EXHIBIT 4

Transport Layer Security

Transport Layer Security (**TLS**), and its now-deprecated predecessor, **Secure Sockets Layer** (**SSL**),[1] are cryptographic protocols designed to provide communications security over a computer network.[2] Several versions of the protocols find widespread use in applications such as web browsing, email, instant messaging, and voice over IP (VoIP). Websites can use TLS to secure all communications between their servers and web browsers.

The TLS protocol aims primarily to provide privacy and data integrity between two or more communicating computer applications.[2]:3 When secured by TLS, connections between a client (e.g., a web browser) and a server (e.g., wikipedia.org) should have one or more of the following properties:

- The connection is *private* (or *secure*) because symmetric cryptography is used to encrypt the data transmitted. The keys for this symmetric encryption are generated uniquely for each connection and are based on a shared secret that was negotiated at the start of the session (see § TLS handshake). The server and client negotiate the details of which encryption algorithm and cryptographic keys to use before the first byte of data is transmitted (see § Algorithm below). The negotiation of a shared secret is both secure (the negotiated secret is unavailable to eavesdroppers and cannot be obtained, even by an attacker who places themselves in the middle of the connection) and reliable (no attacker can modify the communications during the negotiation without being detected).
- The identity of the communicating parties can be *authenticated* using public-key cryptography. This authentication can be made optional, but is generally required for at least one of the parties (typically the server).
- The connection is *reliable* because each message transmitted includes a message integrity check using a message authentication code to prevent undetected loss or alteration of the data during transmission.[2]:3

In addition to the properties above, careful configuration of TLS can provide additional privacy-related properties such as forward secrecy, ensuring that any future disclosure of encryption keys cannot be used to decrypt any TLS communications recorded in the past.[3]

TLS supports many different methods for exchanging keys, encrypting data, and authenticating message integrity (see § Algorithm below). As a result, secure configuration of TLS involves many configurable parameters, and not all choices provide all of the privacy-related properties described in the list above (see the § Key exchange (authentication), § Cipher security, and § Data integrity tables).

Attempts have been made to subvert aspects of the communications security that TLS seeks to provide, and the protocol has been revised several times to address these security threats (see § Security). Developers of web browsers have also revised their products to defend against potential security weaknesses after these were discovered (see TLS/SSL support history of web browsers).[4]

The TLS protocol comprises two layers: the TLS record and the TLS handshake protocols.

TLS is a proposed Internet Engineering Task Force (IETF) standard, first defined in 1999, and the current version is TLS 1.3 defined in RFC 8446 (August 2018). TLS builds on the earlier SSL specifications (1994, 1995, 1996) developed by Netscape Communications[5] for adding the HTTPS protocol to their Navigator web browser.

Contents

Since applications can communicate either with or without TLS (or SSL), it is necessary for the client to indicate to the server the setup of a TLS connection.[6] One of the main ways of achieving this is to use a different port number for TLS connections, for example port 443 for HTTPS. Another mechanism is for the client to make a protocol-specific request to the server to switch the connection to TLS; for example, by making a STARTTLS request when using the mail and news protocols.

Once the client and server have agreed to use TLS, they negotiate a stateful connection by using a handshaking procedure.[7] The protocols use a handshake with an asymmetric cipher to establish not only cipher settings but also a session-specific shared key with which further communication is encrypted using a symmetric cipher. During this handshake, the client and server agree on various parameters used to establish the connection's security:

- The handshake begins when a client connects to a TLS-enabled server requesting a secure connection and the client presents a list of supported cipher suites (ciphers and hash functions).
- From this list, the server picks a cipher and hash function that it also supports and notifies the client of the decision.
- The server usually then provides identification in the form of a digital certificate. The certificate contains the server name, the trusted certificate authority (CA) that vouches for the authenticity of the certificate, and the server's public encryption key.
- The client confirms the validity of the certificate before proceeding.
- To generate the session keys used for the secure connection, the client either:
  - encrypts a random number with the server's public key and sends the result to the server (which only the server should be able to decrypt with its private key); both parties then use the random number to generate a unique session key for subsequent encryption and decryption of data during the session
  - uses Diffie–Hellman key exchange to securely generate a random and unique session key for encryption and decryption that has the additional property of forward secrecy: if the server's private key is disclosed in future, it cannot be used to decrypt the current session, even if the session is intercepted and recorded by a third party.

This concludes the handshake and begins the secured connection, which is encrypted and decrypted with the session key until the connection closes. If any one of the above steps fails, then the TLS handshake fails and the connection is not created.

TLS and SSL do not fit neatly into any single layer of the OSI model or the TCP/IP model.[8][9] TLS runs "on top of some reliable transport protocol (e.g., TCP),"[10] which would imply that it is above the transport layer. It serves encryption to higher layers, which is normally the function of the presentation layer. However, applications generally use TLS as if it were a transport layer,[8][9] even though applications using TLS must actively control initiating TLS handshakes and handling of exchanged authentication certificates.[10]

History and development

SSL and TLS protocols

Protocol	Published	Status
SSL 1.0	Unpublished	Unpublished
SSL 2.0	1995	Deprecated in 2011 (RFC 6176)
SSL 3.0	1996	Deprecated in 2015 (RFC 7568)
TLS 1.0	1999	Deprecation planned in 2020[11]
TLS 1.1	2006	Deprecation planned in 2020[11]
TLS 1.2	2008	
TLS 1.3	2018	

Secure Network Programming

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 pre-existing network applications with security measures.[12]

SSL 1.0, 2.0, and 3.0

Netscape developed the original SSL protocols.[13][14] Version 1.0 was never publicly released because of serious security flaws in the protocol; version 2.0, released in February 1995, contained a number of security flaws which necessitated the design of version 3.0.[15][13] Released in 1996, SSL version 3.0 represented a complete redesign of the protocol produced by Paul Kocher working with Netscape engineers Phil Karlton and Alan Freier, with a reference implementation by Christopher Allen and Tim Dierks of Consensus Development. 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.

Taher Elgamal, chief scientist at Netscape Communications from 1995 to 1998, has been described as the "father of SSL".[16][17]

- SSL 2.0 was deprecated in 2011 by RFC 6176.

In 2014, SSL 3.0 was found to be vulnerable to the POODLE attack that affects all block ciphers in SSL; RC4, the only non-block cipher supported by SSL 3.0, is also feasibly broken as used in SSL 3.0.[18]

- SSL 3.0 was deprecated in June 2015 by RFC 7568.

TLS 1.0

TLS 1.0 was first defined in RFC 2246 in January 1999 as an upgrade of SSL Version 3.0, and written by Christopher Allen and Tim Dierks of Consensus Development. As stated in the RFC, "the differences between this protocol and SSL 3.0 are not dramatic, but they are significant enough 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.[19]:1–2

The PCI Council suggested that organizations migrate from TLS 1.0 to TLS 1.1 or higher before June 30, 2018.[20][21] In October 2018, Apple, Google, Microsoft, and Mozilla jointly announced they would deprecate TLS 1.0 and 1.1 in March 2020.[11]

TLS 1.1

TLS 1.1 was defined in RFC 4346 in April 2006.[22] 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.[19]:2

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

- TLS Extensions definition and AES cipher suites were added.[19]:2

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

TLS 1.3

TLS 1.3 was defined in RFC 8446 in August 2018. It is based on the earlier TLS 1.2 specification. Major differences from TLS 1.2 include:

- Separating key agreement and authentication algorithms from the cipher suites
- Removing support for weak and lesser-used named elliptic curves
- Removing support for MD5 and SHA-224 cryptographic hash functions
- Requiring digital signatures even when a previous configuration is used
- Integrating HKDF and the semi-ephemeral DH proposal
- Replacing resumption with PSK and tickets
- Supporting 1-RTT handshakes and initial support for 0-RTT
- Mandating perfect forward secrecy, by means of using ephemeral keys during the (EC)DH key agreement
- Dropping support for many insecure or obsolete features including compression, renegotiation, non-AEAD ciphers, non-PFS key exchange (among which are static RSA and static DH key exchanges), custom DHE groups, EC point format negotiation, Change Cipher Spec protocol, Hello message UNIX time, and the length field AD input to AEAD ciphers
- Prohibiting SSL or RC4 negotiation for backwards compatibility
- Integrating use of session hash
- Deprecating use of the record layer version number and freezing the number for improved backwards compatibility
- Moving some security-related algorithm details from an appendix to the specification and relegating ClientKeyShare to an appendix
- Adding the ChaCha20 stream cipher with the Poly1305 message authentication code
- Adding the Ed25519 and Ed448 digital signature algorithms
- Adding the x25519 and x448 key exchange protocols

Network Security Services (NSS), the cryptography library developed by Mozilla and used by its web browser Firefox, enabled TLS 1.3 by default in February 2017.[24] TLS 1.3 was added to Firefox 52.0, which was released in March 2017, but it was disabled by default due to compatibility issues for some users.[25] It has been enabled by default since Firefox 60.0.[26]

Google Chrome set TLS 1.3 as the default version for a short time in 2017. It then removed it as the default, due to incompatible middleboxes such as Blue Coat web proxies.[27]

Pale Moon enabled the use of TLS 1.3 as of version 27.4, released in July 2017.[28] During the IETF 100 Hackathon which took place in Singapore, The TLS Group worked on adapting open-source applications to use TLS 1.3.[29][30] The TLS group was made up of individuals from Japan, United Kingdom, and Mauritius via the cyberstorm.mu team.[30] During the IETF 101 Hackathon which took place in London, more work was done on application support of TLS 1.3.[31] During IETF 102 Hackathon, work continued to inter-operate lesser known TLS 1.3 implementations along with application integration.[32]

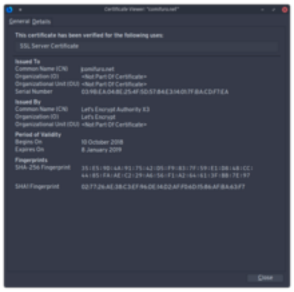
wolfSSL enabled the use of TLS 1.3 as of version 3.11.1, released in May 2017.[33] As the first commercial TLS 1.3 implementation, wolfSSL 3.11.1 supported Draft 18 and now supports Draft 28,[34] the final version, as well as many older versions. A series of blogs was published on the performance difference between TLS 1.2 and 1.3.[35]

In September 2018, the popular OpenSSL project released version 1.1.1 of its library, in which support for TLS 1.3 was "[t]he headline new feature".[36]

The Electronic Frontier Foundation praises TLS 1.3 and warns about "a look-alike protocol brewing called ETS (or eTLS) that intentionally disables important security measures in TLS 1.3".[37]

Digital certificates

A digital certificate certifies the ownership of a public key by the named subject of the certificate, and indicates certain expected usages of that key. This allows others (relying parties) to rely upon signatures or on assertions made by the private key that corresponds to the certified public key.



Example of a website with digital certificate

Certificate authorities

TLS typically relies on a set of trusted third-party certificate authorities to establish the authenticity of certificates. Trust is usually anchored in a list of certificates distributed with user agent software,[38] and can be modified by the relying party.

According to Netcraft, who monitors active TLS certificates, the market-leading certificate authority (CA) has been Symantec since the beginning of their survey (or VeriSign before the authentication services business unit was purchased by Symantec). Symantec currently accounts for just under a third of all certificates and 44% of the valid certificates used by the 1 million busiest websites, as counted by Netcraft.[39]

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 convenient 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) if the certificate authority cooperates (or is compromised).[40][41]

Algorithm

Key exchange or key agreement

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 (TLS_DH), ephemeral Diffie–Hellman (TLS_DHE), elliptic-curve Diffie–Hellman (TLS_ECDH), ephemeral elliptic-curve Diffie–Hellman (TLS_ECDHE), anonymous Diffie–Hellman (TLS_DH_anon),[2] pre-shared key (TLS_PSK)[42] and Secure Remote Password (TLS_SRP).[43]

The TLS_DH_anon and TLS_ECDH_anon key agreement methods do not authenticate the server or the user and hence are rarely used because those are vulnerable to man-in-the-middle attacks. 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 because the

Algorithm	SSL 2.0	SSL 3.0	TLS 1.0	TLS 1.1	TLS 1.2	TLS 1.3	Status
RSA	Yes	Yes	Yes	Yes	Yes	No	Defined for TLS 1.2 in RFCs
DH-RSA	No	Yes	Yes	Yes	Yes	No	
DHE-RSA (forward secrecy)	No	Yes	Yes	Yes	Yes	Yes	
ECDH-RSA	No	No	Yes	Yes	Yes	No	
ECDHE-RSA (forward secrecy)	No	No	Yes	Yes	Yes	Yes	
DH-DSS	No	Yes	Yes	Yes	Yes	No	
DHE-DSS (forward secrecy)	No	Yes	Yes	Yes	Yes	No[45]	
ECDH-ECDSA	No	No	Yes	Yes	Yes	No	
ECDHE-ECDSA (forward secrecy)	No	No	Yes	Yes	Yes	Yes	
PSK	No	No	Yes	Yes	Yes		
PSK-RSA	No	No	Yes	Yes	Yes		
DHE-PSK (forward secrecy)	No	No	Yes	Yes	Yes		
ECDHE-PSK (forward secrecy)	No	No	Yes	Yes	Yes		
SRP	No	No	Yes	Yes	Yes		
SRP-DSS	No	No	Yes	Yes	Yes		
SRP-RSA	No	No	Yes	Yes	Yes		
Kerberos	No	No	Yes	Yes	Yes		
DH-ANON (insecure)	No	Yes	Yes	Yes	Yes		
ECDH-ANON (insecure)	No	No	Yes	Yes	Yes		
GOST R 34.10-94 / 34.10-2001[46]	No	No	Yes	Yes	Yes		Proposed in RFC drafts

Cipher

Cipher security against publicly known feasible attacks

Cipher			Protocol version						Status
Type	Algorithm	Nominal strength (bits)	SSL 2.0	SSL 3.0 [n 1][n 2][n 3][n 4]	TLS 1.0 [n 1][n 3]	TLS 1.1 [n 1]	TLS 1.2 [n 1]	TLS 1.3	
Block cipher with mode of operation	AES GCM[47][n 5]	256, 128	N/A	N/A	N/A	N/A	Secure	Secure	Defined for TLS 1.2 in RFCs
	AES CCM[48][n 5]		N/A	N/A	N/A	N/A	Secure	Secure	
	AES CBC[n 6]		N/A	N/A	Depends on mitigations	Depends on mitigations	Depends on mitigations	N/A	
	Camellia GCM[49][n 5]	256, 128	N/A	N/A	N/A	N/A	Secure	N/A	
	Camellia CBC[50][n 6]		N/A	N/A	Depends on mitigations	Depends on mitigations	Depends on mitigations	N/A	
	ARIA GCM[51][n 5]	256, 128	N/A	N/A	N/A	N/A	Secure	N/A	
	ARIA CBC[51][n 6]		N/A	N/A	Depends on mitigations	Depends on mitigations	Depends on mitigations	N/A	
	SEED CBC[52][n 6]	128	N/A	N/A	Depends on mitigations	Depends on mitigations	Depends on mitigations	N/A	
	3DES EDE CBC[n 6][n 7]	112[n 8]	Insecure	Insecure	Insecure	Insecure	Insecure	N/A	
	GOST 28147-89 CNT[46][n 7]	256	N/A	N/A	Insecure	Insecure	Insecure	N/A	Defined in RFC 4357
	IDEA CBC[n 6][n 7][n 9]	128	Insecure	Insecure	Insecure	Insecure	N/A	N/A	Removed from TLS 1.2
	DES CBC[n 6][n 7][n 9]	56	Insecure	Insecure	Insecure	Insecure	N/A	N/A	
		40[n 10]	Insecure	Insecure	Insecure	N/A	N/A	N/A	Forbidden in TLS 1.1 and later
	RC2 CBC[n 6][n 7]	40[n 10]	Insecure	Insecure	Insecure	N/A	N/A	N/A	
Stream cipher	ChaCha20-Poly1305[57][n 5]	256	N/A	N/A	N/A	N/A	Secure	Secure	Defined for TLS 1.2 in RFCs
	RC4[n 11]	128	Insecure	Insecure	Insecure	Insecure	Insecure	N/A	Prohibited in all versions of TLS by RFC 7465
		40[n 10]	Insecure	Insecure	Insecure	N/A	N/A	N/A	
None	Null[n 12]	–	N/A	Insecure	Insecure	Insecure	Insecure	N/A	Defined for TLS 1.2 in RFCs

Notes

1. RFC 5746 (https://tools.ietf.org/html/rfc5746) must be implemented to fix a renegotiation flaw that would otherwise break this protocol.
2. If libraries implement fixes listed in RFC 5746 (https://tools.ietf.org/html/rfc5746) this violates the SSL 3.0 specification, which the IETF cannot change unlike TLS. 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
4. The POODLE attack breaks all block ciphers (CBC ciphers) used in SSL 3.0 unless mitigated by the client and/or the server. See § Web browsers.
5. AEAD ciphers (such as GCM and CCM) can be used in only TLS 1.2.
6. CBC ciphers can be attacked with the Lucky Thirteen attack if the library is not written carefully to eliminate timing side channels.
7. The Sweet32 attack breaks block ciphers with a block size of 64 bits.[53]

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS

Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS

Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS

Sync your system to PACER to automate legal marketing.

WHAT WILL YOU BUILD? | **sales@docketalarm.com** | **1-866-77-FASTCASE**

fastcase
Smarter legal research.