iOS 5 and TLS 1.2 Interoperability Issues". Retrieved 2012-12-10.
58.Liebowitz, Matt (2011-10-13). "Apple issues huge software security patches". *NBCNews.com*. Retrieved 2012-12-10.
59.MWR Info Security (2012-04-16). "Adventures with iOS UIWebviews". Retrieved 2012-12-10., section "HTTPS (SSL/TLS)"
60.schurtertom (October 11, 2013). "SOAP Request fails randomly on one Server but works on an other on iOS7". Retrieved January 5, 2014.
61.Oracle. "Java Cryptography Architecture Oracle Providers Documentation". Retrieved 2012-08-16.
62.Oracle. "JDK 8 Security Enhancements". Retrieved 2014-03-25.
63.Georgiev, Martin and Iyengar, Subodh and Jana, Suman and Anubhai, Rishita and Boneh, Dan and Shmatikov, Vitaly (2012). *The most dangerous code in the world: validating SSL certificates in non-browser software. Proceedings of the 2012 ACM conference on Computer and communications security*. pp. 38–49. ISBN 978-1-4503-1651-4.
64.Joris Claessens, Valentin Dem, Danny De Cock, Bart Preneel, Joos Vandewalle (2002). "On the Security of Today's Online Electronic Banking Systems". *Computers & Security* **21** (3): 253–265. doi:10.1016/S0167-4048(02)00312-7.
65.Lawrence, Eric (2005-10-22). "IEBlog: Upcoming HTTPS Improvements in Internet Explorer 7 Beta 2". MSDN Blogs. Retrieved 2007-11-25.
66."Bugzilla@Mozilla — Bug 236933 – Disable SSL2 and other weak ciphers". Mozilla Corporation. Retrieved 2007-11-25.
67."Opera 9.5 for Windows Changelog" at Opera.com: "Disabled SSL v2 and weak ciphers."
68."Firefox still sends SSLv2 handshake even though the protocol is disabled". 2008-09-11.
69."Opera 10 for Windows changelog" at Opera.com: "Removed support for SSL v2 and weak ciphers"
70.Pettersen, Yngve (2007-04-30). "10 years of SSL in Opera — Implementer's notes". Opera Software. Retrieved 2007-11-25.
71.National Institute of Standards and Technology (December 2010). "Implementation Guidance for FIPS PUB 140-2 and the Cryptographic Module Validation Program".
72.Eric Rescorla (2009-11-05). "Understanding the TLS Renegotiation Attack". *Educated*

Guesswork. Retrieved 2009-11-27.
73. "SSL_CTX_set_options SECURE_RENEGOTIATION". *OpenSSL Docs*. 2010-02-25. Retrieved 2010-11-18.
74. "GnuTLS 2.10.0 released". *GnuTLS release notes*. 2010-06-25. Retrieved 2011-07-24.
75. "NSS 3.12.6 release notes". *NSS release notes*. 2010-03-03. Retrieved 2011-07-24.
76. A. Langley; N. Modadugu, B. Moeller (June 20120). "Transport Layer Security (TLS) False Start". *Internet Engineering Task Force*. IETF. Retrieved 31 July 2013.
77. Wolfgang, Gruener. "False Start: Google Proposes Faster Web, Chrome Supports It Already". Archived from the original on October 7, 2010. Retrieved 9 March 2011.
78. Brian, Smith. "Limited rollback attacks in False Start and Snap Start". Retrieved 9 March 2011.
79. Adrian, Dimcev. "False Start". *Random SSL/TLS 101*. Retrieved 9 March 2011.
80. Mavrogiannopoulos, Nikos and Vercautern, Frederik and Velichkov, Vesselin and Preneel, Bart (2012). *A cross-protocol attack on the TLS protocol. Proceedings of the 2012 ACM conference on Computer and communications security*. pp. 62–72. ISBN 978-1-4503-1651-4.
81. Thai Duong and Juliano Rizzo (2011-05-13). "Here Come The ⊕ Ninjas".
82. Dan Goodin (2011-09-19). "Hackers break SSL encryption used by millions of sites".
83. "Y Combinator comments on the issue". 2011-09-20.
84. "Security of CBC Ciphersuites in SSL/TLS: Problems and Countermeasures". 2004-05-20. Archived from the original on 2012-06-30.
85. Brian Smith (2011-09-30). "(CVE-2011-3389) Rizzo/Duong chosen plaintext attack (BEAST) on SSL/TLS 1.0 (facilitated by websockets −76)".
86. "Vulnerability in SSL/TLS Could Allow Information Disclosure (2643584)". 2012-01-10.
87. Dan Goodin (2012-09-13). "Crack in Internet's foundation of trust allows HTTPS session hijacking". Ars Technica. Retrieved 2013-07-31.
88. Dennis Fisher (September 13, 2012). "CRIME Attack Uses Compression Ratio of TLS Requests as Side Channel to Hijack Secure Sessions". ThreatPost. Retrieved 2012-09-13.
89. Goodin, Dan (1 August 2013). "Gone in 30 seconds: New attack plucks secrets from HTTPS-protected pages". *Ars Technica*. Condé Nast. Retrieved 2 August 2013.
90. Leyden, John (2 August 2013). "Step into the BREACH: New attack developed to read encrypted web data". *The Register*. Retrieved 2 August 2013.
91. security – Safest ciphers to use with the BEAST? (TLS 1.0 exploit) I've read that RC4 is immune – Server Fault
92. Pouyan Sepehrdad, Serge Vaudenay, Martin Vuagnoux (2011). "Discovery and Exploitation of New Biases in RC4". *Lecture Notes in Computer Science* **6544**: 74–91. doi:10.1007/978-3-642-19574-7_5.
93. Green, Matthew. "Attack of the week: RC4 is kind of broken in TLS". *Cryptography Engineering*. Retrieved March 12, 2013.
94. Nadhem AlFardan, Dan Bernstein, Kenny Paterson, Bertram Poettering and Jacob Schuldt. "On the Security of RC4 in TLS". Royal Holloway University of London. Retrieved March 13, 2013.
95. AlFardan, Nadhem J.; Bernstein, Daniel J.; Paterson, Kenneth G.; Poettering, Bertram; Schuldt, Jacob C. N. (8 July 2013). *On the Security of RC4 in TLS and WPA* (PDF). Retrieved 2 September 2013.

96. AlFardan, Nadhem J.; Bernstein, Daniel J.; Paterson, Kenneth G.; Poettering, Bertram; Schuldt, Jacob C. N. (15 August 2013). "On the Security of RC4 in TLS" (PDF). 22nd USENIX Security Symposium. p. 51. Retrieved 2 September 2013. "Plaintext recovery attacks against RC4 in TLS are feasible although not truly practical"
97. "Security Advisory 2868725: Recommendation to disable RC4". Microsoft. 2013-11-12. Retrieved 2013-12-04.
98. draft-popov-tls-prohibiting-rc4-01
99. John Leyden (1 August 2013). "Gmail, Outlook.com and e-voting 'pwned' on stage in crypto-dodge hack". *The Register*. Retrieved 1 August 2013.
100. "BlackHat USA Briefings". *Black Hat 2013*. Retrieved 1 August 2013.
101. Diffie, Whitfield; van Oorschot, Paul C.; Wiener, Michael J. (June 1992). "Authentication and Authenticated Key Exchanges". *Designs, Codes and Cryptography* **2** (2): 107–125. doi:10.1007/BF00124891. Retrieved 2008-02-11.
102. "Protecting data for the long term with forward secrecy". Retrieved 2012-11-05.
103. Vincent Bernat. "SSL/TLS & Perfect Forward Secrecy". Retrieved 2012-11-05.
104. "SSL Labs: Deploying Forward Secrecy". Qualys.com. 25 June 2013. Retrieved 10 July 2013.
105. Discussion on the TLS mailing list in October 2007
106. Ristic, Ivan (5 August 2013). "SSL Labs: Deploying Forward Secrecy". Qualsys. Retrieved 31 August 2013.
107. Langley, Adam (27 June 2013). "How to botch TLS forward secrecy". *imperialviolet.org*.
108. Daignière, Florent. "TLS "Secrets": Whitepaper presenting the security implications of the deployment of session tickets (RFC 5077) as implemented in OpenSSL". Matta Consulting Limited. Retrieved 7 August 2013.
109. Daignière, Florent. "TLS "Secrets": What everyone forgot to tell you...". Matta Consulting Limited. Retrieved 7 August 2013.
110. "Protecting data for the long term with forward secrecy". Retrieved 2014-03-07.
111. Hoffman-Andrews, Jacob. "Forward Secrecy at Twitter". *Twitter*. Twitter. Retrieved 2014-03-07.
112. "Certificate Pinning".
113. Perspectives Project
114. These certificates are currently X.509, but RFC 6091 also specifies the use of OpenPGP-based certificates.
115. Chris (2009-02-18). "vsftpd-2.1.0 released – Using TLS session resume for FTPS data connection authentication". Scarybeastsecurity. blogspot.com. Retrieved 2012-05-17.
116. "Named-based SSL virtual hosts: how to tackle the problem" (PDF). Retrieved 2012-05-17.

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. pp. 29–40.
- 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). Retrieved 2011-09-23.
- Canvel, Brice. "Password Interception in a SSL/TLS Channel". Retrieved 2007-04-20.
- IETF Multiple Authors. "RFC of change for TLS Renegotiation". Retrieved 2009-12-11.
- Creating VPNs with IPsec and SSL/TLS Linux Journal article by Rami Rosen