X.509 Certificates and Transport Layer Security (TLS) protocol

Laboratory for the class "Information Systems Security" (01TYMOV, 01KRQ)
Politecnico di Torino – AA 2025/26
Prof. Diana Berbecaru

prepared by:
Enrico Bravi (enrico.bravi@polito.it)
Diana Berbecaru (diana.berbecaru@polito.it)

v. 2.0 (05/11/2025)

Contents

1 Purpose of the laboratory

In this laboratory, you will perform exercises related to X.509v3 certificates and their application in the TLS protocol. Thus, you will understand how to generate the necessary keys and create the Certificate Signing Request (CSR). Moreover, you will also put in place a simple Certification Authority (CA) side, which will issue the requested certificate(s) and manage the certificate revocation (by means of CRLs). In the second part of the lab, you will exploit TLS protocol to enable security at the transport level.

More in detail, in the first part of this laboratory you will create asymmetric keys and a Certificate Signing Request (CSR) for the public key. To obtain a certificate, it will also be necessary to create a (demo) Certificate Authority (CA), which will be responsible for managing the CSR and verifying it. In addition you will also

understand how to handle the revocation of certificates by issuing a CRL and how to manage the verification of the certificate status with a CRL.

The second part of the laboratory is dedicated to experimenting with the TLS (Transport Layer Security) protocol for protecting data at the transport level. For this purpose, we will use the OpenSSL library (and some associated tools) offering in-depth support for the configuration and analysis of SSL/TLS channels. In particular, the command `s_client` implements the functionality of an SSL/TLS client (`man openssl-s_client`). The command `s_server` implements instead the functionality of a simple SSL/TLS server (`man openssl-s_server`).

Below you find the main OpenSSL commands (along with the necessary parameters) to be used in this lab.

Certificate request/issue/revocation

The OpenSSL library and tool enable the management of creating CSRs for public keys and issuing the requested certificates. The command in charge of managing the certificate requests is `req`, with the following syntax:

```
openssl req [-help] [-in filename] [-out filename] [-text] [-noout] [-new]
[-key filename] [-verbose] [-quiet]
```

where:

- `-help` prints the basic usage of the command (alternatively, you can use the manpage: `man openssl req`);
- `-in` *filename* specifies the input filename to read a request from;
- `-out` *filename* specifies the output filename to write to or standard output by default;
- `-text` prints out the certificate request in text form;
- `-noout` prevents output of the encoded version of the certificate request;
- `-new` generates a new certificate request. It will prompt the user for the relevant field values. The actual fields prompted for and their maximum and minimum sizes are specified in an openssl configuration file and any requested extensions.

  If the `-key` option is not given, it will generate a new private key using information specified in the configuration file or given with the `-newkey` and `-pkeyopt` options, else by default an RSA key with 2048 bits length.
- `-key` *filename* provides the private key for signing a new certificate or certificate request. Unless `-in` is given, the corresponding public key is placed in the new certificate or certificate request, resulting in a self-signature.

For issuing certificates acting as a CA, the command the OpenSSL provides is `ca`, with the following syntax:

```
openssl ca [-help] [-verbose] [-quiet] [-in file] [-out file] [-cert file]
[-keyfile file] [-gencrl]
```

- `-help` prints the basic usage of the command (alternatively, you can use the manpage: `man openssl ca`);
- `-in` *filename* An input filename containing a single certificate request (CSR) to be signed by the CA;
- `-out` *filename* The output file to output certificates to. The default is standard output. The certificate details will also be printed out to this file in PEM format (except that -spkac outputs DER format);

- `-cert` *filename* The CA certificate, which must match with `-keyfile`;
- `-keyfile` *filename* The CA private key to sign certificate requests with. This must match with `-cert`;
- `-gencrl` generates a CRL based on information in the index file.

TLS

The OpenSSL library implements a simple SSL/TLS client and server, which can be used for testing the SSL/TLS protocol very easily.
The OpenSSL command used to start an SSL/TLS client program is `s_client`, whose syntax is given below (some additional information about CRL flags can be found in `man openssl-verification-options`):

```
openssl s_client [-connect host:port] [-state] [-showcert] [-CAfile file_cert]
[-cipher cipherlist] [-reconnect] [-crl_check] [-extended_crl]
```

where:

- `-connect` *host:port* specifies the host and optional port to connect to. If the host and port are not specified then an attempt is made to connect to the local host on port 4433;
- `-state` prints out the SSL session states;
- `-showcerts` displays the whole server certificate chain: normally only the server certificate itself is displayed;
- `-CAfile` *file* indicates the file containing trusted certificates to use during server authentication and to use when attempting to build the client certificate chain;
- `-cipher` *cipherlist* allows to specify the cipher list sent by the client in the ClientHello message of the Handshake protocol. Although the server determines which cipher suite is used it should take the first supported cipher in the list sent by the client. See the OpenSSL `ciphers` command for more information;
- `-reconnect` allows the client to reconnect to the same server 5 times using the same session ID.
- `-crl_check` checks end entity certificate validity by attempting to look up a valid CRL. If a valid CRL cannot be found, an error occurs.
- `-extended_crl` enable extended CRL features such as indirect CRLs and alternate CRL signing keys.

The OpenSSL command used to start an SSL/TLS server program is `s_server`, whose syntax and parameters are given below:

```
openssl s_server [-www] [-no_dhe] [-key server_pkey.pem] [-cert server_cert.pem]
[-CAfile file_cert] [-{vV}erify depth] [-cipher ciphersuite_list]
```

where:

- `-www` sends a status message back to the client when it connects. This includes lots of information about the ciphers used and various session parameters. The output is in HTML format so this option will normally be used with a web browser;
- `-no_dhe` if this option is set then no DH parameters will be loaded effectively disabling the ephemeral DH cipher suites;

- `-key` *server_pkey.pem* indicates that *server_pkey.pem* contains the private key of the server;
- `-cert` *server_cert.pem* indicates that *server_cert.pem* contains the certificate of the server;
- `-CAfile` *file* indicates the file containing trusted certificates to use during client authentication and to use when attempting to build the server certificate chain. The list is also used in the list of acceptable client CAs passed to the client when a certificate is requested;
- `-verify` *depth*, `-Verify` *depth* indicates the verify depth to use. This specifies the maximum length of the client certificate chain and makes the server request a certificate from the client. With the -verify option a certificate is requested but the client does not have to send one, with the -Verify option the client must supply a certificate or an error occurs;
- `-cipher` *cipherlist*, this option allows the cipher list used by the server to be modified. When the client sends a list of supported ciphers the first client cipher also included in the server list is used. Because the client specifies the preference order, the order of the server cipherlist is irrelevant. See the OpenSSL `ciphers` command for more information.

ATTENTION

Be careful in copying and pasting commands from the laboratory PDF text. There could be some characters that will not be pasted correctly in the terminal. Check every command, and preferably, try to write it without copy-paste.

2 X.509v3 Certificates: Issuance and Revocation of Certificates

In this section, you will experiment with the issuing an X.509 certificate. In particular, you will act as a Certificate Authority (CA), which will issue a certificate upon a request for a specific key.

2.1 Setting up a (demo) Certificate Authority (CA)

OpenSSL allows to setup a demoCA by using one of its utility tool. In this exercise, you will create such a demoCA, which allows to get understand the data structures and the processes involved in certificate management. Create first a test CA by exploiting OpenSSL:

```
/usr/lib/ssl/misc/CA.pl -newca
```

When asked, insert a password to protect the asymmetric private key of the demoCA, e.g. “ciao”. Remember the values you have inserted for “Country”, “State”, and “Organization”, as you will have to use the same values in the following steps.

This command creates a demo CA in the current directory, placing the data in the `demoCA` directory.

Question 1. Find the CA certificate and show its content, indicating the command used.

→

Question 2. Now take a closer look at the fields in the CA certificate. Write down the values for the “Issuer” and for the “Subject”. Which key has been used to sign the demo CA certificate? Indicate below the “Issuer”, the “Subject”, the key that has been used the demoCA certificate.

→

2.2 Create (asymmetric) key pair and the Certificate Signing Request (CSR)

In this exercise you will exploit the demoCA to issue certificates for a server.

For this purpose, create a certificate request for the TLS server:

Starting from the OpenSSL command `genrsa`, run the following OpenSSL command used to generate a 2048-bit RSA key pair, and save it in the `rsa.key.pem`:

```
openssl genrsa -out rsa.key.pem 2048
```

Then extract the public key:

```
openssl rsa -in rsa.key.pem -pubout -out rsa.pubkey.pem
```

Then create the CSR for the created key. Pay attention to indicate the same information (e.g. "Organization", "Country") that you put in the CA certificate.

```
openssl req -new -key rsa.key.pem -out certificate_req.pem
```

Question 3. Find out and write below the command to view the CSR content in text format.

```
→
```

Question 4. With the help of a screen capture illustrate content of the CSR.

```
→
```

2.3 Issue an X.509v3 certificate

Now it's time to issue the certificate associated with the public key created.

Question 5. Write down the command to issue the certificate using the demoCA:

```
→
```

Question 6. What is the main aspect that needs to be verified by the CA before issuing the certificate?

```
→
```

At this point, you should have: the certificate of the CA (in the `cacert.pem` in the `demoCA` directory), and the certificate and the corresponding key.

2.4 Revoking a certificate

Now let's assume that the certificate created needs to be revoked by the CA. In this case, the CA will need to update its database in order to log that this specific certificate cannot be considered valid anymore.

In the first step, the CA revokes the certificate:

```
openssl ca -revoke pkey_cert.pem
```

Question 7. Show the output of the command (with a screen capture):

```
→
```

Now the CA needs to create a CRL that can be used by a relying party to check the status of certificates:

```
openssl ca -gencrl -out demoCA/crl/rootca.crl
```

Question 8. Write down the command to show the content of the generated CRL in text format:

```
→
```

Now it is possible to try to verify the revoked certificate and see which is the result of the verification, using the CRL:

```
cat demoCA/cacert.pem demoCA/crl/rootca.crl > ca_crl.pem
```

```
openssl verify -extended_crl -verbose -CAfile ca_crl.pem -crl_check pkey_cert.pem
```

Question 9. Show the output of the command (with a screen capture):

```
→
```

3 The TLS protocol: applying security above the transport level

3.1 Setting up a TLS channel

Depicted in Fig. 1 is a typical instance of a TLS handshake (TLS v1.2, server authentication only).

The description of the steps is given shortly below:

1. ClientHello: Client sends the random number R_c and the supported cipher suites;
2. ServerHello: Server sends the random number R_s and the chosen cipher suite;
3. Certificate: Server sends its X.509v3 certificate (and chain);
4. ServerKeyExchange: (in case of DH key agreement) Server sends its DH public part (signed);
5. ServerHelloDone: Server signals end of handshake;
6. ClientKeyExchange: (in case of DH key agreement) Client sends its DH public part.
In case of RSA (key exchange): Client sends the Premaster Secret (PS) encrypted with the public key of the server;
Client and server derive master secret, then the keys for encryption ($K_{enc}c$ and $K_{enc}s$), the keys for calculation of MACs ($K_{mac}c$) and $K_{mac}s$ and the IVs.
7. ChangeCipherSpec, Finished: Client sends MAC($K_{mac}c$, handshake messages);
8. ChangeCipherSpec, Finished: Server sends MAC($K_{mac}s$, handshake messages);
9. ApplicationData: Client sends data protected with the keys and algorithms negotiated;
10. ApplicationData: Server sends data protected with the keys and algorithms negotiated.

Start Wireshark so that you can capture and analyze the TLS handshake messages exchanged between your TLS client and the remote TLS server.

Now, from your machine, try to connect to a TLS-enabled server, by using `s_client` program (from the command line):

```
openssl s_client -connect www.polito.it:443 -state -showcerts -verify 1
```

Question 10. Which TLS version and cipher suite have been negotiated?

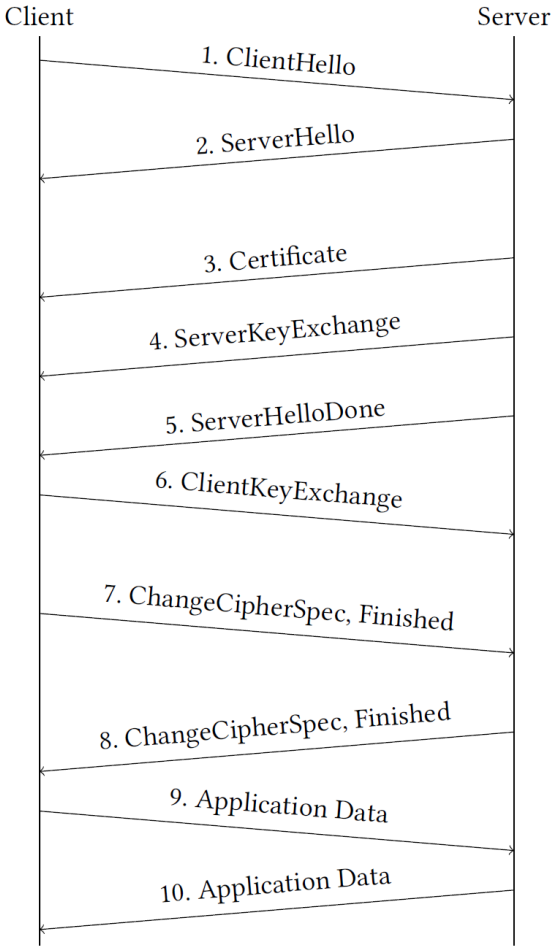

Figure 1: TLS handshake messages (v1.2).

→

Now run the following command:

```
openssl s_client -connect fancyssl.hboeck.de:443 -state -showcerts -verify 1
```

Observe with Wireshark the TLS handshake messages exchanged between your machine and the TLS server in the above command.

Question 11. Which TLS version and cipher suite have been negotiated? Illustrate it with the caption (Wireshark) in your screen.

→

Question 12. Which algorithm has been negociated for key exchange/key agreement? Which algorithm has been negotiated for encrypting the data exchanged? Which algorithm has been negociated for protecting the data for integrity and data authentication (i.e., MAC calculation)?

```
→
```

3.2 Configuration of a TLS server

Try now to configure a TLS server.

Question 13. What do you need in the first place?

```
→
```

First, let's create the key and the correspoding certificate for the server:

```
openssl req -new -keyout server_pkey.pem -out server_creq.pem
```

```
openssl ca -in server_creq.pem -out server_cert.pem
```

At this point you should have: the certificate of the CA (in the `cacert.pem` in the `demoCA` directory), and the certificate and the corresponding key for the TLS server (in the files `server_cert.pem`, `server_key.pem`).

Start the `s_server` with the following command:

```
openssl s_server -www -no_dhe -key server_pkey.pem -cert server_cert.pem
```

The server listens on the port `4433`. From another terminal on the same machine, try to connect with `s_client` and check out the result:

```
openssl s_client -connect 127.0.0.1:4433 -state -showcerts -CAfile
demoCA/cacert.pem
```

Question 14. What is the purpose of the parameter `-no_dhe` used so far?

```
→
```

Question 15. What are and in which case can be used the ephemeral mechanisms for key exchange in TLS?

```
→
```

3.3 Use of Session ID

Question 16. For what purpose is it used the "Session ID" in TLS?

```
→
```

Start the server with the following command:

```
openssl s_server -www -no_dhe -key server_pkey.pem -cert server_cert.pem -CAfile
demoCA/cacert.pem -tls1_2
```

Execute the following `s_client` command, which opens up several consecutive connections by exploiting the Session ID:

```
openssl s_client -connect 127.0.0.1:4433 -state -CAfile demoCA/cacert.pem
-reconnect
```

Question 17. Which parameters of the TLS session remain unchanged in successive connections?

→

3.4 Client authentication in TLS

Now try to configure a TLS connection with client authentication.

Question 18. What do you need in the first place?

→

You can generate an X.509 client certificate in the following way:

1. **Question 19**. Write down the OpenSSL command to create a key (for the client) and the corresponding certificate request for the client:

   →

2. **Question 20**. By exploiting the demoCA you have created above, write down the command to issue the new certificate for the client:

   →

Configure now `s_server` so that to request client authentication during the handshake phase of the SSL/TLS protocol:

```
openssl s_server -www -no_dhe -key rsa.key.pem -cert pkey_cert.pem -CAfile
demoCA/cacert.pem -Verify 1
```

Run the `s_client` command necessary to connect to the `s_server` started above:

```
openssl s_client -connect 127.0.0.1:4433 -state -showcerts -CAfile
demoCA/cacert.pem -cert client_cert.pem -key client_pkey.pem
```

Question 21. Which steps have changed in the handshake phase of the SSL/TLS protocol?

→

3.5 Server's certificate revocation

Now let' simulate a revocation for the server certificate and check how the TLS handshake changes:

First, the CA revokes the server certificate:

Question 22. Write down the command to revoke the server certificate:

→

Now the CA needs to update the CRL:

```
openssl ca -gencrl -out demoCA/crl/rootca.crl
```

```
cat demoCA/cacert.pem demoCA/crl/rootca.crl > ca_crl.pem
```

Now we can start the TLS server, by using the revoked certificate:

```
openssl s_server -www -no_dhe -key rsa.key.pem -cert pkey_cert.pem -CAfile
demoCA/cacert.pem
```

And finally, let's create a client connection in order to see how the verification of the server certificate ends:

```
openssl s_client -connect 127.0.0.1:4433 -state -CAfile ca_crl.pem -showcerts
-cert client_cert.pem -key client_pkey.pem -crl_check -extended_crl
```

Question 23. Show the output of the connection and highlight the points where you can see that the verification of the server certificate failed because of the revocation:

```
→
```

4 Homework

Homework 1. Propose one multiple-choice question about **X.509v3 certificates and PKIs**. You can propose a generic question or a specific example (e.g., involving two parties like Alice and Bob). At least one response must be correct and you must indicate it clearly. In case multiple responses are correct, indicate them all clearly, such as by using a different colour or a bold font.

```
→
```

Homework 2. Propose one multiple-choice question about **TLS protocol**. You can propose a generic question or a specific example (e.g., involving two parties like Alice and Bob). At least one response must be correct and you must indicate it clearly. In case multiple responses are correct, indicate them all clearly, such as by using a different colour or a bold font.

```
→
```