Transport Layer Security

Sean Turner • *IECA*

Transport Layer Security is the standard, widely deployed protocol for securing client-server communications over the Internet. TLS is designed to prevent eavesdropping, tampering, and message forgery for client-server applications. Here, the author looks at the collection of standards that make up TLS, including its history, protocol, and future.

The Transport Layer Security (TLS) protocol, or as it's sometimes referred to, the Secure Sockets Layer (SSL) protocol, is a stateful, connection-oriented, client-server protocol. It's arguably the most widely deployed communications security protocol on the Internet, providing authentication, integrity, and confidentiality for two parties. It's probably most widely known as the protocol that, coupled with HTTP, secures the Web and uses the https URI scheme. There's a good chance that even your nontechie friends and relatives have heard of TLS or SSL because of recent headlines describing vulnerabilities affecting certain implementations; in some instances, these vulnerabilities affected a good-sized chunk of secure Internet Web servers. One thing to keep in mind, though, is that TLS is used for more than just securing the Web. Here, I explore where TLS came from and what problem it was trying to solve, how it works, and what it doesn't do. I also provide a snapshot of anticipated revisions for the next version.

How Did We Get Here?

Before the Web became a thing, people accessed information on the Internet using protocols such as Telnet[1] and FTP,[2] and a little later, Gopher[3]; at least, these were the protocols my friends and I used most. When the Web showed up, the formidable HTTP and HTML combo supplanted these protocols; some might say this was due to the graphical user interface making the Internet more accessible to the average Joe/Jane (that is, nonscientists and engineers), but I digress. These protocols needed to transmit sensitive data from the client to the server in the form of the username/password tuple, which allowed client access to the data on the server. However, the prevailing winds at the time blew toward securing all of the data exchanged between the client and server; this all helped commercialize the Web. I like to think of it as the Web's Jerry McGuire moment: those trying to operate businesses on the Internet needed people to have the confidence to give up their credit-card numbers to make money, and this was going to enable it.

And the Winner Is …

To be clear, TLS wasn't anointed winner right out of the gate — at least two skirmishes occurred.

The first was over where to do it. Although the Internet Security Protocol (IPsec),[4] which was under development around the same time, could have been used instead of TLS, IPsec is realized in kernel space, whereas TLS is realized in the user space. Kernel changes, by all accounts I've heard, take a long time to become ubiquitous, while distributing an application in user space is much quicker. The win here goes to TLS.

The second question was which protocol to use. There were really two competing proposals: Netscape's SSL and Microsoft's Private Communications Technology (PCT).[5] Remember that these were interesting times: these two proposals were developed by the main protagonists in the first installment of the so-called "browser wars." PCT had some issues and wasn't widely deployed; regardless, some say it did its job by getting Netscape to release SSL to a standards development organization (SDO) — more on this next. Depending on how you score it, the win here goes to TLS.

Now, because the Web ended up everywhere, code reuse is common, and designing security protocols is hard, SSL/TLS got adopted by a lot of application-layer protocols.

Something Wicked (Cool) This Way Comes

Netscape first developed SSL 1.0 in the middle of 1994. The 1.0 specification was never distributed beyond Netscape; the secondhand scuttlebutt was that it had some issues with the integrity mechanism (something about the interaction of the MD5-based integrity mechanism and the RC4 cipher interaction) and lacked replay protection. These problems led to SSL 2.0 at the beginning of 1995,[6] but it too had issues, which are documented in RFC 6176.[7] These problems ultimately led most implementations to disable SSL 2.0 by default. SSL 3.0 arrived in 1996 to address the known SSL 2.0 deficiencies, and gained widespread deployment. It was published many years later — some might say posthumously — as RFC 6101.[8]

The King Is Dead, Long Live the King

Development of the protocol formerly known as SSL and henceforth referred to as TLS moved from Netscape to the IETF TLS working group (WG) in mid 1996. TLS 1.0 wasn't a rubberstamp of SSL 3.0; some changes were trivial, such as incrementing the version number and trimming the list of ciphers to those that were public; other revisions were more structural, such as changing the procedures for key generation and authentication. Over the years, the TLS WG has produced three versions — TLS 1.0 in 1999, TLS 1.1 in 2006, and TLS 1.2 in 2008 — tweaking the protocol along the way to address known issues and add features. Work on TLS 1.3 is under way, and it's not entirely clear where it will end up. What follows is a description focusing on pre-1.3 versions.

How Does This Thing Work?

Since SSL 3.0, the protocol's design has basically remained the same, although some underlying details have changed. The protocol has two parts. The *handshaking protocol* comes first; it negotiates the cipher suite, authenticates the server and, optionally, the client, and establishes the session keys. After the handshake comes the *record protocol*, which secures the application data with the session key established in the record protocol and verifies the application data's integrity and origin.

Let's Get This Party Started!

The handshaking protocol actually has three sub-protocols. The *handshake protocol* does the heavy lifting by negotiating the version of the protocol, the session identifier, the peer's public key (optionally), the compression method, the cipher specification, and the master secret.

The *alert protocol*, which isn't normally sent during the handshake protocol, notifies the peer about the cause of a protocol failure. Alerts are categorized as either a *failure alert*, in which case the session is terminated, or a *warning alert*, in which case the recipient gets the choice to end the session. Fatal warnings result in terminated connections that can't be resumed.

The *change cipher specification protocol* informs the peer that the sender wants to change to a new set of keys, which are created from information exchanged by the handshake protocol.

For a new session, the "normal" protocol flow — by which I mean clients and servers negotiating the mandatory-to-implement cipher suite (more about this later) — can begin anytime after a TCP ACK is received, and goes a little like this:

The client sends a *client hello* message to the server, which includes the highest protocol version the client supports and a client-generated random number, as well as a list of client-supported cipher suites and compression methods, and finally an empty session identifier field.

Next, the server returns the following three messages to the client:

- An appropriately named *server hello* message that includes the same fields as the client hello, except here the server indicates the values to be used for that session: the protocol version, the cipher suite, the compression method, and the session identifier, as well as a server-generated random number.
- A *certificate* message that contains a list of certificates, including its own and possibly some intermediate certification authority (CA) certificates, which the client will use to authenticate the server.
- A *server hello done* message to indicate that the server is done with key-exchange-related messages and that it's time for the client to start its part of the key-exchange process.

The client then creates a random pre-master secret and encrypts it with the public key from the server's certificate. The client sends the server the following:

- The encrypted pre-master secret in the "client key-exchange" message.
- A *change cipher spec* message to let the server know to start using the newly established sessions for hashing and encryption messages. Technically, this message isn't a handshake message, but rather an independent TLS message type to avoid pipeline stalls.
- A *finished* message, which the client must send immediately after a change cipher spec message. This lets the peer — in this case, the server — verify that the key exchange and authentication processes were successful. It's the concatenation of the master secret, a finished label (either *client finished* or *server finished*), and a hash of all handshake messages up to but not including this one. The resulting concatenated blob is then fed into a pseudo-random function (PRF). Note that the finished message's format has changed over time; where once it was fixed, it's now cipher-suite-specified.

The server next returns a change cipher spec, indicating that it will use the indicated cipher suites henceforth, and a server finished message.

After the client receives and verifies the server's finished message, it can proceed to protect application data with the record protocol.

Options Abound

It should come as no surprise that there are many other ways to implement TLS because it has come to support an extremely broad number of use cases. These different use cases require different messages to support things such as in-band client authentication as well as different types of cipher suites.

If a server wishes to have a client use an in-band mechanism as opposed to an HTTP-based or Web form-based mechanism, it can request that the client authenticate with the server using client certificates. To do this, the server sends a *certificate request* message along with the other messages in the first flight of messages returned after the client hello. The client returns *client certificate* and *certificate verify* messages with its second flight of messages. Hello extensions,[9] which are supported in TLS versions but no SSL version, let the client more easily obtain Online Certificate Status Protocol[10] responses for the server's certificate as well as the intermediate CA certificates returned in the server certificate message. If X.509 certificates aren't your cup of tea, then another option is to negotiate the use of OpenPGP[11] or raw public keys[12] for the client, server, or both.

Different cipher suites — where a cipher suite specifies a key-exchange algorithm, a bulk encryption algorithm (its mode and its secret key length), a MAC algorithm, and a PRF — require different information. If the key-exchange algorithm being negotiated is an anonymous Diffie-Hellman (DH) or ephemeral DH (DHE), then there is no need to send the server or client certificate message, certificate request message, or certificate verify message. Instead, the server sends a *server key-exchange* message to provide enough information for the client to encrypt the premaster secret returned in the client key-exchange message. An option also exists to negotiate preshared key (PSK) suites,[13] which lets clients and servers negotiate

- only symmetric key algorithms, which are often touted as suitable for performance-constrained use cases because public-key operations are "costly";
- a PSK to authenticate a DHE exchange; and
- the use of RSA and the server's certificate to authenticate the server, as well as the use of the PSK for mutual authentication.

The first PSK case is similar to the anonymous DH option because no certificates are needed, and certificate-related messages aren't exchanged on the server and client key-exchange messages. The second PSK case differs from the first in the parameters included in the server and client key-exchange messages. An option is also available to support Secure Remote Password (SRP) cipher suites,[14] which aren't susceptible to offline dictionary attacks, and let application-layer protocols use username/password tuples in-band. The server never asks the client for a certificate. Depending on the SRP cipher suite, the server either sends a certificate message but no certificate-verify message along with the server key-exchange message to the client, or just the server key-exchange message. DH suites can also be negotiated instead of RSA (that is, key agreement versus key transport). Elliptic curve (EC) variants of DH, DHE, and anonymous DH can also be negotiated. When negotiating EC algorithms, the curves as well as the key formats (such as compressed or uncompressed) that the client supports can be provided in the client hello extensions defined in RFC 4492.[15]

Not to be outdone by the number of key-exchange algorithms, a whole bunch of bulk-encryption algorithms are also available. RC4 and the Advanced Encryption Standard are the most widely used; the rest are national algorithms such as GOST, Camellia, SEED, and ARIA. Authenticated Encryption and Authenticated Data (AEAD) algorithms such as Galois Counter Mode and Counter with CBC-MAC are also supported.

In addition to client authentication and cipher-suite-based options, TLS 1.2 (but maybe not TLS 1.3) supports the session resumption. Here, rather than sending an empty session ID in the client hello, the client sends the session ID of the session to be resumed. The server responds with server hello, change cipher spec, and server finished messages. The client then responds with change cipher spec and client finished messages.

Finally, either clients or servers can initiate a renegotiation of the cipher suites. When the server initiates them, it sends a *hello request* message to the client that in turn triggers the client to initiate a new handshake. Clients can also initiate a renegotiation at any time simply by sending a client hello message.

Time to Make the Donuts

TLS likely got its name from the record protocol because it's the bit of the protocol that performs the transport-like functions — namely, it handles fragmenting, transmission, receiving, and defragmenting. Depending on the algorithms negotiated during the handshake, application data can be compressed, MACed, and encrypted before transmission and subsequently received, decompressed, verified, and decrypted.

Wait, You Mean It Doesn't Do That?

The not-so-subtle point from the previous paragraphs is that TLS can be configured to operate as securely as possible or in some horrifically broken way. It's up to administrators to ensure that their servers are configured properly. Also, cryptographic algorithms weaken over time, so it's a good idea to check the server's configuration file to make sure that strong ciphers are preferred over weak ones.

TLS isn't going to automatically update itself if an implementation bug such as the now famous heartbleed bug is discovered. Deploying updated software or disabling insecure options, my friends, is up to server administrators and those that control their budgets.

TLS is a two-party protocol. Clients are only assured of end-to-end security when the server is the TLS session endpoint, and the application data resides on said server. If the data resides elsewhere, then even with TLS sessions between the servers, the data isn't truly protected from end to end. There have also been numerous attempts to introduce another party into the protocol. What they're called depends on your viewpoint: proxies or men in the middle.

TLS relies on a reliable transport protocol — mostly TCP. Datagram Transport Layer Security (DTLS)[16] is used with unreliable transports such as UDP.

Some have said TLS's time has come and gone, and that it's time to move to an entirely new protocol. Not everyone agrees with this sentiment. The TLS WG is actively working on TLS 1.3 with the following goals in mind:

- Encrypt as much of the handshake as possible to reduce the amount of observable data to both passive and active attackers.
- Reduce handshake latency to primarily support HTTP-based applications with the aim of one roundtrip (1-RTT) for a full handshake and one or zero roundtrips (0-RTT) for repeated handshakes.
- Update payload protection cryptographic mechanisms and algorithms to address known weaknesses in the CBC block cipher modes and to replace RC4.
- Reevaluate handshake content.

Some of the major decisions made to date would do the following:

- Remove support for compression. The application handles this better, and it's been the source of numerous well-known attacks (such as Beast and Crime).
- Remove support for static RSA and DH static key exchange. These types of ciphers don't support perfect forward secrecy (PFS).
- Remove support for non-AEAD ciphers.
- Move EC algorithms to the standards track. Still to be decided is whether an EC-based algorithm will be the mandatory-to-implement algorithm.

Work is ongoing, so if you're interested and have some constructive criticism, please come and get involved (see http://datatracker.ietf.org/wg/tls/documents).

References

1. J. Postel and J. Reynolds, *Telnet Protocol Specification*, IETF RFC 854, May 1983; https://datatracker.ietf.org/doc/rfc854/.
2. J. Postel and J. Reynolds, *File Transfer Protocol (FTP)*, IETF RFC 959, Oct. 1995; http://datatracker.ietf.org/doc/rfc959/.
3. F. Anklesaria et al., *The Internet Gopher Protocol: A Distributed Document Search and Retrieval Protocol*, IETF RFC 1436, Mar. 1993; https://datatracker.ietf.org/doc/rfc1436/.
4. S. Kent and K. Seo, *Security Architecture for the Internet Protocol*, IETF RFC 4301, Dec. 2005; https://datatracker.ietf.org/doc/rfc4301/.
5. J. Benaloh et al., *The Private Communication Technology Protocol*, IETF Internet draft, work in progress, Oct. 1995.
6. K. Hickman, *The SSL Protocol*, IETF Internet draft, work in progress, Apr. 1995.
7. S. Turner and T. Polk, *Prohibiting Secure Sockets Layer (SSL) Version 2.0*, IETF RFC 6176, Mar. 2011; https://datatracker.ietf.org/doc/rfc6176/.
8. A. Freier, P. Karlton, and P. Cocher, *The Secure Sockets Layer (SSL) Protocol Version 3.0*, IETF RFC 6101, Aug. 2011; https://datatracker.ietf.org/doc/rfc6101/.
9. D. Eastlake, *Transport Layer Security (TLS) Extensions: Extension Definitions*, IETF RFC 6066, Jan. 2011; https://datatracker.ietf.org/doc/rfc6066/.
10. Y. Pettersen, *The Transport Layer Security (TLS) Multiple Certificate Status Request Extension*, IETF RFC 6961, June 2013; https://datatracker.ietf.org/doc/rfc6961/.
11. N. Mavrogiannopoulos and D.K Gillmor, *Using OpenPGP Keys for Transport Layer Security (TLS) Authentication*, IETF RFC 6091, Feb. 2011; https://datatracker.ietf.org/doc/rfc6091/.
12. P. Wouters et al., *Using Raw Public Keys in Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)*, IETF RFC 7250, June 2014; https://datatracker.ietf.org/doc/rfc7250/.
13. P. Eronen and H. Tschofenig, *Pre-Shared Key Ciphersuites for Transport Layer Security (TLS)*, IETF RFC 4279, Dec. 2005; https://datatracker.ietf.org/doc/rfc4279/.
14. D. Taylor et al., *Using the Secure Remote Password (SRP) Protocol for TLS Authentication*, IETF RFC 5054; Nov. 2007; https://datatracker.ietf.org/doc/rfc5054/.
15. S. Blake-Wilson et al., *Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS)*, IETF RFC 4492, May 2006; https://datatracker.ietf.org/doc/rfc4492/.
16. E. Rescorla and N. Modadugu, *Datagram Transport Layer Security Version 1.2*, IETF RFC 6347, Jan. 2012; https://datatracker.ietf.org/doc/rfc6347/.

Sean Turner is founder and principal engineer at IECA. He received a BEE from the Georgia Institute of Technology. Turner is a long-time member of IEEE, active in the IETF, and coauthor of *Implementing Email and Security Tokens: Current Standards, Tools, and Practices* (Wiley, 2008). Contact him at turners@ieca.com.