Because the ugly truth about the existing SSL/TLS protocols, is that, even with lots of clever cryptography being used, the current system we have for validating trust is based on often unknown third parties, with a single point of failure.

What Is SSL?

SSL (Secure Sockets Layer) is a standard security technology for establishing an encrypted link between a server and a client—typically a web server (website) and a browser; or a mail server and a mail client (e.g., Outlook).

SSL allows sensitive information such as credit card numbers, social security numbers, and login credentials to be transmitted securely. Normally, data sent between browsers and web servers is sent in plain text—leaving you vulnerable to eavesdropping. If an attacker is able to intercept all data being sent between a browser and a web server they can see and use that information.

More specifically, SSL is a security protocol. Protocols describe how algorithms should be used; in this case, the SSL protocol determines variables of the encryption for both the link and the data being transmitted.

SSL secures millions of peoples' data on the Internet every day, especially during online transactions or when transmitting confidential information. Internet users have come to associate their online security with the lock icon that comes with an SSL-secured website or green address bar that comes with an extended validation SSL-secured website. SSL-secured websites also begin with https rather than http.

What is an SSL Certificate and How Does it Work?

SSL Certificates have a key pair: a public and a private key. These keys work together to establish an encrypted connection. The certificate also contains what is called the "subject," which is the identity of the certificate/website owner.

To get a certificate, you must create a Certificate Signing Request (CSR) on your server. This process creates a private key and public key on your server. The CSR data file that you send to the SSL Certificate issuer (called a Certificate Authority or CA) contains the public key. The CA uses the CSR data file to create a data structure to match your private key without compromising the key itself. The CA never sees the private key.

Once you receive the SSL Certificate, you install it on your server. You also install an intermediate certificate that establishes the credibility of your SSL Certificate by tying it to your CA's root certificate. The instructions for installing and testing your certificate will be different depending on your server.

In the image below, you can see what is called the certificate chain. It connects your server certificate to your CA's (in this case DigiCert's) root certificate through an intermediate certificate.

The most important part of an SSL Certificate is that it is digitally signed by a trusted CA like DigiCert. Anyone can create a certificate, but browsers only trust certificates that come from an organization on their list of trusted CAs. Browsers come with a pre-installed list of trusted

CAs, known as the Trusted Root CA store. In order to be added to the Trusted Root CA store and thus become a Certificate Authority, a company must comply with and be audited against security and authentication standards established by the browsers.

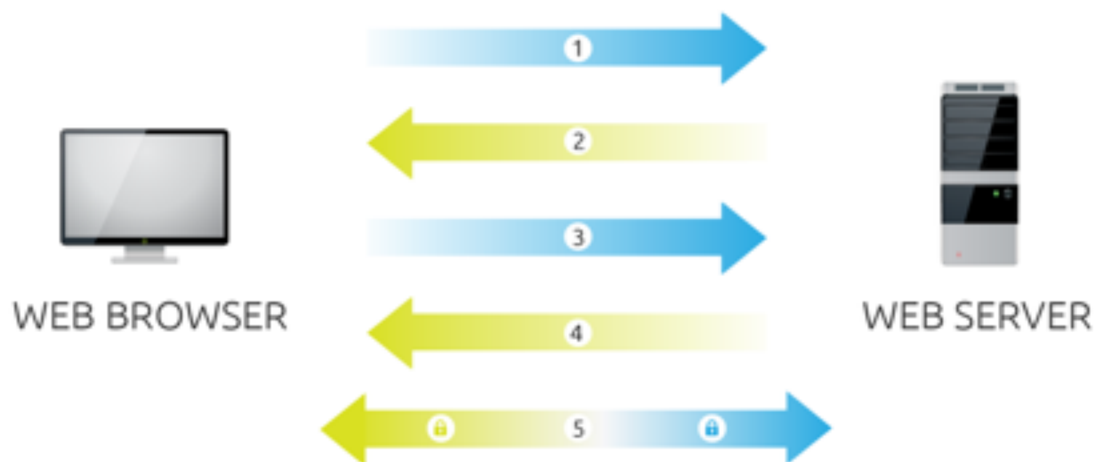
An SSL Certificate issued by a CA to an organization and its domain/website verifies that a trusted third party has authenticated that organization's identity. Since the browser trusts the CA, the browser now trusts that organization's identity too. The browser lets the user know that the website is secure, and the user can feel safe browsing the site and even entering their confidential information.

How Does the SSL Certificate Create a Secure Connection?

When a browser attempts to access a website that is secured by SSL, the browser and the web server establish an SSL connection using a process called an "SSL Handshake" (see diagram below). Note that the SSL Handshake is invisible to the user and happens instantaneously.

Essentially, three keys are used to set up the SSL connection: the public, private, and session keys. Anything encrypted with the public key can only be decrypted with the private key, and vice versa.

Because encrypting and decrypting with private and public key takes a lot of processing power, they are only used during the SSL Handshake to create a symmetric session key. After the secure connection is made, the session key is used to encrypt all transmitted data.



1. **Browser** connects to a web server (website) secured with SSL (https). Browser requests that the server identify itself.

2. **Server** sends a copy of its SSL Certificate, including the server's public key.
3. **Browser** checks the certificate root against a list of trusted CAs and that the certificate is unexpired, unrevoked, and that its common name is valid for the website that it is connecting to. If the browser trusts the certificate, it creates, encrypts, and sends back a symmetric session key using the server's public key.
4. **Server** decrypts the symmetric session key using its private key and sends back an acknowledgement encrypted with the session key to start the encrypted session.
5. **Server** and **Browser** now encrypt all transmitted data with the session key.

The SSL protocol was originally developed at Netscape to enable e-commerce 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.

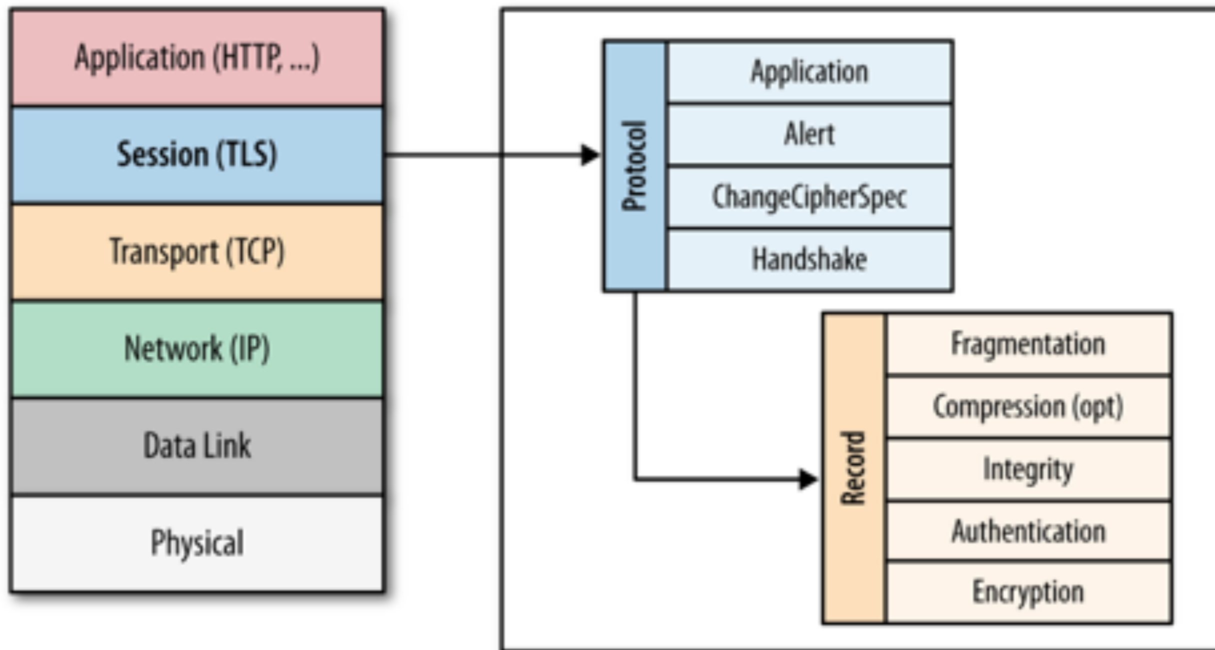


Figure 4-1. Transport Layer Security (TLS)

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 Netscape, the IETF formed an effort to standardize the protocol, resulting in RFC 2246, which became known as TLS 1.0 and is effectively an upgrade to SSL 3.0:

> The differences between this protocol and SSL 3.0 are not dramatic, but they are significant to preclude interoperability between TLS 1.0 and SSL 3.0.

Since the publication of TLS 1.0 in January 1999, two new versions have been produced by the IETF working group to address found security flaws, as well as to extend the capabilities of the protocol: TLS 1.1 in April 2006 and TLS 1.2 in August 2008. Internally the SSL 3.0 implementation, as well as all subsequent TLS versions, are very similar, and many clients continue to support SSL 3.0 and TLS 1.0 to this day, although there are very good reasons to upgrade to newer versions to protect users from known attacks!

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 computer 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 cipher suites 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 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.

As part of the TLS handshake, the protocol also allows both connection peers to authenticate their identity. When used in the browser, this authentication mechanism allows the client to verify that the 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 his 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 cipher suites, 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 small problem for anyone who tries to deviate from the HTTP protocol in any way: some proxy servers may simply relay 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 traffic where there is none.

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

Due to these behaviors, new protocols and extensions to HTTP, such as WebSocket, SPDY, and others, often 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. This solves the immediate problem, but it does have a real downside of not being able to leverage the intermediaries, many of which provide useful services: authentication, caching, security scanning, and so on.

If you have ever wondered why most WebSocket guides will tell you to use HTTPS to deliver data to mobile clients, this is why. As times passes and the intermediaries are upgraded to recognize new protocols, the requirement for HTTPS deployment will also become less relevant—that is, unless your session actually needs the encryption, authentication, and integrity provided by TLS!

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 cipher suite, and verify certificates if necessary. Unfortunately,

each of these steps requires new packet roundtrips (Figure 1) between the client and the server, which adds startup latency to all TLS connections.

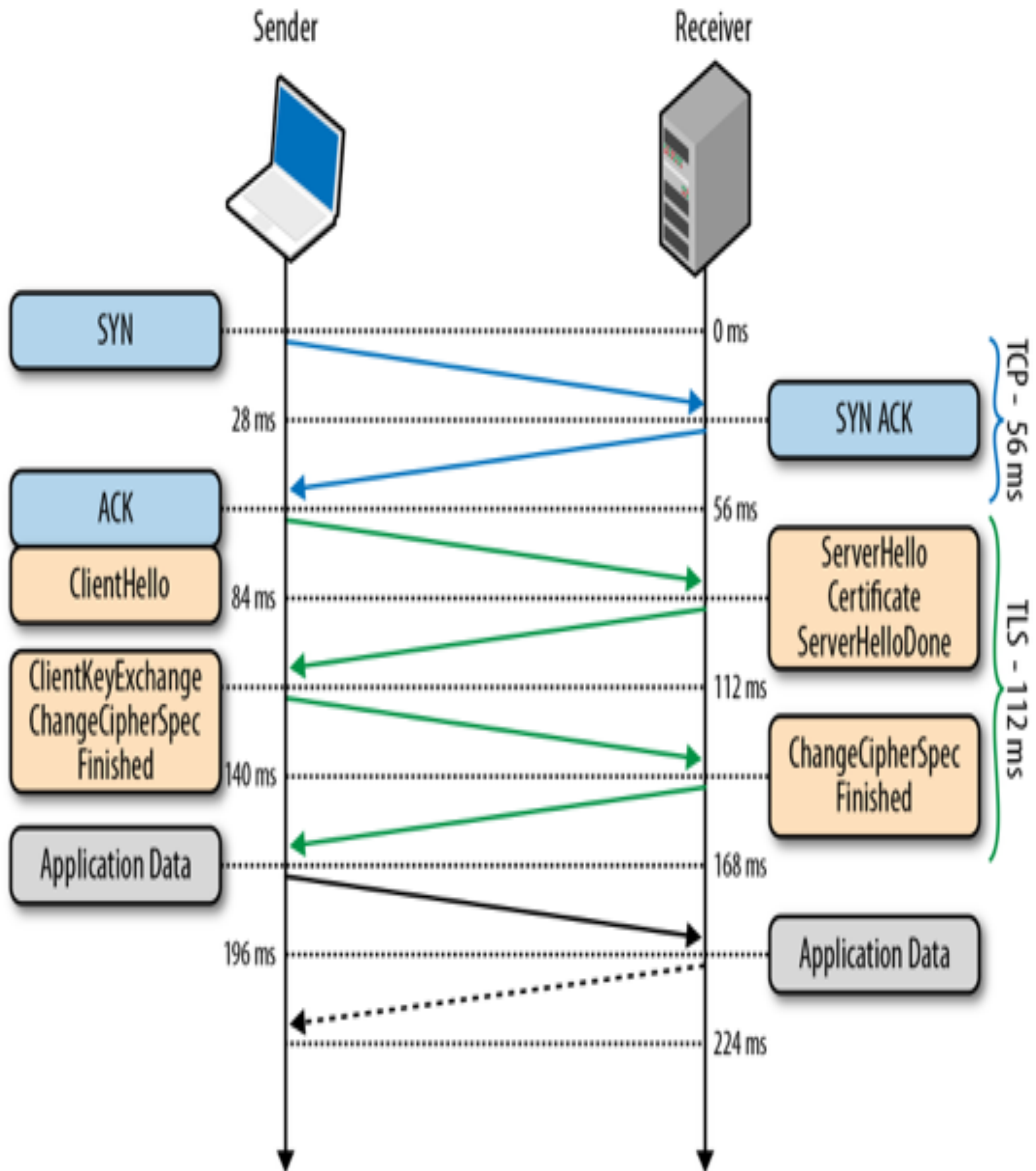


Figure 1. TLS handshake protocol

Figure 1 assumes the same 28 millisecond one-way "light in fiber" delay between New York and London as used in previous TCP connection establishment examples.

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.
56ms	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 cipher suites, and other TLS options it may want to use.
84ms	The server picks the TLS protocol version for further communication, decides on a cipher suite 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.
112ms	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.
140ms	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.
168ms	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.

Negotiating a secure TLS tunnel is a complicated process, and there are many ways to get it wrong. The good news is all the work just shown will be done for us by the server and the browser, and all we need to do is provide and configure the certificates.

Having said that, while our web applications do not have to drive the preceding exchange, it is nonetheless important to realize that every TLS connection will require up to two extra roundtrips on top of the TCP handshake—that's a long time to wait before any application data can be exchanged! If not managed carefully, delivering application data over TLS can add hundreds, if not thousands of milliseconds of network latency.

Optimizing TLS handshake with Session Resumption and False Start

New TLS connections require two roundtrips for a "full handshake" and CPU resources to verify and compute the parameters for the ensuing session. However, the good news is that we don't have to repeat the "full handshake" in every case:

- If the client has previously communicated with the server, an "abbreviated handshake" can be used, which requires one roundtrip and allows the client and server to reduce

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

- False Start is an optional 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 Finishedmessages are sent, but without waiting for the other side to do the same. This optimization reduces new handshake overhead to one roundtrip; see “TLS False Start”.

For best results, both optimizations should be used together to provide a single roundtrip handshake for new and returning visitors, plus computational savings for sessions that can be resumed based on previously negotiated session parameters.

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 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.

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 every key exchange and discard the previous keys. As a result, because the ephemeral keys are 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 ephemeral key, for those session does not help attacker decrypt any of the previous sessions!

The combination of Diffie-Hellman and the use of ephemeral session keys are what enables "Forward Secrecy": even if an attacker gains access to the server's private key they are not able to passively listen in on the active session, nor can they decrypt previously recorded sessions.

Despite the historical dominance of the RSA handshake, it is now being actively phased out to address the weaknesses we saw above: all the popular browsers prefer ciphers that enable forward secrecy (i.e. Diffie-Hellman key exchange), and may only enable certain protocol optimizations when forward secrecy is available. Long story short, consult your server documentation on how to enable forward secrecy.

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 marketing or an outdated benchmark! Always run the performance tests on your own hardware and refer to "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".

For a hands-on example of HTTP Upgrade flow, 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 and arbitrary application protocols. However, while use of TLS addresses reliability, we still need a way to negotiate the protocol!

An HTTPS session could, of course, reuse the HTTP Upgrade mechanism to perform the require negotiation, but this would result in another full roundtrip of latency. What if we could negotiate the protocol as part of the TLS handshake itself?

As the name implies, Application Layer Protocol Negotiation (ALPN) is a TLS extension that introduces support for application protocol negotiation into the TLS handshake (Figure 1), thereby eliminating the need for an extra roundtrip required by the HTTP Upgrade workflow. Specifically, the process is as follows:

- The client appends a new ProtocolNameList field, containing the list of supported application protocols, into theClientHello 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 complete, both the secure tunnel is established, and the client and server are in

agreement as to which application protocol will be used, they can begin communicating immediately.

ALPN eliminates the need for the HTTP Upgrade exchange, saving an extra roundtrip of latency. However, note that the TLS handshake itself still must be performed; hence ALPN negotiation is not any faster than HTTP Upgrade over an unencrypted channel. Instead, it ensures that application protocol negotiation over TLS is *not any slower*.

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 familiar? The end result is functionally equivalent to ALPN.

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 other words, ALPN is a successor to NPN, and NPN is deprecated. Clients and servers that rely on NPN negotiation will have to be upgraded to use ALPN instead.

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 at the start of the handshake. As a result, a web server can inspect the SNI hostname, select the appropriate certificate, and continue the handshake.

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, many 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 these older 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 an ability 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.

Internally, the server could then maintain a cache of session IDs and the negotiated session parameters for each peer. In turn, the client could then also store the session ID information and include the ID in the "ClientHello" message for a subsequent session, which serves as an indication to the server that the client still remembers the negotiated cipher suite and keys from previous handshake and is able to reuse them. Assuming both the client and the server are able to find the shared session ID parameters in their respective caches, then an abbreviated handshake (Figure 2) can take place. Otherwise, a full new session negotiation is required, which will generate a new session ID.

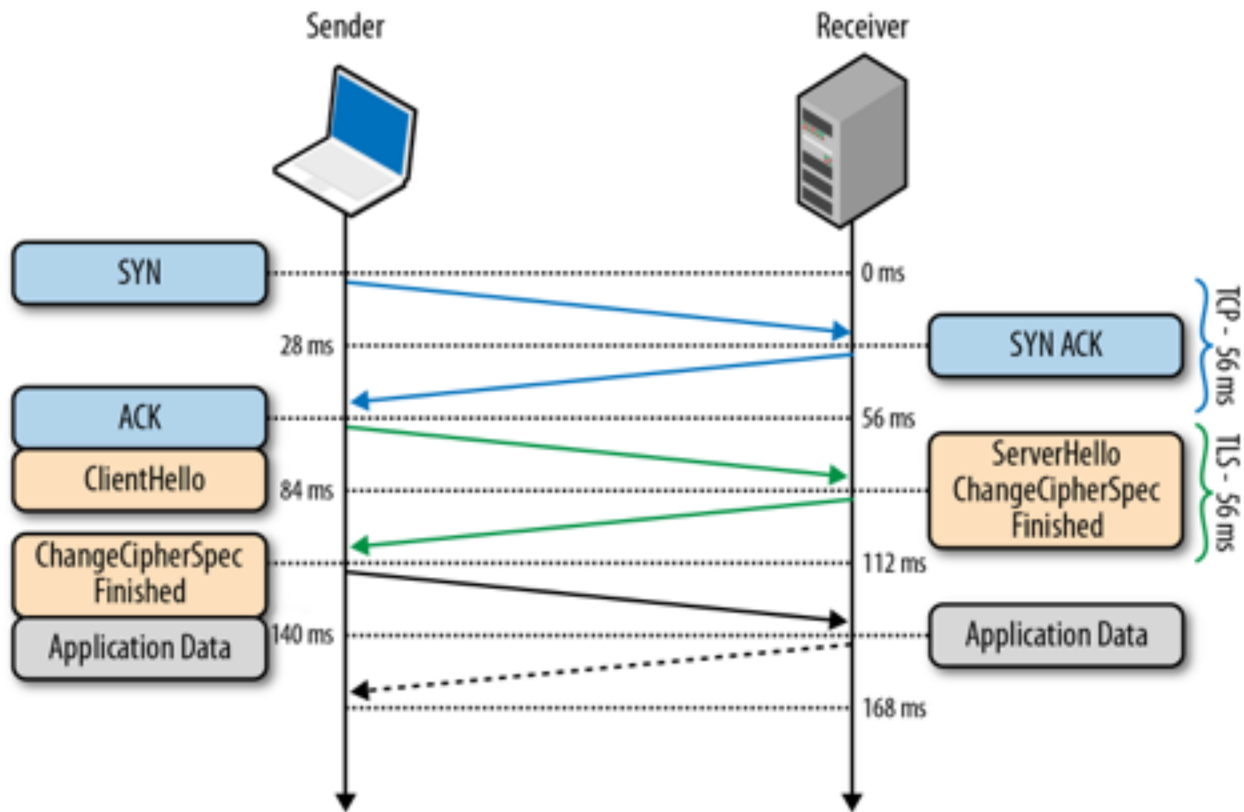


Figure 2. Abbreviated TLS handshake protocol

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.

In practice, most web applications attempt to establish multiple connections to the same host to fetch resources in parallel, which makes session resumption a must-have optimization to reduce latency and computational costs for both sides.

Most modern browsers intentionally wait for the first TLS connection to complete before opening new connections to the same server: subsequent TLS connections can reuse the SSL session parameters to avoid the costly handshake.

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 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 multiserver 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 indicated that it supports Session Tickets, in the last exchange of the full TLS handshake, the server can include a New Session Ticket record, which includes all of the 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 ClientHellomessage 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.

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 may be needed to periodically 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 computer 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.

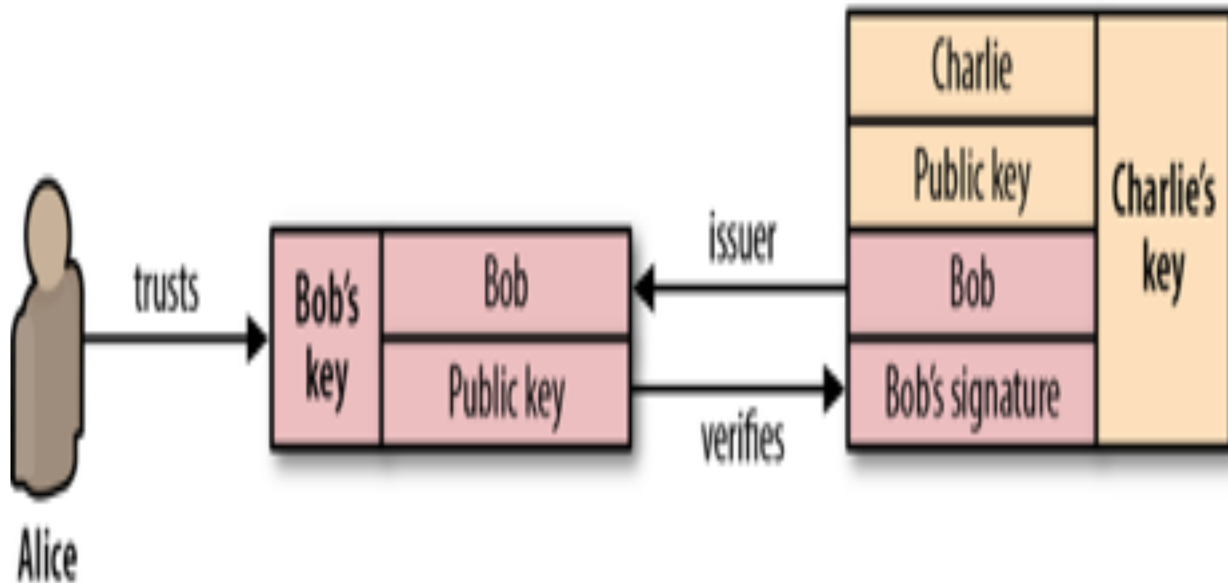


Figure 4-4. Chain of trust for Alice, Bob, and Charlie

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 gets compromised, this allows us to build and grow the list of trusted parties.

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): 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.

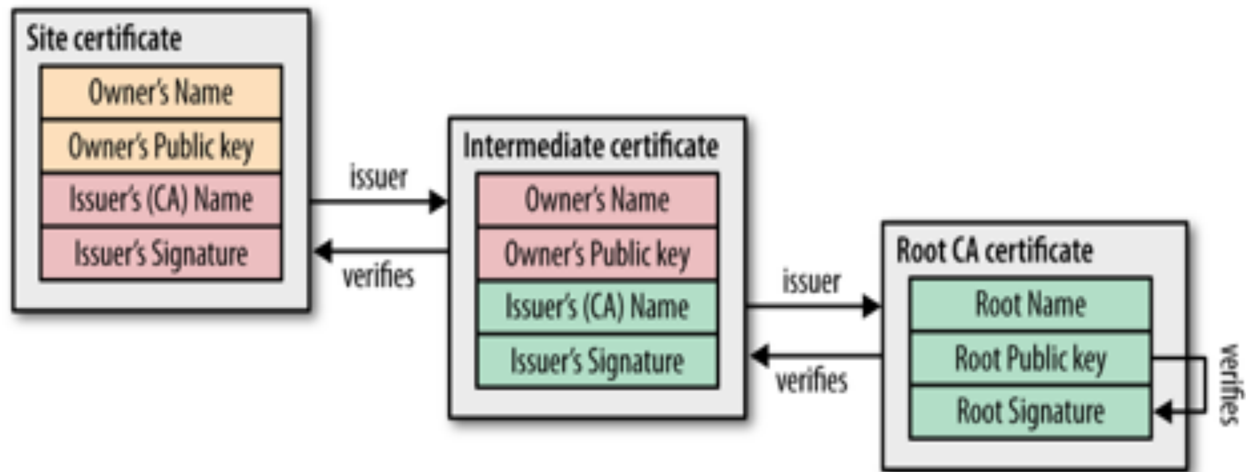


Figure 4. CA signing of digital certificates

Every browser allows you to inspect the chain of trust of your secure connection (Figure 5), usually accessible by clicking on the lock icon beside the URL.

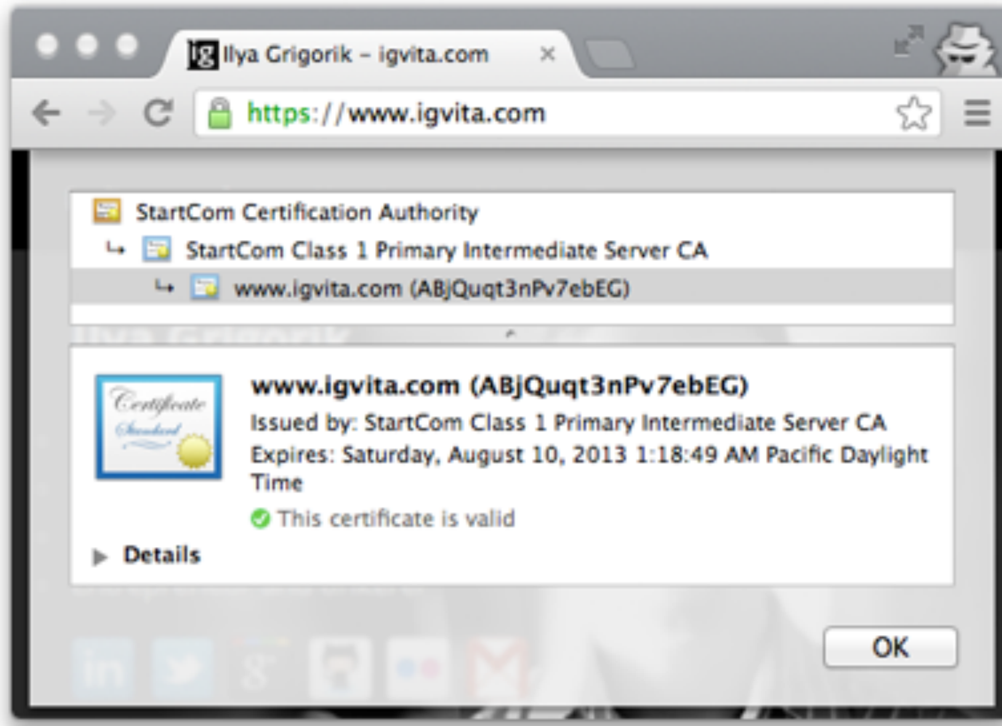


Figure 5. Certificate chain of trust for igvita.com (Google Chrome, v25)

- 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.

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, there are hundreds of well-known and trusted certificate authorities, which is also a common complaint against the system. The large number of CAs creates a potentially large attack surface area against the chain of trust in your browser.

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 6) 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 walks up the certificate chain.

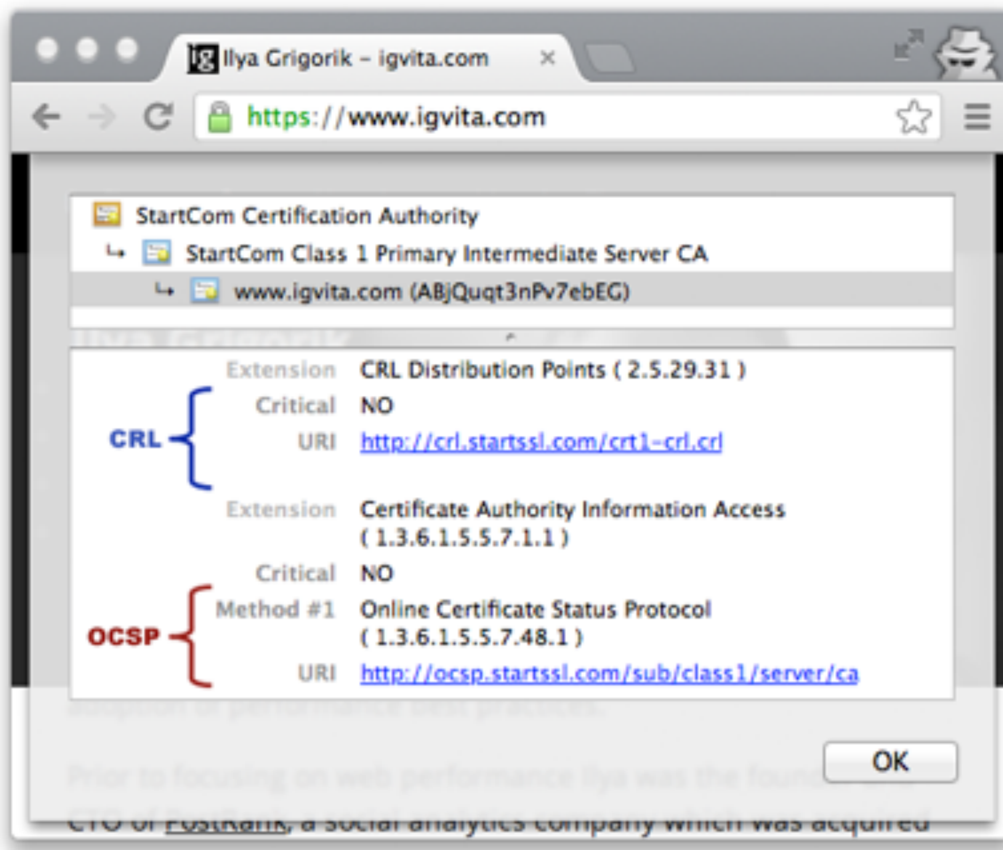


Figure 6. CRL and OCSP instructions for igvita.com (Google Chrome, v25)

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 and check the presence of the serial number within it—if it is present, then it has been revoked.

The CRL file itself can be published periodically or on every update and can be delivered via HTTP, or any other file transfer protocol. The list is also signed by the CA, and is usually allowed to be cached for a specified interval. In practice, this workflow works quite well, but there are instances where CRL mechanism may be insufficient:

- 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.

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, which contains all the revoked serial numbers, OCSP allows the verifier to query the certificate database directly for just the serial number in question while validating the certificate chain.

As a result, the OCSP mechanism should consume much less bandwidth and is able to provide real-time validation. However, no mechanism is perfect! The requirement to perform real-time OCSP queries creates several problems of its own:

- 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.
- The client must block on OCSP requests before proceeding with the navigation.
- Real-time OCSP requests may impair the client's privacy because the CA knows which sites the client is visiting.

In practice, CRL and OCSP mechanisms are complementary, and most certificates will provide instructions and endpoints for both.

The more important part is the client support and behavior: some browsers distribute their own CRL lists, others fetch and cache the CRL files from the CAs. Similarly, some browsers will perform the real-time OCSP check but will differ in their behavior if the OCSP request fails. If you are curious, check your browser and OS certificate revocation settings!

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 10). 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 10. TLS record structure

A typical workflow for delivering application data is as follows:

- Record protocol receives application data.
- Received data is divided into blocks: maximum of 2^{14} bytes, or 16 KB per record.
- Application data is optionally compressed.
- Message authentication code (MAC) or HMAC is added.
- Data 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 data using negotiated cipher, verify MAC, extract and deliver the data to the application above it.

Once again, the good news is 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 you should 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 overhead due to record framing, 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.

Optimizing for TLS

Due to the layered architecture of the network protocols, running an application over TLS is no different from communicating directly over TCP. As such, there are no, or at most minimal, application modifications that you will need to make to deliver it over TLS. That is, assuming you have already applied the "Optimizing for TCP" best practices.

However, what you should investigate are the operational pieces of your TLS deployments: how and where you deploy your servers, size of TLS records and memory buffers, certificate sizes, support for abbreviated handshakes, and so on. Getting these parameters right on your servers can make an enormous positive difference in the user experience, as well as in your operational costs.

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 in "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 the case. Modern hardware has made great improvements to help minimize these costs, and what once required additional hardware can now be done directly on the CPU. Large organizations such as Facebook, Twitter, and Google, which offer TLS to hundreds of millions of users, perform all the necessary TLS negotiation and computation in software and on commodity hardware.

Previous experiences notwithstanding, techniques such as "TLS Session Resumption" are still important optimizations, which will help you decrease the computational costs and latency of public key cryptography performed during the TLS handshake. There is no reason to spend CPU cycles on work that you don't need to do.

Speaking of optimizing CPU cycles, make sure to upgrade your SSL libraries to the latest release, and build your web server or proxy against them! For example, recent versions of

OpenSSL have made significant performance improvements, and chances are your system default OpenSSL libraries are outdated.

Early Termination

The connection setup latency imposed on every TLS connection, new or resumed, is an important area of optimization. First, recall that every TCP connection begins with a three-way handshake (explained in “Three-Way Handshake”), which takes a full roundtrip for the SYN/SYN-ACK packets. Following that, the TLS handshake (explained in “TLS Handshake”) requires up to two additional roundtrips for the full process, or one roundtrip if an abbreviated handshake (explained in “Optimizing TLS handshake with Session Resumption and False Start”) can be used.

In the worst case, before any application data can be exchanged, the TCP and TLS connection setup process will take three roundtrips! Following our earlier example of a client in New York and the server in London, with a roundtrip time of 56 milliseconds (Table 1), this translates to 168 milliseconds of latency for a full TCP and TLS setup, and 112 milliseconds for a TLS session that is resumed. Even worse, the higher the latency between the peers, the worse the penalty, and 56 milliseconds is definitely an optimistic number.

Because all TLS sessions run over TCP, all the advice for “Optimizing for TCP” applies here as well. If TCP connection reuse was an important consideration for unencrypted traffic, then it is a critical optimization for all applications running over TLS—if you can avoid doing the handshake, do so. However, if you have to perform the handshake, then you may want to investigate using the "early termination" technique.

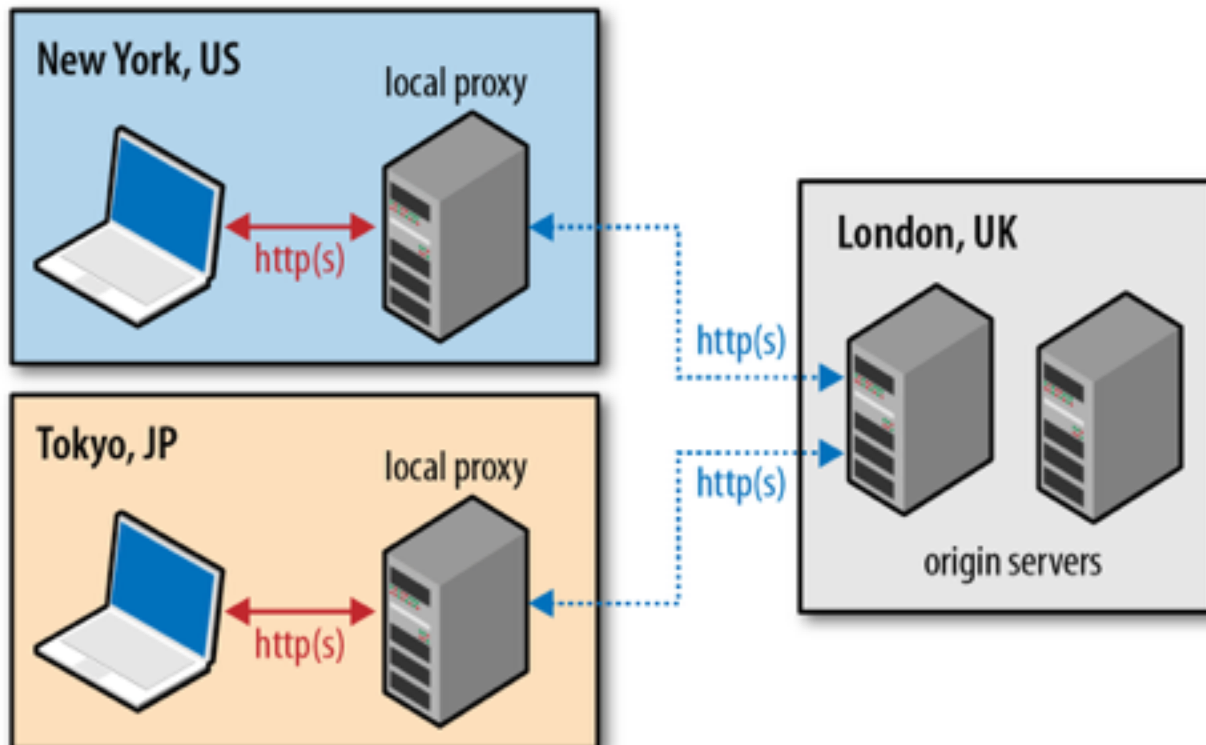


Figure 8. Early termination of client connections

The simplest way to accomplish this is to replicate or cache your data and services on servers around the world instead of forcing every user to traverse across oceans and continental links to the origin servers. Of course, this is precisely the service that many content delivery networks (CDNs) are set up to offer. However, the use case for geo-distributed servers does not stop at optimized delivery of static assets.

A nearby server can also terminate the TLS session, which means that the TCP and TLS handshake roundtrips are much quicker and the total connection setup latency is greatly reduced. In turn, the same nearby server can then establish a pool of long-lived, secure connections to the origin servers and proxy all incoming requests and responses to and from the origin servers.

In a nutshell, move the server closer to the client to accelerate TCP and TLS handshakes! Most CDN providers offer this service, and if you are adventurous, you can also deploy your own infrastructure with minimal costs: spin up cloud servers in a few data centers around the globe, configure a proxy server on each to forward requests to your origin, add geographic DNS load balancing, and you are in business.

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 will allow you to eliminate an entire roundtrip 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! You should double-check and verify your 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 are not exclusive and can work together to provide best performance coverage both for new and older clients.

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 9) when the handshake is only partially complete.

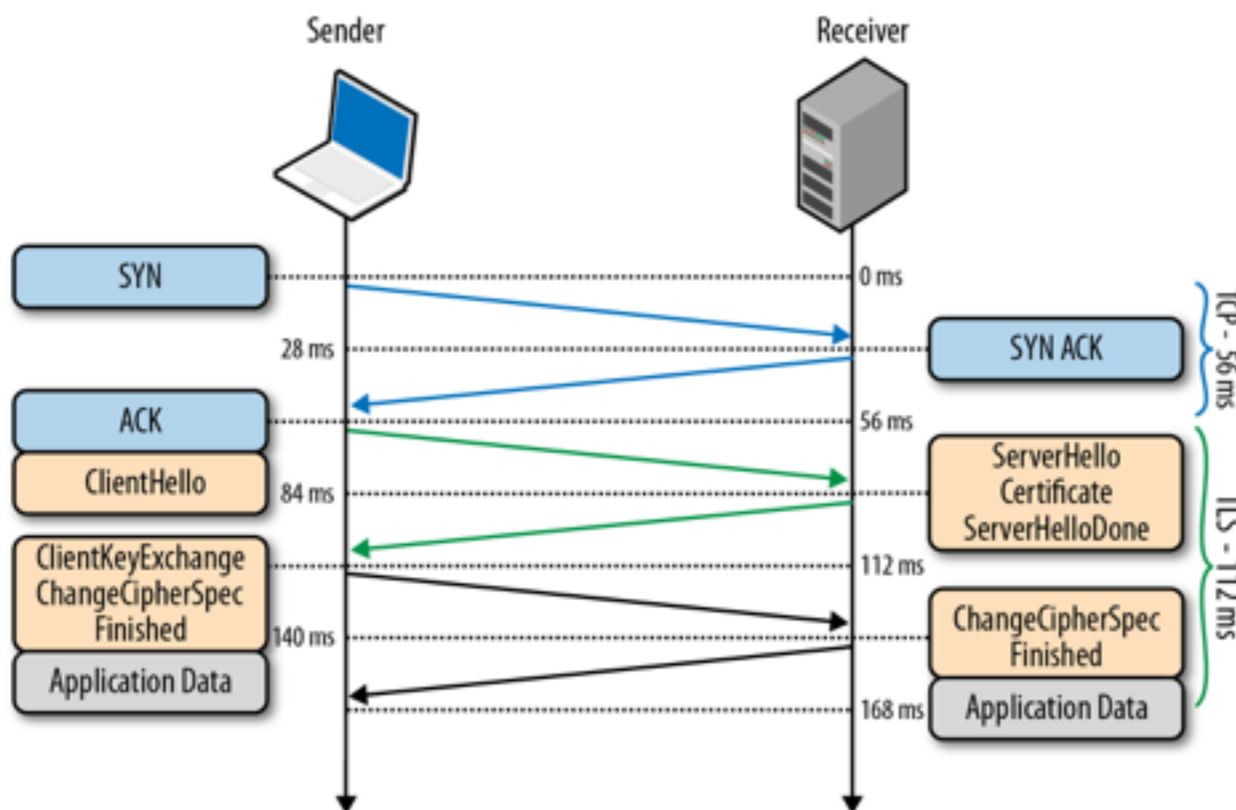


Figure 9. TLS handshake with False Start

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.

TLS Record Size

All application data delivered via TLS is transported within a record protocol (Figure 10). 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 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 byte by byte and as soon as possible.

```
▽ [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 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:

- When the connection is new and TCP congestion window is low, or when the connection has been idle for some time (see “Slow-Start Restart”), each TCP packet should carry exactly 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.

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 payload—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, this is a setting and perhaps even a compile-time constant or flag on your TLS server. For details on how to configure these values, check the documentation of your server.

TLS Compression

A little-known feature of TLS is built-in support for lossless compression of data transferred within the record protocol: the compression algorithm is negotiated during the TLS handshake, and compression is applied prior to encryption of each record. However, in practice, you should disable TLS compression on your server for several reasons:

- The "CRIME" attack, published in 2012, leverages TLS compression to recover secret authentication cookies and allows the attacker to perform session hijacking.
- Transport-level TLS compression is not content aware and will end up attempting to recompress already compressed data (images, video, etc.).

Double compression will waste CPU time on both the server and the client, and the security breach implications are quite serious: disable TLS compression. In practice, most browsers disable support for TLS compression, but you should nonetheless also explicitly disable it in the configuration of your server to protect your users.

> Instead of relying on TLS compression, make sure your server is configured to Gzip all text-based assets and that you are using an optimal compression format for all other media types, such as images, video, and audio.

Certificate-Chain Length

Verifying the chain of trust requires that the browser traverse the chain, starting from the site certificate, and recursively verifying the certificate of the parent until it reaches a trusted root. Hence, the first optimization you should make is to verify that the server does not forget to include all the intermediate certificates when the handshake is performed. If you forget, many browsers will still work, but they will instead be forced to pause the verification and fetch the intermediate certificate on their own, verify it, and then continue. This will most likely require a new DNS lookup, TCP connection, and an HTTP GET request, adding hundreds of milliseconds to your handshake.

> How does the browser know from where to fetch it? The child certificate will usually contain the URL for the parent.

Conversely, make sure you do not include unnecessary certificates in your chain! Or, more generally, you should aim to minimize the size of your certificate chain. Recall that server certificates are sent during the TLS handshake, which is likely running over a new TCP connection that is in the early stages of its slow-start algorithm. If the certificate chain exceeds TCP’s initial congestion window (Figure 12), then we will inadvertently add yet another roundtrip to the handshake: certificate length will overflow the congestion window and cause the server to stop and wait for a client ACK before proceeding.

```
▷ [4 Reassembled TCP Segments (5341 bytes): #98(1402), #99(1460), #101(1176), #102(1303)]
▽ Secure Sockets Layer
  ▽ TLSv1.1 Record Layer: Handshake Protocol: Certificate
      Content Type: Handshake (22)
      Version: TLS 1.1 (0x0302)
      Length: 5327
    ▽ Handshake Protocol: Certificate
        Handshake Type: Certificate (11)
        Length: 5323
        Certificates Length: 5320
      ▷ Certificates (5320 bytes)
```

Figure 12. WireShark capture of a 5,323-byte TLS certificate chain

The certificate chain in Figure 12 is over 5 KB in size, which will overflow the initial congestion window size of older servers and force another roundtrip of delay into the handshake. One possible solution is to increase the initial congestion window; see "Increasing TCP's Initial Congestion Window". In addition, you should investigate if it is possible to reduce the size of the sent certificates:

- Minimize the number of intermediate CAs. Ideally, your sent certificate chain should contain exactly two certificates: your site and the CA's intermediary certificate; use this as a criteria in the selection of your CA. The third certificate, which is the CA root, should already be in the browser's trusted root and hence should not be sent.
- It is not uncommon for many sites to include the root certificate of their CA in the chain, which is entirely unnecessary: if your browser does not already have the certificate in its trust store, then it won't be trusted, and including the root certificate won't change that.
- A carefully managed certificate chain can be as low as 2 or 3 KB in size, while providing all the necessary information to the browser to avoid unnecessary roundtrips or out-of-band requests for the certificates themselves. Optimizing your TLS handshake mitigates a critical performance bottleneck, since every new TLS connection is subject to its overhead.

OCSP Stapling

Every new TLS connection requires that the browser must verify the signatures of the sent certificate chain. However, there is one more step we can't forget: the browser also needs to verify that the certificate is not revoked. To do so, it may periodically download and cache the CRL of the certificate authority, but it may also need to dispatch an OCSP request during

the verification process for a "real-time" check. Unfortunately, the browser behavior for this process varies wildly:

- Some browsers may use their own update mechanism to push updated CRL lists instead of relying on on-demand requests.
- Some browsers may do only real-time OCSP and CRL checks for Extended Validation (EV) certificates.
- Some browsers may block the TLS handshake on either revocation method, others may not, and this behavior will vary by vendor, platform, and version of the browser.

Unfortunately, it is a complicated space with no single best solution. However, one optimization that can be made for some browsers is OCSP stapling: the server can include (staple) the OCSP response from the CA to its certificate chain, allowing the browser to skip the online check. Moving the OCSP fetch to the server allows the server to cache the signed OCSP response and save the extra request for many clients. However, there are also a few things to watch out for:

- OCSP responses can vary from 400 to 4,000 bytes in size. Stapling this response to your certificate chain may once again overflow your TCP congestion window—pay close attention to the total size.
- Only one OCSP response can be included, which may still mean that the browser will have to issue an OCSP request for other intermediate certificates, if it has not been cached already.

Finally, to enable OCSP stapling, you will need a server that supports it. The good news is popular servers such as Nginx, Apache, and IIS meet this criteria. Check the documentation of your own server for support and configuration instructions.

HTTP Strict Transport Security (HSTS)

HTTP Strict Transport Security is a security policy mechanism that allows the server 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.
- All insecure links and client requests should be automatically converted to HTTPS on the client before the request is sent.
- In case of a certificate error, an error message is displayed, and the user is not allowed to circumvent the warning.

- *max-age* specifies the lifetime of the specified HSTS ruleset in seconds (e.g., max-age=31536000 is equal to a 365-day cache lifetime).
- Optionally, the UA can be instructed to remember ("pin") the fingerprint of a host in the specified certificate chain for future access, effectively limiting the scope of authorities who can authenticate the certificate.

HSTS converts the origin to an HTTPS-only destination and helps protect the application from a variety of passive and active network attacks against the user. Performance wise, it also helps eliminate unnecessary HTTP-to-HTTPS redirects by shifting this responsibility to the client, which will automatically rewrite all links to HTTPS.

Attacks against SSL/TLS

BEAST (**"Browser Exploit Against SSL/TLS, SSL")** Beast is an exploit first, revealed in late September 2011, that leverages weaknesses in cipher block chaining (CBC) to exploit the Secure Sockets Layer (SSL) protocol. The CBC vulnerability can enable man-in-the-middle (MITM) attacks against SSL in order to silently decrypt and obtain authentication tokens, providing hackers with access to the data passed between a Web server and the browser accessing the server.

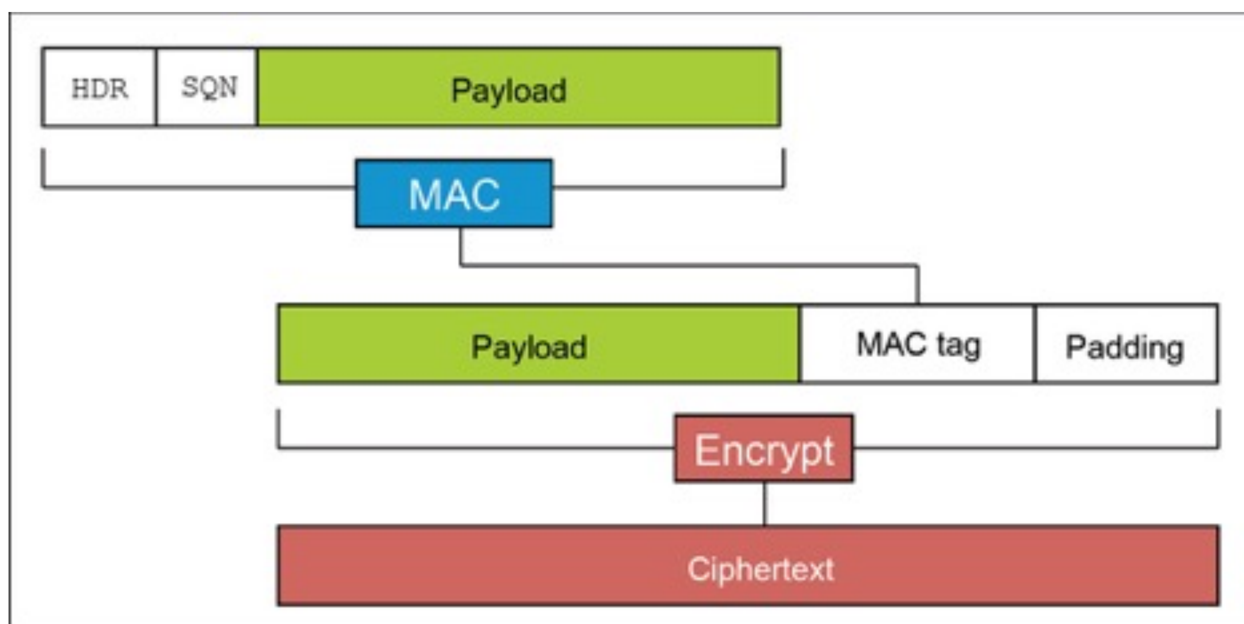
While SSL BEAST attacks affect only the Transport Layer Security (TLS) 1.0 version of SSL and not later versions such as TLS 1.1 and 1.2, TLS 1.0 remains the overwhelmingly predominant version used by both Web servers and browsers. Following a Javascript-based demonstration of the SSL BEAST attack by researchers Juliano Rizzo and Thai Duong, developers of Google Chrome and other major Web browsers started taking steps to create workarounds for mitigating the risks of SSL BEAST attacks.

CRIME ("**Compression Ratio Info-leak Made Easy**") is a security exploit against secret web cookies over connections using the HTTPS and SPDY protocols that also use data compression.[1][2] When used to recover the content of secret authentication cookies, it allows an attacker to perform session hijacking on an authenticated web session, allowing the launching of further attacks. The vulnerability exploited is a combination of chosen plaintext attack and inadvertent information leakage through data compression similar to that described in 2002 by the cryptographer John Kelsey. It relies on the attacker being able to observe the size of the ciphertext sent by the browser while at the same time inducing the browser to make multiple carefully crafted web connections to the target site. The attacker then observes the change in size of the compressed request payload, which contains both the secret cookie that

is sent by the browser only to the target site, and variable content created by the attacker, as the variable content is altered. When the size of the compressed content is reduced, it can be inferred that it is probable that some part of the injected content matches some part of the source, which includes the secret content that the attacker desires to discover. Divide and conquer techniques can then be used to hone in on the true secret content in a relatively small number of probe attempts that is a small multiple of the number of secret bytes to be recovered.

The CRIME exploit was created by the security researchers Juliano Rizzo and Thai Duong, who also created the BEAST exploit. The exploit was due to be revealed in full at the 2012 ekoparty security conference. Rizzo and Duong presented CRIME as a general attack that works effectively against a large number of protocols, including but not limited to SPDY (which always compresses request headers), TLS (which may compress records) and HTTP (which may compress responses)

Lucky Thirteen Attack –



Lucky Thirteen uses a technique known as a padding oracle that works against the main cryptographic engine in TLS that performs encryption and ensures the integrity of data. It processes data into 16-byte chunks using a routine known as MEE, which runs data through a MAC (Message Authentication Code) algorithm, then encodes and encrypts it. The routine adds "padding" data to the ciphertext so the resulting data can be neatly aligned in 8- or 16-byte boundaries. The padding is later removed when TLS decrypts the ciphertext.

The attacks start by capturing the ciphertext as it travels over the Internet. Using a long-discovered weakness in TLS's CBC, or cipher block chaining, mode, attackers replace the last several blocks with chosen blocks and observe the amount of time it takes for the server to respond. TLS messages that contain the correct padding will take less time to process. A mechanism in TLS causes the transaction to fail each time the application encounters a TLS message that contains tampered data, requiring attackers to repeatedly send malformed messages in a new session following each previous failure. By sending large numbers of TLS messages and statistically sampling the server response time for each one, the scientists were able to eventually correctly guess the contents of the ciphertext.

Attack Scenarios:

The Certificate Authority System A certificate from an intermediate CA is trusted if there is a valid chain of trust all the way back to a root CA. Any certificate authority can issue and sign a certificate for any identity / public key. The system is only as secure as the weakest certificate authority.

Attack Scenario 1. An attacker compromises the private (signing) key of a certificate authority. 2. The attacker can now sign a certificate for the attacker’s public key, with the identity “google.com”. 3. A man-in-the-middle attack is now possible, as long as the compromised CA is continued to be trusted all the way back to a root CA. Note: This attack works even if a root CA (or any other CA) has signed a certificate for google.com’s real public key

Attack Scenario 2. DigiNotar DigiNotar was a Dutch root certificate authority, trusted by all major web browsers. On July 10th, 2011 a hacker gained access to DigiNotar’s private signing key, and created a wildcard certificate for Google (*.google.com) using the hacker’s public key. The certificate was used to perform man-in-the-middle attacks on web users in Iran before it was detected and revoked at the end of August.

DigiNotar (The Fallout) At least 531 other fraudulent certificates were found to have been issued by hackers using DigiNotar’s compromised key. DigiNotar was removed as a root CA from all major web browsers. DigiNotar’s intermediate CAs and all certificates they had signed were considered invalid due to the broken chain of trust.

Because the ugly truth about the existing SSL/TLS protocols, is that, even with lots of clever cryptography being used, the current system we have for validating trust is based on often unknown third parties, with a single point of failure.

How can we make certificate verification more secure?

VTE Information