Transport Layer Security (TLS)

NETWORKING 101, CHAPTER 4

The SSL protocol was originally developed at Netscape to enable ecommerce transaction security on the Web, which required encryption to protect customers' personal data, as well as authentication and integrity guarantees to ensure a safe transaction. To achieve this, the SSL protocol was implemented at the application layer, directly on top of TCP (Figure 4-1), enabling protocols above it (HTTP, email, instant messaging, and many others) to operate unchanged while providing communication security when communicating across the network.

When SSL is used correctly, a third-party observer can only infer the connection endpoints, type of encryption, as well as the frequency and an approximate amount of data sent, but cannot read or modify any of the actual data.

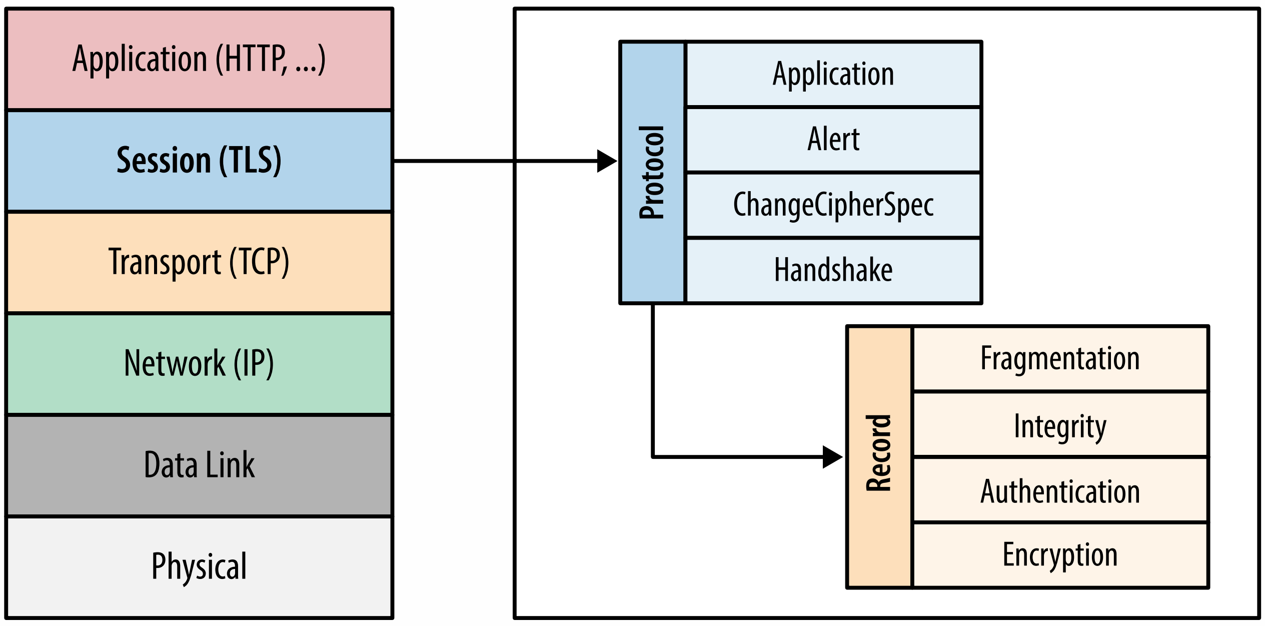


Figure 4-1. Transport Layer Security (TLS)

Note

When the SSL protocol was standardized by the IETF, it was renamed to Transport Layer Security (TLS). Many use the TLS and SSL names interchangeably, but technically, they are different, since each describes a different version of the protocol.

SSL 2.0 was the first publicly released version of the protocol, but it was quickly replaced by SSL 3.0 due to a number of discovered security flaws. Because the SSL protocol was proprietary to published in January 1999 and became known as TLS 1.0. Since then, the IETF has continued iterating on the protocol to address security flaws, as well as to extend its capabilities: TLS 1.1 (RFC 4346) was published in April 2006, TLS 1.2 (RFC 5246) in August 2008, and work is now underway to define TLS 1.3.

That said, don't let the abundance of versions numbers mislead you: your servers should always prefer and negotiate the latest stable version of the TLS protocol to ensure the best security, capability, and performance guarantees. In fact, some performance-critical features, such as HTTP/2, explicitly require the use of TLS 1.2 or higher and will abort the connection otherwise. Good security and performance go hand in hand.

Note

TLS was designed to operate on top of a reliable transport protocol such as TCP. However, it has also been adapted to run over datagram protocols such as UDP. The Datagram Transport Layer Security (DTLS) protocol, defined in RFC 6347, is based on the TLS protocol and is able to provide similar security guarantees while preserving the datagram delivery model.

Encryption, Authentication, and Integrity §

The TLS protocol is designed to provide three essential services to all applications running above it: encryption, authentication, and data integrity. Technically, you are not required to use all three in every situation. You may decide to accept a certificate without validating its authenticity, but you should be well aware of the security risks and implications of doing so. In practice, a secure web application will leverage all three services.

Encryption

A mechanism to obfuscate what is sent from one host to another.

Authentication

A mechanism to verify the validity of provided identification material.

Integrity

A mechanism to detect message tampering and forgery.

In order to establish a cryptographically secure data channel, the connection peers must agree on which ciphersuites will be used and the keys used to encrypt the data. The TLS protocol specifies a well-defined handshake sequence to perform this exchange, which we will examine in detail in TLS Handshake. The ingenious part of this handshake, and the reason TLS works in practice, is due to its use of public key cryptography (also known as asymmetric key cryptography), which allows the peers to negotiate a shared secret key without having to establish any prior knowledge of each other, and to do so over an unencrypted channel.

server is who it claims to be (e.g., your bank) and not someone simply pretending to be the destination by spoofing its name or IP address. This verification is based on the established chain of trust — see Chain of Trust and Certificate Authorities. In addition, the server can also optionally verify the identity of the client — e.g., a company proxy server can authenticate all employees, each of whom could have their own unique certificate signed by the company.

Finally, with encryption and authentication in place, the TLS protocol also provides its own message framing mechanism and signs each message with a message authentication code (MAC). The MAC algorithm is a one-way cryptographic hash function (effectively a checksum), the keys to which are negotiated by both connection peers. Whenever a TLS record is sent, a MAC value is generated and appended for that message, and the receiver is then able to compute and verify the sent MAC value to ensure message integrity and authenticity.

Combined, all three mechanisms serve as a foundation for secure communication on the Web. All modern web browsers provide support for a variety of ciphersuites, are able to authenticate both the client and server, and transparently perform message integrity checks for every record.

Proxies, Intermediaries, TLS, and New Protocols on the Web §

The extensibility and the success of HTTP created a vibrant ecosystem of various proxies and intermediaries on the Web: cache servers, security gateways, web accelerators, content filters, and many others. In some cases we are aware of their presence (explicit proxies), and in others they are completely transparent to the end user.

Unfortunately, the very success and the presence of these servers has created a problem for anyone who tries to deviate from the HTTP/1.x protocol in any way: some proxy servers may simply relay new HTTP extensions or alternative wire formats they cannot interpret, others may continue to blindly apply their logic even when they shouldn't, and some, such as security appliances, may infer malicious intent where there is none.

In other words, in practice, deviating from the well-defined semantics of HTTP/1.x on port 80 often leads to unreliable deployments: some clients have no problems, while others fail with unpredictable and hard-to-reproduce behaviors — e.g., the same client may see different behaviors as it migrates between different networks.

Due to these behaviors, new protocols and extensions to HTTP, such as WebSocket, HTTP/2, and others, have to rely on establishing an HTTPS tunnel to bypass the intermediate proxies and provide a reliable deployment model: the encrypted tunnel obfuscates the data from all intermediaries. If you have ever wondered why most WebSocket guides will tell you to use HTTPS to deliver data to mobile clients, this is why.

HTTPS Everywhere §

Unencrypted communication—via HTTP and other protocols—creates a large number of privacy, security, and integrity vulnerabilities. Such exchanges are susceptible to interception, manipulation, and impersonation, and can reveal users credentials, history, identity, and other sensitive information. Our applications need to protect themselves, and our users, against these threats by delivering data over HTTPS.

HTTPS protects the integrity of the website

Encryption prevents intruders from tampering with exchanged data—e.g. rewriting content, injecting unwanted and malicious content, and so on.

HTTPS protects the privacy and security of the user

Encryption prevents intruders from listening in on the exchanged data. Each unprotected request can reveal sensitive information about the user, and when such data is aggregated across many sessions, can be used to de-anonymize their identities and reveal other sensitive information. All browsing activity, as far as the user is concerned, should be considered private and sensitive.

HTTPS enables powerful features on the web

A growing number of new web platform features, such as accessing users geolocation, taking pictures, recording video, enabling offline app experiences, and more, require explicit user opt-in that, in turn, requires HTTPS. The security and integrity guarantees provided by HTTPS are critical components for delivering a secure user permission workflow and protecting their preferences.

To further the point, both the Internet Engineering Task Force (IETF) and the Internet Architecture Board (IAB) have issued guidance to developers and protocol designers that strongly encourages adoption of HTTPS:

- IETF: Pervasive Monitoring Is an Attack
- IAB: Statement on Internet Confidentiality

As our dependency on the Internet has grown, so have the risks and the stakes for everyone that is relying on it. As a result, it is our responsibility, both as the application developers and users, to ensure that we protect ourselves by enabling HTTPS everywhere.

Note

The HTTPS-Only Standard published by the White House's Office of Management and Budget is a great resource for additional information on the need for HTTPS, and hands-on advice for deploying it.

Let's Encrypt

A common objection and roadblock towards widespread adoption of HTTPS has been the

Certificate Authorities. The Let’s Encrypt ⊘ project launched in 2015 solves this particular problem:

> *"Let’s Encrypt is a free, automated, and open certificate authority brought to you by the Internet Security Research Group (ISRG). The objective of Let’s Encrypt and the ACME protocol is to make it possible to set up an HTTPS server and have it automatically obtain a browser-trusted certificate, without any human intervention."*

Visit the project website to learn how to set it up on your own site. There are no restrictions, now anyone can obtain a trusted certificate for their site, free of charge.

TLS Handshake §

Before the client and the server can begin exchanging application data over TLS, the encrypted tunnel must be negotiated: the client and the server must agree on the version of the TLS protocol, choose the ciphersuite, and verify certificates if necessary. Unfortunately, each of these steps requires new packet roundtrips (Figure 4-2) between the client and the server, which adds startup latency to all TLS connections.

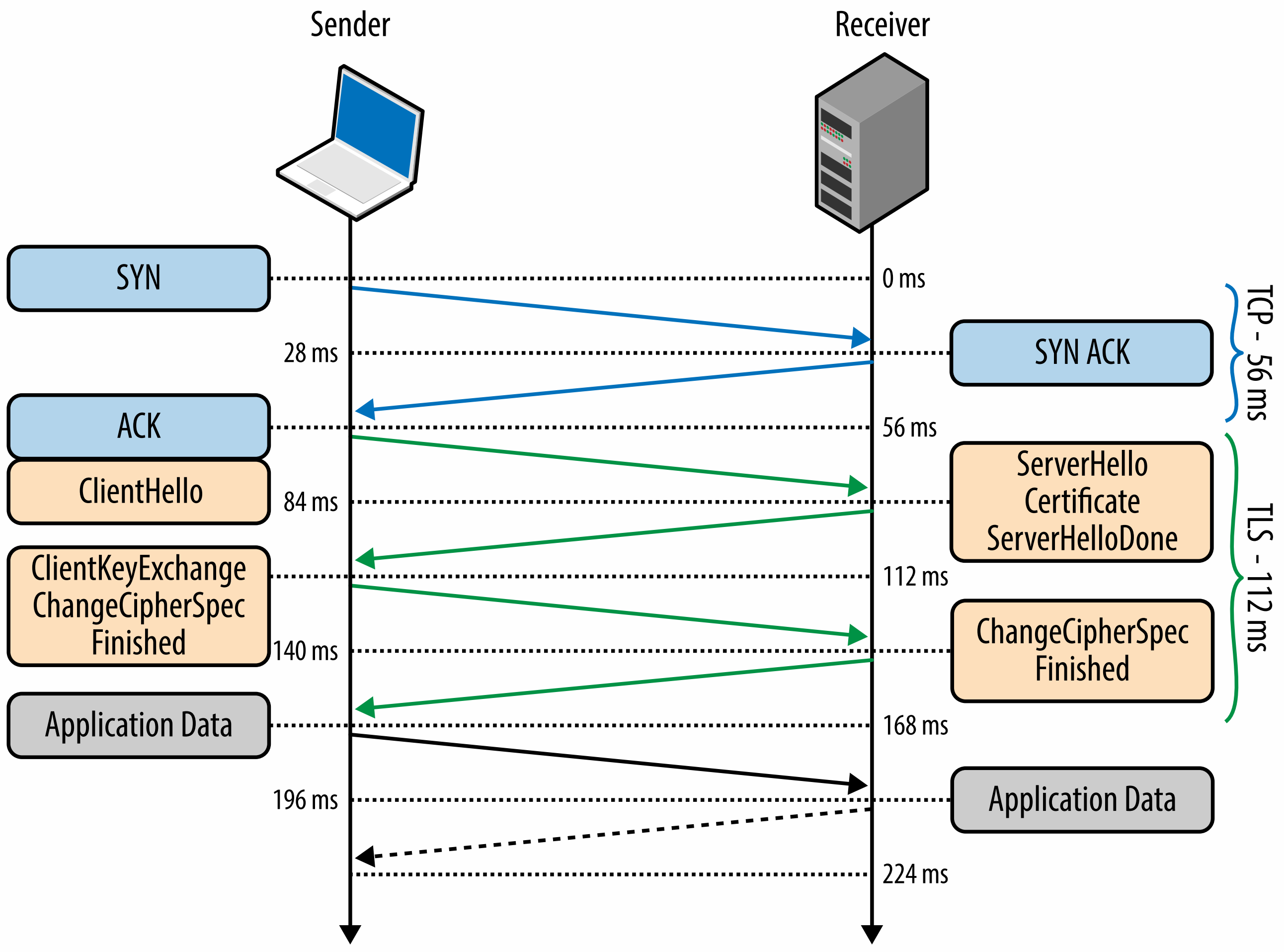


Figure 4-2. TLS handshake protocol

Figure 4-2 assumes the same (optimistic) 28 millisecond one-way "light in fiber" delay between New York and London as used in previous TCP connection establishment examples; see Table 1-1.

`0 ms`

TLS runs over a reliable transport (TCP), which means that we must first complete the TCP three-way handshake, which takes one full roundtrip.

`56 ms`

With the TCP connection in place, the client sends a number of specifications in plain text, such as the version of the TLS protocol it is running, the list of supported ciphersuites, and other TLS options it may want to use.

`84 ms`

The server picks the TLS protocol version for further communication, decides on a ciphersuite from the list provided by the client, attaches its certificate, and sends the response back to the client. Optionally, the server can also send a request for the client's certificate and parameters for other TLS extensions.

`112 ms`

Assuming both sides are able to negotiate a common version and cipher, and the client is happy with the certificate provided by the server, the client initiates either the RSA or the Diffie-Hellman key exchange, which is used to establish the symmetric key for the ensuing session.

`140 ms`

The server processes the key exchange parameters sent by the client, checks message integrity by verifying the MAC, and returns an encrypted `Finished` message back to the client.

`168 ms`

The client decrypts the message with the negotiated symmetric key, verifies the MAC, and if all is well, then the tunnel is established and application data can now be sent.

As the above exchange illustrates, new TLS connections require two roundtrips for a "full handshake"—that's the bad news. However, in practice, optimized deployments can do much better and deliver a consistent 1-RTT TLS handshake:

- False Start is a TLS protocol extension that allows the client and server to start transmitting encrypted application data when the handshake is only partially complete—i.e., once `ChangeCipherSpec` and `Finished` messages are sent, but without waiting for the other side to do the same. This optimization reduces handshake overhead for new TLS connections to one roundtrip; see Enable TLS False Start.

CPU overhead by reusing the previously negotiated parameters for the secure session; see TLS Session Resumption.

The combination of both of the above optimizations allows us to deliver a consistent 1-RTT TLS handshake for new and returning visitors, plus computational savings for sessions that can be resumed based on previously negotiated session parameters. Make sure to take advantage of these optimizations in your deployments.

Note

One of the design goals for TLS 1.3 is to reduce the latency overhead for setting up the secure connection: 1-RTT for new, and 0-RTT for resumed sessions!

RSA, Diffie-Hellman and Forward Secrecy §

Due to a variety of historical and commercial reasons the RSA handshake has been the dominant key exchange mechanism in most TLS deployments: the client generates a symmetric key, encrypts it with the server's public key, and sends it to the server to use as the symmetric key for the established session. In turn, the server uses its private key to decrypt the sent symmetric key and the key-exchange is complete. From this point forward the client and server use the negotiated symmetric key to encrypt their session.

The RSA handshake works, but has a critical weakness: the same public-private key pair is used both to authenticate the server and to encrypt the symmetric session key sent to the server. As a result, if an attacker gains access to the server's private key and listens in on the exchange, then they can decrypt the the entire session. Worse, even if an attacker does not currently have access to the private key, they can still record the encrypted session and decrypt it at a later time once they obtain the private key.

By contrast, the Diffie-Hellman key exchange allows the client and server to negotiate a shared secret without explicitly communicating it in the handshake: the server's private key is used to sign and verify the handshake, but the established symmetric key never leaves the client or server and cannot be intercepted by a passive attacker even if they have access to the private key.

Note

For the curious, the Wikipedia article on Diffie-Hellman key exchange is a great place to learn about the algorithm and its properties.

Best of all, Diffie-Hellman key exchange can be used to reduce the risk of compromise of past communication sessions: we can generate a new "ephemeral" symmetric key as part of each and never communicated and are actively renegotiated for each the new session, the worst-case scenario is that an attacker could compromise the client or server and access the session keys of the current and future sessions. However, knowing the private key, or the current ephemeral key, does not help the attacker decrypt any of the previous sessions!

Combined, the use of Diffie-Hellman key exchange and ephemeral sessions keys enables "perfect forward secrecy" (PFS): the compromise of long-term keys (e.g. server's private key) does not compromise past session keys and does not allow the attacker to decrypt previously recorded sessions. A highly desirable property, to say the least!

As a result, and this should not come as a surprise, the RSA handshake is now being actively phased out: all the popular browsers prefer ciphers that enable forward secrecy (i.e., rely on Diffie-Hellman key exchange), and as an additional incentive, may enable certain protocol optimizations only when forward secrecy is available—e.g. 1-RTT handshakes via TLS False Start.

Which is to say, consult your server documentation on how to enable and deploy forward secrecy! Once again, good security and performance go hand in hand.

Performance of Public vs. Symmetric Key Cryptography §

Public-key cryptography is used only during initial setup of the TLS tunnel: the certificates are authenticated and the key exchange algorithm is executed.

Symmetric key cryptography, which uses the established symmetric key is then used for all further communication between the client and the server within the session. This is done, in large part, to improve performance—public key cryptography is much more computationally expensive. To illustrate the difference, if you have OpenSSL installed on your computer, you can run the following tests:

- `$> openssl speed ecdh`
- `$> openssl speed aes`

Note that the units between the two tests are not directly comparable: the Elliptic Curve Diffie-Hellman (ECDH) test provides a summary table of operations per second for different key sizes, while AES performance is measured in bytes per second. Nonetheless, it should be easy to see that the ECDH operations are much more computationally expensive.

The exact performance numbers vary significantly based on used hardware, number of cores, TLS version, server configuration, and other factors. Don't fall for an outdated benchmark! Always run the performance tests on your own hardware and refer to Reduce Computational Costs for additional context.

Application Layer Protocol Negotiation (ALPN) §

Two network peers may want to use a custom application protocol to communicate with each other. One way to resolve this is to determine the protocol upfront, assign a well-known port to it (e.g., port 80 for HTTP, port 443 for TLS), and configure all clients and servers to use it. However, in practice, this is a slow and impractical process: each port assignment must be approved and, worse, firewalls and other intermediaries often permit traffic only on ports 80 and 443.

As a result, to enable easy deployment of custom protocols, we must reuse ports 80 or 443 and use an additional mechanism to negotiate the application protocol. Port 80 is reserved for HTTP, and the HTTP specification provides a special `Upgrade` flow for this very purpose. However, the use of `Upgrade` can add an extra network roundtrip of latency, and in practice is often unreliable in the presence of many intermediaries; see Proxies, Intermediaries, TLS, and New Protocols on the Web.

Note

For a hands-on example of HTTP Upgrade workflow, flip ahead to Upgrading to HTTP/2.

The solution is, you guessed it, to use port 443, which is reserved for secure HTTPS sessions running over TLS. The use of an end-to-end encrypted tunnel obfuscates the data from intermediate proxies and enables a quick and reliable way to deploy new application protocols. However, we still need another mechanism to negotiate the protocol that will be used within the TLS session.

Application Layer Protocol Negotiation (ALPN), as the name implies, is a TLS extension that addresses this need. It extends the TLS handshake (Figure 4-2) and allows the peers to negotiate protocols without additional roundtrips. Specifically, the process is as follows:

- The client appends a new `ProtocolNameList` field, containing the list of supported application protocols, into the `ClientHello` message.
- The server inspects the `ProtocolNameList` field and returns a `ProtocolName` field indicating the selected protocol as part of the `ServerHello` message.

The server may respond with only a single protocol name, and if it does not support any that the client requests, then it may choose to abort the connection. As a result, once the TLS handshake is finished, both the secure tunnel is established, and the client and server are in agreement as to which application protocol will be used; the client and server can immediately begin exchanging messages via the negotiated protocol.

History and Relationship of NPN and ALPN §

Next Protocol Negotiation (NPN) is a TLS extension, which was developed as part of the SPDY effort at Google to enable efficient application protocol negotiation during the TLS handshake. Sound

ALPN is a revised and IETF approved version of the NPN extension. In NPN, the server advertised which protocols it supports, and the client then chose and confirmed the protocol. In ALPN, this exchange was reversed: the client now specifies which protocols it supports, and the server then selects and confirms the protocol. The rationale for the change is that this brings ALPN into closer alignment with other protocol negotiation standards.

In short, ALPN is a successor to NPN.

Server Name Indication (SNI) §

An encrypted TLS tunnel can be established between any two TCP peers: the client only needs to know the IP address of the other peer to make the connection and perform the TLS handshake. However, what if the server wants to host multiple independent sites, each with its own TLS certificate, on the same IP address — how does that work? Trick question; it doesn't.

To address the preceding problem, the Server Name Indication (SNI) extension was introduced to the TLS protocol, which allows the client to indicate the hostname the client is attempting to connect to as part of the TLS handshake. In turn, the server is able to inspect the SNI hostname sent in the `ClientHello` message, select the appropriate certificate, and complete the TLS handshake for the desired host.

TLS, HTTP, and Dedicated IPs §

The TLS+SNI workflow is identical to `Host` header advertisement in HTTP, where the client indicates the hostname of the site it is requesting: the same IP address may host many different domains, and both SNI and `Host` are required to disambiguate between them.

Unfortunately, some older clients (e.g., most IE versions running on Windows XP, Android 2.2, and others) do not support SNI. As a result, if you need to provide TLS to such clients, then you may need a dedicated IP address for each and every host.

TLS Session Resumption §

The extra latency and computational costs of the full TLS handshake impose a serious performance penalty on all applications that require secure communication. To help mitigate some of the costs, TLS provides a mechanism to resume or share the same negotiated secret key data between multiple connections.

Session Identifiers §

The first Session Identifiers (RFC 5246) resumption mechanism was introduced in SSL 2.0, which allowed the server to create and send a 32-byte session identifier as part of its `ServerHello` message during the full TLS negotiation we saw earlier. With the session ID in place, both the client and server can store the previously negotiated session parameters—keyed by session ID—and reuse them for a subsequent session.

Specifically, the client can include the session ID in the `ClientHello` message to indicate to the server that it still remembers the negotiated cipher suite and keys from previous handshake and is able to reuse them. In turn, if the server is able to find the session parameters associated with the advertised ID in its cache, then an abbreviated handshake (Figure 4-3) can take place. Otherwise, a full new session negotiation is required, which will generate a new session ID.

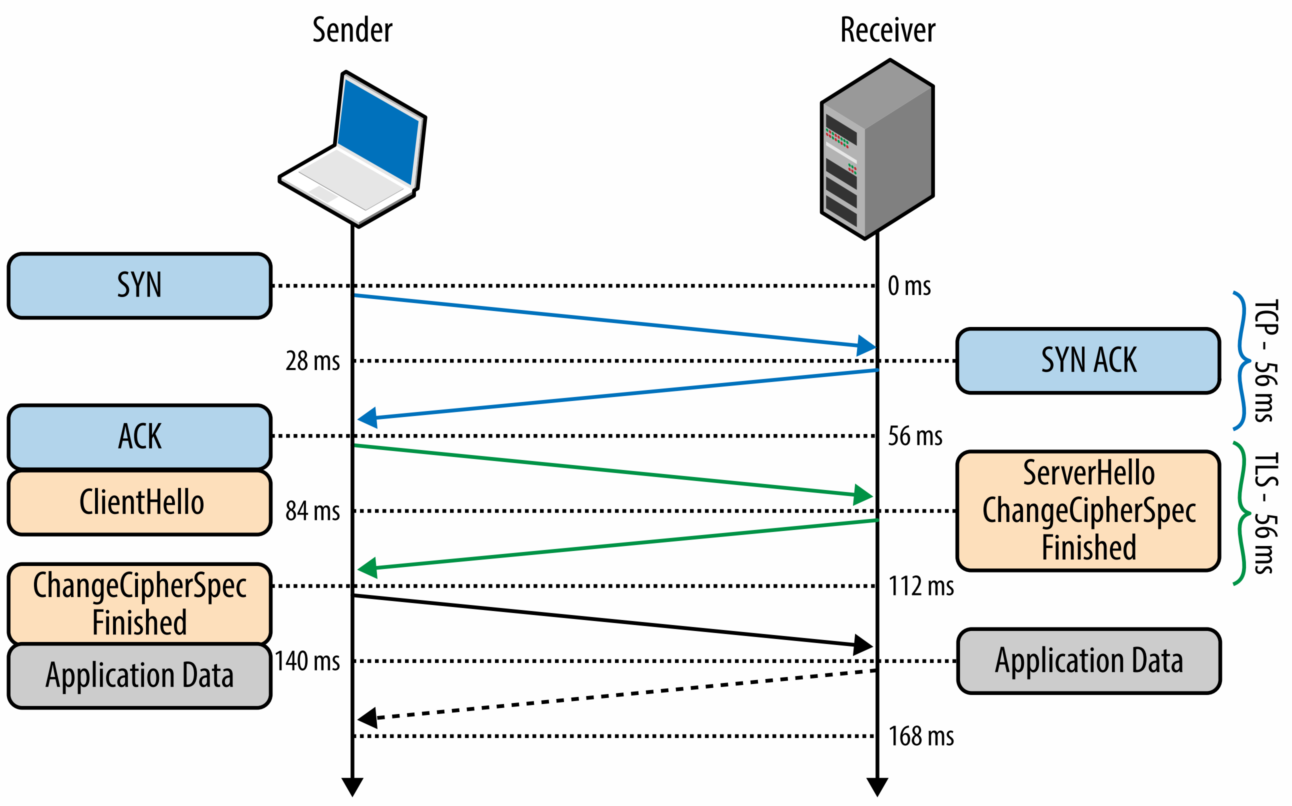


Figure 4-3. Abbreviated TLS handshake protocol

Leveraging session identifiers allows us to remove a full roundtrip, as well as the overhead of public key cryptography, which is used to negotiate the shared secret key. This allows a secure connection to be established quickly and with no loss of security, since we are reusing the previously negotiated session data.

Note

Session resumption is an important optimization both for HTTP/1.x and HTTP/2 deployments. The abbreviated handshake eliminates a full roundtrip of latency and significantly reduces computational costs for both sides.

In fact, if the browser requires multiple connections to the same host (e.g. when HTTP/1.x is in *additional connections to the same server, such that they can be "resumed" and reuse the same session parameters. If you've ever looked at a network trace and wondered why you rarely see multiple same-host TLS negotiations in flight, that's why!*

However, one of the practical limitations of the Session Identifiers mechanism is the requirement for the server to create and maintain a session cache for every client. This results in several problems on the server, which may see tens of thousands or even millions of unique connections every day: consumed memory for every open TLS connection, a requirement for a session ID cache and eviction policies, and nontrivial deployment challenges for popular sites with many servers, which should, ideally, use a shared TLS session cache for best performance.

None of the preceding problems are impossible to solve, and many high-traffic sites are using session identifiers successfully today. But for any multi-server deployment, session identifiers will require some careful thinking and systems architecture to ensure a well operating session cache.

Session Tickets §

To address this concern for server-side deployment of TLS session caches, the "Session Ticket" (RFC 5077) replacement mechanism was introduced, which removes the requirement for the server to keep per-client session state. Instead, if the client indicates that it supports session tickets, the server can include a `New Session Ticket` record, which includes all of the negotiated session data encrypted with a secret key known only by the server.

This session ticket is then stored by the client and can be included in the `SessionTicket` extension within the `ClientHello` message of a subsequent session. Thus, all session data is stored only on the client, but the ticket is still safe because it is encrypted with a key known only by the server.

The session identifiers and session ticket mechanisms are respectively commonly referred to as *session caching* and *stateless resumption* mechanisms. The main improvement of stateless resumption is the removal of the server-side session cache, which simplifies deployment by requiring that the client provide the session ticket on every new connection to the server—that is, until the ticket has expired.

Note

In practice, deploying session tickets across a set of load-balanced servers also requires some careful thinking and systems architecture: all servers must be initialized with the same session key, and an additional mechanism is required to periodically and securely rotate the shared key across all servers.

Chain of Trust and Certificate Authorities §

Authentication is an integral part of establishing every TLS connection. After all, it is possible to carry out a conversation over an encrypted tunnel with any peer, including an attacker, and unless we can be sure that the host we are speaking to is the one we trust, then all the encryption work could be for nothing. To understand how we can verify the peer's identity, let's examine a simple authentication workflow between Alice and Bob:

- Both Alice and Bob generate their own public and private keys.
- Both Alice and Bob hide their respective private keys.
- Alice shares her public key with Bob, and Bob shares his with Alice.
- Alice generates a new message for Bob and signs it with her private key.
- Bob uses Alice's public key to verify the provided message signature.

Trust is a key component of the preceding exchange. Specifically, public key encryption allows us to use the public key of the sender to verify that the message was signed with the right private key, but the decision to approve the sender is still one that is based on trust. In the exchange just shown, Alice and Bob could have exchanged their public keys when they met in person, and because they know each other well, they are certain that their exchange was not compromised by an impostor—perhaps they even verified their identities through another, secret (physical) handshake they had established earlier!

Next, Alice receives a message from Charlie, whom she has never met, but who claims to be a friend of Bob's. In fact, to prove that he is friends with Bob, Charlie asked Bob to sign his own public key with Bob's private key and attached this signature with his message (Figure 4-4). In this case, Alice first checks Bob's signature of Charlie's key. She knows Bob's public key and is thus able to verify that Bob did indeed sign Charlie's key. Because she trusts Bob's decision to verify Charlie, she accepts the message and performs a similar integrity check on Charlie's message to ensure that it is, indeed, from Charlie.

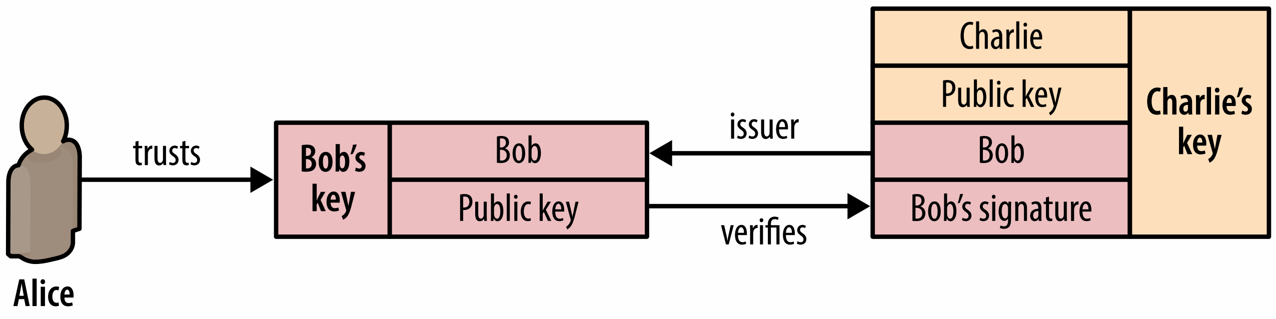


Figure 4-4. Chain of trust for Alice, Bob, and Charlie

What we have just done is established a chain of trust: Alice trusts Bob, Bob trusts Charlie, and by transitive trust, Alice decides to trust Charlie. As long as nobody in the chain is compromised, this

Authentication on the Web and in your browser follows the exact same process as shown. Which means that at this point you should be asking: whom does your browser trust, and whom do you trust when you use the browser? There are at least three answers to this question:

Manually specified certificates
: Every browser and operating system provides a mechanism for you to manually import any certificate you trust. How you obtain the certificate and verify its integrity is completely up to you.

Certificate authorities
: A certificate authority (CA) is a trusted third party that is trusted by both the subject (owner) of the certificate and the party relying upon the certificate.

The browser and the operating system
: Every operating system and most browsers ship with a list of well-known certificate authorities. Thus, you also trust the vendors of this software to provide and maintain a list of trusted parties.

In practice, it would be impractical to store and manually verify each and every key for every website (although you can, if you are so inclined). Hence, the most common solution is to use certificate authorities (CAs) to do this job for us (Figure 4-5): the browser specifies which CAs to trust (root CAs), and the burden is then on the CAs to verify each site they sign, and to audit and verify that these certificates are not misused or compromised. If the security of any site with the CA's certificate is breached, then it is also the responsibility of that CA to revoke the compromised certificate.

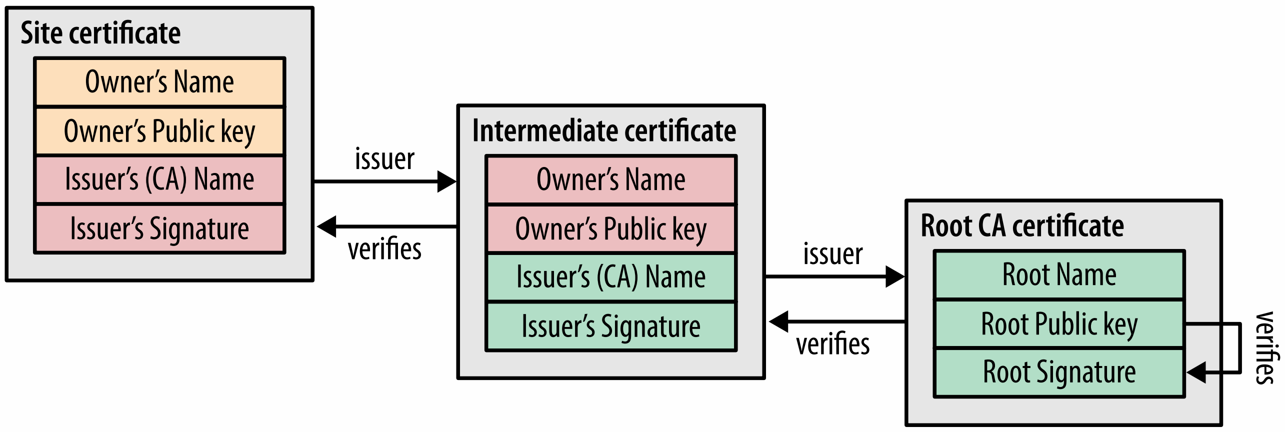
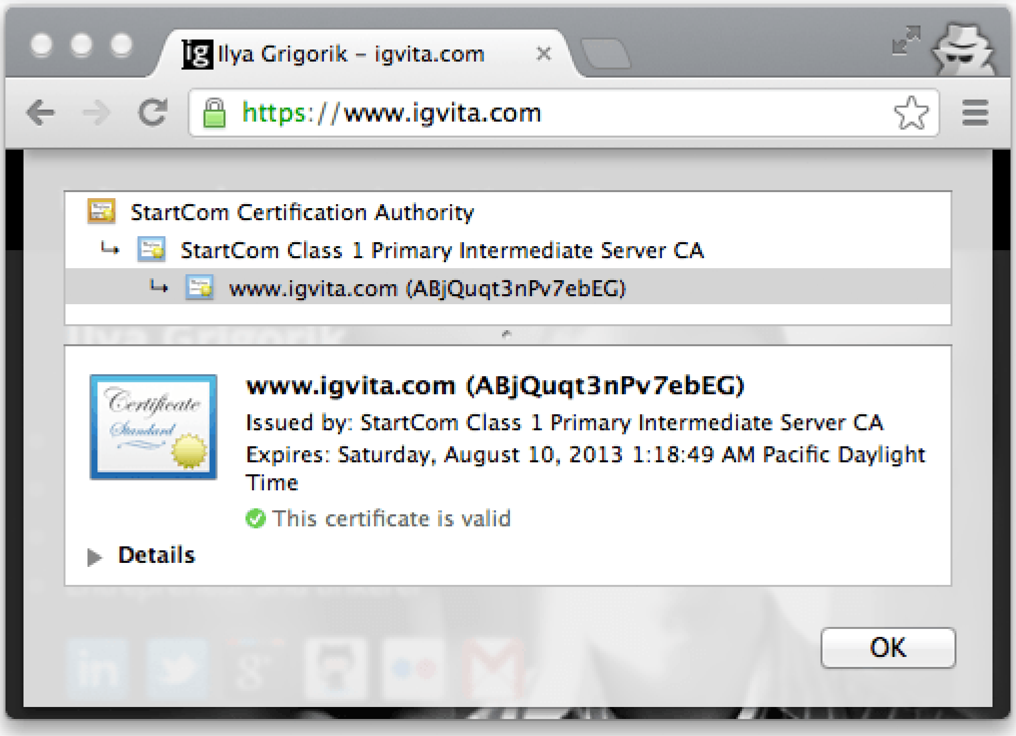


Figure 4-5. CA signing of digital certificates

Every browser allows you to inspect the chain of trust of your secure connection (Figure 4-6), usually accessible by clicking on the lock icon beside the URL.



StartCom Certification Authority
↳ StartCom Class 1 Primary Intermediate Server CA
↳ www.igvita.com (ABjQuqt3nPv7ebEG)

www.igvita.com (ABjQuqt3nPv7ebEG)
Issued by: StartCom Class 1 Primary Intermediate Server CA
Expires: Saturday, August 10, 2013 1:18:49 AM Pacific Daylight Time
This certificate is valid
Details

Figure 4-6. Certificate chain of trust for igvita.com (Google Chrome, v25)

- igvita.com certificate is signed by StartCom Class 1 Primary Intermediate Server.
- StartCom Class 1 Primary Intermediate Server certificate is signed by the StartCom Certification Authority.
- StartCom Certification Authority is a recognized root certificate authority.

The "trust anchor" for the entire chain is the root certificate authority, which in the case just shown, is the StartCom Certification Authority. Every browser ships with a pre-initialized list of trusted certificate authorities ("roots"), and in this case, the browser trusts and is able to verify the StartCom root certificate. Hence, through a transitive chain of trust in the browser, the browser vendor, and the StartCom certificate authority, we extend the trust to our destination site.

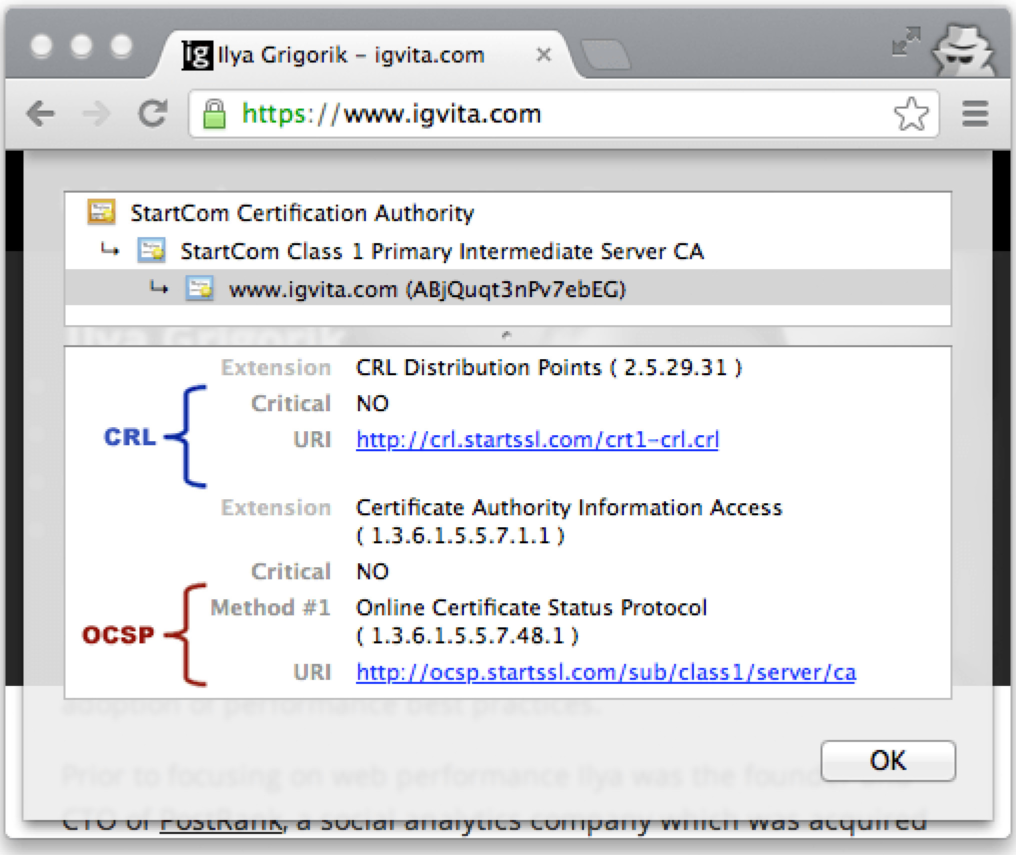
Certificate Transparency §

Every operating system vendor and every browser provide a public listing of all the certificate authorities they trust by default. Use your favorite search engine to find and investigate these lists. In practice, you'll find that most systems rely on hundreds of trusted certificate authorities, which is also a common complaint against the system: the large number of trusted CAs creates a large attack surface area against the chain of trust in your browser.

The good news is, the Certificate Transparency ⊘ project is working to address these flaws by providing a framework—a public log—for monitoring and auditing of issuance of all new certificates. Visit the project website to learn more.

Certificate Revocation §

Occasionally the issuer of a certificate will need to revoke or invalidate the certificate due to a number of possible reasons: the private key of the certificate has been compromised, the certificate authority itself has been compromised, or due to a variety of more benign reasons such as a superseding certificate, change in affiliation, and so on. To address this, the certificates themselves contain instructions (Figure 4-7) on how to check if they have been revoked. Hence, to ensure that the chain of trust is not compromised, each peer can check the status of each certificate by following the embedded instructions, along with the signatures, as it verifies the certificate chain.



StartCom Certification Authority
↳ StartCom Class 1 Primary Intermediate Server CA
↳ www.igvita.com (ABjQuqt3nPv7ebEG)

CRL	Extension	CRL Distribution Points (2.5.29.31)
	Critical	NO
	URI	http://crl.startssl.com/crt1-crl.crl
	Extension	Certificate Authority Information Access (1.3.6.1.5.5.7.1.1)
OCSP	Critical	NO
	Method #1	Online Certificate Status Protocol (1.3.6.1.5.5.7.48.1)
	URI	http://ocsp.startssl.com/sub/class1/server/ca

Figure 4-7. CRL and OCSP instructions for igvita.com (Google Chrome, v25)

Certificate Revocation List (CRL) §

Certificate Revocation List (CRL) is defined by RFC 5280 and specifies a simple mechanism to check the status of every certificate: each certificate authority maintains and periodically publishes a list of revoked certificate serial numbers. Anyone attempting to verify a certificate is then able to download the revocation list, cache it, and check the presence of a particular serial number within it—if it is present, then it has been revoked.

This process is simple and straightforward, but it has a number of limitations:

- The growing number of revocations means that the CRL list will only get longer, and each client must retrieve the entire list of serial numbers.
- There is no mechanism for instant notification of certificate revocation—if the CRL was cached by the client before the certificate was revoked, then the CRL will deem the revoked certificate valid until the cache expires.
- The need to fetch the latest CRL list from the CA may block certificate verification, which can add significant latency to the TLS handshake.
- The CRL fetch may fail due to variety of reasons, and in such cases the browser behavior is undefined. Most browsers treat such cases as "soft fail", allowing the verification to proceed—yes, yikes.

Online Certificate Status Protocol (OCSP) §

To address some of the limitations of the CRL mechanism, the Online Certificate Status Protocol (OCSP) was introduced by RFC 2560, which provides a mechanism to perform a real-time check for status of the certificate. Unlike the CRL file, which contains all the revoked serial numbers, OCSP allows the client to query the CA's certificate database directly for just the serial number in question while validating the certificate chain.

As a result, the OCSP mechanism consumes less bandwidth and is able to provide real-time validation. However, the requirement to perform real-time OCSP queries creates its own set of problems:

- The CA must be able to handle the load of the real-time queries.
- The CA must ensure that the service is up and globally available at all times.
- Real-time OCSP requests may impair the client's privacy because the CA knows which sites the client is visiting.
- The client must block on OCSP requests while validating the certificate chain.
- The browser behavior is, once again, undefined and typically results in a "soft fail" if the OCSP fetch fails due to a network timeout or other errors.

Note

As a real-world data point, Firefox telemetry shows that OCSP requests time out as much as 15% of the time, and add approximately 350 ms to the TLS handshake when successful—see hpbn.co/ocsp-performance .

OCSP Stapling §

For the reasons listed above, neither CRL or OSCP revocation mechanisms offer the security and performance guarantees that we desire for our applications. However, don’t despair, because OCSP stapling (RFC 6066, "Certificate Status Request" extension) addresses most of the issues we saw earlier by allowing the validation to be performed by the server and be sent ("stapled") as part of the TLS handshake to the client:

- Instead of the client making the OCSP request, it is the server that periodically retrieves the signed and timestamped OCSP response from the CA.
- The server then appends (i.e. "staples") the signed OCSP response as part of the TLS handshake, allowing the client to validate both the certificate and the attached OCSP revocation record signed by the CA.

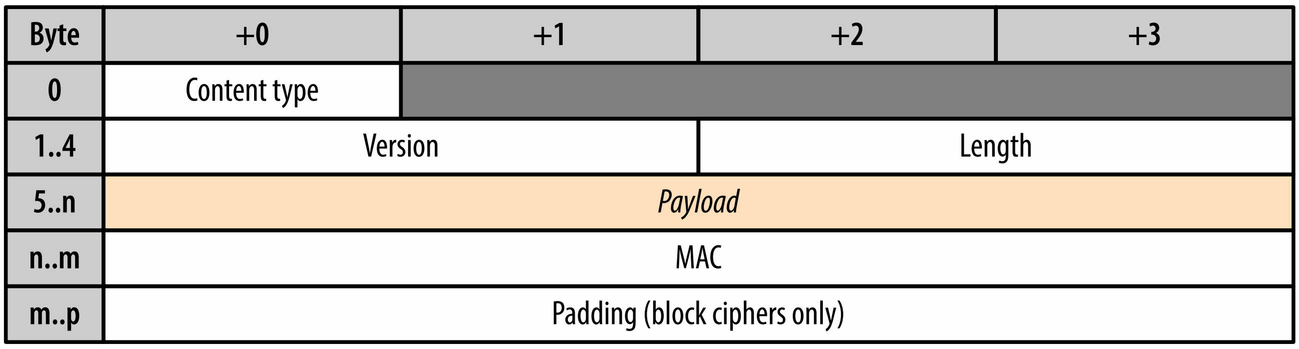
This role reversal is secure, because the stapled record is signed by the CA and can be verified by the client, and offers a number of important benefits:

- The client does not leak its navigation history.
- The client does not have to block and query the OCSP server.
- The client may "hard-fail" revocation handling if the server opts-in (by advertising the OSCP "Must-Staple" flag) and the verification fails.

In short, to get both the best security and performance guarantees, make sure to configure and test OCSP stapling on your servers.

TLS Record Protocol

Not unlike the IP or TCP layers below it, all data exchanged within a TLS session is also framed using a well-defined protocol (Figure 4-8). The TLS Record protocol is responsible for identifying different types of messages (handshake, alert, or data via the "Content Type" field), as well as securing and verifying the integrity of each message.



Byte	+0	+1	+2	+3
0	Content type			
1..4	Version		Length	
5..n	*Payload*			
n..m	MAC			
m..p	Padding (block ciphers only)			

Figure 4-8. TLS record structure

A typical workflow for delivering application data is as follows:

- Record protocol receives application data.

- Message authentication code (MAC) or HMAC is added to each record.
- Data within each record is encrypted using the negotiated cipher.

Once these steps are complete, the encrypted data is passed down to the TCP layer for transport. On the receiving end, the same workflow, but in reverse, is applied by the peer: decrypt record using negotiated cipher, verify MAC, extract and deliver the data to the application above it.

The good news is that all the work just shown is handled by the TLS layer itself and is completely transparent to most applications. However, the record protocol does introduce a few important implications that we need to be aware of:

- Maximum TLS record size is 16 KB
- Each record contains a 5-byte header, a MAC (up to 20 bytes for SSLv3, TLS 1.0, TLS 1.1, and up to 32 bytes for TLS 1.2), and padding if a block cipher is used.
- To decrypt and verify the record, the entire record must be available.

Picking the right record size for your application, if you have the ability to do so, can be an important optimization. Small records incur a larger CPU and byte overhead due to record framing and MAC verification, whereas large records will have to be delivered and reassembled by the TCP layer before they can be processed by the TLS layer and delivered to your application —skip ahead to Optimize TLS Record Size for full details.

Optimizing for TLS §

Deploying your application over TLS will require some additional work, both within your application (e.g. migrating resources to HTTPS to avoid mixed content), and on the configuration of the infrastructure responsible for delivering the application data over TLS. A well tuned deployment can make an enormous positive difference in the observed performance, user experience, and overall operational costs. Let’s dive in.

Reduce Computational Costs §

Establishing and maintaining an encrypted channel introduces additional computational costs for both peers. Specifically, first there is the asymmetric (public key) encryption used during the TLS handshake (explained TLS Handshake). Then, once a shared secret is established, it is used as a symmetric key to encrypt all TLS records.

As we noted earlier, public key cryptography is more computationally expensive when compared with symmetric key cryptography, and in the early days of the Web often required additional hardware to perform "SSL offloading." The good news is, this is no longer necessary and what once required dedicated hardware can now be done directly on the CPU. Large organizations such as Facebook, Twitter, and Google, which offer TLS to billions of users, perform all the

> *In January this year (2010), Gmail switched to using HTTPS for everything by default. Previously it had been introduced as an option, but now all of our users use HTTPS to secure their email between their browsers and Google, all the time. In order to do this we had to deploy no additional machines and no special hardware. On our production frontend machines, SSL/TLS accounts for less than 1% of the CPU load, less than 10 KB of memory per connection and less than 2% of network overhead. Many people believe that SSL/TLS takes a lot of CPU time and we hope the preceding numbers (public for the first time) will help to dispel that.*
>
> *If you stop reading now you only need to remember one thing: SSL/TLS is not computationally expensive anymore.*
>
> *Adam Langley (Google)*

> *We have deployed TLS at a large scale using both hardware and software load balancers. We have found that modern software-based TLS implementations running on commodity CPUs are fast enough to handle heavy HTTPS traffic load without needing to resort to dedicated cryptographic hardware. We serve all of our HTTPS traffic using software running on commodity hardware.*
>
> *Doug Beaver (Facebook)*

> *Elliptic Curve Diffie-Hellman (ECDHE) is only a little more expensive than RSA for an equivalent security level… In practical deployment, we found that enabling and prioritizing ECDHE cipher suites actually caused negligible increase in CPU usage. HTTP keepalives and session resumption mean that most requests do not require a full handshake, so handshake operations do not dominate our CPU usage. We find 75% of Twitter's client requests are sent over connections established using ECDHE. The remaining 25% consists mostly of older clients that don't yet support the ECDHE cipher suites.*
>
> *Jacob Hoffman-Andrews (Twitter)*

To get the best results in your own deployments, make the best of TLS Session Resumption—deploy, measure, and optimize its success rate. Eliminating the need to perform the costly public key cryptography operations on every handshake will significantly reduce both the computational and latency costs of TLS; there is no reason to spend CPU cycles on work that you don't need to do.

Note

Speaking of optimizing CPU cycles, make sure to keep your servers up to date with the latest version of the TLS libraries! In addition to the security improvements, you will also often see performance benefits. Security and performance go hand-in-hand.

An unoptimized TLS deployment can easily add many additional roundtrips and introduce significant latency for the user—e.g. multi-RTT handshakes, slow and ineffective certificate revocation checks, large TLS records that require multiple roundtrips, and so on. Don’t be that site, you can do much better.

A well-tuned TLS deployment should add **at most one extra roundtrip** for negotiating the TLS connection, regardless of whether it is new or resumed, and avoid all other latency pitfalls: configure session resumption, and enable forward secrecy to enable TLS False Start.

Note

To get the best end-to-end performance, make sure to audit both own and third-party services and servers used by your application, including your CDN provider. For a quick report-card overview of popular servers and CDNs, check out istlsfastyet.com .

Optimize Connection Reuse §

The best way to minimize both latency and computational overhead of setting up new TCP+TLS connections is to optimize connection reuse. Doing so amortizes the setup costs across requests and delivers a much faster experience to the user.

Verify that your server and proxy configurations are setup to allow keepalive connections, and audit your connection timeout settings. Many popular servers set aggressive connection timeouts (e.g. some Apache versions default to 5s timeouts) that force a lot of unnecessary renegotiations. For best results, use your logs and analytics to determine the optimal timeout values.

Leverage Early Termination §

As we discussed in Primer on Latency and Bandwidth, we may not be able to make our packets travel faster, but we can make them travel a shorter distance. By placing our "edge" servers closer to the user (Figure 4-9), we can significantly reduce the roundtrip times and the total costs of the TCP and TLS handshakes.

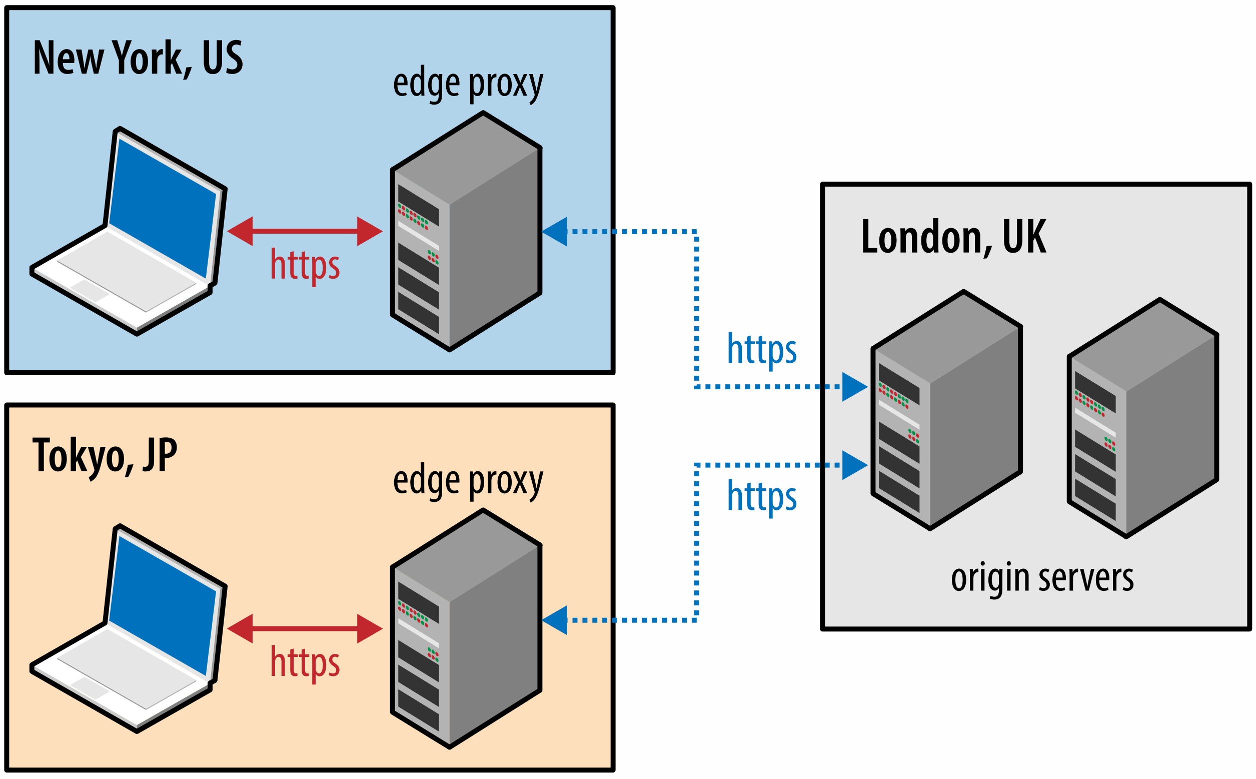


Figure 4-9. Early termination of client connections

A simple way to accomplish this is to leverage the services of a content delivery network (CDN) that maintains pools of edge servers around the globe, or to deploy your own. By allowing the user to terminate their connection with a nearby server, instead of traversing across oceans and continental links to your origin, the client gets the benefit of "early termination" with shorter roundtrips. This technique is equally useful and important for static and dynamic content: static content can also be cached and served by the edge servers, whereas dynamic requests can be routed over established connections from the edge to origin.

Uncached Origin Fetch §

The technique of using a CDN or a proxy server to fetch a resource, which may need to be customized per user or contains other private data, and hence is not a globally cacheable resource at the edge, is commonly known as an "uncached origin fetch."

While CDNs work best when the data is cached in geo-distributed servers around the world, the uncached origin fetch still provides a very important optimization: the client connection is terminated with the nearby server, which can dramatically reduce the handshake latency costs. In turn, the CDN, or your own proxy server, can maintain a "warm connection pool" to relay the data to the origin servers, allowing you to return a fast response back to the client.

In fact, as an additional layer of optimization, some CDN providers will use nearby servers on both sides of the connection! The client connection is terminated at a nearby CDN node, which then relays the request to the CDN node close to the origin, and the request is then routed to the origin. The hop within the CDN network allows the traffic to be routed over the optimized CDN backbone,

Configure Session Caching and Stateless Resumption §

Terminating the connection closer to the user is an optimization that will help decrease latency for your users in all cases, but once again, no bit is faster than a bit not sent—send fewer bits. Enabling TLS session caching and stateless resumption allows us to eliminate an entire roundtrip of latency and reduce computational overhead for repeat visitors.

Session identifiers, on which TLS session caching relies, were introduced in SSL 2.0 and have wide support among most clients and servers. However, if you are configuring TLS on your server, do not assume that session support will be on by default. In fact, it is more common to have it off on most servers by default—but you know better! Double-check and verify your server, proxy, and CDN configuration:

- Servers with multiple processes or workers should use a shared session cache.
- Size of the shared session cache should be tuned to your levels of traffic.
- A session timeout period should be provided.
- In a multi-server setup, routing the same client IP, or the same TLS session ID, to the same server is one way to provide good session cache utilization.
- Where "sticky" load balancing is not an option, a shared cache should be used between different servers to provide good session cache utilization, and a secure mechanism needs to be established to share and update the secret keys to decrypt the provided session tickets.
- Check and monitor your TLS session cache statistics for best performance.

In practice, and for best results, you should configure both session caching and session ticket mechanisms. These mechanisms work together to provide best coverage both for new and older clients.

Enable TLS False Start §

Session resumption provides two important benefits: it eliminates an extra handshake roundtrip for returning visitors and reduces the computational cost of the handshake by allowing reuse of previously negotiated session parameters. However, it does not help in cases where the visitor is communicating with the server for the first time, or if the previous session has expired.

To get the best of both worlds—a one roundtrip handshake for new and repeat visitors, and computational savings for repeat visitors—we can use TLS False Start, which is an optional protocol extension that allows the sender to send application data (Figure 4-10) when the handshake is only partially complete.

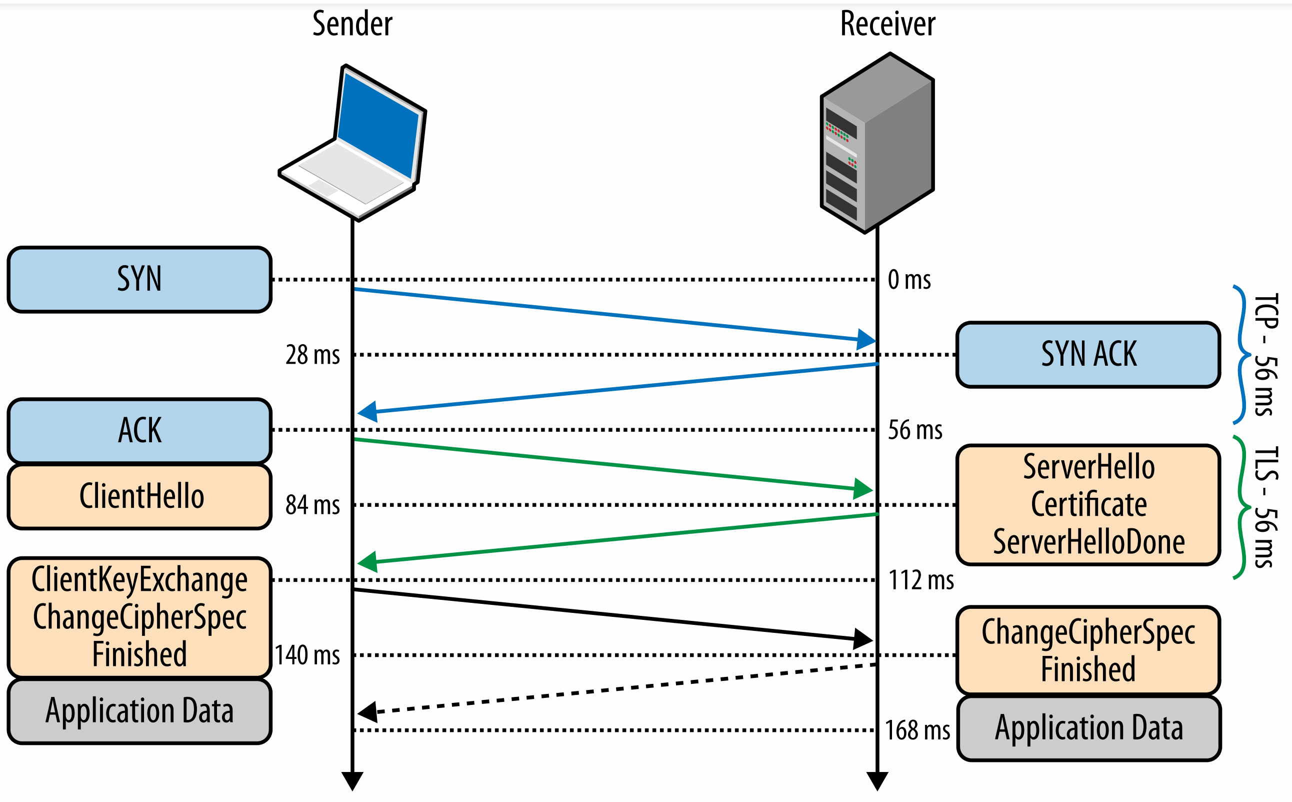


Figure 4-10. TLS handshake with False Start

False Start does not modify the TLS handshake protocol, rather it only affects the protocol timing of when the application data can be sent. Intuitively, once the client has sent the `ClientKeyExchange` record, it already knows the encryption key and can begin transmitting application data—the rest of the handshake is spent confirming that nobody has tampered with the handshake records, and can be done in parallel. As a result, False Start allows us to keep the TLS handshake at one roundtrip regardless of whether we are performing a full or abbreviated handshake.

Deploying TLS False Start §

Because False Start is only modifying the timing of the handshake protocol, it does not require any updates to the TLS protocol itself and can be implemented unilaterally—i.e., the client can simply begin transmitting encrypted application data sooner. Well, that's the theory.

In practice, even though TLS False Start should be backwards compatible with all existing TLS clients and servers, enabling it by default for all TLS connections proved to be problematic due to some poorly implemented servers. As a result, all modern browsers are capable of using TLS False Start, but will only do so when certain conditions are met by the server:

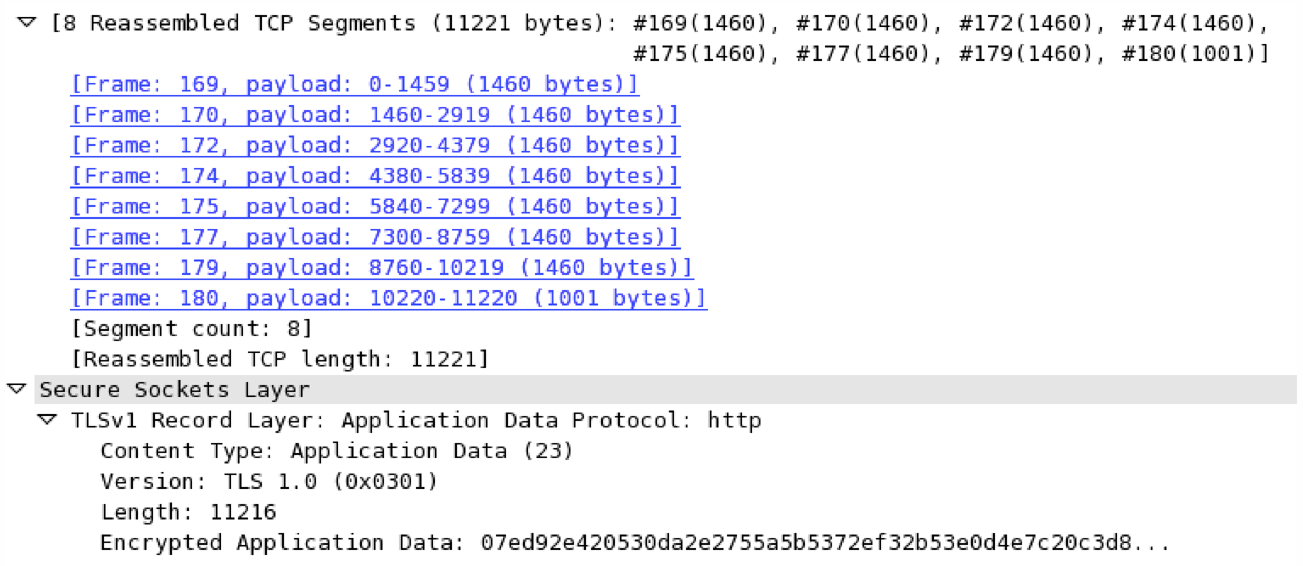
- Chrome and Firefox require an ALPN protocol advertisement to be present in the server handshake, and that the cipher suite chosen by the server enables forward secrecy.
- Safari requires that the cipher suite chosen by the server enables forward secrecy.
- Internet Explorer uses a combination of a blacklist of known sites that break when TLS False Start is enabled, and a timeout to repeat the handshake if the TLS False Start handshake

To enable TLS False Start across all browsers the server should advertise a list of supported protocols via the ALPN extension—e.g., "h2, http/1.1"—and be configured to support and prefer cipher suites that enable forward secrecy.

Optimize TLS Record Size §

All application data delivered via TLS is transported within a record protocol (Figure 4-8). The maximum size of each record is 16 KB, and depending on the chosen cipher, each record will add anywhere from 20 to 40 bytes of overhead for the header, MAC, and optional padding. If the record then fits into a single TCP packet, then we also have to add the IP and TCP overhead: 20-byte header for IP, and 20-byte header for TCP with no options. As a result, there is potential for 60 to 100 bytes of overhead for each record. For a typical maximum transmission unit (MTU) size of 1,500 bytes on the wire, this packet structure translates to a minimum of 6% of framing overhead.

The smaller the record, the higher the framing overhead. However, simply increasing the size of the record to its maximum size (16 KB) is not necessarily a good idea. If the record spans multiple TCP packets, then the TLS layer must wait for all the TCP packets to arrive before it can decrypt the data (Figure 4-11). If any of those TCP packets get lost, reordered, or throttled due to congestion control, then the individual fragments of the TLS record will have to be buffered before they can be decoded, resulting in additional latency. In practice, these delays can create significant bottlenecks for the browser, which prefers to consume data in a streaming fashion.



```
▽ [8 Reassembled TCP Segments (11221 bytes): #169(1460), #170(1460), #172(1460), #174(1460),
                                             #175(1460), #177(1460), #179(1460), #180(1001)]
    [Frame: 169, payload: 0-1459 (1460 bytes)]
    [Frame: 170, payload: 1460-2919 (1460 bytes)]
    [Frame: 172, payload: 2920-4379 (1460 bytes)]
    [Frame: 174, payload: 4380-5839 (1460 bytes)]
    [Frame: 175, payload: 5840-7299 (1460 bytes)]
    [Frame: 177, payload: 7300-8759 (1460 bytes)]
    [Frame: 179, payload: 8760-10219 (1460 bytes)]
    [Frame: 180, payload: 10220-11220 (1001 bytes)]
    [Segment count: 8]
    [Reassembled TCP length: 11221]
▽ Secure Sockets Layer
  ▽ TLSv1 Record Layer: Application Data Protocol: http
      Content Type: Application Data (23)
      Version: TLS 1.0 (0x0301)
      Length: 11216
      Encrypted Application Data: 07ed92e420530da2e2755a5b5372ef32b53e0d4e7c20c3d8...
```

Figure 4-11. WireShark capture of 11,211-byte TLS record split over 8 TCP segments

Small records incur overhead, large records incur latency, and there is no one value for the "optimal" record size. Instead, for web applications, which are consumed by the browser, the best strategy is to dynamically adjust the record size based on the state of the TCP connection:

one TLS record, and the TLS record should occupy the full maximum segment size (MSS) allocated by TCP.

- When the connection congestion window is large and if we are transferring a large stream (e.g., streaming video), the size of the TLS record can be increased to span multiple TCP packets (up to 16KB) to reduce framing and CPU overhead on the client and server.

Note

If the TCP connection has been idle, and even if Slow-Start Restart is disabled on the server, the best strategy is to decrease the record size when sending a new burst of data: the conditions may have changed since last transmission, and our goal is to minimize the probability of buffering at the application layer due to lost packets, reordering, and retransmissions.

Using a dynamic strategy delivers the best performance for interactive traffic: small record size eliminates unnecessary buffering latency and improves the *time-to-first-{HTML byte, …, video frame}*, and a larger record size optimizes throughput by minimizing the overhead of TLS for long-lived streams.

To determine the optimal record size for each state let's start with the initial case of a new or idle TCP connection where we want to avoid TLS records from spanning multiple TCP packets:

- Allocate 20 bytes for IPv4 framing overhead and 40 bytes for IPv6.
- Allocate 20 bytes for TCP framing overhead.
- Allocate 40 bytes for TCP options overhead (timestamps, SACKs).

Assuming a common 1,500-byte starting MTU, this leaves 1,420 bytes for a TLS record delivered over IPv4, and 1,400 bytes for IPv6. To be future-proof, use the IPv6 size, which leaves us with 1,400 bytes for each TLS record, and adjust as needed if your MTU is lower.

Next, the decision as to when the record size should be increased and reset if the connection has been idle, can be set based on pre-configured thresholds: increase record size to up to 16 KB after X KB of data have been transferred, and reset the record size after Y milliseconds of idle time.

Typically, configuring the TLS record size is not something we can control at the application layer. Instead, often this is a setting and sometimes a compile-time constant for your TLS server. Check the documentation of your server for details on how to configure these values.

TLS optimizations at Google §

As of early 2014, Google's servers use small TLS records that fit into a single TCP segment for the first 1 MB of data, increase record size to 16 KB after that to optimize throughput, and then reset

Similarly, if your servers are handling a large number of TLS connections, then minimizing memory usage per connection can be a vital optimization. By default, popular libraries such as OpenSSL will allocate up to 50 KB of memory per connection, but as with the record size, it may be worth checking the documentation or the source code for how to adjust this value. Google's servers reduce their OpenSSL buffers down to about 5 KB.

Optimize the Certificate Chain §

Verifying the chain of trust requires that the browser traverse the chain, starting from the site certificate, and recursively verify the certificate of the parent until it reaches a trusted root. Hence, it is critical that the provided chain includes all the intermediate certificates. If any are omitted, the browser will be forced to pause the verification process and fetch the missing certificates, adding additional DNS lookups, TCP handshakes, and HTTP requests into the process.

Note

How does the browser know from where to fetch the missing certificates? Each child certificate typically contains a URL for the parent. If the URL is omitted and the required certificate is not included, then the verification will fail.

Conversely, do not include unnecessary certificates, such as the trusted roots in your certificate chain—they add unnecessary bytes. Recall that the server certificate chain is sent as part of the TLS handshake, which is likely happening over a new TCP connection that is in the early stages of its slow-start algorithm. If the certificate chain size exceeds TCP's initial congestion window, then we will inadvertently add additional roundtrips to the TLS handshake: certificate length will overflow the congestion window and cause the server to stop and wait for a client ACK before proceeding.

In practice, the size and depth of the certificate chain was a much bigger concern and problem on older TCP stacks that initialized their initial congestion window to 4 TCP segments—see Slow-Start. For newer deployments, the initial congestion window has been raised to 10 TCP segments and should be more than sufficient for most certificate chains.

That said, verify that your servers are using the latest TCP stack and settings, and optimize and reduce the size of your certificate chain. Sending fewer bytes is always a good and worthwhile optimization.

Configure OCSP Stapling §

Every new TLS connection requires that the browser must verify the signatures of the sent certificate chain. However, there is one more critical step that we can't forget: the browser also needs to verify that the certificates have not been revoked.

To verify the status of the certificate the browser can use one of several methods: Certificate Revocation List (CRL), Online Certificate Status Protocol (OCSP), or OCSP Stapling. Each method has its own limitations, but OCSP Stapling provides, by far, the best security and performance guarantees-refer to earlier sections for details. Make sure to configure your servers to include (staple) the OCSP response from the CA to the provided certificate chain. Doing so allows the browser to perform the revocation check without any extra network roundtrips and with improved security guarantees.

- OCSP responses can vary from 400 to 4,000 bytes in size. Stapling this response to your certificate chain will increase its size—pay close attention to the total size of the certificate chain, such that it doesn't overflow the initial congestion window for new TCP connections.
- Current OCSP Stapling implementations only allow a single OCSP response to be included, which means that the browser may have to fallback to another revocation mechanism if it needs to validate other certificates in the chain—reduce the length of your certificate chain. In the future, OCSP Multi-Stapling should address this particular problem.

Most popular servers support OCSP stapling. Check the relevant documentation for support and configuration instructions. Similarly, if using or deciding on a CDN, check that their TLS stack supports and is configured to use OCSP stapling.

Enable HTTP Strict Transport Security (HSTS) §

HTTP Strict Transport Security is an important security policy mechanism that allows an origin to declare access rules to a compliant browser via a simple HTTP header—e.g., "*Strict-Transport-Security: max-age=31536000*". Specifically, it instructs the user-agent to enforce the following rules:

- All requests to the origin should be sent over HTTPS. This includes both navigation and all other same-origin subresource requests—e.g. if the user types in a URL without the https prefix the user agent should automatically convert it to an https request; if a page contains a reference to a non-https resource, the user agent should automatically convert it to request the https version.
- If a secure connection cannot be established, the user is not allowed to circumvent the warning and request the HTTP version—i.e. the origin is HTTPS-only.
- *max-age* specifies the lifetime of the specified HSTS ruleset in seconds (e.g., `max-age=31536000` is equal to a 365-day lifetime for the advertised policy).
- *includeSubdomains* indicates that the policy should apply to all subdomains of the current origin.

HSTS converts the origin to an HTTPS-only destination and helps protect the application from a variety of passive and active network attacks. As an added bonus, it also offers a nice performance optimization by eliminating the need for HTTP-to-HTTPS redirects: the client automatically rewrites all requests to the secure origin before they are dispatched!

Note

Make sure to thoroughly test your TLS deployment before enabling HSTS. Once the policy is cached by the client, failure to negotiate a TLS connection will result in a hard-fail—i.e. the user will see the browser error page and won't be allowed to proceed. This behavior is an explicit and necessary design choice to prevent network attackers from tricking clients into accessing your site without HTTPS.

HSTS Preload List §

The HSTS mechanism leaves the very first request to an origin unprotected from active attacks—e.g. a malicious party could downgrade the client's request and prevent it from registering the HSTS policy. To address this, most browsers provide a separate "HSTS preload list" mechanism that allows an origin to request to be included in the list of HSTS-enabled sites that ships with the browser.

Once you're confident in your HTTPS deployment, consider submitting your site to the HSTS preload list via hstspreload.appspot.com .

Enable HTTP Public Key Pinning (HPKP) §

One of the shortcomings of the current system—as discussed in Chain of Trust and Certificate Authorities—is our reliance on a large number of trusted Certificate Authorities (CA's). On the one hand, this is convenient, because it means that we can obtain a valid certificate from a wide pool of entities. However, it also means that any one of these entities is also able to issue a valid certificate for our, and any other, origin without their explicit consent.

Note

The compromise of the DigiNotar certificate authority is one of several high-profile examples where an attacker was able to issue and use fake—but valid—certificates against hundreds of high profile sites.

Public Key Pinning enables a site to send an HTTP header that instructs the browsers to remember ("pin") one or more certificates in its certificate chain. By doing so, it is able to scope

- The origin can pin it’s leaf certificate. This is the most secure strategy because you are, in effect, hard-coding a small set of specific certificate signatures that should be accepted by the browser.
- The origin can pin one of the parent certificates in the certificate chain. For example, the origin can pin the intermediate certificate of its CA, which tells the browser that, for this particular origin, it should only trust certificates signed by that particular certificate authority.

Picking the right strategy for which certificates to pin, which and how many backups to provide, duration, and other criteria for deploying HPKP are important, nuanced, and beyond the scope of our discussion. Consult your favorite search engine, or your local security guru, for more information.

Note

HPKP also exposes a "report only" mode that does not enforce the provided pin but is able to report detected failures. This can be a great first step towards validating your deployment, and serve as a mechanism to detect violations.

Update Site Content to HTTPS §

To get the best security and performance guarantees it is critical that the site actually uses HTTPS to fetch all of its resources. Otherwise, we run into a number of issues that will compromise both, or worse, break the site:

- Mixed "active" content (e.g. scripts and stylesheets delivered over HTTP) will be blocked by the browser and may break the functionality of the site.
- Mixed "passive" content (e.g. images, video, audio, etc., delivered over HTTP) will be fetched, but will allow the attacker to observe and infer user activity, and degrade performance by requiring additional connections and handshakes.

Audit your content and update your resources and links, including third-party content, to use HTTPS. The Content Security Policy (CSP) mechanism can be of great help here, both to identify HTTPS violations and to enforce the desired policies.

```
Content-Security-Policy: upgrade-insecure-requests 1
Content-Security-Policy-Report-Only: default-src https:;
  report-uri https://example.com/reporting/endpoint 2
```

1. Tells the browser to upgrade all (own and third-party) requests to HTTPS.
2. Tells the browser to report any non-HTTPS violations to designated endpoint.

CSP provides a highly configurable mechanism to control which asset are allowed to be used, and how and from where they can be fetched. Make use of these capabilities to protect your site and your users.

Performance Checklist §

As application developers we are shielded from most of the complexity of the TLS protocol—the client and server do most of the hard work on our behalf. However, as we saw in this chapter, this does not mean that we can ignore the performance aspects of delivering our applications over TLS. Tuning our servers to enable critical TLS optimizations and configuring our applications to enable the client to take advantage of such features pays high dividends: faster handshakes, reduced latency, better security guarantees, and more.

With that in mind, a short checklist to put on the agenda:

- Get best performance from TCP; see Optimizing for TCP.
- Upgrade TLS libraries to latest release, and (re)build servers against them.
- Enable and configure session caching and stateless resumption.
- Monitor your session caching hit rates and adjust configuration accordingly.
- Configure forward secrecy ciphers to enable TLS False Start.
- Terminate TLS sessions closer to the user to minimize roundtrip latencies.
- Use dynamic TLS record sizing to optimize latency and throughput.
- Audit and optimize the size of your certificate chain.
- Configure OCSP stapling.
- Configure HSTS and HPKP.
- Configure CSP policies.
- Enable HTTP/2; see HTTP/2.

Testing and Verification §

Finally, to verify and test your configuration, you can use an online service, such as the Qualys SSL Server Test to scan your public server for common configuration and security flaws. Additionally, you should familiarize yourself with the `openssl` command-line interface, which will help you inspect the entire handshake and configuration of your server locally.

```
$> openssl s_client -state -CAfile root.ca.crt -connect igvita.com:443

CONNECTED(00000003)
```

```
SSL_connect:SSLv2/v3 write client hello A
SSL_connect:SSLv3 read server hello A
depth=2 /C=IL/O=StartCom Ltd./OU=Secure Digital Certificate Signing
        /CN=StartCom Certification Authority
verify return:1
depth=1 /C=IL/O=StartCom Ltd./OU=Secure Digital Certificate Signing
        /CN=StartCom Class 1 Primary Intermediate Server CA
verify return:1
depth=0 /description=ABjQuqt3nPv7ebEG/C=US
        /CN=www.igvita.com/emailAddress=ilya@igvita.com
verify return:1
SSL_connect:SSLv3 read server certificate A
SSL_connect:SSLv3 read server done A 1
SSL_connect:SSLv3 write client key exchange A
SSL_connect:SSLv3 write change cipher spec A
SSL_connect:SSLv3 write finished A
SSL_connect:SSLv3 flush data
SSL_connect:SSLv3 read finished A
---
Certificate chain 2
 0 s:/description=ABjQuqt3nPv7ebEG/C=US
     /CN=www.igvita.com/emailAddress=ilya@igvita.com
   i:/C=IL/O=StartCom Ltd./OU=Secure Digital Certificate Signing
     /CN=StartCom Class 1 Primary Intermediate Server CA
 1 s:/C=IL/O=StartCom Ltd./OU=Secure Digital Certificate Signing
     /CN=StartCom Class 1 Primary Intermediate Server CA
   i:/C=IL/O=StartCom Ltd./OU=Secure Digital Certificate Signing
     /CN=StartCom Certification Authority
---
Server certificate
-----BEGIN CERTIFICATE-----
... snip ...
---
No client certificate CA names sent
---
SSL handshake has read 3571 bytes and written 444 bytes 3
---
New, TLSv1/SSLv3, Cipher is RC4-SHA
Server public key is 2048 bit
Secure Renegotiation IS supported
Compression: NONE
Expansion: NONE
SSL-Session:
    Protocol  : TLSv1
    Cipher    : RC4-SHA
    Session-ID: 269349C84A4702EFA7 ... 4
    Session-ID-ctx:
    Master-Key: 1F5F5F33D50BE6228A ...
    Key-Arg   : None
    Start Time: 1354037095
    Timeout   : 300 (sec)
    Verify return code: 0 (ok)
---
```

1. Client completed verification of received certificate chain.
2. Received certificate chain (two certificates).
3. Size of received certificate chain.
4. Issued session identifier for stateful TLS resume.

In the preceding example, we connect to igvita.com on the default TLS port (443), and perform the TLS handshake. Because the `s_client` makes no assumptions about known root certificates, we manually specify the path to the root certificate which, at the time of writing, is the StartSSL Certificate Authority for the example domain. Your browser already has common root certificates and is thus able to verify the chain, but `s_client` makes no such assumptions. Try omitting the root certificate, and you will see a verification error in the log.

Inspecting the certificate chain shows that the server sent two certificates, which added up to 3,571 bytes. Also, we can see the negotiated TLS session variables—chosen protocol, cipher, key—and we can also see that the server issued a session identifier for the current session, which may be resumed in the future.

« Back to the Table of Contents