Cryptographic Engineering

working draft

Matthew D. Green
Johns Hopkins University

Excerpt: Chapter 9, Cryptographic protocols

§9.2.1 Transport Layer Security (TLS) (book pages 181–186)

Posted for 601.445/601.645 Practical Cryptographic Systems, Fall 2026
Lecture 15, Wed Oct 21: TLS design and PKI — required

`https://practicalcrypto.org`

This is an excerpt from an unfinished book draft: expect gaps, rough edges, and errors, and note that section numbers and page numbers may change in later drafts. Cross-references to other chapters point to material that is not included here. Copyright © 2026 Matthew D. Green. All rights reserved. Posted for the students of the course above; **please do not redistribute** — link to the course site instead. Built 2026-10-19.

9.2.1 Transport Layer Security (TLS)

With these considerations in mind, a good place to begin any discussion of network encryption is with the Transport Layer Security protocol. TLS is, by far, the most widely-used encryption protocol on the Internet. It is also one of the most mature, having evolved over nearly thirty years.

TLS is used to secure most of the session-oriented (TCP) communications on the Internet. Most notably, it is the encryption protocol that powers HTTPS, which is used for most web connections, and for the underlying API connections that many apps use to communicate with servers. It is also used for secure email transport between SMTP mail servers, as well as the IMAP and POP3 protocols used by most email clients. A variant called *Datagram TLS* (DTLS) also exists to support datagram-oriented (UDP) connections, though it's not used as widely.

A brief history: from SSL to TLS. The protocol we know as TLS was originally developed within Netscape Communications back around 1995, and was named Secure Sockets Layer (SSL). The first version of SSL must have been terrible, since the protocol was not made public until SSLv2 the following year. A deluge of vulnerabilities led to a further revision in SSLv3 [320]. In 1999, the protocol was standardized by the Internet Engineering Task Force under the name Transport Layer Security v1.0 [14].[15] TLS v1.1 was published in 2006, and TLS v1.2 in 2008. The continued discovery of major flaws built momentum for a substantial protocol redesign in TLS v1.3 that completed in 2018 [274].

A serious problem with TLS 1.2 and earlier versions is that, in the words of designer Eric Rescorla, these versions make use of "pre-historic cryptography." Much of this is due to backwards compatibility requirements with 1990s implementations of the protocol, which were designed before many subtle cryptographic attacks were well-understood. In the following sections we will focus mainly on TLS 1.3.

DRAFT

What happens inside a TLS connection? From the perspective of an application, TLS is designed to be a (near) drop-in replacement for a standard TCP socket. In this setting, a *client* application opens a connection by specifying a domain name and port for a *server*, and the two then send and receive arbitrary bytes of data as in a normal TCP connection. Under the hood, a typical TLS 1.3 connection proceeds through the following stages:

Negotiation. In the first stage of a TLS protocol, the client and server agree on parameters for the connection. This includes the specific version of TLS to use, as well as the encryption algorithms to use within it. To enable *cryptographic agility*, TLS is designed to support multiple different cryptographic primitives: this allows servers to

[15] Allegedly Microsoft refused to come on board unless the protocol was re-named, since they felt that SSL was a Netscape-specific brand.

rapidly upgrade their security when a flaw is found in some primitive, or when operators have specific requirements.[16] In TLS 1.3, the signature scheme, key exchange protocol, and symmetric encryption parameters are negotiated separately. The symmetric parameters are specified together as a *ciphersuite* containing a hash function for use in key derivation (KDF) and an authenticated symmetric encryption scheme, *e.g.,* `TLS_CHACHA20_POLY1305_SHA256`.[17]

Key exchange. In most TLS connections, the client and server next run a key exchange to agree on a shared master key. Most TLS 1.3 connections use the elliptic curve or finite-field Diffie-Hellman algorithms, although several new extensions support "hybrid" key exchanges that fold in post-quantum key exchange algorithms such as ML-KEM [299]. To ensure that the key exchange is authentic, one or both sides will sign their messages using a digital signature scheme. Verifying the authenticity of the signature typically relies on obtaining the other party's certificate, which will also be exchanged at this time.

Record encryption. Once the two sides have successfully agreed on a master key for the session, this key is "pluralized" using a KDF into a set of encryption keys, including at least one for data flowing in each direction. Data transmitted by either side is next broken into chunks called *records* that are optionally padded, then encrypted and sent onwards via the TCP protocol. TLS 1.3 only supports authenticated encryption schemes for its record encryption, which ensures that the data cannot be tampered with.[18] Record encryption generally continues until the connection terminates, although in some cases TLS connections can *re-key* when the connection state changes or too much data has been sent.

DRAFT

We will now focus on specific portions of this protocol.

The TLS handshake. The negotiation and key exchange phases of a TLS connection are called the *handshake*. Because each round-trip between client and server induces latency in connection establishment, TLS tries to group these messages together so they use the smallest number of communication rounds. In practice this sometimes requires "optimistic" behavior on the part of the client: for example, a client might send key exchange messages for one or more algorithms even before the server has explicitly chosen the key exchange algorithm it will use. A typical handshake looks like this:

[16] A common example is the ability to support post-quantum cryptography algorithms. Some servers also deploy FIPS-approved ciphersuites to facilitate compliance with government rules.

[17] In versions of TLS prior to 1.3, a ciphersuite contained a list of *all* algorithms to be used: key exchange, digital signature, hash/MAC, and symmetric encryption algorithm. This was broken up in TLS 1.3, and ciphersuites now identify only the symmetric algorithm and KDF.

[18] Previous versions of TLS supported an explicit MAC on each TLS record, which was often separate from the encryption algorithm used. Poor combinations of encryption, MAC and padding led to many vulnerabilities.

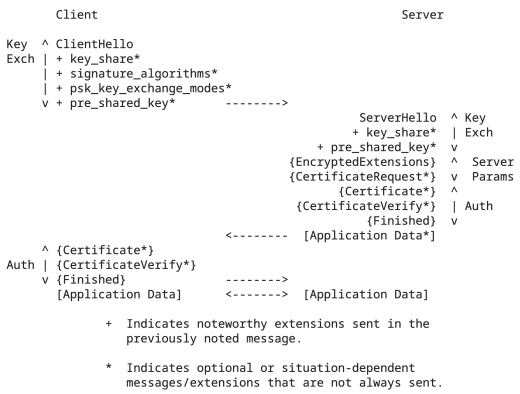

Figure 9.4: Example of a basic TLS 1.3 handshake. Source: RFC 8446 [274].

DRAFT

1. The client sends a ClientHello message to the server. This specifies the version of TLS the client is using, a random nonce ("client random"), a session ID, as well as a list of algorithms and encryption schemes that the client supports. In TLS 1.3, clients will typically attach one or more KeyShare values: these contain the first message in a key exchange algorithm such as Diffie-Hellman or elliptic-curve Diffie-Hellman.

   For backwards compatibility reasons, TLS 1.3 ClientHello messages are disguised in the format of a TLS 1.2 message, which means that servers that only support older versions of the protocol will be able to respond to them.

2. The server then responds with a ServerHello that contains a random nonce ("server random"), as well as the server's KeyShare for the key exchange scheme, and the server's Certificate. Most critically, the server transmits a CertificateVerify message that contains a digital signature computed over the hash of the handshake messages it has sent and received thus far. If the server has been able to complete the key exchange algorithm successfully, it also derives a MAC key and attaches a Finished message that contains a MAC of the handshake transcript so far.[19]

   If all has gone well, the server can now compute a set of symmetric encryption keys to use for communication with the client, and can begin sending encrypted application data at this time.

3. To complete the protocol, the client verifies all of the data received from the server, including, crucially, the certificate and CertificateVerify signature, which ensures that the handshake messages it received are authentic. If all goes well, it computes a set

[19]The main purpose of the Finished message is to ensure that the two parties receive the same key at the end of the protocol. However it also provides a belt-and-suspenders protection that (combined with the CertificateVerify signature) ensures that nobody has tampered with handshake messages.

of shared symmetric keys from the key exchange, then sends its own Finished message with a MAC of the handshake messages. From this point onwards the client can send encrypted application data to the server.

This simple description leaves out many optional messages and extensions. For example, in the example above, only the server provides a certificate to prove its authenticity: clients are assumed to be unauthenticated. If a server desires mutual authentication (mTLS), it can attach a CertificateRequest message requesting that the client transmit its own Certificate and CertificateVerify message. It is also possible that the KeyShare message sent by the client will not match any algorithm accepted by the server, and the server will need to send a HelloRetryRequest that restarts the handshake. The numerous options and complexity of the TLS handshake are part of the reason that TLS libraries have produced so many implementation vulnerabilities.

Key exchange and derivation. Most key exchange in TLS v1.3 is conducted using elliptic curve Diffie-Hellman (ECDH) using a small set of standard elliptic curves, although finite-field Diffie-Hellman is also supported over one of several pre-selected fields. Some IETF drafts have proposed extensions for adding post-quantum cryptographic (PQC) primitives such as ML-KEM, usually in a "hybrid" mode that combines the output of the PQC key agreement with the result of an ECDH handshake [299].[20]

Once a shared secret has been established by the key exchange, each side passes this value into a key derivation function (KDF) as illustrated in Figure 9.5, and the result is further pluralized into a set of distinct encryption and MAC keys for various purposes.

Signatures, certificates and revocation. TLS uses the X.509 format for all certificates. In TLS v1.3 the server's long term keypairs are used only for digital signatures: however, many servers also use the same certificate to support older versions of TLS in which keys can be used for encryption as well. The most popular signature algorithms for TLS are RSA and ECDSA. For legacy reasons, RSA signatures are very widely used, while ECDSA usage is slowly increasing.

An important consideration when using TLS is that verifying a server's certificate may require a full *certificate chain*, which could comprise several certificates. This can add space to the TLS handshake. Additionally, clients are required to verify that certificates have not been revoked. This is typically done using a separate protocol called the Online Certificate Status Protocol (OCSP). Some servers optimize this process by "stapling" a short-term validity ticket to their signatures, using a process called OCSP stapling. See §7.3.6 for many more details.

Session resumption. Since public-key operations are expensive, most versions of TLS/SSL contain a provision for *resumption* of past sessions. Here we will assume that a

[20] As of this writing, Google has activated support for this mode in Chromium browsers [7].

DRAFT

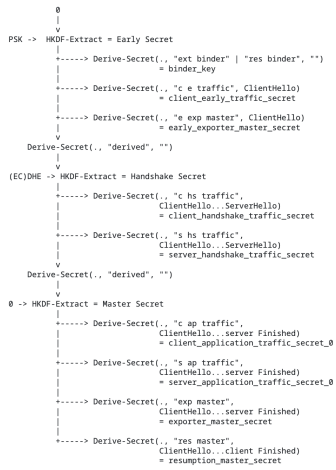

Figure 9.5: Keys derived within the TLS 1.3 handshake (right side). The PSK alone can be used to derive several "early" secrets. The (EC)DHE handshake is combined with the early secret to produce a handshake secret, that is further derived into keys for encrypting portions of the handshake. All of these results are used to produce the application traffic secrets, that are used to encrypt record data. Source: RFC 8446 [274].

DRAFT

normal session has been established in the recent past, and that the resulting key material is still available to both client and server. In this case, the pair can execute a more efficient handshake that simply updates the session keys without the need for a full key exchange.

TLS 1.3 enables this via a long-term secret known as a "pre-shared key" (PSK). In principle, PSKs are just arbitrary symmetric secrets that can be established via some out-of-band process, and labeled with some name known to client and server. In practice, most TLS 1.3 implementations use PSKs mainly for session resumption: at the conclusion of each successful handshake, the client and server will compute a resumption_master_secret (see Figure 9.5), and will store this as a named PSK. When resuming a session, the client can transmit a Pre-Shared key message that contains the identifier of a specific PSK value. Most clients will also send a Key Share message as a backup, just in case the server does not find the matching PSK.

A disadvantage of session resumption and PSKs is that, used alone, they do not provide any forward secrecy: that is, an attacker who compromises a PSK will be able to decrypt past connections. A server can optionally combine a PSK with a fresh key exchange, although this reduces the efficiency advantage of session resumption.

0-RTT and "early data." Notice that the standard TLS handshake does not provide the client with encryption keys until after it receives the server's Server Hello message. This

necessitates a full round-trip between client and server before the client can, for example, send a simple HTTP GET request. For machines located on different continents, this can add up to half a second of latency to session establishment: long enough that even human beings might notice! Many teeth have been gnashed over this problem, mainly by engineers at Google.[21]

To improve responsiveness, TLS features a 0-RTT mode. This mode is simply an extension of session resumption, and so it requires that the client and server have previously conducted a full round-trip handshake and established a PSK. If this is the case, then the client can immediately transmit application data called "early data" along with the initial Client Hello message. A first caveat is that this "early data" does not receive the benefit of forward secrecy. A second caveat is that this early data can be subject to *replay attacks* in which an attacker records and re-transmits the client's first message to the same server. If replay attacks are a concern (something that varies between applications), servers can prevent them by waiting until the client sends a valid Finished message before processing the application data, or they can keep track of ClientHello messages and reject repeated messages.[22]

Session tickets. A disadvantage of the TLS handshake is that the server must store some amount of state for sessions that may resume in the future (in TLS 1.3, this is equivalent to the PSK value.) This can overwhelm servers that receive millions of connection requests, and it can cause challenges when many different servers handle connections for the same endpoint. One way to address these problems is to use *session tickets*: in this approach, the server encrypts any relevant session state under a single master secret key that it stores locally (or shares with other servers), and sends the encrypted "ticket" to the client for storage. The client can send back the encrypted ticket the next time it connects. The main concern with session tickets is that an adversarial client might attempt to tamper with a session ticket, or replay an older/expired one. Servers defend against this by using authenticated encryption and storing an relatively short-term expiration date within the ticket.

Record encryption. The purpose of the TLS handshake is to establish a shared symmetric encryption key that both sides can use to encrypt application data. The handshake key derivation procedure (Figure 9.5) produces at least two such keys: one for data sent by the client, and a second one for data sent by the server.[23]

Both handshake messages and application data are packed into *records*, which are a simple data structure that contains up to 2^{14} bytes of payload. Each record contains fields

[21] The results of this tooth-gnashing include specialized protocols such as QUIC [176].

[22] Some other defenses are discussed in the TLS specification [274, §8]. These defenses are particularly important in applications where replay attacks can cause persistent changes to the state of a server-based application. They can be challenging to implement in applications where multiple servers sit behind a load balancer: here the servers may need to coordinate in order to avoid replays.

[23] Separate keys are derived for "early data" and normal application traffic.

DRAFT