The today's laboratory task is to implement a client-server application that takes advantage of the features offered by the `openssl` command for ensuring a secure communication channel and verifying the identity of parties involved. You will have to implement both a client and a server. Client and server will be able to establish an authenticated (only the server is authenticated) and secure session (cryptography is enabled).
In particular, you should implement the authentication server side as follows:

- create a mini *Certification Authority;*
- create a *server certificate* using `openssl` commands, this latter signed by the mini-CA;
- create a *client* that, before starting a conversation with the server side, checks the identity of the server ensuring that the personal certificate sent by the server is signed by a CA (*Certification Authority*) that it recognizes.

The server will be developed using the `openssl` commands and the client will be developed in Java using the `javax.net.ssl` library.

We will proceed in 4 steps:

A) Generation of the certificate for the *Certification Authority;*
B) Generation of the certificate for the s*erver;*
C) Development of the *Java client;*
D) Starting and testing of *server* and *client* applications*;*

A) Generation of the certificate for the *Certification Authority*
Unless you have a contract with a root Certification Authority, you must generate a key and a 'self-signed' certificate for the CA.

A.1) Generating of a RSA key for the CA. This is the private key which will be used for signing all the server certificates later. Thus `ca.key` should be kept as safe as possible.

```
$ openssl genrsa -aes256 -out ca.key  2048
```

A.2) Generating a CSR (*Certificate Signing Request*), that is a encrypted text file used for requesting an assignment of a SSL certificate. You should put in this file all the information that the CA needs to sign a SSL certificate. In particular, you should specify the common name field (CN) and attach a public key which will be embedded in the certificate you're requesting.

```
$ openssl req -new -key ca.key > ca.csr
```

A.3) Generating a CA X.509 certificate valid for 100 days which, with your own key, can be used to release server certificates (we will self-sign it by the CA itself). Note the *–trustout* option.

```
$ openssl x509 -req -days 100 -trustout -signkey ca.key < ca.csr > ca.cert
```

After the above steps you should have a public CA certificate (ca.cert), and its private key (ca.key).

B) Generation of the certificate for the s*erver*

B.1) Generating of a RSA key for the server

```
$ openssl genrsa -aes256 -out server.key 2048
```

B.2) Generating a CSR (*Certificate Signing Request*)

```
$ openssl req -new -key server.key -out server.csr
```

B.3) Request of a certificate for the server valid for 100 days. This certificate will be the very first one signed by your CA, so we set its serial to 1. An actual CA maintainance system should store and increment this serial number properly.

```
$ openssl x509 -req -days 100 -CA ca.cert -CAkey ca.key -set_serial 1 < server.csr > server.cert
```

After the above steps, you should have a server.cert which identifies your server, and its private key.

C) Development of the *Java client*

We will develop a Java client application that connects to an SSL server, receives an authentication certificate from the server and validates it compared to a set of certificates present in its truststore. Java has a separate truststore, which by default is usually at *<java-home>/lib/security/cacerts*. In our case we will add the certificate of the CA in a new *truststore* before any conversation is established between the parties. When the CA certificate is installed, all the certificate signed by your CA will be automatically validated (provided they are not expired, nor revoked: note that revocation is checked only if you configured your system for checking CRLs or accessing OCSP servers properly).

C.1) The following command allows to create a keystore using the **keytool** command

```
$ keytool -import -alias alias_name -file ca.cert -keystore /path_to/my_key_store
```

C.2) the code stub you can use for your client app:

```
import javax.net.ssl.SSLSocket;
import javax.net.ssl.SSLSocketFactory;
import java.io.*;

public class Client {

 public static void  main(String[] arstring) {
        try {
        SSLSocketFactory f = (SSLSocketFactory) SSLSocketFactory.getDefault();
        SSLSocket sslsocket =(SSLSocket) f.createSocket("localhost", 4433);

        InputStream inputstream = System.in;
        InputStreamReader inputstreamreader = new InputStreamReader(inputstream);
        BufferedReader bufferedreader = new BufferedReader(inputstreamreader);

        OutputStream outputstream = sslsocket.getOutputStream();
```

```
            OutputStreamWriter outputstreamwriter = new OutputStreamWriter(outputstream);
            BufferedWriter bufferedwriter = new BufferedWriter(outputstreamwriter);

            String string = null;
            while ((string = bufferedreader.readLine()) != null) {
                  bufferedwriter.write(string + '\n');
                  bufferedwriter.flush();
                   }
            } catch (Exception exception) {
                exception.printStackTrace();
            }
        }
    }
```

D) Start *server and client*

D.1)

```
$ openssl s_server -key server.key -cert server.cert
```

D.2)

```
$ java -Djavax.net.ssl.trustStore=./path_to/my_key_store
  -Djavax.net.ssl.trustStorePassword=key_store_password
  -Djava.protocol.handler.pkgs=com.sun.net.ssl.internal.www.protocol
  -Djavax.net.debug=ssl Client
```

Other useful commands:

```
$ keytool -list -keystore nome_key_store
```

used to list the content of the keystore, then the certificates installed (requires knowledge of the password chosen during the generation).

```
$ keytool -printcert -file ca.cert
```

used to display the contents of a certificate stored in a file.

```
$ openssl s_client
```

You can use this for debugging the server side.

Experiments you might want to do:

1. How the Java Client reacts if the CA certificate is not installed in the truststore beforehand?
2. How your browser reacts when connecting to your '`openssl s_server`'?
3. Describe the steps required for enabling a browser correctly validating your server when you point the browser to.
4. Attach a real HTTP server instead of `s_server`. Hint: you might want to consider first experimenting with the `-www` and `-WWW` options, and then installing a small

httpd daemon like minihttpd (http://acme.com/software/mini_httpd/). If you're brave enough, try configuring apache for SSL: APACHE + SSL

5. Experiment observing the SSL traffic using Wireshark. Which are the initial packets you notice? What you see after a connection is established?