Published in Demystifying Security

Ravi Follow
May 3, 2020 · 15 min read · Listen

Save

Making Sense of SSL/TLS

The minimum that every software developer should know about SSL/TLS

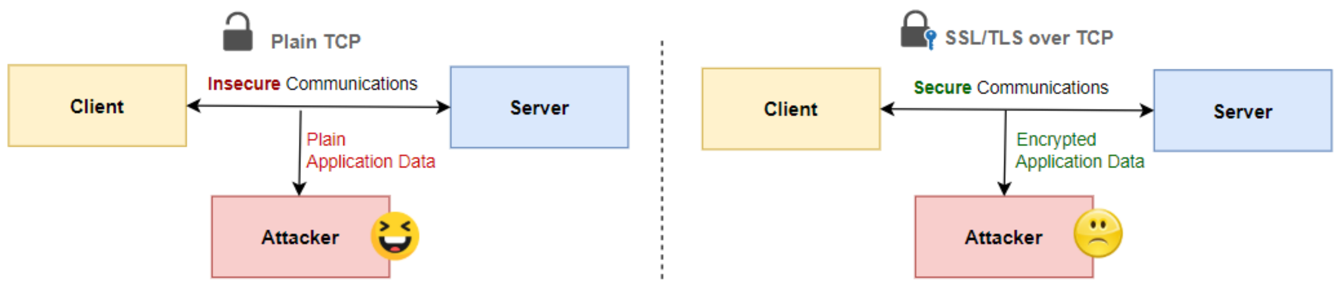
Photo by Nick Fewings on Unsplash

If you are a software developer, there is a good chance that you have had to deal with SSL/TLS (at least a bit) as it is widely used for securing network applications such as web applications, emails and file transfers. It is easy to learn just enough about digital certificates, PKI, and trust. This article is intended to help you understand the foundational concepts of SSL/TLS, easily.

Table of Contents:

1. Overview

SSL and TLS enable two parties, say a *client* and a *server,* to communicate securely even over an insecure medium such as the Internet.



- The data they are exchanging isn't read by any other party, including attackers. (*Confidentiality* using symmetric shared-key/symmetric *encryption*)
- They are talking to the right sender or receiver. *(Authentication* using *X.509 Certificate validation)*
- They have received exactly the same data that the other party has sent, i.e., there is no *message loss* or *tampering,* and that origin of the data is indeed the other party. *(Message integrity and authenticity* using *Message Authentication Codes(MAC))*

And, the client/server can be assured of the above, even if:

- The two are communicating over an insecure medium where an attacker can easily intercept the network traffic.
- The two parties have had no prior interactions for establishing a mutual trust or for exchanging any encryption keys/identities, before-hand.

Think about it for a moment — pretty amazing, isn't it?

1.1. SSL/TLS vs. HTTPS

You may have heard about SSL/TLS in the context of HTTPS and might wonder how they are related. HTTPS is one of the many applications of SSL/TLS: when you interact with an HTTP**S** (**S** for 'secure') website like *https://www.medium.com*, it is SSL/TLS that secures the web-traffic (made of HTTP interactions) between your browser and the website.

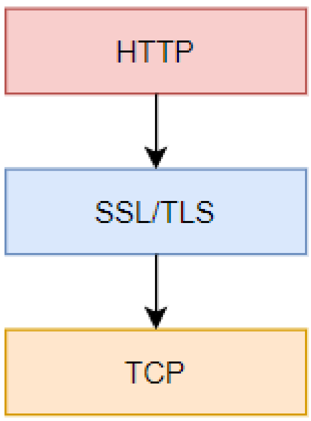


Figure 2: HTTPS is HTTP over SSL/TLS

SSL and TLS stand for *Secure Sockets Layer (SSL)* and SSL's modern counterpart *Transport Layer Security (TLS),* respectively. The last SSL version SSL v3 was deprecated in 2015. While the name SSL still lives on, people usually mean *TLS* when they use the terms *SSL or SSL/TLS*. We'll use the terms SSL/TLS and TLS interchangeably in this article, henceforth.

. . .

2. The TLS Protocol

I don't want to bore you with the gory details of how TLS protocols and how they relate to the TCP/IP stack, but there are a few essential things you should know about this topic that we cover in the following sub-sections.

2.1. How TLS Relates to TCP/IP Protocol Stack

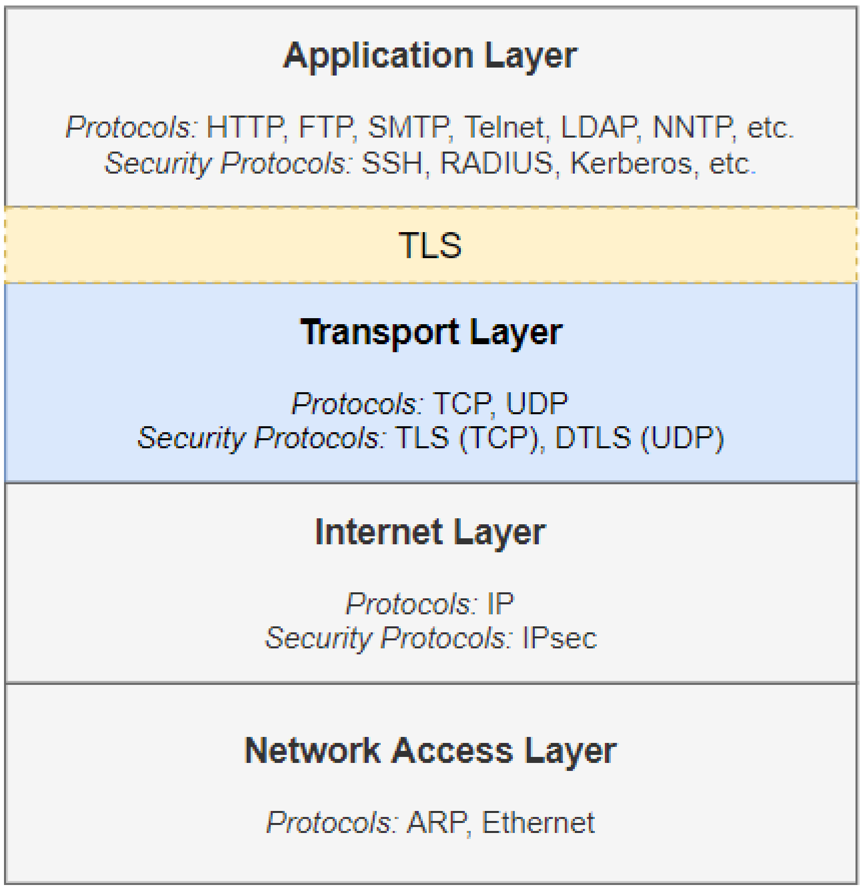

Application Layer
Protocols: HTTP, FTP, SMTP, Telnet, LDAP, NNTP, etc.
Security Protocols: SSH, RADIUS, Kerberos, etc.
TLS
Transport Layer
Protocols: TCP, UDP
Security Protocols: TLS (TCP), DTLS (UDP)
Internet Layer
Protocols: IP
Security Protocols: IPsec
Network Access Layer
Protocols: ARP, Ethernet

Figure 3: TCP/IP Protocol Stack

Despite what its name might suggest, TLS (for *Transport-Layer* Security) is not a *transport-layer* security protocol. As illustrated by the yellow box in the figure, it sits between application layer protocols such as HTTP, SMTP, FTP, NNTP, LDAP and POP, and the transport protocol TCP. Sitting in the middle of the two layers, TLS secures application layer data transmitted over TCP. For instance, in HTTPS, TLS secures HTTP traffic transmitted over TCP.

DTLS — a sibling of TLS — is to UDP what TLS is to TCP.

2.2. TLS Protocol Overview

TLS is a *security protocol*. It establishes a secure TCP session between the *client* and the *server* and uses the session to transmit application layer data securely.

TLS is also a *cryptographic protocol*, i.e., it uses cryptographic techniques like symmetric and asymmetric encryption, Message Authentication Codes (MAC), digital signatures, encryption key exchange, etc. for establishing a secure session and exchanging data securely. However, you don't need to be familiar with those techniques to use and understand TLS at a high-level. To know more about how TLS uses cryptography for securing communications, see How TLS Secures Communications.

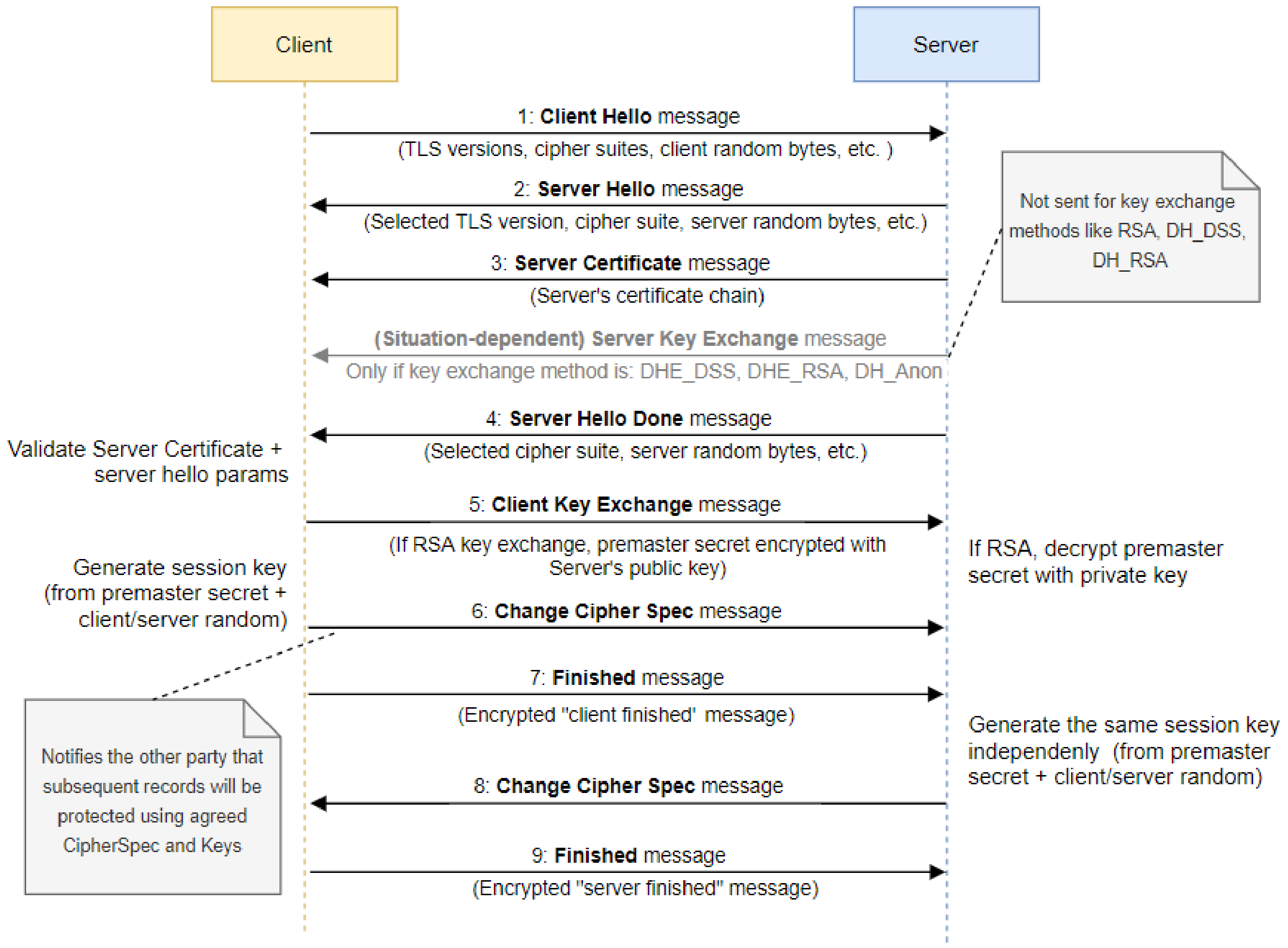
The TLS protocol comprises of a handful of sub-protocols. The two main sub-protocols are:

- *Handshake* protocol: The *handshake protocol* is used to negotiate session parameters like TLS version, cipher suite, data encryption keys, etc. In effect, it establishes a secure session for exchanging data securely,
- *Record* protocol. The *record protocol,* on the other hand, is used to transmit data securely using the session parameters negotiated during the handshake protocol. For example, it encrypts data using session encryption keys and algorithms generated during the handshake.

TLS has a few other sub-protocols:

- *Change cipher spec protocol*: It is used in TLS 1.2 and earlier to modify the cryptographic parameters of a session.

The *handshake protocol* is where most of the heavy-lifting is done. Figure 4 below illustrates an example of the message flow between the two parties in one-way TLS.



Client
Server
1: **Client Hello** message
(TLS versions, cipher suites, client random bytes, etc.)
2: **Server Hello** message
(Selected TLS version, cipher suite, server random bytes, etc.)
3: **Server Certificate** message
(Server's certificate chain)
(Situation-dependent) **Server Key Exchange** message
Only if key exchange method is: DHE_DSS, DHE_RSA, DH_Anon
Not sent for key exchange methods like RSA, DH_DSS, DH_RSA
4: **Server Hello Done** message
(Selected cipher suite, server random bytes, etc.)
Validate Server Certificate + server hello params
5: **Client Key Exchange** message
(If RSA key exchange, premaster secret encrypted with Server's public key)
If RSA, decrypt premaster secret with private key
Generate session key (from premaster secret + client/server random)
6: **Change Cipher Spec** message
7: **Finished** message
(Encrypted "client finished' message)
Notifies the other party that subsequent records will be protected using agreed CipherSpec and Keys
Generate the same session key independenly (from premaster secret + client/server random)
8: **Change Cipher Spec** message
9: **Finished** message
(Encrypted "server finished" message)

Figure 4: Representative Message Flow in Full Handshake Protocol for 1-way TLS (Version 1.2)

The boldface text in Figure 4 above represents TLS protocol messages. The text in brackets below the arrows indicates the payload. We won't go into the details of these interactions in this article, for the sake of brevity. For a detailed overview of these messages and the sub-protocols, see RFC 5246: TLS Protocol v1.2 or RFC 8446: TLS Protocol v1.3.

. . .

3. Public Key Infrastructure (PKI)

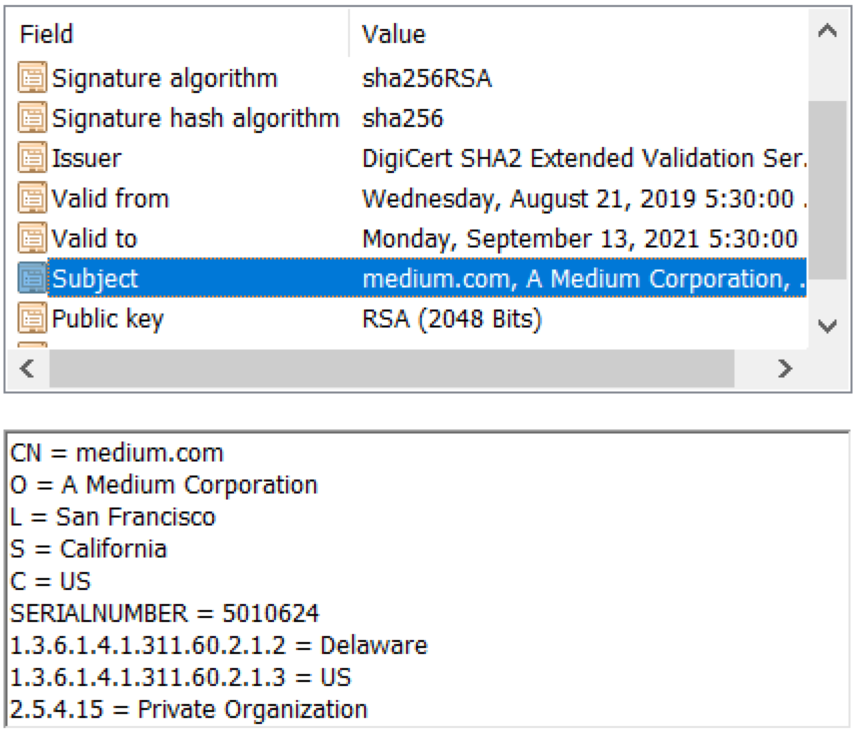
Public Key Infrastructure (PKI), which has components like digital certificates, A full-blown introduction to PKI is hard to cover in a single article. We'll limit our coverage here. The following sub-sections introduce SSL/TLS certificates, CA, and certificate validation, which in turn are perhaps the first things that come to mind when we think about SSL/TLS.

3.1. Understanding TLS Certificates

A server's *TLS certificate* is a *public-key certificate* that binds the server's identity — such as its DNS name, to the server's *public key*. The public key is the public part of the public-private key pair used for asymmetric encryption of secrets, key exchange, and authentication in TLS.

A client's TLS certificate does a similar thing for the client's identity. It is used in *"two-way TLS"* (more on this in a bit).

TLS certificates are specific types of X.509 digital certificates, which have their purpose defined as "*server authentication*" or "*client authentication*" via the certificate's extended key usage (EKU) field. For more information about the field and TLS X.509 certificates in general, see Understanding TLS Certificates. The following figure shows an example.



Field	Value
Signature algorithm	sha256RSA
Signature hash algorithm	sha256
Issuer	DigiCert SHA2 Extended Validation Ser.
Valid from	Wednesday, August 21, 2019 5:30:00 .
Valid to	Monday, September 13, 2021 5:30:00
Subject	medium.com, A Medium Corporation, .
Public key	RSA (2048 Bits)

```
CN = medium.com
O = A Medium Corporation
L = San Francisco
S = California
C = US
SERIALNUMBER = 5010624
1.3.6.1.4.1.311.60.2.1.2 = Delaware
1.3.6.1.4.1.311.60.2.1.3 = US
2.5.4.15 = Private Organization
```

A certificate authority (CA) is a trusted authority that signs digital certificates, typically after validating the server or client's credentials, such as ownership of the subject name (like server's DNS name) and the public key, etc.

A CA is analogous to a government authority that issues driver licenses. Various agencies trust a driver's license, as they trust the issuing authority and know that the license cannot be forged easily. Similarly, a CA issues digital certificates, which are computationally difficult to forge. So, when you see a digital certificate signed by CA you trust, you might choose to trust the contents of the certificate.

Of course, if a CA is compromised, the trustworthiness of the certificates that were signed by it also falls apart. So, if you have an in-house *private/corporate CA* service issuing certificates, you must secure them adequately. Many companies use certificates signed by private CAs for internal-facing applications and certificates signed by public CAs (like Verizon, DigiCert, GlobalSign, etc.) for external-facing applications like websites, eCommerce applications, etc.

3.3. The 'Why' and 'What' of TLS Certificate Validation

With the background about TLS certificates and CA out of the way, let's discuss the purpose of certificate validation and how it is done.

During the TLS handshake, typically a client authenticates the server. How it works is that the server sends its TLS certificate(s) to the client, which in turn validates them. If the server's certificate(s) is/are successfully validated, the client has successfully authenticated the server.

The checks that a client performs while validating the server's TLS certificate(s) are listed below (see Understanding TLS Certificates for a description of these fields):

- The certificate's integrity is intact, i.e., it has not been modified or tampered with since its issuance, using its "*signature*" field.
- The certificate is still valid, i.e., it hasn't expired. A certificate is valid within the period specified by "*valid from*" and "*valid to*" fields.
- The client (such as a browser in HTTPS) trusts the CA that has signed the

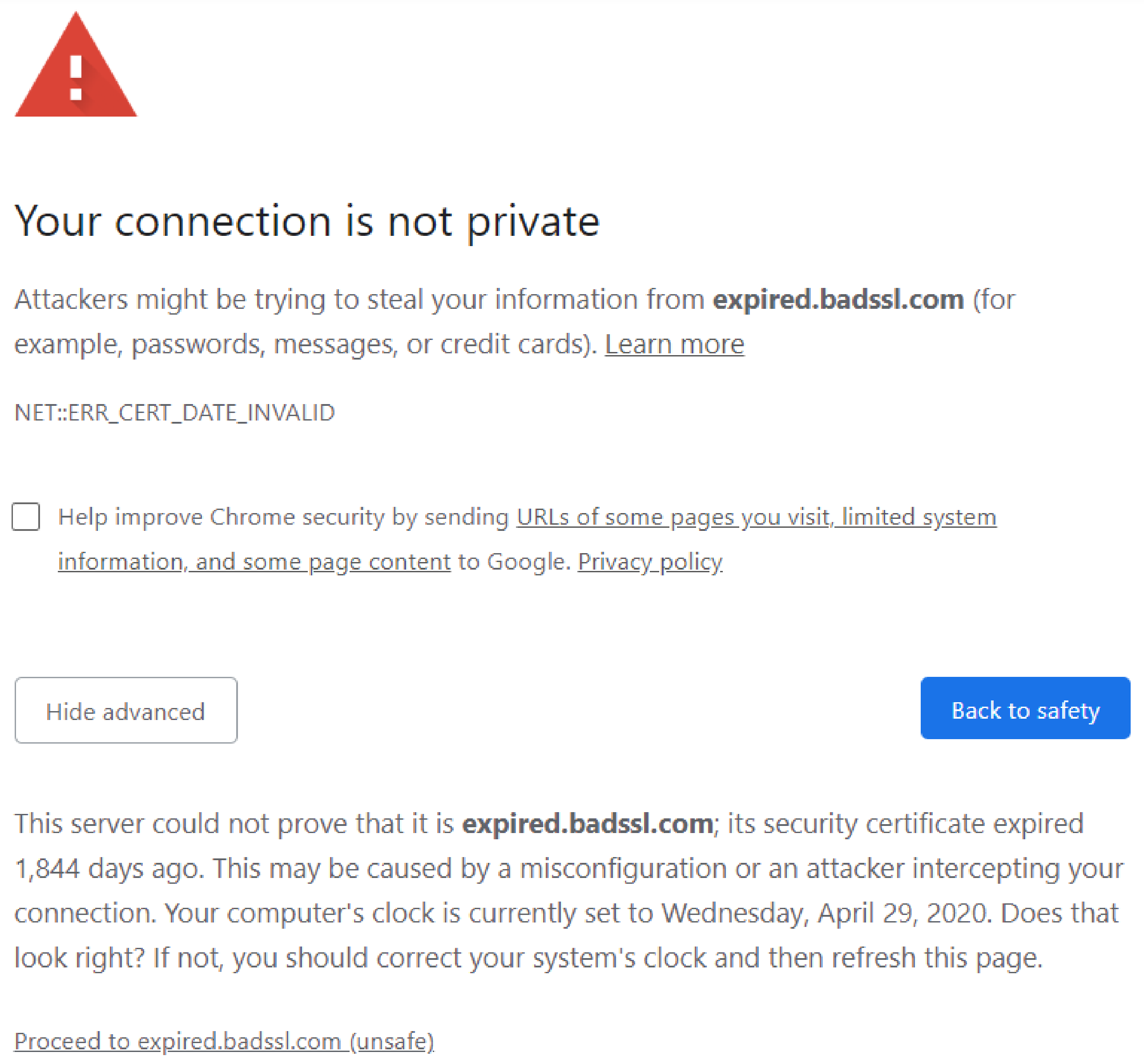
- The client trusts the issuer of the server's certificate (the CA), or alternatively, the certificate itself. The two models of trust are called "*chain-of-trust*" or "*direct trust.*" More on these trust models in a bit.
- The "*subject*" in the server's certificate is the same as the server hostname that the client is accessing. For example, if I hit the URL "https://medium.com/new-story," the hostname is "medium.com," and therefore the server serving the request must present a certificate that is issued to (i.e., has a subject "medium.com").
- The certificate is not revoked. (This is an advanced topic that warrants a separate article of its own, so we'll not cover it here.)

In *"one-way TLS,"* the client authenticates the server, but the server does not authenticate the client. In *"two-way TLS,"* mutual authentication takes place, i.e., both the client and the server authenticate each other. Client certificate validation of the server-side works much the same as way as client-side server certificate validation. Both use TLS certificates and rely on the validation of each other's X.509 certificates for authentication.

3.4. What Happens If Certificate Validation Fails?

What happens upon certificate validation is what the client (or the server in case of client authentication) chooses to do. In most cases, the request is blocked. For example, if one node of a cluster fails to validate another node's certificate, communication between the nodes will fail.

In the case of HTTPS (an application of TLS as we mentioned earlier), most browsers will fail the request, describe the situation to the user, and allow the user to proceed by taking action. See the figure below for an example.



Your connection is not private

Attackers might be trying to steal your information from **expired.badssl.com** (for example, passwords, messages, or credit cards). Learn more

NET::ERR_CERT_DATE_INVALID

Help improve Chrome security by sending URLs of some pages you visit, limited system information, and some page content to Google. Privacy policy

This server could not prove that it is **expired.badssl.com**; its security certificate expired 1,844 days ago. This may be caused by a misconfiguration or an attacker intercepting your connection. Your computer's clock is currently set to Wednesday, April 29, 2020. Does that look right? If not, you should correct your system's clock and then refresh this page.

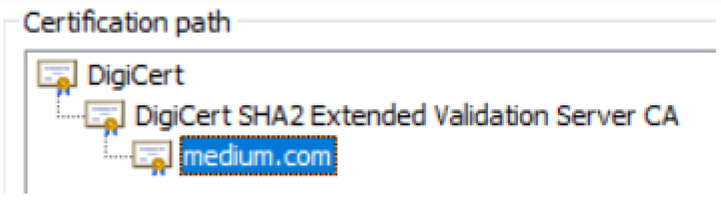
Proceed to expired.badssl.com (unsafe)

Figure 6: Accessing an HTTPS site that has Expired Certificate

3.5. How 'Chain-of-Trust' Works

Earlier, we said that during a TLS handshake, the server (and optionally the client) sends its certificates to the other party for validation. Certificate validation tasks include verifying that the certificate is signed by a trusted third-party — the CA. This task is where the *chain-of-trust* trust model comes in.

The server/client sends not just its own certificate for validation, but the entire certificate chain. To understand the certificate chain, let's consider a real example. If you accessed "https://www.medium.com" currently, the server serving the request


Certification path

DigiCert
DigiCert SHA2 Extended Validation Server CA
medium.com

Figure 7: Certification Path for the medium.com Certificate

In the above screenshot showing the “https://www.medium.com” server’s certificate, at the lowest level is the sender’s certificate, which is assigned to “medium.com.” That certificate is signed by an *intermediate CA* “DigiCert SHA2 Extended Validation Server CA.” The intermediate CA’s certificate itself is signed by a *root CA* “DigiCert High Assurance EV Root CA (DigiCert).”

To generalize, a certificate chain is a hierarchy of certificates comprising of:

- An end-entity certificate (“medium.com” certificate in the example above), which is signed by the immediate intermediate CA (if applicable).
- One or more intermediate CA certificates, each signed by the intermediate CA up in the hierarchy. In the example above, there is a single intermediate CA, and its certificate is signed by the root CA directly.
- The root CA certificate, which signs the first intermediate CA in the hierarchy, and whose own certificate is self-signed.

You may have noticed earlier that certificates signed by public CAs are typically signed by an intermediate CA (and not the root CA). This is because CA/Browser Forum’s Baseline Requirements for the Issuance and Management of Publicly-Trusted Certificates prohibits issuing certificates by root CAs.

So, how is certificate chain-of-trust used? During TLS handshake, when the client is validating the server’s certificate chain (or vice versa), among other things, it’ll check if the certificate can be linked all the way back to a trusted root CA. If the server (or client’s) certificate can be linked to a trusted root CA, itis trusted; Otherwise, it is not. This is why the root CA is also known as the “trust anchor.”

The question then arises: How do the server/client entities know which root CA’s to automatically takes you to the website as DigiCert's root certificate is in the trusted root CAs list.

This model of trust establishment where the client trusts the server by verifying that the server's certificate is signed by a CA it trusts is called the *"chain-of-trust"* model. In this model, the client trusts the CA — the trusted arbitrator, rather than the server directly.

3.6. 'Direct Trust' and Self-Signed Certificates

Alternatively to the "chain-of-trust" model, one can add the server's TLS certificate directly (and not the CA's certificate) in the client's truststore, and then the client will trust the server's certificate. This model is called the *"direct trust"* model. This model is useful in cases where you use self-signed certificates — certificates that are not signed by any CA. Self-signed certificates are often used in development and testing, but you shouldn't use them in production.

. . .

4. TLS Configuration

TLS implementations typically let you configure TLS versions and cipher suites that the client/server can use. One or more of these may be set on both client- and server-side. The specific version and cipher suite that are used to secure a particular TLS session are negotiated on-the-fly during the TLS handshake.

Why do people restrict the versions and cipher suites in the first place? All SSL versions and some of the older versions of TLS are considered vulnerable. Many cipher suites are either vulnerable or do not provide enough security by design. Therefore, many companies will disallow the use of older versions and certain cipher suites.

4.1. What are Cipher Suites?

A cipher suite is a named combination of cryptographic algorithms that the client/server may use during the TLS handshake. To understand how TLS uses cryptographic techniques, see How TLS Secures Communications.

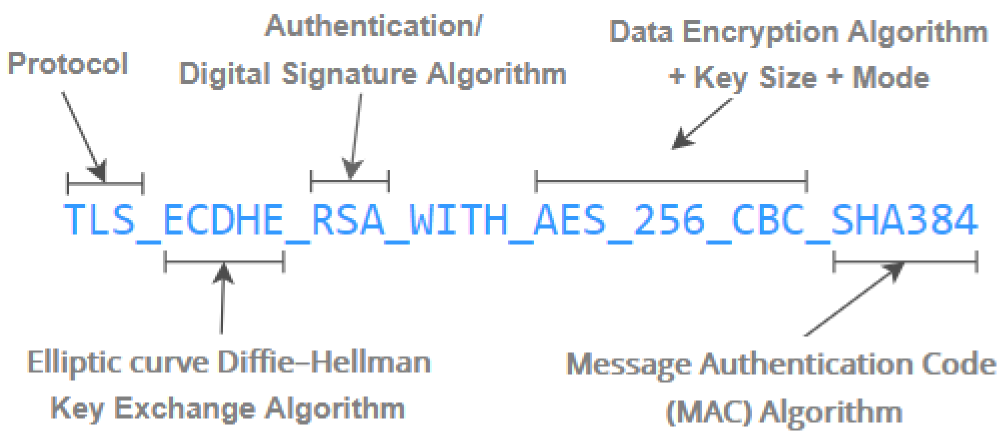

Protocol
Authentication/
Digital Signature Algorithm
Data Encryption Algorithm
+ Key Size + Mode
TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384
Elliptic curve Diffie–Hellman
Key Exchange Algorithm
Message Authentication Code
(MAC) Algorithm

Figure 8: An Example TLS 1.2 Cipher Suite

Each version of TLS specifies a set of cipher suites that may be used with that version. For instance, TLS 1.2 supports 37 different cipher suites, according to this article from thesslstore.com. TLS implementations (such as OpenSSL and Java TLS implementation, etc.) may support a subset of those cipher suites. Many products allow you to restrict and reorder the list of cipher suites (as well as TLS versions) that they can use.

The format of cipher suites has changed in TLS 1.3. An example of a TLS 1.3 cipher suite is TLS_AES_256_GCM_SHA384, which is in a different format from the TLS 1.2 cipher suite shown in Figure 8 above. TLS 1.3 cipher suites do not contain key exchange and digital signature algorithms. TLS 1.2 cipher suites cannot be used with TLS 1.3, and vice versa.

Servers must restrict the cipher suites that they allow as SSL/TLS includes support for cipher suites that provide minimal or even no security at all. Here’s what RFC 5246: TLS Protocol v1.2 says on the topic:

“TLS supports a range of key sizes and security levels, including some that provide no or minimal security. A proper implementation will probably not support many cipher suites. For instance, anonymous Diffie-Hellman is strongly discouraged because it cannot prevent man-in-the-middle attacks. Applications should also enforce minimum and maximum key sizes. For example, certificate chains containing 512- bit RSA keys or signatures are not appropriate for high-security

Most SSL/TLS clients/servers support multiple SSL/TLS versions, such as “SSL v3”, “TLS v1.1” and “TLS v1.2”. A client and a server will typically use the highest version that they both support. For example, if the client supports “SSL v3”, “TLS v1.1” and “TLS 1.2”, and the server supports TLS “1.2” and TLS “1.3”, they’ll end up negotiating TLS 1.2.

Which versions should you use or enable for your server or client products?

To be able to answer that question, let’s first get a high-level understanding of TLS versions. The following figure depicts the timeline of SSL/TLS releases/deprecations. *SSL v1* is missing in the figure, as it wasn’t publicly released due to serious security flaws.

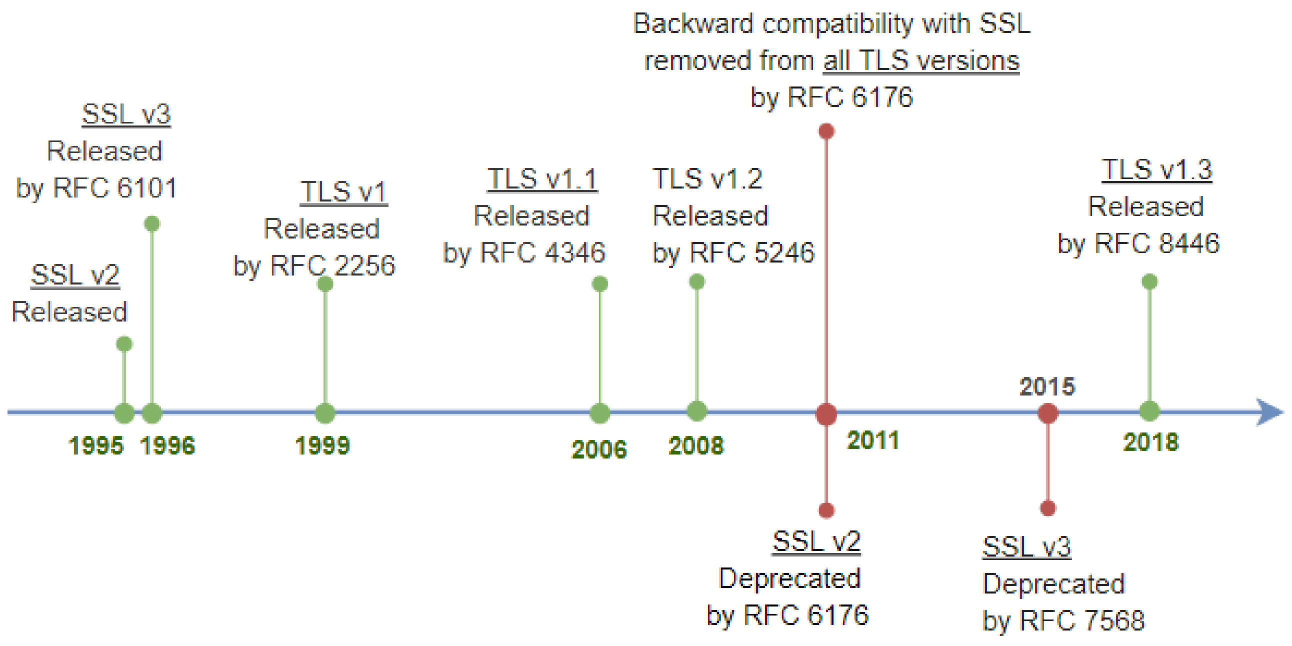

Backward compatibility with SSL removed from all TLS versions by RFC 6176
SSL v3 Released by RFC 6101
SSL v2 Released
TLS v1 Released by RFC 2256
TLS v1.1 Released by RFC 4346
TLS v1.2 Released by RFC 5246
TLS v1.3 Released by RFC 8446
1995
1996
1999
2006
2008
2011
2015
2018
SSL v2 Deprecated by RFC 6176
SSL v3 Deprecated by RFC 7568

Figure 9: Timeline of SSL/TLS Releases and Deprecations and RFCs (Based on Data Gathered from Wikipedia)

As shown in the above figure, all SSL versions — SSL v2, v3 — were deprecated by IETF long, so you should avoid using them altogether. They have many exploitable security vulnerabilities; SSLv3, for example, is vulnerable to the POODLE attack, which enables attackers to gain access to user’s private data such as passwords and cookies (as per this article from Mozilla).

At the time of writing (in April 2020):

- While TLS v1.0 and v1.1 are not officially deprecated by IETF yet (at the time of writing), many browsers have already started deprecating them. According to Wikipedia:

> *"In October 2018, Apple, Google, Microsoft, and Mozilla jointly announced they would deprecate TLS 1.0 and 1.1 in March 2020"*

- Many companies now explicitly disallow the use of TLS 1.0 and have deprecated the use of TLS 1.1 in their environments.
- Many companies already require that their server applications be configured for using TLS 1.2.
- TLS 1.3 is relatively new and isn't widely supported yet, but is gaining traction fast. Many companies have started issuing policies that advise including support for TLS 1.3.

So, at the time of writing of this article, you should consider enabling only TLS 1.2 and 1.3 in your client/server applications and disabling all SSL versions and older versions of TLS.

. . .

5. Conclusion

This article provided a quick overview of some of the foundational concepts of SSL/TLS, which hopefully gives you a better picture of how the different pieces fit together.

Here are the key takeaways:

- SSL/TLS secures communications among two parties. HTTPS is an application of SSL/TLS.
- The SSL protocol is actually long deprecated. Its modern counterpart is TLS. When people say SSL, they usually mean TLS.

- TLS comprises of two main sub-protocols: the *handshake protocol* and the *record protocol*. The former is used to negotiate session parameters like TLS version, ciphers, and encryption keys and the latter uses the negotiated parameters to secure the data.
- In *"one-way TLS,"* the server authenticates the client, but the client doesn't authenticate the server. In *"two-way TLS,"* both the client and the server authenticate each other.
- TLS certificates are specific types of X.509 certificates, which have their purpose specified as *"server authentication"* and/ *"client authentication."*
  Certificate validation is the bedrock for authentication in TLS. It includes checking whether the certificate is trusted. A certificate is trusted either via *"chain-of-trust"* or *"direct trust."*
- Cipher suites specify the cryptographic algorithms that TLS may use while setting up a secure session, as well as during data exchange.

. . .

6. References and Further Reading

1. Guidelines for the Selection, Configuration, and Use of Transport Layer Security (TLS) Implementations, NIST Special Publication 800–52r2
2. RFC 5246: TLS Protocol v1.2, RFC 8446: TLS Protocol v1.3
3. RFC 2818: HTTP over TLS
4. RFC 5280: X.509 Standard
5. https://www.ssl.com/faqs/
6. https://geekflare.com/tls-101/
7. https://help.ubuntu.com/lts/serverguide/certificates-and-security.html.en
8. https://dzone.com/articles/ssl-in-java

11. https://docs.oracle.com/javase/10/security/java-secure-socket-extension-jsse-reference-guide.htm