

SSL/TLS

We are introducing a layer between TCP and Application Level. This is **TLS**. The protocol has been always named **SSL**, after a certain version it has become **TLS**. This may make a little bit of confusion, but they are the same thing. The current version is **TLS 1.2**.

What are the services that the Application Layer asks to TCP?

In TCP the sender and the receiver are PORTS, port numbers. When an application is working and it is using the network a port is associated to it. With IPSEC we provided security from port to port, now we are securing a specific application, so a specific port. In most cases we can prefer TLS over IPSEC, because IPSEC is a way to secure all the communications between hosts, I want you to be completely aware about the fact that TLS secures a specific application. Is this invisible to applications? TLS is not invisible, IPSEC is.

Okay, another way of describing this is “end to end security”. *End* has not a formal definition, it depends on the cases. With *end* we mean a party that is communicating. Whatsapp is offering *end-to-end* communication, it means that the Server is not able to understand the message, they are encrypted.

When we find **https** in our browser when we are surfing on a certain site, it means that the connection between us and that site is secure because it relies on TLS.

TLS is designed to work under the web browsing, of course you can use it for other applications but the main use it is for browsing. TLS is providing confidentiality, data integrity and authentication, your web browser will be sure about the identity of the web server it is communicating with. But TLS authentication is *unilateral*, only **Server** is authenticated. It is not completely true, because installing a certificate on your browser you can reach mutual authentication, so also you will be authenticated by the Server.

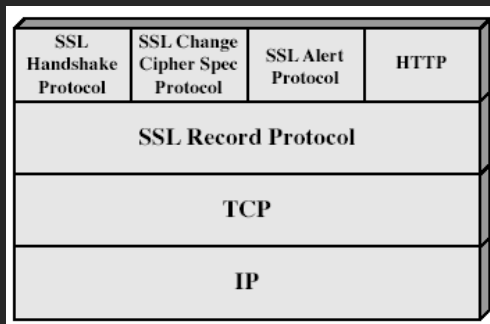
Note that there can be two cases: either Client and Server exchange keys according to a Private/Public Key or they use a pre-shared secret key. In this latter case we are talking about **TLS-PSK**.

When we make a connection through TLS we have what is called **TLS Handshaking**, we will see in details it later, but the main goal is the following:

1. The peer starts a conversation for understanding what algorithms client and Server can both support, then they agree with some algorithms to be used
2. Then they do key exchange and authentication
3. After that they can start to run normal conversation, encrypted of course with the chosen symmetric cipher and messages are also authenticated

If you want to be a little bit more precise, at the very beginning, client and server start talking about what ciphers they know. The Server has an ordered list of choices (ordered by security criteria), and from the known algorithms client-side the Server chooses the best one. As I already mentioned you can use authentication, and after authentication you can derive session keys, if it is not based on public keys you can base it on some shared secret. This is the case of PSK.

The *handshaking* of the TLS protocol is the most complex part, it is the part that may fail, and if it fails the connection is not established. After the handshaking you will have all the parameters for establishing secure conversation.



According to the original design, so SSL, the protocol was based on two different layers of the protocol: the first layer is the one which performs the setup of the connection, so in this phase all the useful parameters in order to secure the conversation are exchanged. The other layer is the Record Protocol, which acts after the setup session, so it is the layer in charge to encrypt and decrypt the real data of the conversation.

TLS Session vs TLS Connection

SSL session: is the result of an handshaking, the two parties are having an handshaking, they establish some parameters, and all such parameters describe a session, now within the session you can establish some several connections, using the same parameters. Why that? Because handshaking here is demanding and you do not want to start a new handshaking for every connection. So a session is a logical association between the client and the Server.

SSL connection: an exchange of messages within a session. More connections can take place in one single session. Made to save some computation.

In order to better understand the difference between session and connection we need to inspect some parameters. The parameters define the status of a session/connection.

Session parameters

Session identifier: it is just a string of bits

Peer certificate: it is a description of the X.509 version of the certificates of the 2 parties. This parameter can be null, for example when the connection is not based on the PKI.

Compression method: data is compressed before being sent, so both the parties have to agree on a compression method and it is described here.

Cipher spec: the specification of the cipher approach you are using, for instance what data encryption algorithm the peers are using.

Master secret: it is a shared secret between Client and Server. Here there is the result of the handshaking.

Is resumable: it is a flag that indicates whether the current session can be used for initiating new connections.

Connection parameters

Random numbers: for each connection client and server are choosing random numbers, *nonces*.

Server MAC secret: key used with MAC operations by the Server

Client MAC secret: key used with MAC operations by the Client

Server write key: key for encryption and decryption server side, actually they are two keys

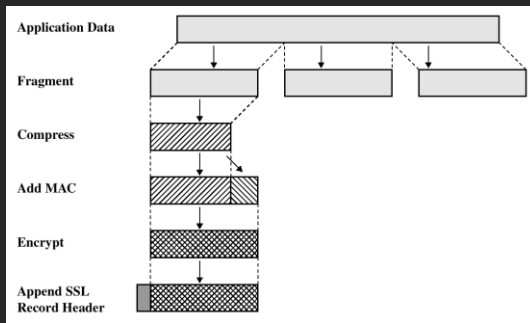
Client write key: key for encryption and decryption client side, actually they are two keys

Initialization vectors: the seed for initialize encryptions, because in most cases we use CBC for symmetric encryption and CBC is using an *initialization vector*.

Sequence numbers: self explaining. Useful also against replay attacks

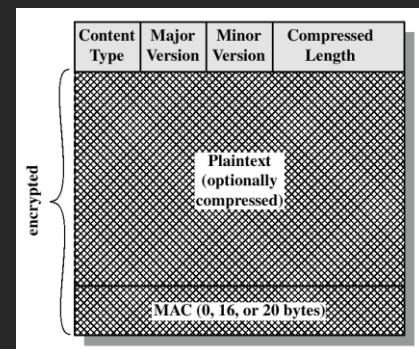
When you are considering a connection you have some practical informations, they are keys, obtained by the Master Key information of the session.

The most important protocol offering encryption, confidentiality and authentication and data integrity is the **Record Protocol**. In order to get confidentiality there are many possible encryption algorithms: *IDEA*, *RC2-40*, *DES-40*, *DES*, *3DES*, *Fortezza*, *RC4-40*, *RC4-128*.



Record Protocol takes place after the Handshaking phase. This is the scheme of Record Protocol. Remember that it takes place after the handshake, so once all the MACs and Cipher Specs have been decided. As we can see, the application data is compressed, authenticated and encrypted, and the the SSL Record Header is appended to the encrypted fragment.

This on the side is a single *record* of the Record Protocol. The last part of the record (MAC) is optional and it depends on whether data integrity is being used or not. Content type is defining what is the upper protocol asking the service to the record protocol, so it defines the protocol towards which this record is sent. Major and Minor version are specifying the upper and lower versions of SSL/TLS. Compressed Length is specifying the length of the compressed record. Then follows the encrypted part, which is the Plaintext and the optional MAC.



SSL payload is the normal information transferred by the protocol. We can have a normal payload like for example some information that have some meaning in the upper protocol (often HTTP), or you can have another kind of payload, for example when the record is a part of the handshaking procedure. There are 4 kinds of payload:

1. **Change Cipher Spec Protocol**: it is just one byte, it acts like a signal saying “ok, we are done with our handshaking, now let’s start the game by encrypting/decrypting and authenticating messages”.
2. **Alert Protocol**: composed by two bytes, we will see later.
3. **Handshake Protocol**: one byte of Type, three bytes of Length and then the content.
4. **Other Upper-Layer Protocol**: one byte of *Opaque Content*.

HANDSHAKE PROTOCOL

It is the most important part of the protocol. It is complex and somewhat long, indeed when people, normal users, are connecting to a TLS enabled website, they can feel a small delay that is related to the handshaking phase. The handshaking phase has 4 different phases:

1. Hello
2. Server offers informations
3. Client offers informations
4. End of connection

On slide 39 they are shown parameters of the handshake procedure.

PHASE 1 – Client Hello

In this phase the Client presents itself to the Server, giving it some informations.

Parameters:

- Version: last supported version of SSL/TLS supported by the client

- Client Random: 32-byte (256-bit), composed by 32 bit of timestamp and 28 bytes random. The whole 32-byte are representing the Client's *Master Secret*
- Session ID: if 0 it is a new session, if $\neq 0$ means we are resuming some old connection or to open a connection within an existing session
- Cipher suite: list of algorithms supported by client, in decreasing preference order
- Compression algorithms.

PHASE 2 – Server Hello

The Server Hello phase is a little bit more complex.

In this phase the Server responds to the Client, deciding for him the algorithms to be used. If client and Server have no matching ciphers (they have no ciphers in common) then a *handshake failure alert* is raised.

Then the server is sending optional informations to the client offering him its digital certificates, and informations for the *Server Key Exchange*, all this in order to construct the shared key, and in order to construct such shared keys the server sends informations depending on the type of construction that is chosen in the previous step. Then the Server may ask the client for a certificate. Then the Server Hello is done.

PHASE 3 – Client responds to the Server

The client answers for the Servers requests of the previous phase, so he offers it the requested certificate, and completes the Key Exchange (so sends the *Client Key Exchange* message). Then the Client confirms the end of its handshake session.

PHASE 4 – Handshake termination

Here the server too confirms the end of its handshake session.

Fixed Diffie-Hellman

Remind that the original Diffie-Hellman is vulnerable to *man-in-the-middle attack* if the parties are not authenticated. In the case of the **Fixed Diffie-Hellman** what happens is that the two parties are offering digital certificates and in the extra fields allowed by version 3 of standard X.509 there are useful informations like the public key for implementing Diffie-Hellman. You remind that the public key of the Diffie-Hellman case is a set of informations like the prime number, the generator for the \mathbb{Z}_p^* group in the case we are making the right choice for D-H, and actually in most cases it happens that the Server is offering a digital certificate, but the client is not offering a digital certificate often. So it happens that the public parameters Server-side are offered by a digital certificate so these public parameters are always present. So this was Fixed DH (informations for the public key of the two parties are clearly maintained in their certificates). This also means that you cannot choose at any time the different parameters, changing the prime number for instance.

Keep in mind that in some cases the parameters are not compatible and the Diffie-Hellman key exchange cannot take place.

Ephemeral Diffie-Hellman

It is the case where information for doing Diffie-Hellman is exchanged by signing the messages, using private keys like RSA. So the parties sign the messages and do authentication. In theory you can consider this possibility more secure because all parameters can be changed at any time. In the case of Fixed DH.

Anonymous Diffie-Hellman

This is the weakest case, and it has become not allowed anymore since TLS 1.2. It is weak against *man-in-the-middle*.

"ssllabs.com time..."

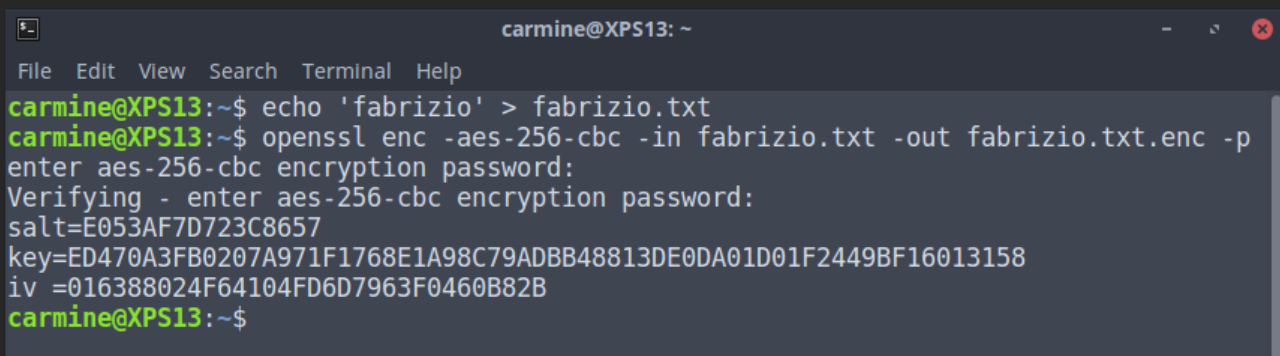
OpenSSL practice

Encryption

Let's try some encryption. In order to do some encryption we should create a file. We create a normal file for example with `'echo qwerty > file.txt'`. Now let's create the encrypted file. The command is `enc`. For details run `'man enc'`. The standard way to encrypt a file using `enc` is:

```
> openssl enc -aes-256-cbc -in input_filename.txt -out output_filename.txt.enc -p
```

The option `-p` is for showing details about the encrypting process. What details? If I want to encrypt a file I need a key, the encryption key, and since we have chosen `aes-256` we should be able to generate such an encryption key, in most cases I don't like to do that, so I ask the software to generate a key randomly and the software is able to do that starting from a secret, that is a password needed at runtime so a prompt will appear and a password will be asked. Example:



```
carmine@XPS13: ~  
File Edit View Search Terminal Help  
carmine@XPS13:~$ echo 'fabrizio' > fabrizio.txt  
carmine@XPS13:~$ openssl enc -aes-256-cbc -in fabrizio.txt -out fabrizio.txt.enc -p  
enter aes-256-cbc encryption password:  
Verifying - enter aes-256-cbc encryption password:  
salt=E053AF7D723C8657  
key=ED470A3FB0207A971F1768E1A98C79ADBB48813DE0DA01D01F2449BF16013158  
iv =016388024F64104FD6D7963F0460B82B  
carmine@XPS13:~$
```

We see the salt, the key, and the iv (initialization vector, so the seed). The encrypted file is a binary file, it is something not comfortable because it is not easily sent over the network, you have to encapsulate the binary file in something different. So we can ask to encrypt also with an encoding. So we can add the `-a` option for encoding, then the outcome will no longer be a binary file, so it will be opened with normal text editors. Notice that the encrypted file is bigger. There is a precise reason.

What if we want to decrypt? We use the same command, but we have to add `-d` to the command.

```
> openssl enc -aes-256-cbc -in file.txt.enc -out newfile.txt -d -p
```

Note: if we encrypted with `-a` parameter we must decrypt with `-a` parameter as well

Note: in order to see all the possible ciphers run the following command:

```
> openssl list -cipher-commands
```

Digital Signatures and Key pair generation

I want to show you digital signatures through openssl. Let's say we want a digital signature based on RSA, in this moment we have nothing. We achieve this by using the *genrsa* tool of the software, that is able to generate a private key. The command will be something like:

```
> openssl genrsa -out filename.pem 4096
```

The last number is the size of the key. This works, but the private key given as outcome is not encrypted and we don't like it, we want it encrypted. So we do:

```
> openssl genrsa -aes256 -out filename.pem 4096
```

A password will be asked, like when we have done the encryption in the previous section. Now the key is encrypted, and in the file *filename.pem* containing such encrypted private key there are printed also informations in order to decrypt. It is written the algorithm used for encrypting.

Now we need another file containing the public key, because the previous process just created a private key. In order to get the public key associated to the private key we can just use another tool that is the *rsa* tool, it will take as input the private key and will write the public key in another file.

```
> openssl dgst -sha512 -sign private.pem -out file.txt.signature file.txt
```

The file that has *signature* extension is a binary file. In order to verify the signature we need the public key. But we have it in a file so it is easy.

```
> openssl dgst -sha512 -verify public.pem -signature file.txt.signature file.txt
```