**The Transport Layer Security (TLS) Protocol**

The primary goal of the TLS protocol is to provide **privacy and data**

**integrity** between two communicating applications. The protocol is

composed of two layers: the TLS Record Protocol and the TLS Handshake

Protocol.

Cryptographic security: TLS should be used to establish a secure

connection between two parties.

HTTPS is the secured version of HTTP: HyperText Transfer Protocol.

When that exchange of data is encrypted with SSL/TLS, then we call it HTTPS. The 'S' stands for Secure.

SSL, which stands for **Secure Sockets Layer**, is a cryptographic security protocol that protects your information as it transmits across the internet. A protocol basically means a set of rules that computers use to communicate with each other. It’s kind of like their value system.

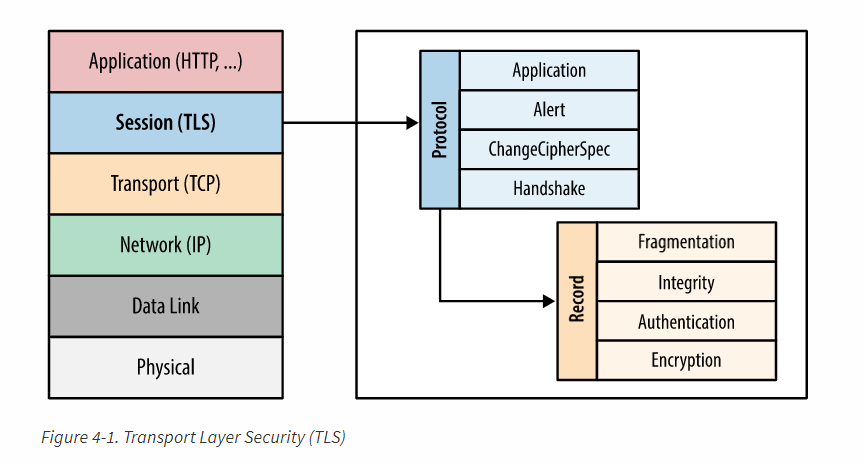
SSL was designed to thwart any unauthorized third party from intercepting and tampering with sensitive data while it’s in transit.

TLS (**Transport Layer Security**): More Secure Version of SSL

Due to the recognized security flaws in SSL, security experts realized that a better and more secure protocol needed to be developed. TLS 1.0 was a **successor** to SSL 3.0 and was first defined in 1999. Since then, three more versions of TLS have been released, with TLS 1.3 (which was released in 2018) being the most current.

The SSL protocol was originally developed at Netscape to enable ecommerce transaction security on the Web, which required encryption to protect customers’ personal data, as well as authentication and integrity guarantees to ensure a safe transaction. To achieve this, the SSL protocol was implemented at the application layer, directly **on top of TCP** ([Figure 4-1](https://hpbn.co/transport-layer-security-tls/#ssl-layer)), 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.

*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 was published in January 1999 and became known as TLS 1.0. Since then, the IETF has continued iterating on the protocol to address security flaws, as well as to extend its capabilities: TLS 1.1 (RFC 4346) was published in April 2006, TLS 1.2 (RFC 5246) in August 2008, and work is now underway to define TLS 1.3.

**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.

When sending information online, we run into three major security problems:

* How can we know whether the person we are communicating with is really who they say they are? **Authentication**
* How can we know that the data hasn’t been tampered with since they sent it? **Integrity**
* How can we prevent other people from seeing and accessing the data? **Encryption**

TLS uses a range of cryptographic techniques to address each of these three problems. Together, they allow the protocol to**authenticate the other party in a connection, check the integrity of data and provide encrypted protection.**

***Encryption***

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

***Authentication***

A mechanism to verify the validity of provided identification material.

***Integrity***

A mechanism to detect message tampering and forgery.

## 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.

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

As part of the TLS handshake, the protocol also allows both 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](https://hpbn.co/transport-layer-security-tls/#chain-of-trust-and-certificate-authorities). In addition, the server can also optionally verify the identity of the client — e.g., a company proxy server can authenticate all employees, each of whom could have their own unique certificate signed by the company.

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

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

**HTTPS Everywhere**

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

*HTTPS protects* ***the integrity of the website***

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

*HTTPS protects the privacy and security of the user*

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

*HTTPS enables powerful features on the web*

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

**Transport Layer Security is a protocol that establishes an encrypted session between two computers on the Internet. It verifies the identity of the server and prevents hackers from intercepting and modifying any data.**

TLS (and its predecessor SSL) allows users to securely transmit sensitive data when using the [HTTPS protocol](https://protonvpn.com/blog/public-wifi-and-https/). In other words, HTTPS is **HTTP layered on top of TLS**. This technology is ideal for applications such as banking, information authentication, email exchange, and any other procedure requiring a higher level of privacy and security. TLS helps provide an enhanced layer of protection by encrypting the otherwise readable data, making it difficult for hackers to obtain private information.



What is a **TLS certificate**?

Digital certificates, also known as identity certificates or public key certificates, are **digital files** that are used to **certify the ownership of a public key**. TLS certificates are a type of digital certificate, **issued by a** **Certificate Authority** (CA). The CA signs the certificate, certifying that they have verified that it belongs to the owners of the domain name which is the subject of the certificate.

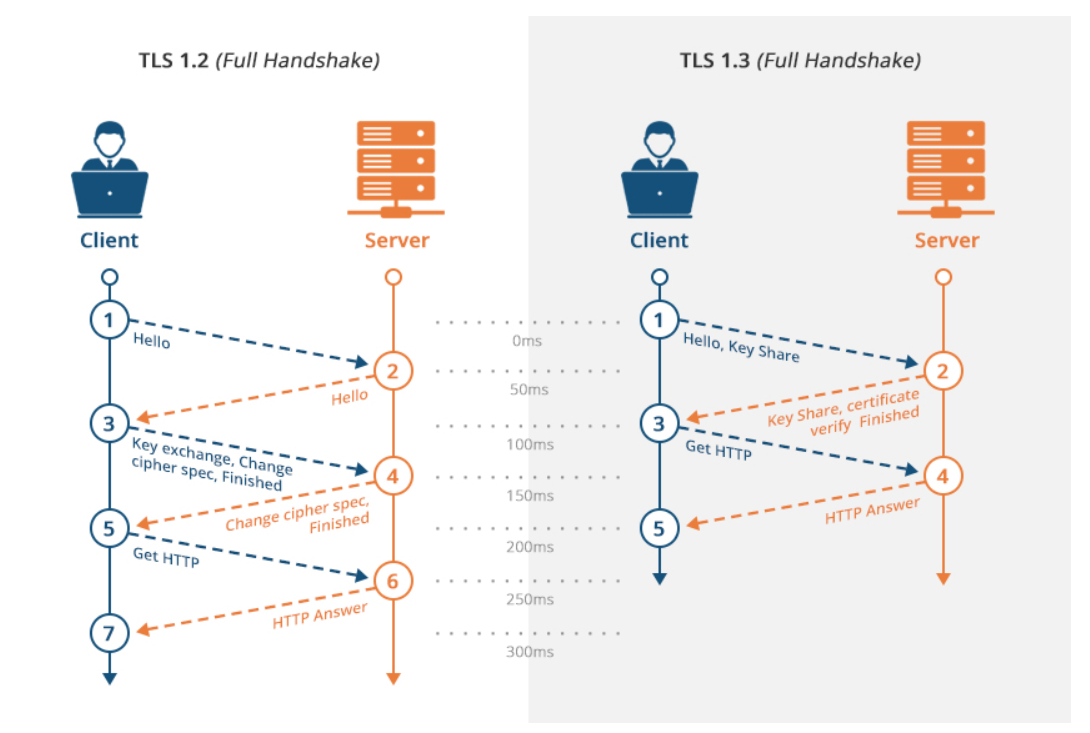
**TLS certificates** usually contain the following information:

* The subject domain name
* The subject organization
* The name of the issuing CA
* Additional or alternative subject domain names, including subdomains, if any
* Issue date
* Expiry date
* **The public key** (The private key, however, is a secret.)
* The digital signature by the CA

### Symmetric and Asymmetric keys

* Keys must be used at least twice in any encrypted conversation. The key needs to be used to encrypt the conversation by the sender and the key also needs to be used on the other end to decrypt the conversation. Before the advent of Public key cryptography, the key used for encryption and decryption were the same; they were symmetrical. This posed the problem I discussed earlier. In order for two parties to exchange encrypted communications, they must have exchanged that key in the past via some secure method that ensured other people had not copied it. This is a pretty big hurdle to overcome and it contributed to the lack of adoption by the general public.
* Public Key Encryption solved this problem through the use of asymmetric keys. Users now generate two keys, not one. The Private key, which needs to remain a secret and never shared, and the Public key, which can be shared and seen by everyone. Senders can encrypt information by using the widely available public key of the recipient. The information can only be decrypted by the recipient’s private key. Conversely, if the recipient needs to reply, then the sender’s public key is used to encrypt the reply to ensure that only they can decrypt it. In this manner, the role of sender and receiver change in every leg of the conversation. While this may seem complicated, there is an entire infrastructure, aptly named Public Key Infrastructure (PKI), that makes public key retrieval and private key usage very easy. I discuss that in the Current Implementations section of this article.
* Given this explanation, you may wonder why anyone would ever use Symmetric encryption any more. The main reason is that it is faster to encrypt and decrypt. If you’re just sending a few pages of documents or emails, there is no noticeable difference. But if you are encrypting gigabytes or terabytes of data on a hard drive, it can make a big difference. Symmetric encryption is also considered stronger than asymmetric encryption, but both are sufficiently hard to break so that is not a practical issue for most people.

[**SSL and TLS**](https://www.ssl2buy.com/wiki/ssl-vs-tls) are **handshake** protocols. They are part of a **server/client architecture** that involves requests and responses to negotiate a connection between two computers. Here is an illustration of a TLS 1.3 handshake:



#### Digital signatures

In asymmetric cryptography, the public and private key can also be used to create a digital signature. A digital signature assures that the person sending the message is who they claim to be.  
   
Typically, we use the recipient’s public key to encrypt the data and the recipient then uses their private key to decrypt the data. However, using the scheme of digital signatures, there’s no way to authenticate the source of the message.

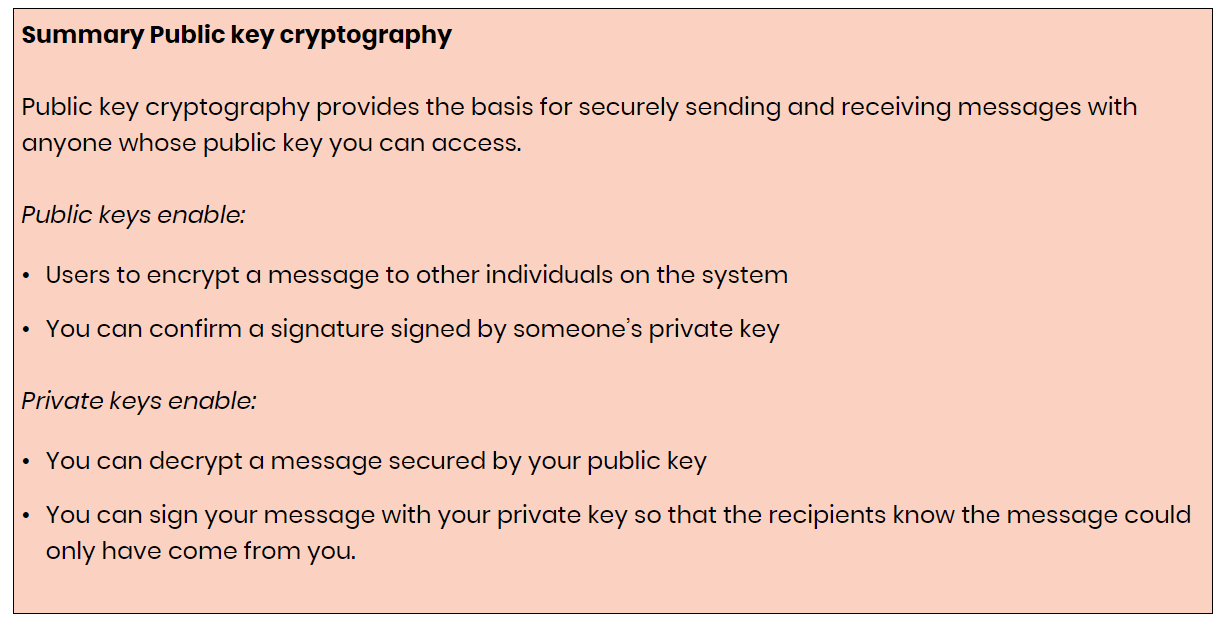
**To create a digital signature using a public and private key**, **Bob digitally signs his email to Alice using his private key**. When Alice receives the message from Bob, she can **verify the digital signature on the message came from Bob by using his public key**. As the digital signature uses Bob’s private key, Bob is the only person who could create the signature.

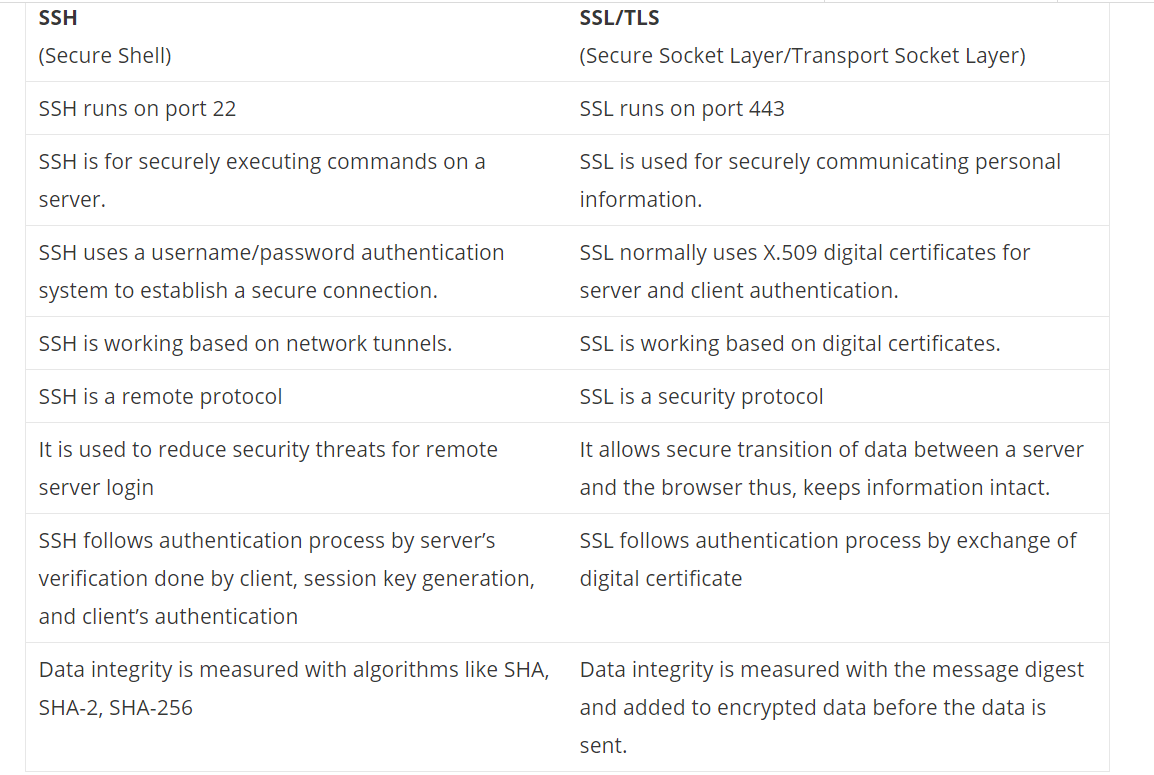
When **encrypting**,

**Sender** use **Recipient’s public key** to encrypt a message, and **sign** the message with sender’s private key.

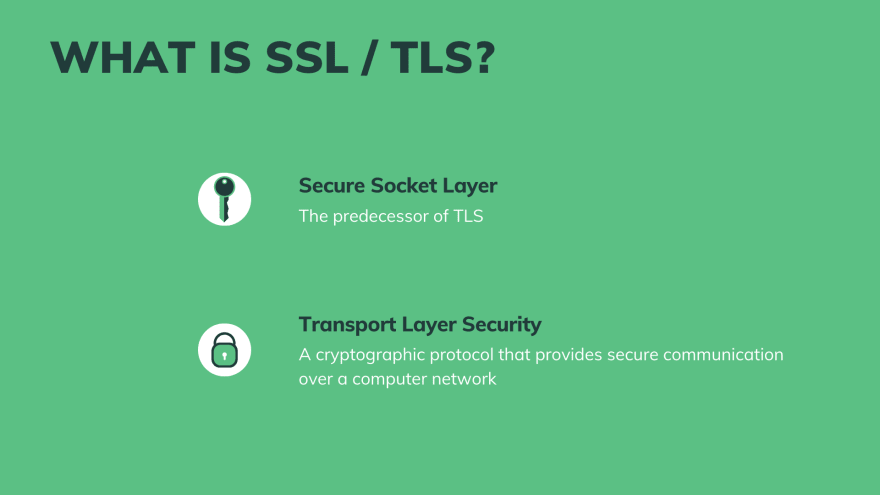
When **Decryption**:

**Recipient** decrypts messages with **Recipient’s private key**, and check signature with sender’s public key.





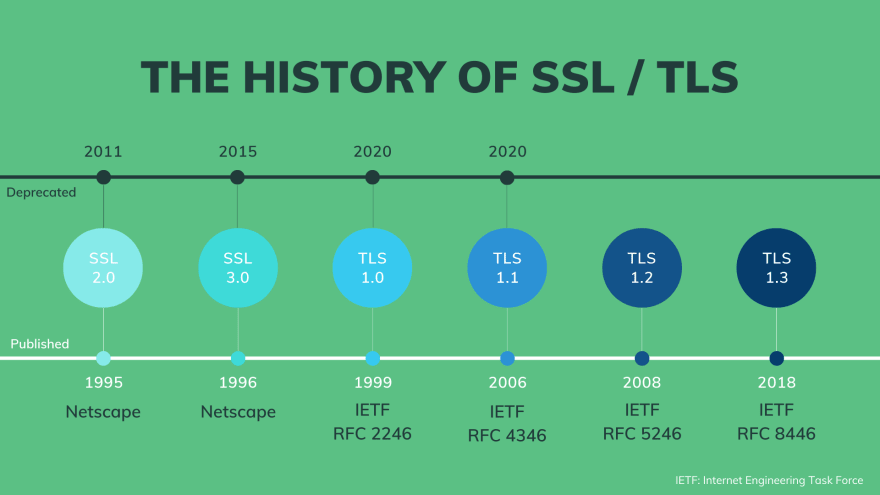
# **1. What is SSL/TLS?**

[](https://res.cloudinary.com/practicaldev/image/fetch/s--ZR5PBInQ--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/dev-to-uploads.s3.amazonaws.com/i/ndqpe3af104ynstvfsp1.png)

SSL stands for Secure Socket Layer. It is the predecessor of TLS.

TLS the short form of Transport Layer Security, which is a cryptographic protocol that provides secure communication over a computer network.

# **2. The history of SSL/TLS**

[](https://res.cloudinary.com/practicaldev/image/fetch/s--SfAucjE3--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/dev-to-uploads.s3.amazonaws.com/i/0lb1buik85z6dkrn274b.png)

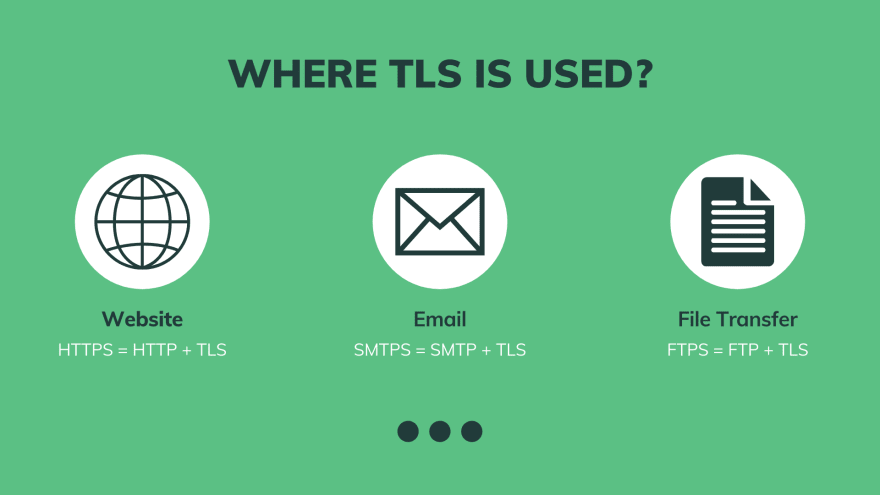
Here's a bit of the history of SSL and TLS:

* SSL was originally developed by Netscape, and it was first published in 1995 with version 2.0
* SSL version 1.0 was never publicly released because of some serious security flaws.
* In 1996, the SSL version 3.0 was published as a complete redesign of the protocol.
* Then 3 years later, TLS 1.0 was first defined in RFC 2246 by IETF as an upgrade of SSL Version 3.0
* It took 7 years to upgrade it to TLS 1.1 in 2006
* TLS 1.2 came right after that in 2008.
* Then finally after 10 years in the making, we got TLS 1.3 with a huge improvements in 2018.

So at the moment which SSL/TLS version still exist?

* The SSL 2.0 was deprecated in 2011
* SSL 3.0 was deprecated in 2015
* And recently, in March 2020, TLS 1.0 and TLS 1.1 was also gone. That means only TLS 1.2 and 1.3 are still active.

# **3. Where is TLS being used?**

[](https://res.cloudinary.com/practicaldev/image/fetch/s---P4dbmV1--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/dev-to-uploads.s3.amazonaws.com/i/3ugpxzufw1v6w6rj1qye.png)

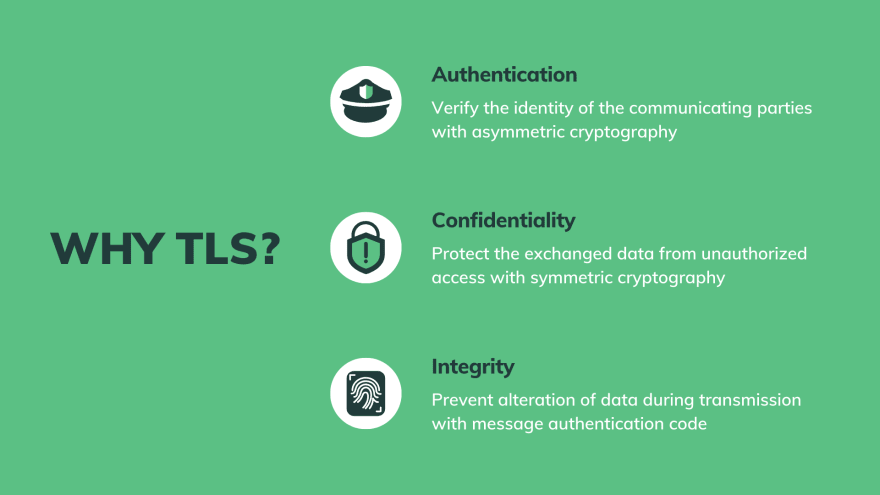
First, it is widely used on the web. All websites that you visit with HTTPS are secured by TLS, or we often say HTTP over TLS.

Similarly, email with SMTPS protocol is in fact SMTP and TLS.

Then FTPS for secure file transfer protocol is also FTP plus TLS.

And there are many other applications of TLS that I don’t have enough time to mention.

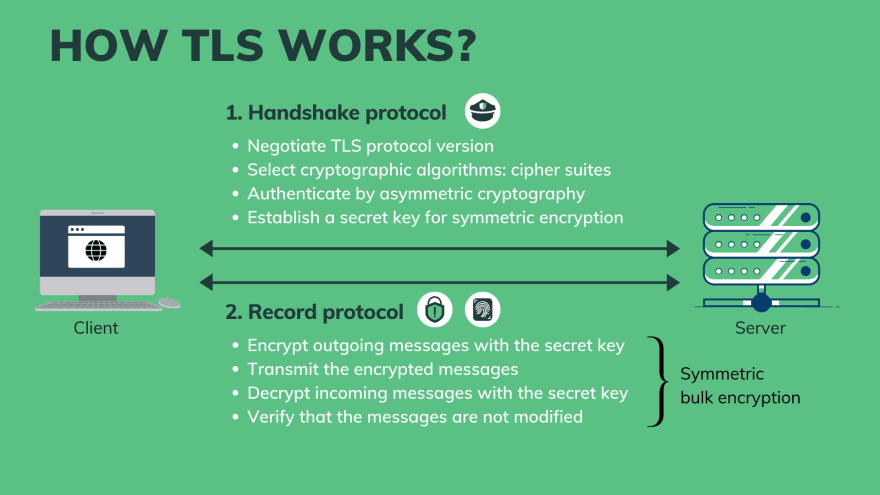
# **4. Why do we need TLS?**

[](https://res.cloudinary.com/practicaldev/image/fetch/s--jOdBgHVU--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/dev-to-uploads.s3.amazonaws.com/i/yhz67zrdmc1vbowilecu.png)

Because TLS gives us 3 things:

* **Authentication**
  + TLS verifies the identity of the communicating parties, which normally be clients and servers.
  + With the help of asymmetric cryptography, TLS makes sure that we will go to the authentic website, and not a fake one.
* **Confidentiality**
  + TLS protects the exchanged data from unauthorized access by encrypting it with symmetric encryption algorithms.
* **Integrity**
  + TLS recognizes any alteration of data during transmission by checking the message authentication code, which we will learn about in a moment.

# **5. How does TLS work?**

[](https://res.cloudinary.com/practicaldev/image/fetch/s--fM6AhQI1--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/dev-to-uploads.s3.amazonaws.com/i/627y8nkwng8w79fsi6y5.png)

Basically, TLS consists of 2 phases, or 2 protocols:

* **Handshake protocol**

In this phase, the client and server will:

* + Negotiate the protocol version
  + Select cryptographic algorithm (or cipher suites)
  + Authenticate each other by asymmetric cryptography
  + Establish a shared secret key that will be used for symmetric encryption in the next phase.

So the main purpose of the handshake is for authentication and key exchange.

* **Record protocol**

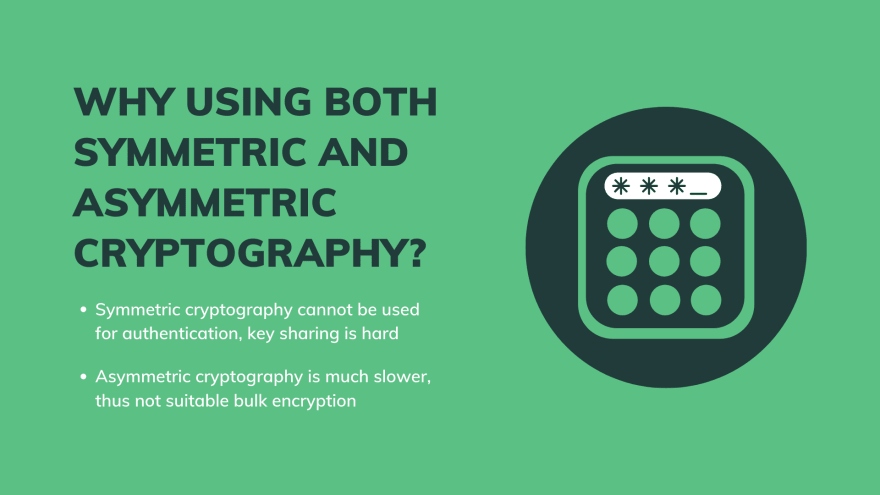
In this phase:

* + All outgoing messages will be encrypted with the shared secret key established in the handshake.
  + Then the encrypted messages are transmitted to the other side.
  + They will be verified to see if there’s any modification during transmission or not.
  + If not, the messages will be decrypted with the same symmetric secret key.

So we will achieve both confidentiality and integrity in this record protocol.

And because the amount of encrypted data in this phase is large, this is often called bulk encryption.

# **6. Why TLS uses both symmetric and asymmetric cryptography?**

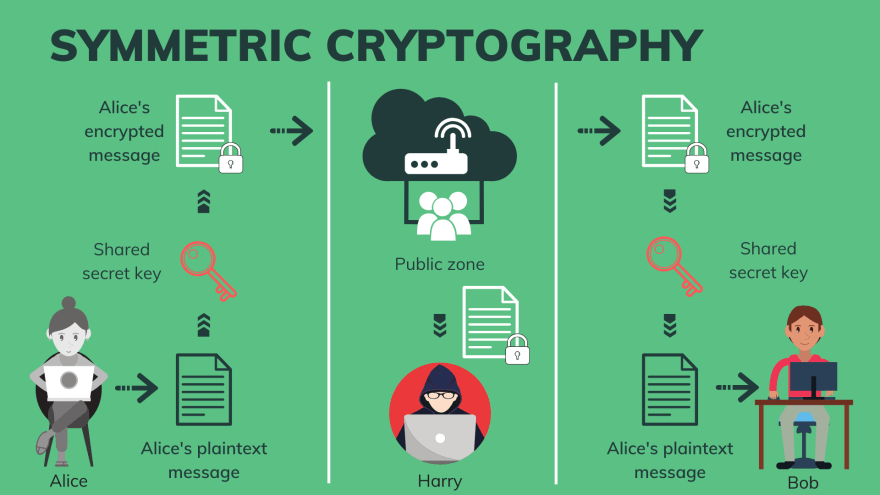
[](https://res.cloudinary.com/practicaldev/image/fetch/s--Ewy3EYpj--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/dev-to-uploads.s3.amazonaws.com/i/6hijxlo3nousvgbfgeut.png)

Why not just use one for all purposes?

Well, it’s easy to see that **symmetric cryptography can’t provide authentication**. Since there’s only 1 secret key for both client and server, they know nothing about each other to verify. Not to mention that how they come up with the same key without leaking it to the public is hard.

How about asymmetric cryptography? Sounds like a good candidate. Unfortunately, **it’s much slower than symmetric cryptography**. And by “much”, I mean from 100 times to even 10000 times slower. So it’s clearly not suitable for bulk encryption.

# **7. Symmetric cryptography**

[](https://res.cloudinary.com/practicaldev/image/fetch/s---BD6KB_C--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/dev-to-uploads.s3.amazonaws.com/i/c9wckcz85djydhfal2ob.png)

Alright, now let’s learn more about symmetric cryptography. I guess you’ve already known the basics.

First of all, Alice has a plaintext message that she wants to send to Bob, but doesn’t want any one in the public zone to read it.

So **she encrypts the message with a secret key that they have shared with each other** before. Then she sends the encrypted message to Bob via the public internet.

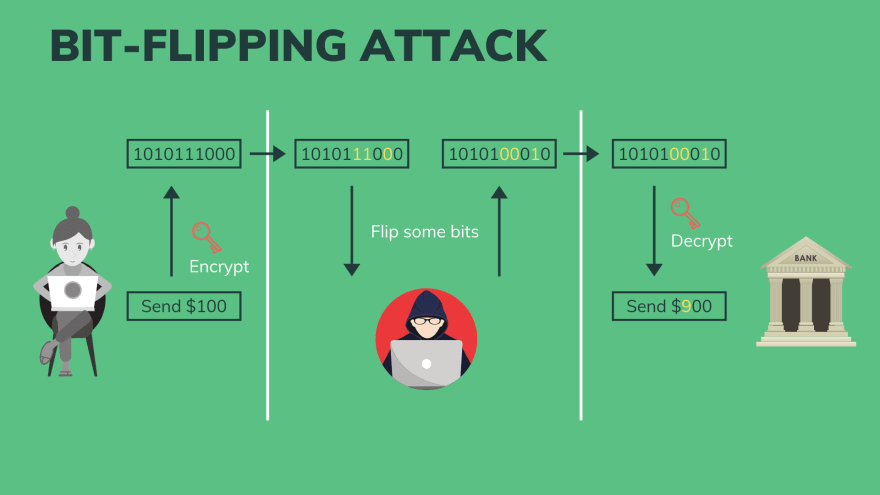
Upon receiving the encrypted message, **Bob will easily use the same secret key to decrypt it.**

Since the same key is used for encryption and decryption, it’s kind of symmetric, so we have the name symmetric cryptography.

Now there might be a hacker Harry, who can catch their exchanged message on the public network. However, the message is already encrypted, and Harry doesn’t have the secret key, so he won’t be able to decrypt it.

But he can still change it!

## 7.1 Bit-flipping attack

[](https://res.cloudinary.com/practicaldev/image/fetch/s--aXErqwZZ--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/dev-to-uploads.s3.amazonaws.com/i/1xpcibbd4haxeh2b0lya.png)

There’s one technique called bit-flipping attack that works like this:

Let’s say this time Alice is not talking to Bob, but talking to her online bank. And she wants to send 100 dollars to someone. The message is encrypted with a secret key and sent to the bank via the internet.

Now Harry catches the encrypted message. Although he can’t decrypt it, he can flip some of its bits from 1 to 0 and from 0 to 1, then forward that modified message to the bank.

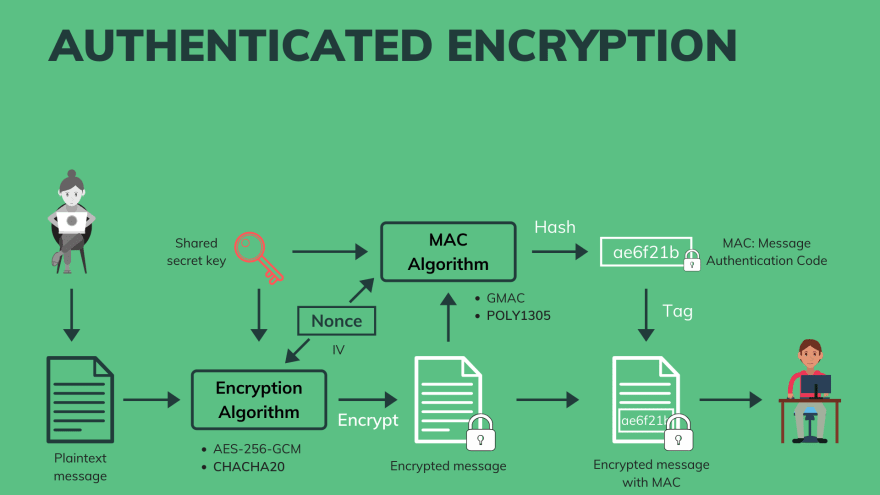
Now when the bank decrypts it, they will get a different plaintext content. In this case, it has become 900 dollars instead of 100.

So it’s very dangerous. That’s why we need to make sure that the encrypted message hasn’t been altered during transmission.

But how?

## 7.2 Authenticated Encryption (AE)

One way to do that is to use Authenticated Encryption. The idea is to not just encrypt, but also authenticate the encrypted message.

[](https://res.cloudinary.com/practicaldev/image/fetch/s--fVTm5C6a--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/dev-to-uploads.s3.amazonaws.com/i/g48ih13sohj4zlad9ch0.png)

**The first step is to encrypt.**

Alice’s plaintext message goes through a symmetric **encryption algorithm**, such as AES-256-GCM or CHACHA20.

This encryption algorithm also takes **a shared secret key** and a **random nonce**, or an initialization vector (IV) as input. And it will return the encrypted message.

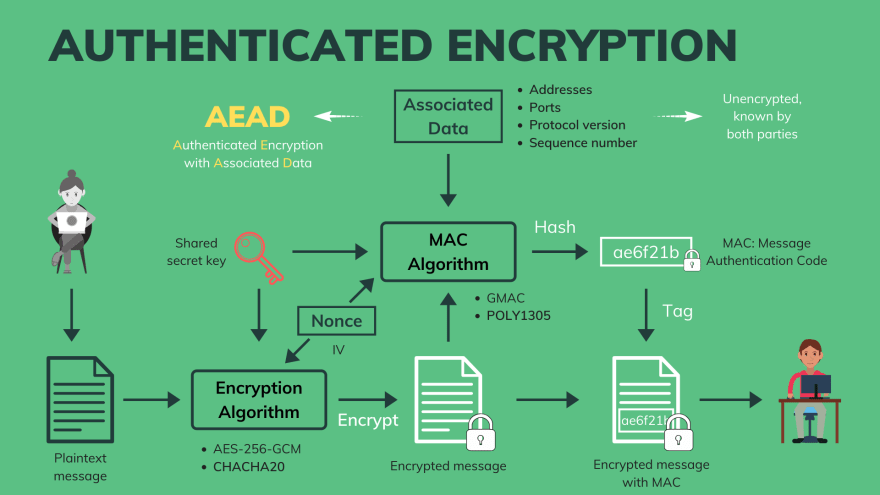
**The second step is to authenticate.**

**The encrypted message, the secret key, and the nonce** become inputs of a MAC algorithm, such as GMAC if you use AES-256-GCM, or POLY1305 if you use CHACHA20 encryption algorithm.

This MAC algorithm acts like a cryptographic hash function, and its output is a MAC, or message authentication code.

Now this MAC will be tagged along with the encrypted message, and the final result will be sent to Bob. Because of this, we sometimes call this MAC an authentication tag.

**Add some Associated Data (AD)**

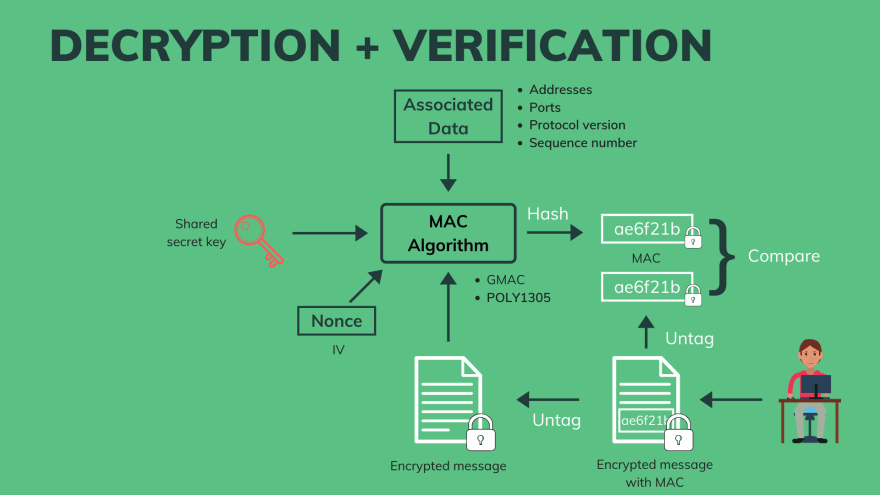
[](https://res.cloudinary.com/practicaldev/image/fetch/s--u6v1yc4w--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/dev-to-uploads.s3.amazonaws.com/i/by4s8o9wuunmb2tw7zkp.png)

In TLS 1.3, besides the encrypted message, we also want to authenticate some associated data, such as: the addresses, the ports, the protocol version, or the sequence number. This information is unencrypted and known by both communicating parties.

So the associated data is also an input of the MAC algorithm. And because of this, the whole process is called Authenticated Encryption with Associated Data, or in short, **AEAD**.

**Decryption and MAC verification**

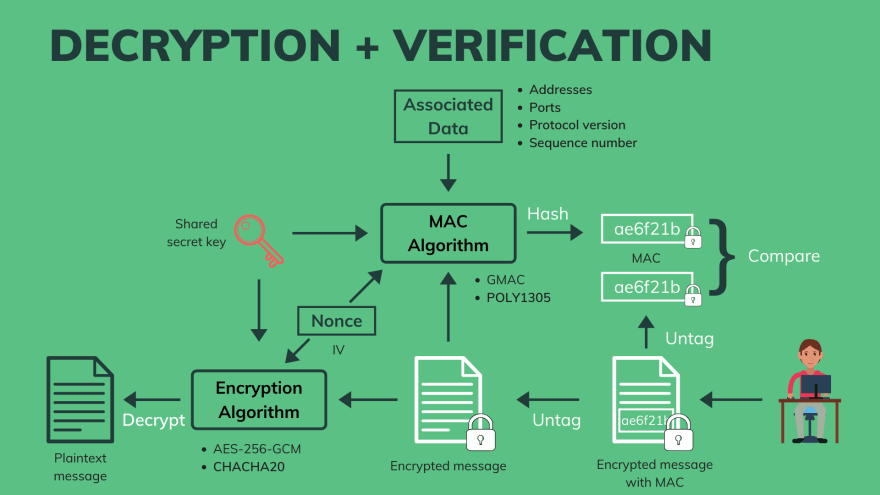
Now let's see how Bob can check that the encrypted message hasn’t been changed during transmission.

[](https://res.cloudinary.com/practicaldev/image/fetch/s--vpOtpKRs--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/dev-to-uploads.s3.amazonaws.com/i/02ir295spos6dii1dtck.png)

It’s simply a reverse process. Starting with the encrypted message with MAC, we untag the MAC from the encrypted message.

Then the encrypted message will go to the MAC algorithm together with the shared secret key and the nonce. Note that this is the same nonce that is used in the encryption process. Usually the nonce is padded to the encrypted message before sending to the receiver.

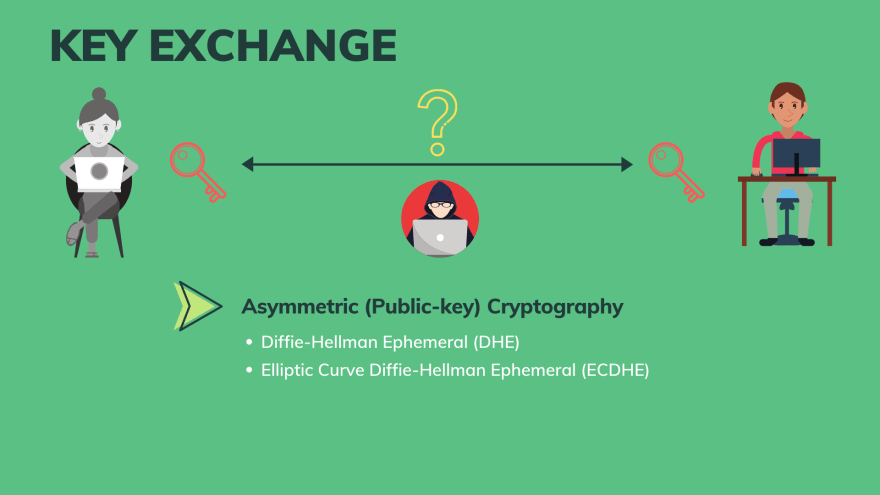
The associated data will also go into the MAC algorithm. And the output of it will be another MAC.

[](https://res.cloudinary.com/practicaldev/image/fetch/s--lwpm6ToG--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/dev-to-uploads.s3.amazonaws.com/i/niaw3gmz718b6s6rnfwm.png)

Now Bob can simply compare the 2 MAC values. If they’re different then he knows that the encrypted message has been changed. Else, he can safely decrypt the message and use it with the confident that it’s the same plaintext message that Alice sent.

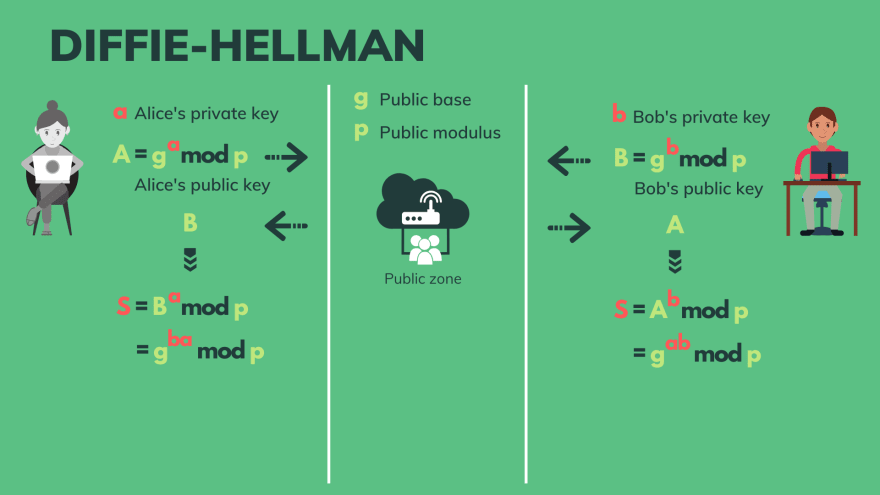
## 7.3 Secret key exchange

However, there’s 1 question: How Bob and Alice **share the secret key** with each other without leaking it to the public?

[](https://res.cloudinary.com/practicaldev/image/fetch/s--imtDi0Yp--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/dev-to-uploads.s3.amazonaws.com/i/lglvuajoqugqkishyzah.png)

Well, the answer is: they need to use asymmetric or public-key cryptography for that purpose. Specifically, they can use either Diffie-Hellman Ephemeral, or Elliptic-Curve Diffie-Hellman Ephemeral.

## 7.3.1 Diffie-Hellman key exchange

[](https://res.cloudinary.com/practicaldev/image/fetch/s--D-EFgolV--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/dev-to-uploads.s3.amazonaws.com/i/klhx6tgnjzkiq9w9n8o6.png)

Let’s see how Diffie Hellman key-exchange works!

**First, Alice and Bobs both agree on 2 numbers**: the base g, and the modulus p. These numbers are known publicly by everyone.

Then each of them secretly choose a private number. **Alice’s private key is number a**, and **Bob’s private key is number b**.

**Then Alice computes her public key and sends it to Bob**:

**A** = (g^a) mod p

**Similarly, Bob computes his public key and sends it to Alice**:

**B** = (g^b) mod p

**Then Alice will receive Bob’s public key B, and Bob will receive Alice’s public key A**.

Now the magic happens!

Alice computes:

**S** = (B^a) mod p

Bob computes:

**S** = (A^b) mod p

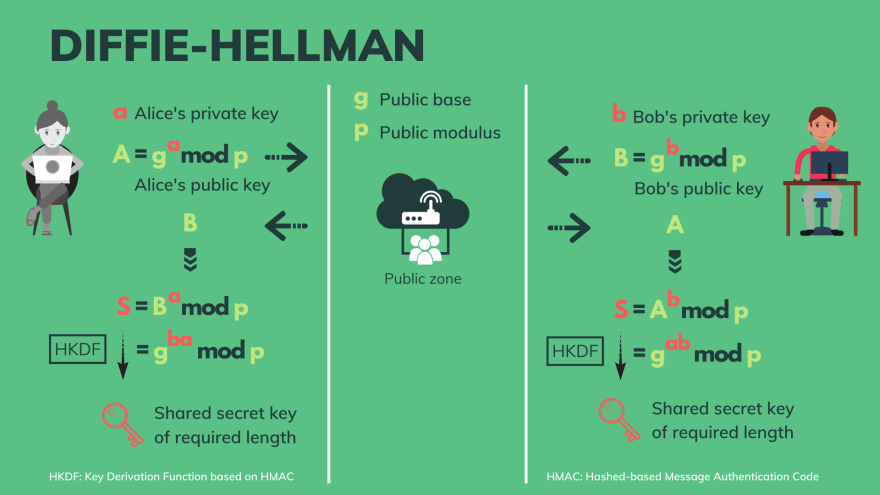
**And these 2 values magically equal to the same number S**.

Why? Let's do the math:

(B^a) mod p = (g^b)^a mod p = ( g^(b\*a) ) mod p

(A^b) mod p = (g^a)^b mod p = ( g^(a\*b) ) mod p

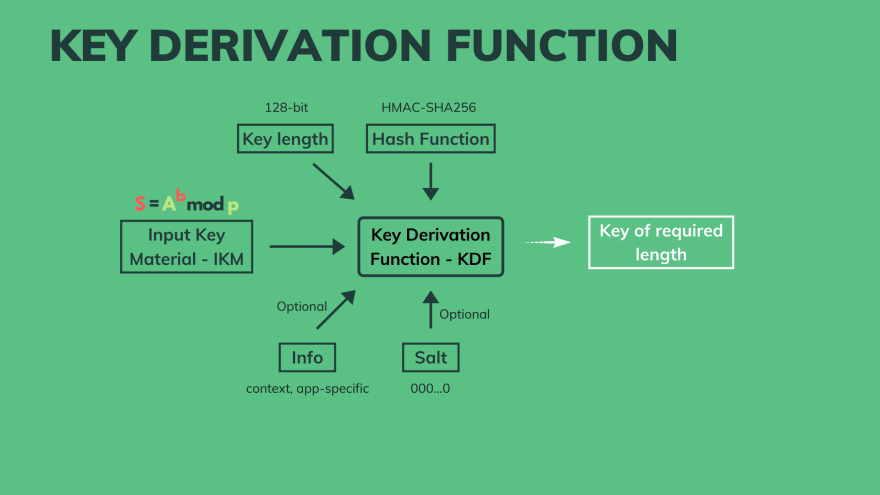
**So Alice and Bob come up with the same secret number S** without leaking it to the public.

[](https://res.cloudinary.com/practicaldev/image/fetch/s--KkJ0THo6--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/dev-to-uploads.s3.amazonaws.com/i/qiun4qnrbh5htimuhpvz.png)

## 7.3.2 Key Derivation Function - KDF

**Each encryption algorithm may require a secret key of different length**. So to make the secret key, Alice and Bob must put S to the same key derivation function (KDF), and the output will be a shared secret key of required length.

In TLS 1.3, we use a HMAC-based key derivation function, so that’s why the name HKDF.

[](https://res.cloudinary.com/practicaldev/image/fetch/s--IXNPCm4E--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/dev-to-uploads.s3.amazonaws.com/i/edabm5a7ke8xm7096wx5.png)

Generally, the KDF takes following inputs:

* An input key material (or IKM). In our case, the IKM is the number S.
* How long we want the output key to be, such as 128-bit.
* A cryptographic hash function, such as HMAC-SHA256.
* Optionally some context or application-specific information
* An optional salt.

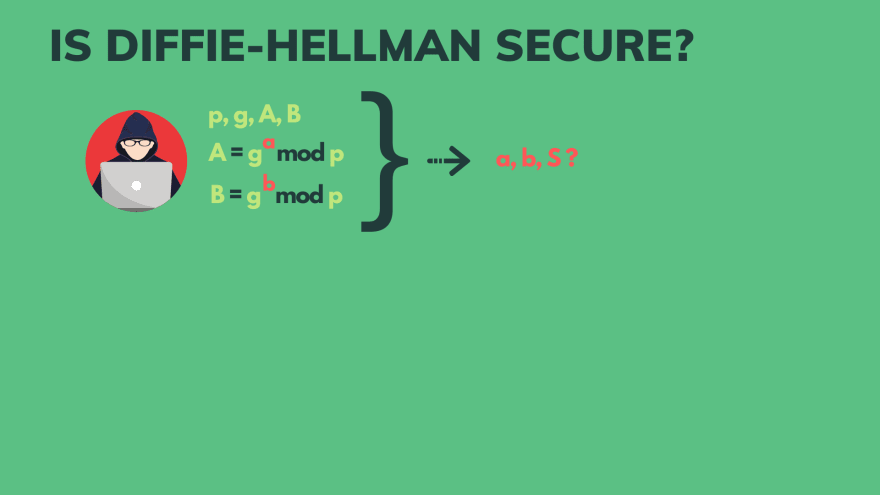
With all of these inputs, KDF will produce a secret key of required length.

## 7.3.3 Trapdoor function

Now let's get back to the Diffie-Hellman key exchange.

We know that **p, g, A, B are known to the public**, which means the hacker, Harry, also has access to those numbers.

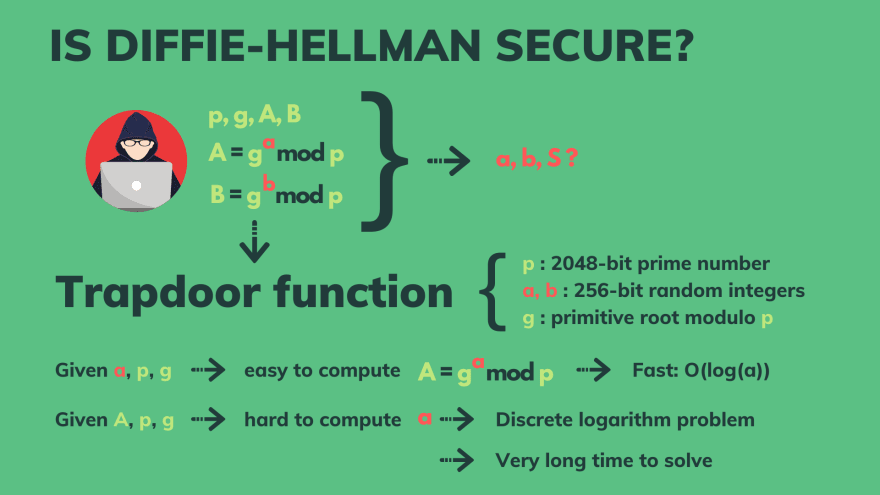
We may wonder: Is this key exchange mechanism secure? Or given p, g, A, B, can Harry figure out the **secret numbers: a, b, S**?

[](https://res.cloudinary.com/practicaldev/image/fetch/s--MsfrB4Ij--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/dev-to-uploads.s3.amazonaws.com/i/bvizk41w4j43ofw9uyld.png)

Fortunately, these functions will be come trapdoors if we choose good values for p, g, a, b.

For example:

* Choose p as a 2048-bit prime number,
* Choose g as a primitive root modulo p,
* And choose a, b to be 256-bit random integers.

[](https://res.cloudinary.com/practicaldev/image/fetch/s--ZjfMnNxT--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/dev-to-uploads.s3.amazonaws.com/i/xs2x9vltcf7dem4btmo2.png)

A trapdoor function basically means it’s easy to compute in one way but hard in the other. In this case:

* Given p, g, a, its’s easy to compute A.
* But given p, g, A, it’s very hard to compute a.

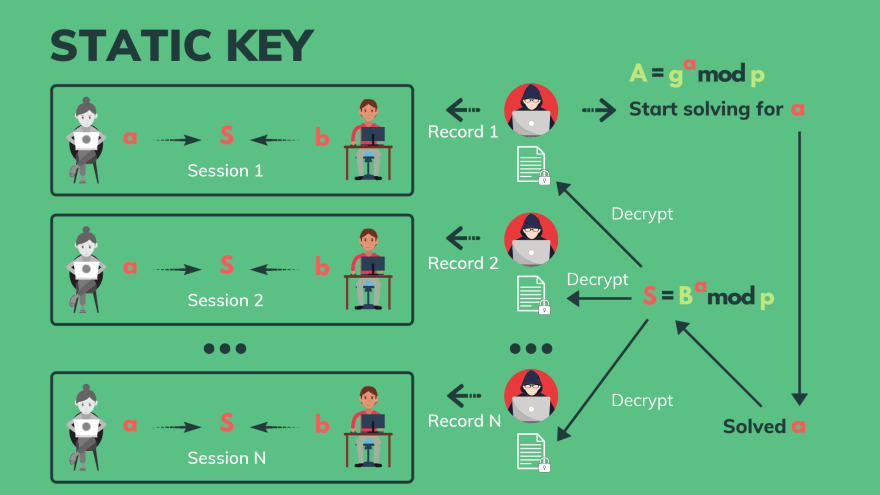
It’s easy to see that A can be computed pretty fast with O(log(a)) time complexity. It’s a well-known [Modular exponentiation problem](https://en.wikipedia.org/wiki/Modular_exponentiation).

Computing a, on the other hand, is much harder. It’s a [Discrete logarithm problem](https://en.wikipedia.org/wiki/Discrete_logarithm), which takes our current-generation of computers a very long time to solve.

So we’re at least safe for now, or until the next generation of strong quantum-computers comes into play.

However, for now, “a long time to solve” doesn’t mean unsolvable, right?

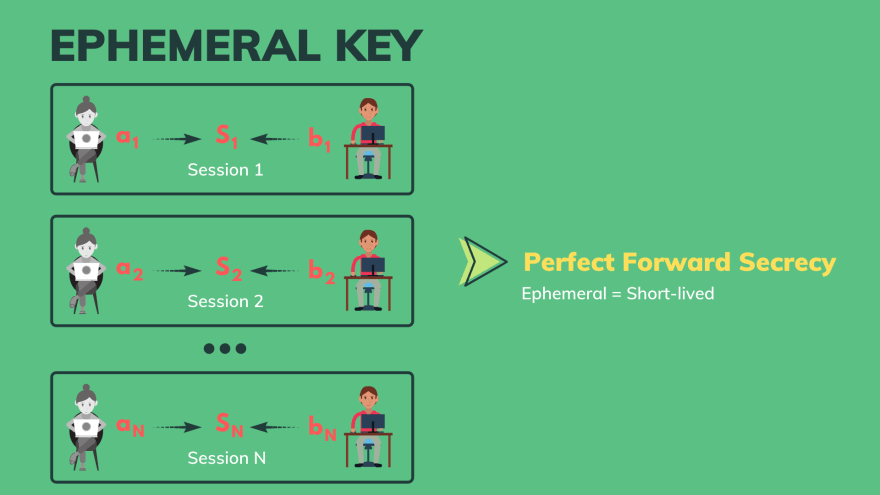
## 7.3.4 Static or Ephemeral key?

[](https://res.cloudinary.com/practicaldev/image/fetch/s--KK2feu2v--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/dev-to-uploads.s3.amazonaws.com/i/yrjca18zd7te7z626rrj.png)

If Alice and Bob use the same private keys a, and b for every sessions that they communicate, then what happens is, Harry can record all of those sessions, and start solving for a from the session 1.

Although it will take him a long time to solve it, let’s say after session N, he gets the right a. Now he can use it to compute the secret number S, and thus, he would be able to decrypt all of the recorded conversations.

Does it sound scary? How can we prevent it?

[](https://res.cloudinary.com/practicaldev/image/fetch/s--gHZ6O-Nf--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/dev-to-uploads.s3.amazonaws.com/i/zgaezee4pjc2s6ejiawu.png)

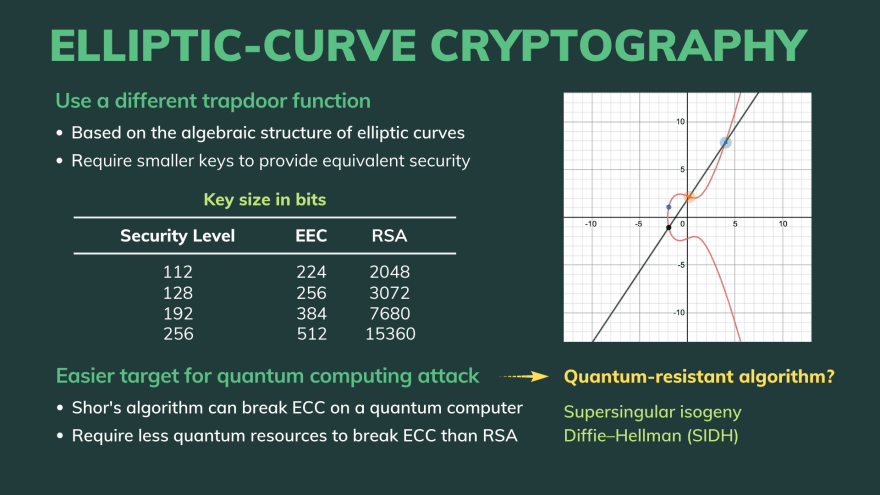
The answer is ephemeral key. As the name may suggest, **we use different private key or each session**. So even if Harry can solve the secret key for 1 session, he could not use it for other ones.

This is called perfect forward secrecy in TLS.

So now you understand what Diffie-Hellman Ephemeral means. It’s just Diffie-Hellman with ephemeral or short-lived keys.

How about Elliptic-Curve Diffie-Hellman Ephemeral?

# **8. Elliptic-Curve Cryptography**

[](https://res.cloudinary.com/practicaldev/image/fetch/s--O1bz4f1G--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/dev-to-uploads.s3.amazonaws.com/i/ax54m5xjp0quwyvg898f.png)

Elliptic-curve cryptography (or ECC) is an approach to asymmetric cryptography, where the algorithm is similar, but a different trapdoor function is used.

That trapdoor function is based on the algebraic structure of elliptic curves. And that’s why the name.

One amazing value of elliptic curve cryptography is: it requires smaller keys to provide the equivalent security level. You can see it in this comparison with RSA.

The U.S. National Security Agency (NSA) used to protect their top secret with ECC 384-bits key, which provides the same security level with a RSA-7680 bit key.

Sounds amazing, right?

However, Elliptic curve cryptography is an easier target for quantum-computing attack. [Shor’s algorithm](https://en.wikipedia.org/wiki/Shor%27s_algorithm) can break ECC on a hypothetical quantum computer with less amount of quantum resources than to break RSA.

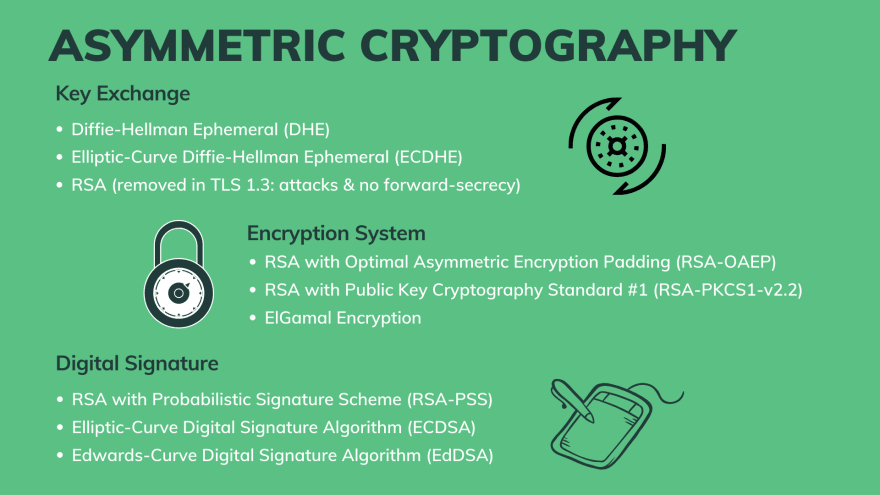
There might be decades before that strong quantum computer actually be built and used. But have we prepared anything for that yet? Is there any quantum-resistant algorithm?

Yes, there is [Supersingular Isogeny Diffie-Hellman](https://en.wikipedia.org/wiki/Supersingular_isogeny_key_exchange) key exchange algorithm, which is also based on the Elliptic Curve Cryptography.

But that’s another story though.

# **9. Asymmetric cryptography**

Now let’s get back to asymmetric cryptography! It’s an awesome technology that has a wide range of applications.

[](https://res.cloudinary.com/practicaldev/image/fetch/s--D7_nmjKL--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/dev-to-uploads.s3.amazonaws.com/i/i8panmd180g4w5z3fv1q.png)

We’ve already explored 1 of its application, which is for symmetric secret key exchange, with Diffie-Hellman Ephemeral and Elliptic-Curve Diffie-Hellman Ephemeral.

In fact, RSA algorithm was also used for key exchange in the past, but it has been removed in TLS 1.3 due to various attacks and no forward-secrecy capability.

Asymmetric cryptography is also used in encryption system. Here are asymmetric encryption algorithms:

* RSA with optimal asymmetric encryption padding (RSA-OAEP).
* RSA with public key cryptography standard 1 (RSA-PKCS1) with the latest version 2.2
* Elgamal Encryption algorithm.

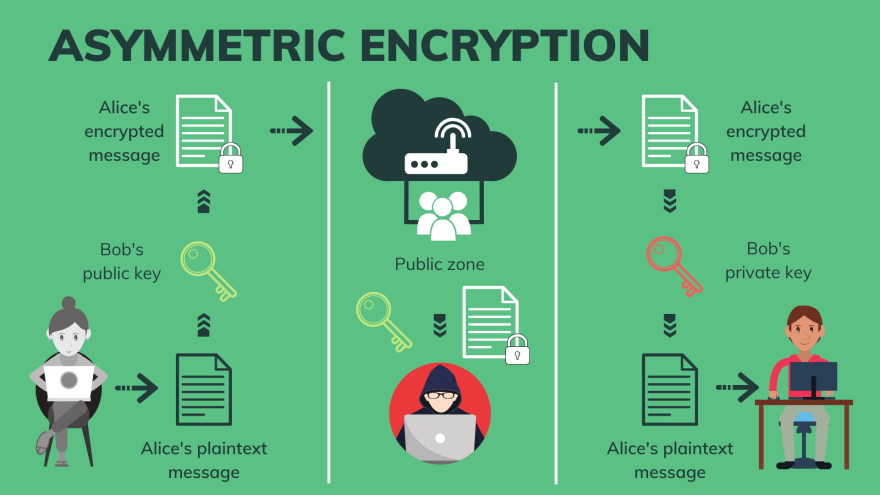
And finally, another important feature of asymmetric cryptography is for **digital signature, which TLS uses extensively for authentication**.

Some popular digital signature algorithms used in TLS are:

* RSA with Probabilitic Signature Scheme.
* Elliptic-Curve Digital Signature Algorithm.
* Edwards-Curve Digital Signature Algorithm.

We will learn about digital signature shortly. But before that, let’s learn how asymmetric encryption system works.

## 9.1 Asymmetric Encryption

[](https://res.cloudinary.com/practicaldev/image/fetch/s--ih9lKTqi--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/dev-to-uploads.s3.amazonaws.com/i/qq8946qj1405vkq8j5ut.png)

Similar as in symmetric encryption, Alice has a plaintext message that she wants to send to Bob.

But this time, there’s no shared secret key. Instead, **Alice encrypt the message with Bob’s public key, and send the encrypted message to Bob.**

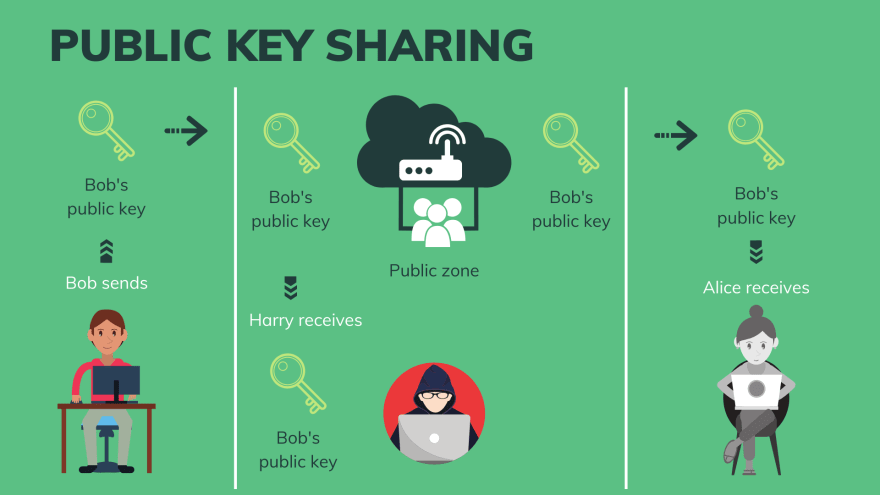
**When Bob receives the message, he uses his private key to decrypt it.**

Although the public key and private key are different, they are still connected by some trapdoor function, just like what we’ve seen in the Diffie-Hellman algorithm.

The idea is: **keys come in pair, and only the private key of the same pair can decrypt the message encrypted with its public key**.

Because of this, even when Harry the hacker has access to both Alice’s encrypted message and Bob’s public key, he cannot use that public key to decrypt the message.

## 9.2 Public key sharing

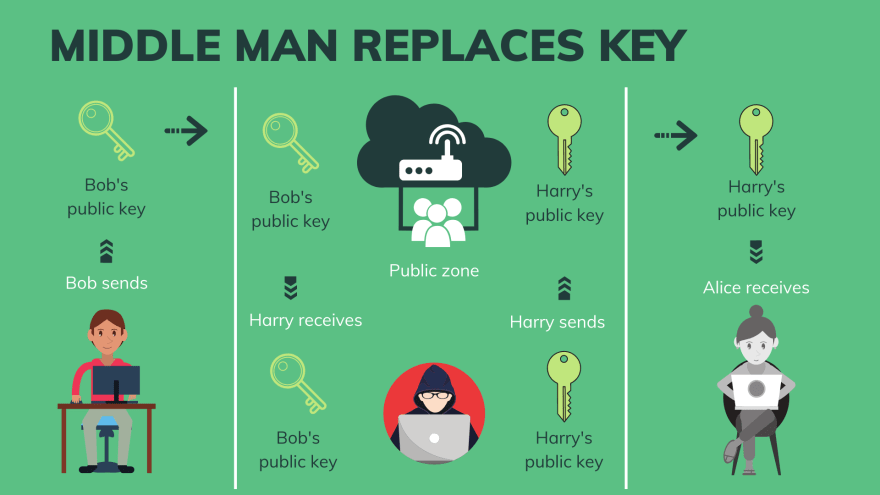
[](https://res.cloudinary.com/practicaldev/image/fetch/s---Y2kFQCU--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/dev-to-uploads.s3.amazonaws.com/i/d0rhcuwf85uuvie5r7ne.png)

The public key sharing is very simple**. Bob just send the key to Alice directly over the public internet** without the fear that the key can be used to decrypt any messages.

The key is public, so anyone can use it to encrypt messages that only Bob can read, even if they have never talked to each other before. It’s really mind-blowing, isn’t it?

However, life’s not that so easy!

## 9.3 Man-in-the-middle swaps the key

[](https://res.cloudinary.com/practicaldev/image/fetch/s--0LytWvl---/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/dev-to-uploads.s3.amazonaws.com/i/4dgg5bm4ughvmthfsy5o.png)

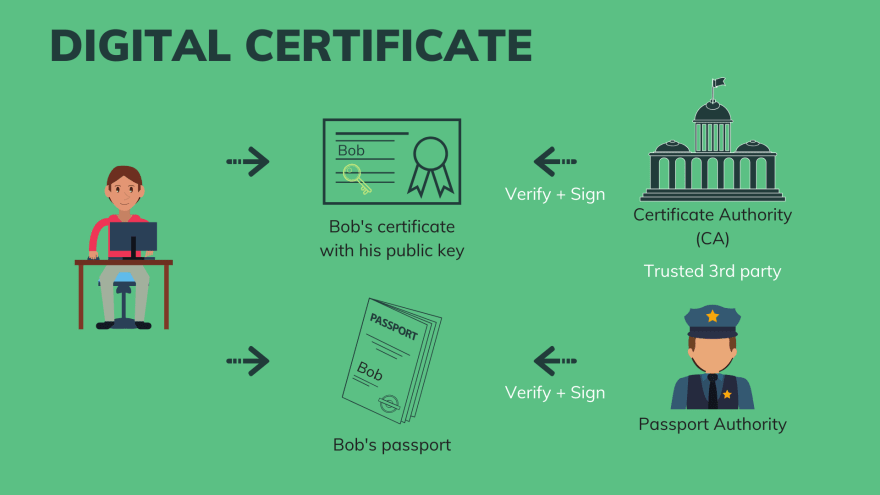
Although we know that Harry cannot decrypt the message with Bob’s public key, **he can still interfere with the public key sharing, and replace Bob’s public key with his own public key**.

**Now when Alice receive the key, she still thinks it’s Bob’s public key, but it’s in fact Harry’s. So if Alice encrypts her message with this key, Harry would be able to decrypt it with his private key**.

The reason this can happen is because a key is simply just a number, and there’s no identity information to tell us who its owner is.

So what can we do? Obviously, we should put the key together with some identity information. And that’s nothing else but a digital certificate.

## 9.4 Digital certificate

[](https://res.cloudinary.com/practicaldev/image/fetch/s--e5OyZU4t--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/dev-to-uploads.s3.amazonaws.com/i/6x1y71uy3ipqu1l0y05i.png)

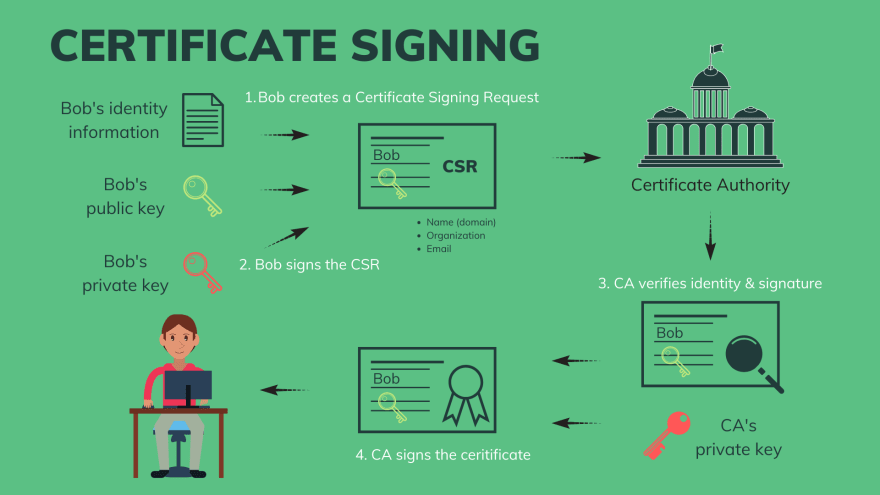
So **Bob puts his key inside his certificate**, which has his name and other identity information on it. The certificate acts like a passport in the real world.

But how do we know it’s really Bob who owns that certificate? What stops Harry from making a fake certificate under Bob’s name but with Harry’s public key?

Well, just like in the real world, the passport must be issued by a passport authority after a process of identity verification. In digital world, the certificate must be verified and signed by a certificate authority.

This certificate authority and passport authority are trusted third party, who helps us prevent creation of fake passport and digital certificates.

**Certificate signing**

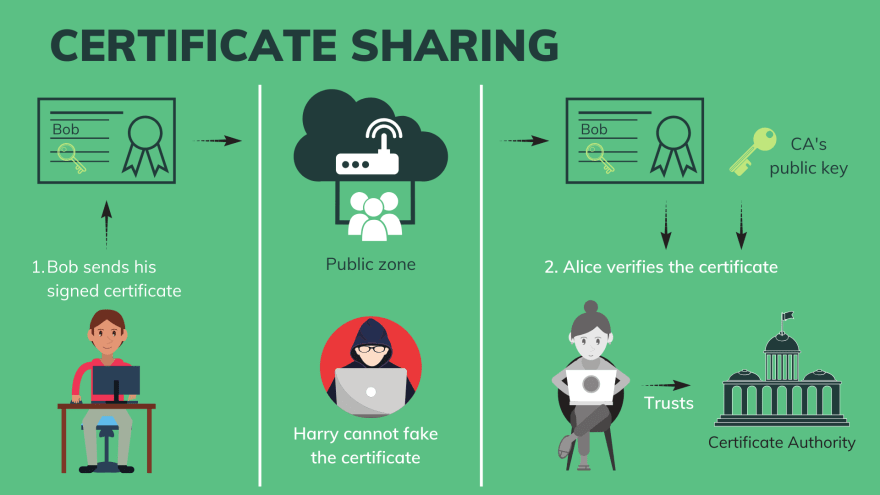
[](https://res.cloudinary.com/practicaldev/image/fetch/s---gjgu4eT--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/dev-to-uploads.s3.amazonaws.com/i/qa3ed04i785g6sunvwmg.png)

**The certificate signing process happens like this**:

**CA : Certificate Authority**

* Bob has a pair of public and private key.
* In the first step, he creates a certificate signing **request**, or CSR. This CSR contains **his public key** and **some identity information**, such as his name, organization, and email.
* Then the second step, he **signs the CSR with his private key**, and sends it to the certificate authority.
* The certificate authority will verify Bob’s identity in the certificate. They can contact him to ask for more proof if necessary.
* Then they use Bob’s public key in the certificate to verify his signature. This is to make sure that Bob really owns the private key that paired with the public key in the certificate.
* If everything is valid, the **CA will sign the certificate with their own private key**, and send it back to Bob.

**Certificate sharing**

[](https://res.cloudinary.com/practicaldev/image/fetch/s--nMeofZsf--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/dev-to-uploads.s3.amazonaws.com/i/dfha869i7tb2rhbwovxd.png)

Now **Bob will share with Alice** this **certificate,** which contains his public key, instead of sending just the public key as before.

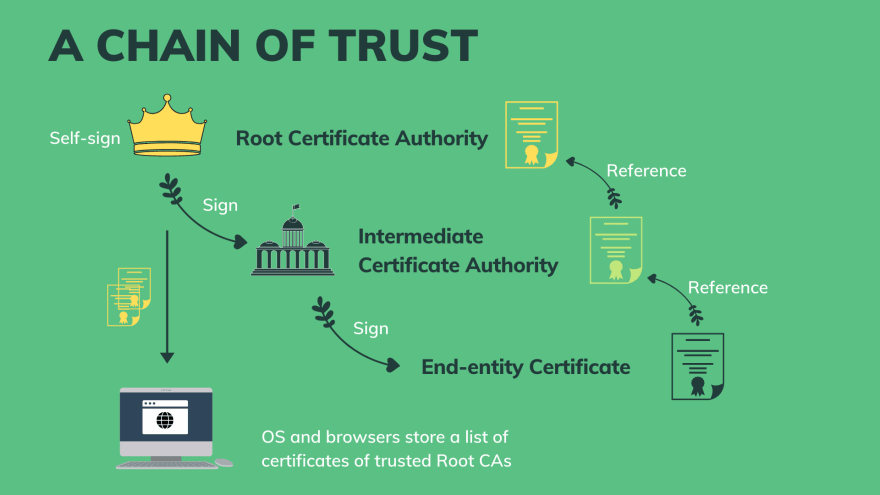
Upon receiving the certificate, **Alice can easily verify its authenticity with the public key of the Certificate authority**.

Because of this, Harry cannot replace Bob’s public key with his key anymore, since he doesn’t have the CA’s private key to sign the fake certificate.

Note that this only works because we all trust the Certificate Authority. If somehow the CA is not trustworthy, for example, if they give Harry their private key, then we’re in a serious trouble!

## 9.5 Certificate Authority - A chain of trust

In reality, there’s a chain of certificate authorities.

[](https://res.cloudinary.com/practicaldev/image/fetch/s--uarduLh1--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/dev-to-uploads.s3.amazonaws.com/i/q8oytic2nbywqj15n2d3.png)

At the top level is a root certificate authority, who signs their own certificate, and also signs the certificate of their subordinate, which is an intermediate certificate authority.

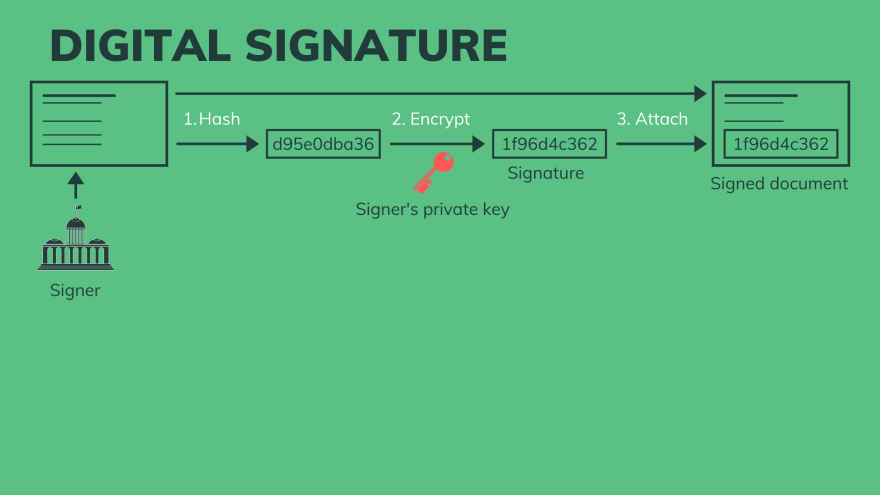
This authority can sign the certificate of other intermediate authorities, or they can sign the end-entity certificate (or leaf certificate).

Each certificate will reference back to the certificate of their higher level authority, up to the root.

Your operating systems and browsers store a list of certificates of trusted root certificate authorities. That way they can easily verify the authenticity of all certificates.

## 9.6 Digital signature

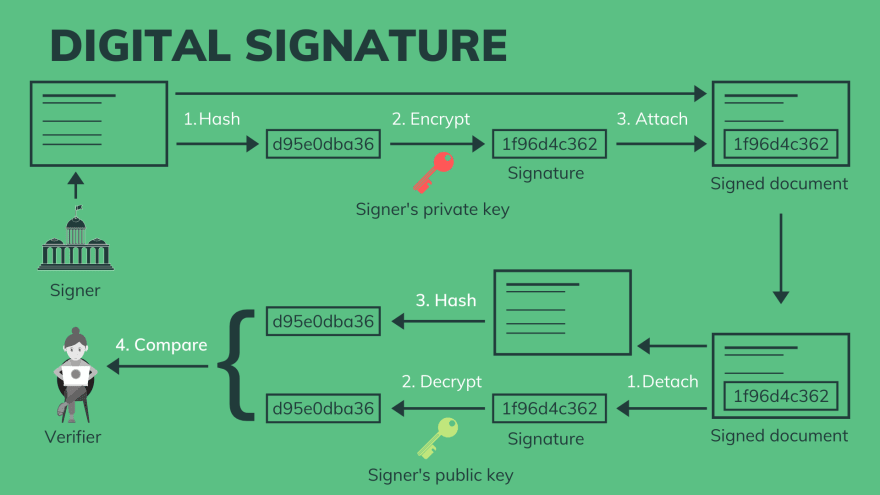
We’ve talked a lot about signing a digital signature, so let’s see how it really works!

[](https://res.cloudinary.com/practicaldev/image/fetch/s--RPySdjlN--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/dev-to-uploads.s3.amazonaws.com/i/ctr1kbvxifx1lwcf6e4l.png)

To sign a document:

* The signer first need to hash it.
* Then the hash value is encrypted using the signer’s private key.
* The result will be the digital signature.
* Then this signature will be attached to the original document.

And that’s it for the signing process. Now how can we verify that the signature is valid?

[](https://res.cloudinary.com/practicaldev/image/fetch/s--iD3N4hcH--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/dev-to-uploads.s3.amazonaws.com/i/nailgasrcrzaajb12d53.png)

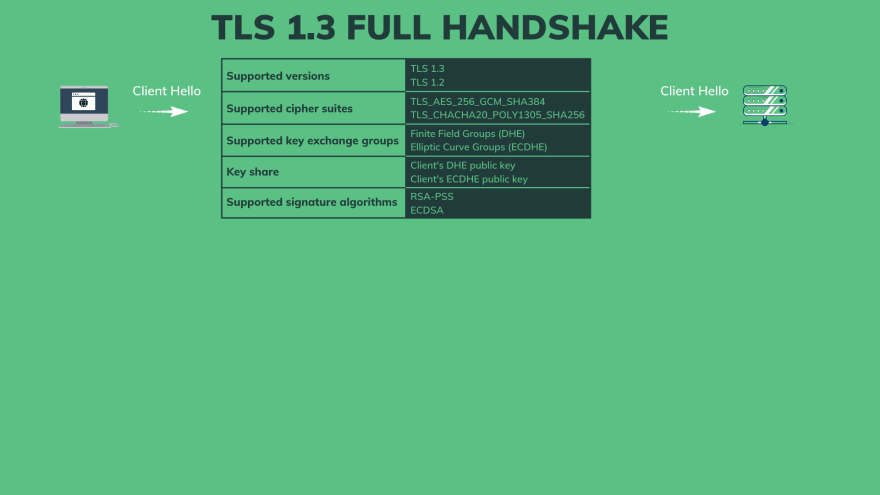
Well, we just do a reversed process:

* First we detach the signature from the document
* Decrypt it with the signer’s public key to get a hash value.
* Then we hash the document with the same hash algorithm used in the signing process.
* The result is another hash value.
* Then we just compare the 2 hash values.
* If they’re the same then the signature is valid.

# **10. TLS 1.3 handshake protocol**

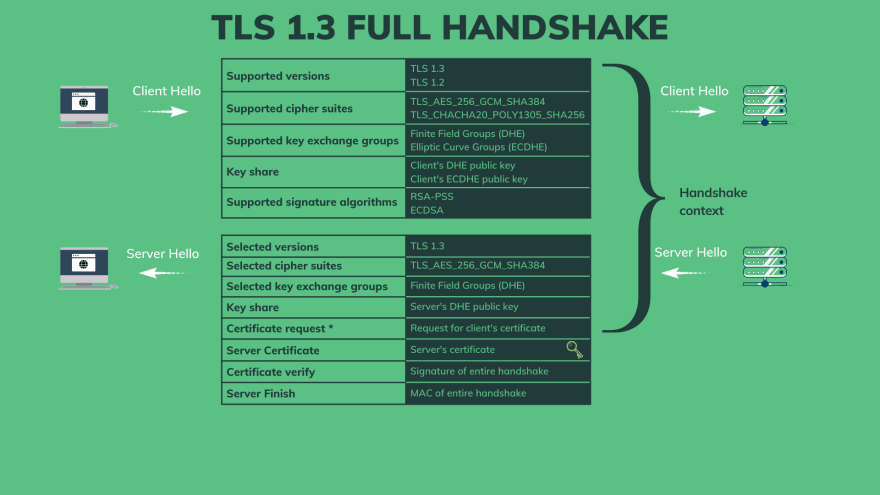
OK, so now with all the knowledge we’ve gained so far, let’s take a closer look at how they’re used in the TLS handshake protocol.

## 10.1 Full handshake

[](https://res.cloudinary.com/practicaldev/image/fetch/s--2nST-rCH--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/dev-to-uploads.s3.amazonaws.com/i/2i12vggsnis0naqxhv34.png)

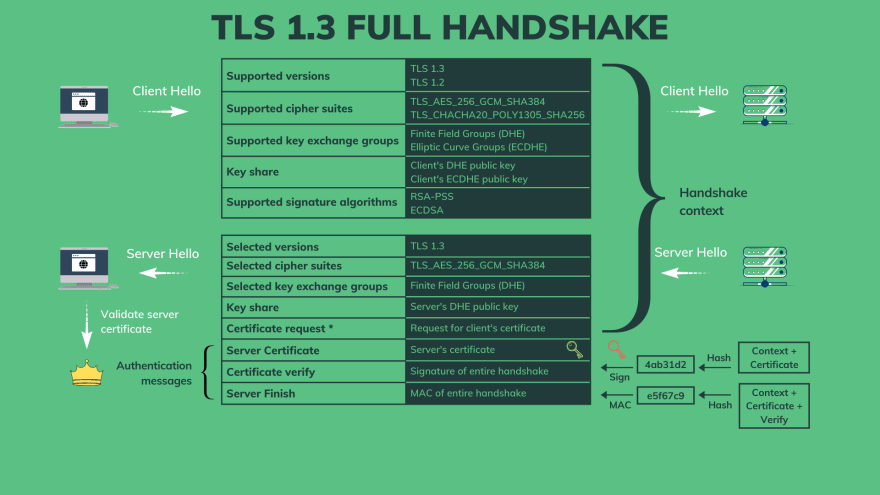
The TLS 1.3 full handshake **starts with a hello message that the client sends to the server**. Actually this message contains a lot of things, but here I just list some important information:

* First, a list of protocol version that client supports.
* Then a list of supported AEAD symmetric cipher suites. In this case, there are 2 options: AES-256-GCM or CHACHA20-POLY1305
* After that, there’s a list of supported key exchange groups. For example, this client supports both Finite field Diffie-Hellman Ephemeral and Elliptic-Curve Diffie-Hellman Ephemeral.
* That’s why client also shares its 2 public keys, 1 for Diffie-Hellman, and the other for Elliptic-Curve Diffie-Hellman. This way, the server will be able to compute the shared secret key no mater what algorithm it chooses.
* The last field client sends in this message is a list of signature algorithms it supports. This is for server to choose which algorithm it should use to sign the whole handshake. We will see how it works in a bit.

[](https://res.cloudinary.com/practicaldev/image/fetch/s--mmLHEmko--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/dev-to-uploads.s3.amazonaws.com/i/mowf2zm6q21gyqsdl1th.png)

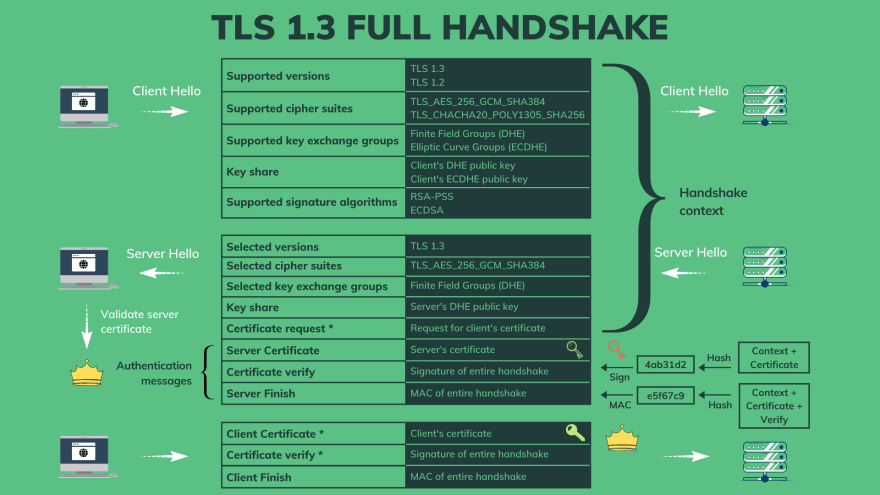
After receiving the client hello message, **the server also sends back its hello message**,which contains:

* The selected protocol version TLS 1.3
* The selected cipher suites: AES-256-GCM
* The selected key exchange method: Diffie-Hellman Ephemeral
* **And the server’s public key** for that chosen method.
* The next field is a request for the **client’s certificate**, which is optional and will only be sent if the server wants to authenticate the client by its certificate.
* Normally on a **HTTPS website**, **only the server side needs to send its certificate to the client**. And that is sent in the next field of this message.

[](https://res.cloudinary.com/practicaldev/image/fetch/s--dXqkm5gy--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/dev-to-uploads.s3.amazonaws.com/i/7bmi3ncwju9ezjz2t6xa.png)

* The next field is certificate verify, which is, in fact, the signature of the entire handshake up to this point. Here’s how it is generated: The whole data from the beginning of the handshake up to the certificate request is called a handshake context. We concatenate this context with the server’s certificate, hash it, and sign the hash value with the server’s private key using 1 of the signature algorithms that the client supports.
* In a similar fashion, the server finish is generated by concatenating the handshake context, the certificate, and the certificate verify, hash it, and put the hash value through the MAC algorithm of the chosen cipher suite. The result is the MAC of the entire handshake.

Here the server certificate, certificate verify, and server finish are called authentication messages, because they are used to authenticate the server. With the signature and MAC of the entire handshake, TLS 1.3 is safe against several types of man-in-the-middle Downgrade attacks.

[](https://res.cloudinary.com/practicaldev/image/fetch/s--gTodBW6o--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/dev-to-uploads.s3.amazonaws.com/i/ymvzwxg30cbdob621fze.png)

**Now after the client receives the hello message from server, it will validate the server’s certificate with the root authority**, and check the signature and MAC of the entire handshake to make sure it’s not been tampered with.

**If everything is good then the client sends its finish message** with the MAC of the entire handshake up to this point, and optionally the client’s certificate and certificate verify in case the server has requested.

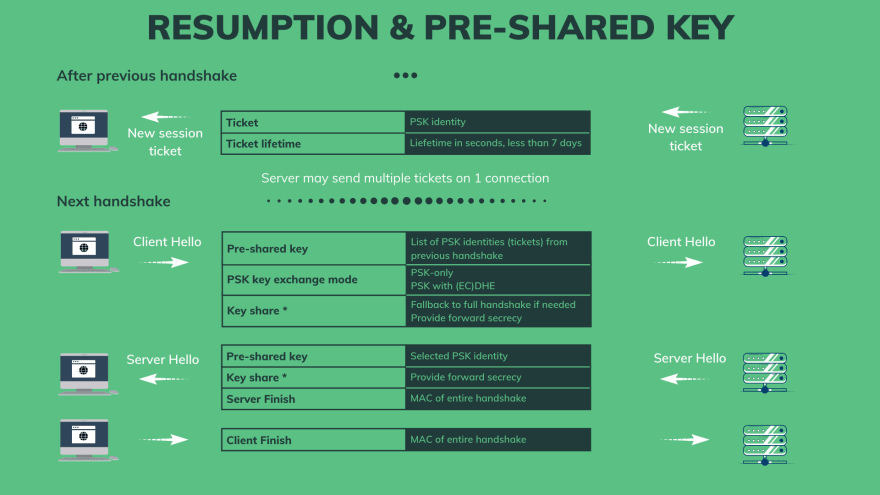
And that’s the whole flow of the full TLS handshake.

## 10.2 Abbreviated handshake with PSK resumption

To improve the performance, the client and server don’t always go through this full handshake. Sometimes, they perform abbreviated handshake by using preshared key resumption.

The idea is: after the previous handshake, the client and server already know each other, so they don’t need to authenticate again.

So the server may send one or multiple session tickets to the client, which can be used as the pre-shared key (PSK) identity in the next handshake. It goes with a ticket lifetime as well as some other information.

[](https://res.cloudinary.com/practicaldev/image/fetch/s--r8ihXoKe--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/dev-to-uploads.s3.amazonaws.com/i/jn5mt50d8nsqyrd1vo1i.png)

Now in the next handshake, the client will send a simple hello message, which contains:

* A list of PSK identities (or tickets) obtained from the previous handshake
* A PSK key exchange mode, which can be either PSK only, or PSK with Diffie-Hellman.
* If the PSK with Diffie-Hellman mode is used, then the client also needs to share its Diffie-Hellman public key. This will provide perfect forward secrecy, as well as allow the server to fallback to full handshake if needed.

When the server receives this client hello message, it sends back its hello with:

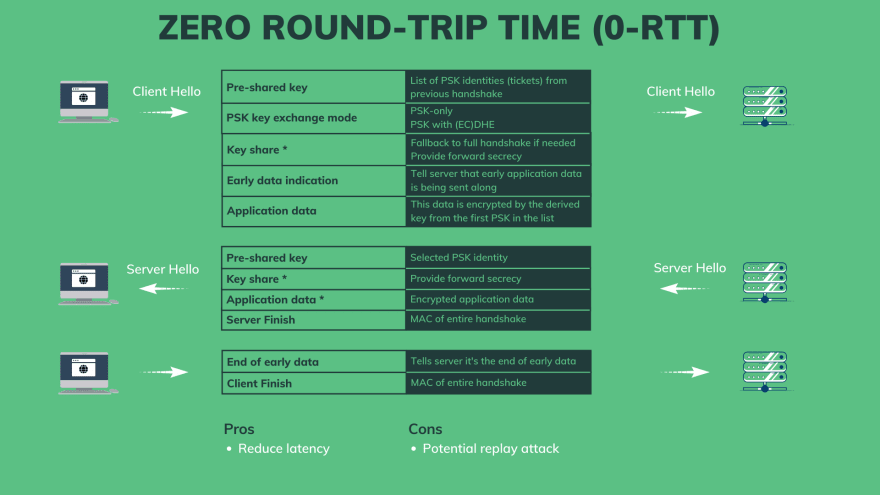
* The selected pre-shared key identity
* The optional Diffie-Hellman public key of the server
* And the server Finish just like in the full handshake.

Finally the client sends back its Finish, and that’s the end of the PSK resumption.

As you can see, there’s no certificate authentication between client and server in this abbreviated handshake.

This also opens up an opportunity for zero round-trip time (0-RTT) data, which means, the client doesn’t need to wait for the handshake to complete to send its first application data to the server.

## 10.3 0-RTT handshake

[](https://res.cloudinary.com/practicaldev/image/fetch/s--pkZpO6nu--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/dev-to-uploads.s3.amazonaws.com/i/ra5s19bgffxu212qyhz7.png)

In 0-RTT, client sends the application data together with the client hello message. This data is encrypted using the key derived from the first PSK in the ticket list.

And it also adds 1 more field: early data indication to tell the server that there’s early application data being sent along.

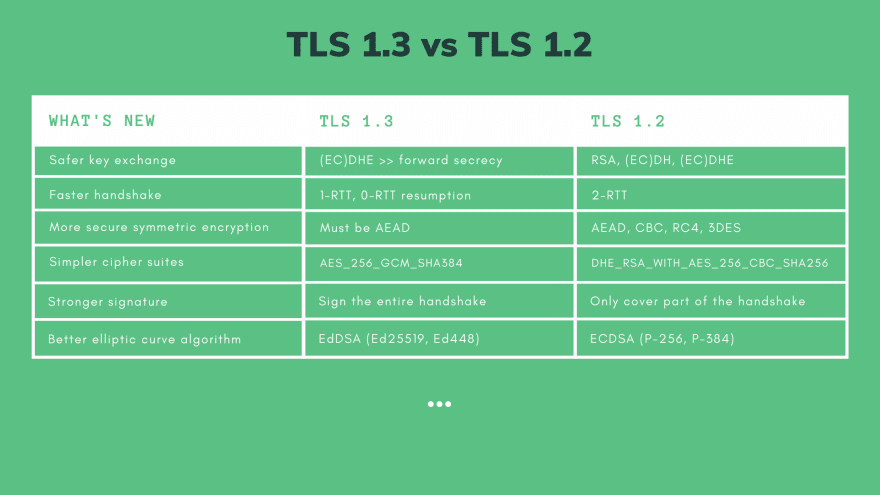
If the server accepts this 0-RTT request, it will sends back the server hello just like in the normal PSK resumption, and optionally some application data as well.

The client will finish with a message containing the MAC, and an end-of-early-data indicator. That’s how 0 round-trip time works in TLS 1.3.

Its pros is reduce the latency by 1 round trip time. But the cons is openning up a potential threat of replay attack. Which means, the hacker can just copy and send the same encrypted 0-RTT request to the server multiple times. To avoid this, the server application must be implemented in a way that’s resilient to duplicate requests.

## 11. Compare TLS 1.3 vs TLS 1.2

Now before we finish, let’s do a quick comparison of TLS 1.3 and TLS 1.2 to see what’s new!

[](https://res.cloudinary.com/practicaldev/image/fetch/s--JaAtPCej--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/dev-to-uploads.s3.amazonaws.com/i/36u728t4pild17fldy1e.png)

1. TLS 1.3 has safer key exchange mechanisms, where the vulnerable RSA and other static key exchange methods are removed, leaving only ephemeral Diffie-Hellman or Elliptic-Curve Diffie-Hellman remain, therefore achieved perfect forward secrecy.
2. TLS 1.3 handshake is at least 1 round-trip faster than TLS 1.2.
3. Symmetric encryption in TLS 1.3 is more secure because AEAD cipher suite is mandatory, and it also removes some weak algorithms from the list such as Block Cipher Mode (CBC), RC4, or Triple DES.
4. The cipher suite in TLS 1.3 is also simpler, since it only contains the AEAD algorithm and a hash algorithm. The key exchange and signature algorithms are moved to separate fields. While in TLS 1.2, they’re merged into the cipher suite. This makes the number of recommended cipher suites become too big, 37 options in TLS 1.2 if i remember correctly. While in TLS 1.3, there are only 5.
5. Next, TLS 1.3 also give us stronger signature, since it signs the entire handshake, not just cover some part of it as in TLS 1.2.
6. Last but not least, Elliptic-curve cryptography gets a significant attention in TLS 1.3, with some better curves algorithm added, such as Edward-curve digital signature algorithm, which is faster without sacrificing security.

And that’s everything I want to share with you in this article. Thanks for reading, and I’ll catch you guys in the next one!

# How to create & sign SSL/TLS certificates

[#security](https://dev.to/t/security) [#tutorial](https://dev.to/t/tutorial) [#webdev](https://dev.to/t/webdev) [#beginners](https://dev.to/t/beginners)

[techschoolguru profile imageTECH SCHOOL](https://dev.to/techschoolguru)Apr 12 ・Updated on Aug 28 ・10 min read

### [SSL/TLS (2 Part Series)](https://dev.to/techschoolguru/series/5898)

[1A complete overview of SSL/TLS and its cryptographic system](https://dev.to/techschoolguru/a-complete-overview-of-ssl-tls-and-its-cryptographic-system-36pd)[2How to create & sign SSL/TLS certificates](https://dev.to/techschoolguru/how-to-create-sign-ssl-tls-certificates-2aai)

In the [previous article](https://dev.to/techschoolguru/a-complete-overview-of-ssl-tls-and-its-cryptographic-system-36pd), we’ve talked about how digital certificates help with authentication and provide a safe and reliable key exchange process in TLS.

Today we will learn exactly how to generate a certificate and have it signed by a Certificate Authority (CA).

For the purpose of this tutorial, we won’t submit our Certificate Signing Request (CSR) to a real CA. Instead, we will play both roles: the certificate authority and the certificate applicant.

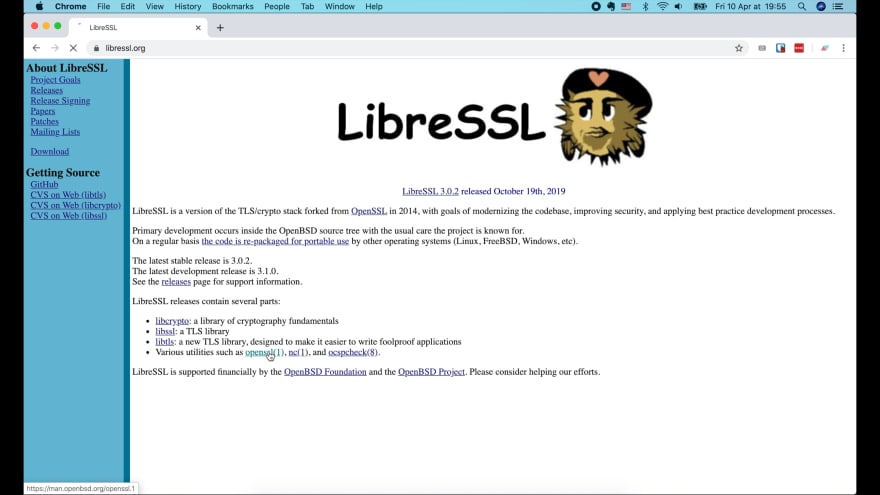
So here's what we're gonna do:

1. In the first step, we will generate a private key and its self-signed certificate for the CA. They will be used to sign the CSR later.
2. In the second step, we will generate a private key and its paired CSR for the web server that we want to use TLS.
3. Then finally we will use the CA’s private key to sign the web server’s CSR and get back the signed certificate.

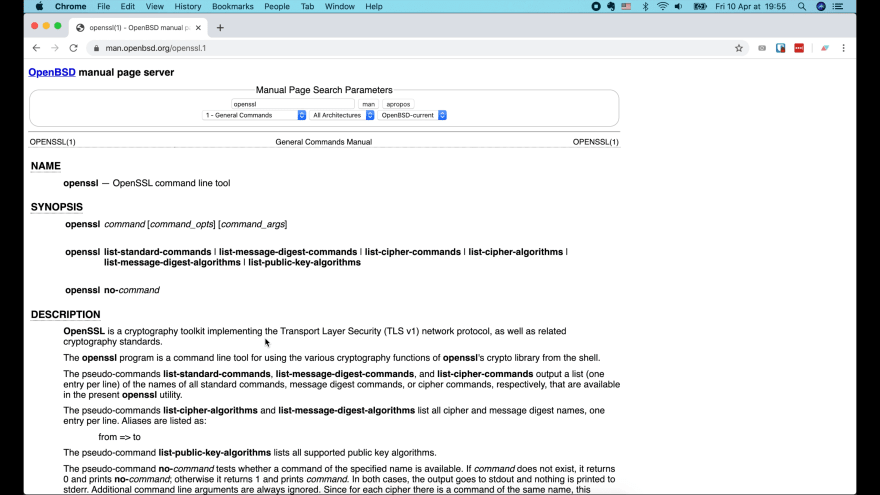
In order to do all of these things, we need to have openssl installed. If you’re on a Mac, it’s probably already there. You can run this command to see which version it’s running:

openssl version

In my case, it’s [LibreSSL](https://www.libressl.org/) version 2.8.3.

[](https://res.cloudinary.com/practicaldev/image/fetch/s--VW4hE1RL--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/dev-to-uploads.s3.amazonaws.com/i/u0g4hf1nevxzno4cvuqo.png)

We can access its manual documentation at [this link](https://man.openbsd.org/openssl.1).

[](https://res.cloudinary.com/practicaldev/image/fetch/s--8FZ9L-k8--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/dev-to-uploads.s3.amazonaws.com/i/tq1e2ar4flz5dv76gxh6.png)

## 1. Generate CA'private key and certificate

The first command we’re gonna used is [openssl req](https://man.openbsd.org/openssl.1" \l "req), which stands for request. This command is used to create and process certificate signing request. It can also be used to create a self-signed certificate for the CA, which is exactly what we want in the first step.

openssl req -x509 -newkey rsa:4096 -days 365 -keyout ca-key.pem -out ca-cert.pem

The -x509 option is used to tell openssl to output a self-signed certificate instead of a certificate request. In case you don’t know, [X509](https://en.wikipedia.org/wiki/X.509) is just a standard format of the public key certificate.

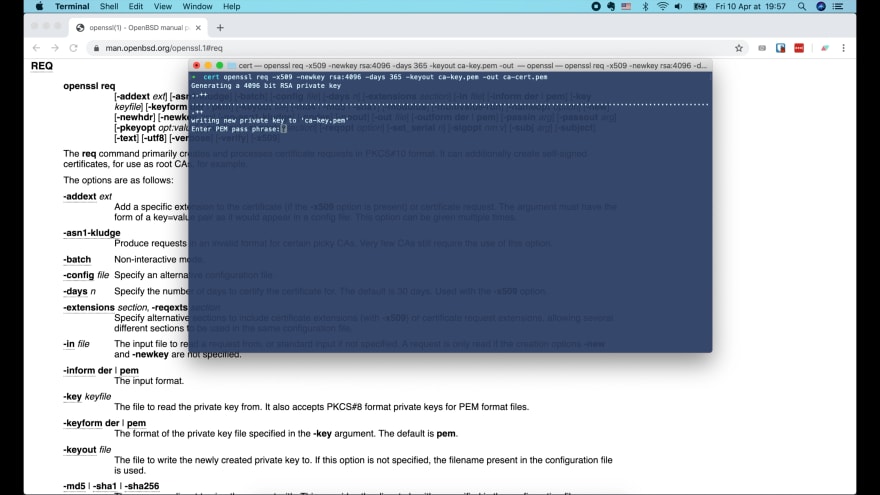
The -newkey rsa:4096 option basically tells openssl to create both a new RSA private key (4096-bit) and its certificate request at the same time. As we’re using this together with -x509 option, it will output a certificate instead of a certificate request.

The next option is -days 365, which specifies the number of days that the certificate is valid for.

Then we use the -keyout option to tell openssl to write the created private key to ca-key.pem file

And finally the -out option to tell it to write the certificate to ca-cert.pem file.

When we run this command, openssl will start generating the private key.

[](https://res.cloudinary.com/practicaldev/image/fetch/s--k7uMBtMh--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/dev-to-uploads.s3.amazonaws.com/i/ud7gi0xpkrxm1ydawb8m.png)

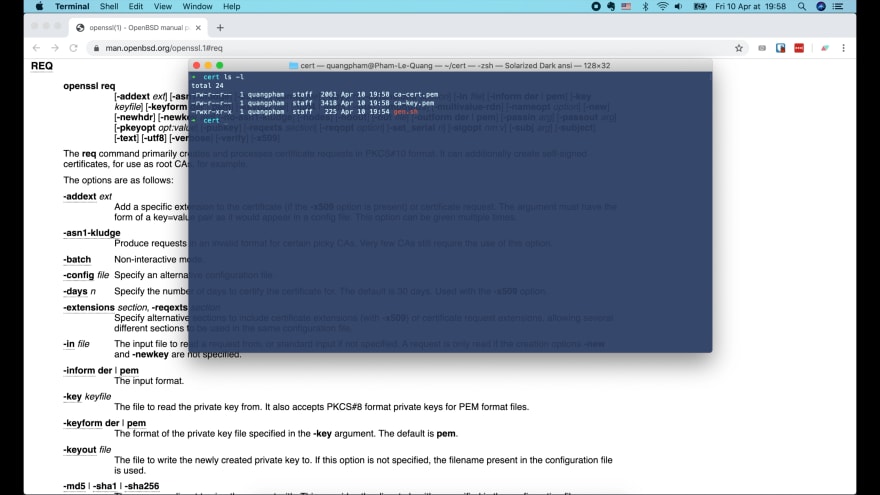
Once the key is generated, we will be asked to provide a pass phrase, which will be used to encrypt the private key before writing it to the PEM file.

Why is it encrypted? Because if somehow the private key file is hacked, the hacker cannot use it to do anything without knowing the pass phrase to decrypt it first.

Next, openssl will ask us for some identity information to generate the certificate:

* The country code
* The state or province name
* The organisation name
* The unit name
* The common name (or domain name)
* The email address

And that’s it! The certificate and private key files will be successfully generated.

[](https://res.cloudinary.com/practicaldev/image/fetch/s--8RG481_o--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/dev-to-uploads.s3.amazonaws.com/i/4sww5640rd34xklp8kgs.png)

If we cat the private key file ca-key.pem, we can see it says ENCRYPTED PRIVATE KEY:

-----BEGIN ENCRYPTED PRIVATE KEY-----

MIIJnzBJBgkqhkiG9w0BBQ0wPDAbBgkqhkiG9w0BBQwwDgQILfki090rvloCAggA

MB0GCWCGSAFlAwQBKg...GNYc7i9SVDBoA==

-----END ENCRYPTED PRIVATE KEY-----

The certificate ca-cert.pem, on the other hand, is not encrypted, but only base64-encoded, because it just contains the public key, the identity information, and the signature that should be visible to everyone.

-----BEGIN CERTIFICATE-----

MIIFxjCCA64CCQCNT+eP2vjJxzANBgkqhkiG9w0BAQsFADCBpDELMAkGA1UEBhMC

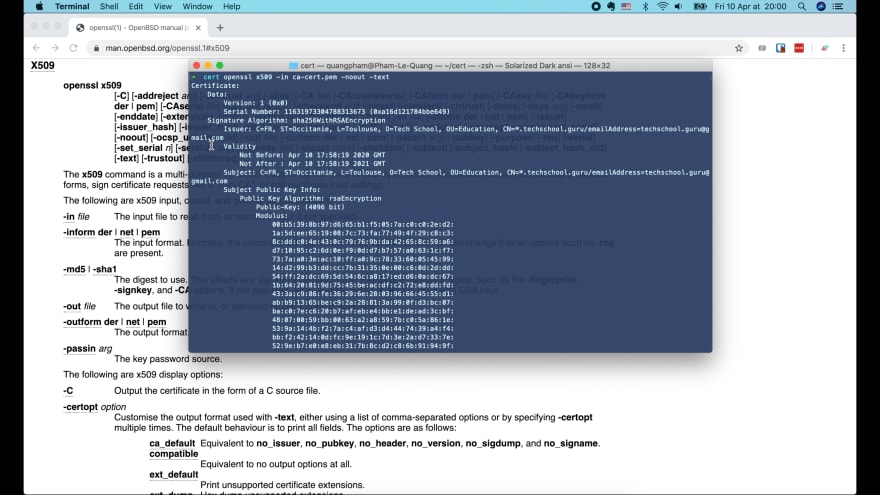
RlIxEjAQBgNVBAgMC...udJwE7HnnA7lpA

-----END CERTIFICATE-----

We can use the [openssl x509](https://man.openbsd.org/openssl.1" \l "x509) command to display all the information encoded in this certificate. This command can also be used to sign certificate requests, which we see in a moment.

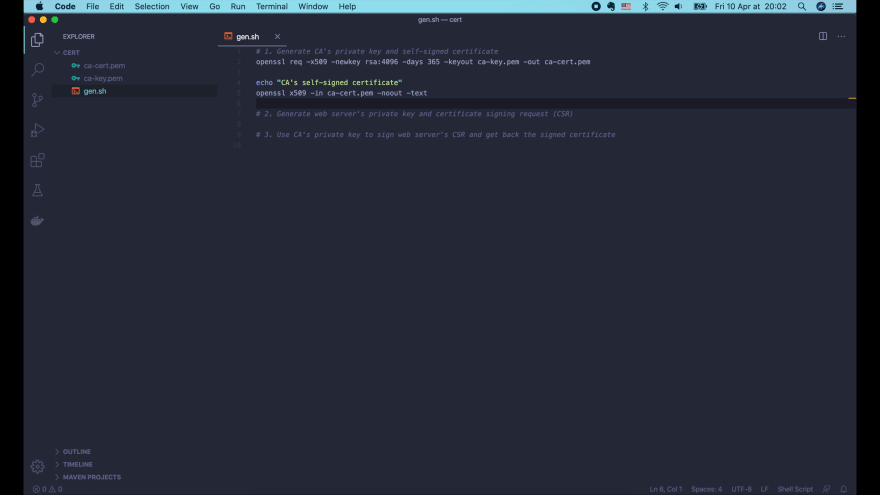
openssl x509 -in ca-cert.pem -noout -text

Here we use the -in option to pass in the CA’s certificate file. And the -noout option to tell it to not output the original base64-encoded value. Instead, we use the -text option because we want to display it in a readable text format.

[](https://res.cloudinary.com/practicaldev/image/fetch/s--dStacSoc--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/dev-to-uploads.s3.amazonaws.com/i/dcketza3v19b6j07auqn.png)

Now we can see all information of the certificate, such as the version, the serial number. The issuer and the subject are the same in this case because this is a self-signed certificate. Then the RSA public key and signature.

I’m gonna copy this command and save it to a gen.sh script. With this script, I want to automate the process of generating a set of keys and certificates.

[](https://res.cloudinary.com/practicaldev/image/fetch/s--lTS072aa--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/dev-to-uploads.s3.amazonaws.com/i/6e7f8i191318dvhbq8qj.png)

Before moving to the 2nd step, I’m gonna show you another way to provide the identity information without entering it interactively as before. To do this, we must add the -subj (subject) option to the openssl req command:

openssl req -x509 -newkey rsa:4096 -days 365 -keyout ca-key.pem -out ca-cert.pem -subj "/C=FR/ST=Occitanie/L=Toulouse/O=Tech School/OU=Education/CN=\*.techschool.guru/emailAddress=techschool.guru@gmail.com"

In this subject string:

* /C=FR is for Country
* /ST=Occitanie is for STate or province
* /L=Toulouse is for Locality name or city
* /O=Tech School is for Organisation
* /OU=Education is for Organisation Unit
* /CN=\*.techschool.guru is for Common Name or domain name
* /emailAddress=techschool.guru@gmail.com is for email address

Now let’s add command rm \*.pem (to remove all pem files) at the top of the gen.sh script, and run it in the terminal.

rm \*.pem

# 1. Generate CA's private key and self-signed certificate

openssl req -x509 -newkey rsa:4096 -days 365 -keyout ca-key.pem -out ca-cert.pem -subj "/C=FR/ST=Occitanie/L=Toulouse/O=Tech School/OU=Education/CN=\*.techschool.guru/emailAddress=techschool.guru@gmail.com"

echo "CA's self-signed certificate"

openssl x509 -in ca-cert.pem -noout -text

# 2. Generate web server's private key and certificate signing request (CSR)

# 3. Use CA's private key to sign web server's CSR and get back the signed certificate

We still being prompted for a pass phrase, but it doesn’t ask for identity information anymore, because we already provided them in the subject option.

## 2. Generate web server's private key and CSR

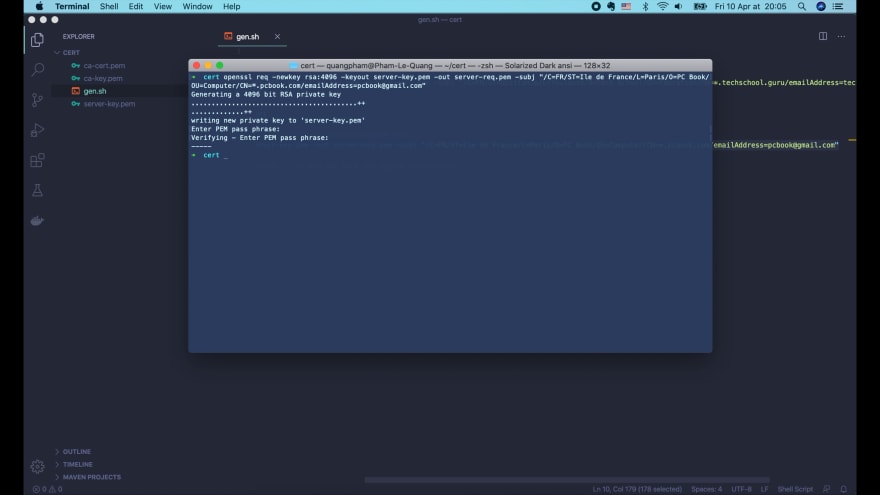
Now the next step is to generate a private key and CSR for our web server.

It’s almost the same as the command we used in the 1st step. Except that, this time we don’t want to self-sign it, so we should remove the -x509 option. The -days option should be removed as well, since we don’t create a certificate, but just a CSR.

openssl req -newkey rsa:4096 -keyout server-key.pem -out server-req.pem -subj "/C=FR/ST=Ile de France/L=Paris/O=PC Book/OU=Computer/CN=\*.pcbook.com/emailAddress=pcbook@gmail.com"

The name of the output key should be server-key.pem. The output certificate request file should be server-req.pem. And the subject should contain our web server’s information.

Now, when we run this command, the encrypted private key and the certificate signing request files will be generated.

[](https://res.cloudinary.com/practicaldev/image/fetch/s--1Hp94TI4--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/dev-to-uploads.s3.amazonaws.com/i/nwe8g3zz2hs33bkiq9zb.png)

This time, in the server-req.pem file, it says CERTIFICATE REQUEST, not CERTIFICATE as in the ca-cert.pem file. That's because it’s not a certificate as before, but a certificate signing request instead.

-----BEGIN CERTIFICATE REQUEST-----

MIIE2DCCAsACAQAwgZIxCzAJBgNVBAYTAkZSMRYwFAYDVQQIDA1JbGUgZGUgRnJh

bmNlMQ4wDAYDVQQHDAVQ...pWofr2eOeBQ4Q=

-----END CERTIFICATE REQUEST-----

So now let’s move to step 3 and sign this request.

## 3. Sign the web server's certificate request

To sign the certificate, we will use the same [openssl x509](https://man.openbsd.org/openssl.1" \l "x509) command that we’ve used to display certificate before. Let’s open the terminal and run this:

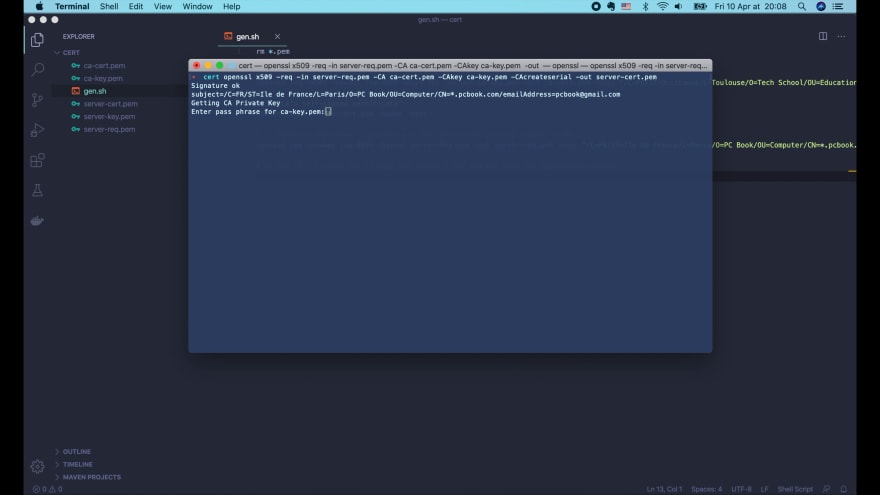
openssl x509 -req -in server-req.pem -CA ca-cert.pem -CAkey ca-key.pem -CAcreateserial -out server-cert.pem

In this command, we use the -req option to tell openssl that we’re gonna pass in a certificate request. We use the -in option follow by the name of the request file: server-req.pem.

Next we use the -CA option to pass in the certificate file of the CA: ca-cert.pem. And the -CAkey option to pass in the private key of the CA: ca-key.pem.

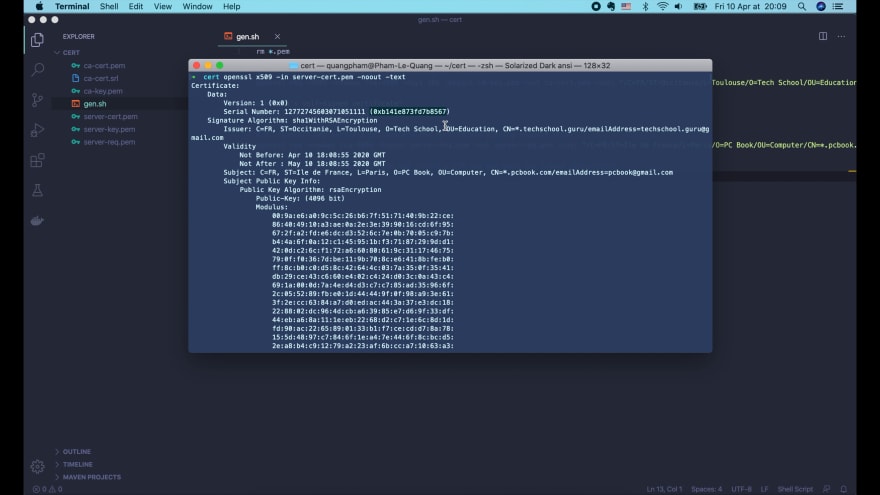
Then 1 important option is -CAcreateserial. Basically the CA must ensure that each certificate it signs goes with a unique serial number. So with this option, a file containing the next serial number will be generated if it doesn’t exist.

Finally we use the -out option to specify the file to write the output certificate to.

[](https://res.cloudinary.com/practicaldev/image/fetch/s--UuXGhyF7--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/dev-to-uploads.s3.amazonaws.com/i/rmcn4b9w4rt492r41osf.png)

Now as you can see here, because the CA’s private key is encrypted, openssl is asking for the pass phrase to decrypt it before it can be used to sign the certificate. It’s a countermeasure in case the CA’s private key is hacked.

OK, now we’ve got the signed certificate for our web server. Let’s print it out in plain text format.

[](https://res.cloudinary.com/practicaldev/image/fetch/s--J0ao4ffB--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/dev-to-uploads.s3.amazonaws.com/i/3qkp9ma6nhp9wlui7plr.png)

This is its unique serial number 0xb141e873fd7b8567. We can also see a ca-cert.srl file, which contains the same serial number.

B141E873FD7B8567

By default, the certificate is valid for 30 days. We can change it by adding the -days option to the signing command.

openssl x509 -req -in server-req.pem -days 60 -CA ca-cert.pem -CAkey ca-key.pem -CAcreateserial -out server-cert.pem

Now the validity duration has changed to 60 days.

A certificate can be used for multiple websites with different domain names. We can do that by specifying the Subject Alternative Name extension when signing the certificate request.

The -extfile option of the openssl x509 command allows us to state the file containing the extensions. We can see the format of the config file in [this page](https://man.openbsd.org/x509v3.cnf.5#Subject_alternative_name).

There are several things that we can use as the alternative name, such as email, DNS, or IP. I will create a new file server-ext.cnf with this content:

subjectAltName=DNS:\*.pcbook.com,DNS:\*.pcbook.org,IP:0.0.0.0

Here I set DNS to multiple domain names: \*.pcbook.com and \*.pcbook.org. I also set IP to 0.0.0.0 which will be used when we develop on localhost.

Now in the certificate signing command, let’s add the -extfile option and pass in the name of the extension config file:

openssl x509 -req -in server-req.pem -days 60 -CA ca-cert.pem -CAkey ca-key.pem -CAcreateserial -out server-cert.pem -extfile server-ext.cnf

Now the result certificate file has a new extensions section with all the subject alternative names that we’ve chosen:

Certificate:

...

Signature Algorithm: sha1WithRSAEncryption

Issuer: C=FR, ST=Occitanie, L=Toulouse, O=Tech School, OU=Education, CN=\*.techschool.guru/emailAddress=techschool.guru@gmail.com

Validity

Not Before: Apr 10 18:17:05 2020 GMT

Not After : Jun 9 18:17:05 2020 GMT

Subject: C=FR, ST=Ile de France, L=Paris, O=PC Book, OU=Computer, CN=\*.pcbook.com/emailAddress=pcbook@gmail.com

Subject Public Key Info:

Public Key Algorithm: rsaEncryption

Public-Key: (4096 bit)

Modulus:

00:cb:e2:2b:c3:68:...

Exponent: 65537 (0x10001)

X509v3 extensions:

X509v3 Subject Alternative Name:

DNS:\*.pcbook.com, DNS:\*.pcbook.org, IP Address:0.0.0.0

Signature Algorithm: sha1WithRSAEncryption

5e:67:4d:f7:91:89:fc:...

So looks like our automate script is ready, except for the fact that we have to enter a lot of password to protect the private keys.

In case we just want to use this for development and testing, we can tell openssl to not encrypt the private key, so that it won’t ask us for the pass phrase.

We do that by adding the -nodes option to the openssl req command like this:

rm \*.pem

# 1. Generate CA's private key and self-signed certificate

openssl req -x509 -newkey rsa:4096 -days 365 -nodes -keyout ca-key.pem -out ca-cert.pem -subj "/C=FR/ST=Occitanie/L=Toulouse/O=Tech School/OU=Education/CN=\*.techschool.guru/emailAddress=techschool.guru@gmail.com"

echo "CA's self-signed certificate"

openssl x509 -in ca-cert.pem -noout -text

# 2. Generate web server's private key and certificate signing request (CSR)

openssl req -newkey rsa:4096 -nodes -keyout server-key.pem -out server-req.pem -subj "/C=FR/ST=Ile de France/L=Paris/O=PC Book/OU=Computer/CN=\*.pcbook.com/emailAddress=pcbook@gmail.com"

# 3. Use CA's private key to sign web server's CSR and get back the signed certificate

openssl x509 -req -in server-req.pem -days 60 -CA ca-cert.pem -CAkey ca-key.pem -CAcreateserial -out server-cert.pem -extfile server-ext.cnf

echo "Server's signed certificate"

openssl x509 -in server-cert.pem -noout -text

Now if we run gen.sh again, it will not ask for passwords anymore. And if we look at the private key file, it will be PRIVATE KEY, and not ENCRYPTED PRIVATE KEY as before.

-----BEGIN PRIVATE KEY-----

MIIJQwIBADANBgkqhkiG9w0BAQEFAASCCS0wggkpAgEAAoICAQDL4ivDaIzDM3my

VDzT2Mw5R9bicXS...AxAt2Ldmc4=

-----END PRIVATE KEY-----

## 4. Verify a certificate

One last thing before we finish, I will show you how to verify if a certificate is valid or not. We can do that with the [openssl verify](https://man.openbsd.org/openssl.1" \l "verify) command:

openssl verify -CAfile ca-cert.pem server-cert.pem

We just pass in the trusted CA’s certificate and the certificate that we want to verify. If it returns OK then the certificate is valid.

And that’s it for today’s article. I hope it’s useful for you. Thanks for reading and I’ll see you guys in the next one!

## What HTTPS Does

HTTPS verifies the identity of a website or web service for a connecting client, and encrypts nearly all information sent between the website or service and the user. Protected information includes cookies, user agent details, URL paths, form submissions, and query string parameters. HTTPS is designed to prevent this information from being read or changed while in transit.

HTTPS is a combination of HTTP and Transport Layer Security (TLS). TLS is a network protocol that establishes an encrypted connection to an authenticated peer over an untrusted network.

Browsers and other HTTPS clients are configured to trust a set of certificate authorities [[2]](https://https.cio.gov/#footnote-2) that can issue cryptographically signed certificates on behalf of web service owners. These certificates communicate to the client that the web service host demonstrated ownership of the domain to the certificate authority at the time of certificate issuance. This prevents unknown or untrusted websites from masquerading as a Federal website or service.

## What HTTPS Doesn’t Do

HTTPS has several important limitations. **IP addresses** and **destination domain** **names** **are not encrypted during communication**. Even encrypted traffic can reveal some information indirectly, such as time spent on site, or the size of requested resources or submitted information.

HTTPS only guarantees the integrity of the connection between two systems, not the systems themselves. It is not designed to protect a web server from being hacked or compromised, or to prevent the web service from exposing user information during its normal operation. Similarly, if a user’s system is compromised by an attacker, that system can be altered so that its future HTTPS connections are under the attacker’s control. The guarantees of HTTPS may also be weakened or eliminated by compromised or malicious certificate authorities.