PKI and TLS

Project 2, EITF55 Security, 2024

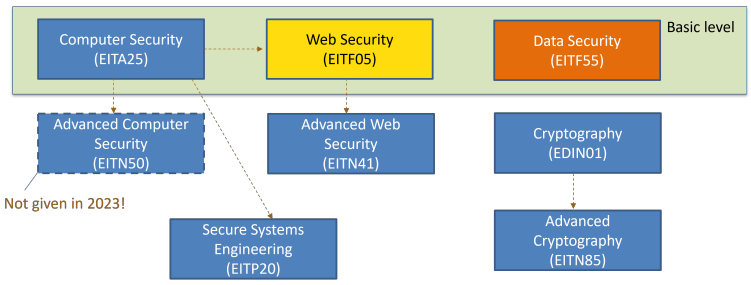
Ben Smeets and Karim Khalil
Dept. of Electrical and Information Technology, Lund University, Sweden

Last revised by Christian Gehrmann on
2025–01–20 at 16:32

What you will learn

In this project you will

- Study PKI certificates for servers and clients.
- Setup a PKI infrastructure, and key enrollment.
- Learn to use OpenSSL for handling certificate sign requests.
- Learn about Java TLS and the use of KeyStore and TrustStore.
- Learn about differences in how stacks use TLS.
- Intercept and analyse TLS data traffic.



Contents

1 General instructions

There are a number of exercises that guide your project work and you should use the exercises to structure your report. A list of tasks that should be included in your project report is given in section 9. While writing your report:

- Give clear indications where you put your answers to the assignments.
- For convenience name the report xyz_Project2_eitf55, where xyz corresponds to your group id.
- You should submit the reports electronically in pdf or word format and use the subject "Project2_EITF55" in the email that contains the report. Send it using the email address(es) specified on the course home page.

DO READ this entire document before you start coding and testing. The document contains useful information that will save you time if you are not familiar with the tools to generate certificates. Many of your previous year colleague students admitted that after reading the guides that are given here they got their program working correctly. In this instruction you will find several commands that are useful for debugging your implementation, configuration and listing the contents of your keystores.

BEFORE YOU ASK FOR HELP you should collect the help information that is stated in some of the exercises. Failing to do so may cause you to be sent back to gather this information first.

1.1 Checklist

You should submit

Item	Description
1	Report with your group number and names on it.
2	Printout of your certificates and include in the report.
3	Code of your Server and Client (as text file, not pdf) using server authentication only.
4	Code of your Server and Client (as text file, not pdf) using server and client authentication AND specific cipher suite selection.
5	Answers to the assignment questions in section 9.

2 Introduction

TLS is a one of the most used secure communication protocols. Every modern engineer working with data communication, automation, embedded systems, and web design should know how to setup and use TLS. In this project you will go through the main steps to have TLS support in an application and how to configure the required keys. One can use the TLS protocol directly in a program via a secure socket interface and web applications make use of TLS via the https protocol. TLS is derived from SSL and still today people speak of a "(secure) SSL connection" even if the underlying protocol is TLS. One of the nice features of TLS, is that integrated in the protocol is an authentication and session key agreement protocol. There are several options how to use TLS. TLS version 1.3 is to be used and not TLS 1.2 or older. This will limit some of the choices further on. In the course lectures you can read more about this. Here, in this project, we will only consider TLS in conjunction with RSA-based server and client authentication. You already studied RSA in Project 1. In TLS, the RSA algorithm has several roles. It is used in the authentication of the keys and it is used in the establishment of the session keys. The latter is, in principle, a simple step consisting of the encryption of a random value by the connecting client using the server's public key. By the properties of the RSA public-key crypto scheme it is only the server than that can decrypt this random value. From the random value the client and the server will compute their shared session key that will protect the subsequent data they will exchange. To perform the authentication of the RSA keys TLS assumes that the keys are organised in what is referred to as a Public Key Infrastructure (PKI). A PKI is usually a tree-like structure of approved keys where the data containers of the approved public keys are called certificates. To verify a certificate in the tree one uses the certificate on the previous level (closer to the root) of the tree, see Figure 1.

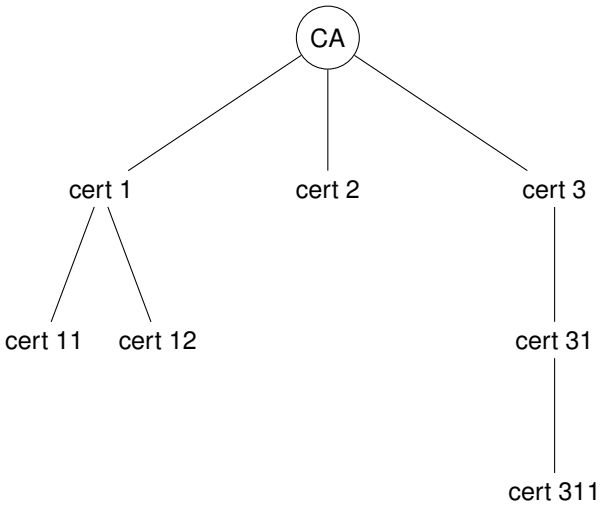


Figure 1: Example of PKI: tree organization with CA and other certificates.

The certificate at the root of the tree (lacking a previous level) is crafted such that it verifies it self. It is often called the root certificate (some even call it the root key). The root certificate is different from the the subsequent certificates in that it cannot be cryptographically verified. Instead the entity that needs to trust the root key must secure the use of the root key by other means. To create the PKI one establishes first a public and private key whose public key will be used in the root certificate. The entity that does this and who will keep the secret corresponding to the root certificate public key is called the Certificate Authority or just CA. The root certificate is therefore also called a CA certificate. The CA will issue certificates by creating signed (with its private key) approvals of other public keys. This step is called enrollment of a public key into the PKI.

In TLS, a server can be setup to use a self-signed certificate. However the client that connects to a

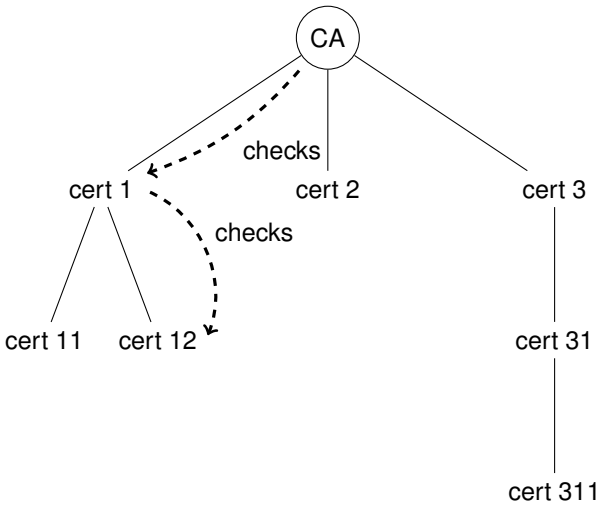


Figure 2: Complete certificate chain in PKI tree.

server and receives such a self-signed certificate cannot determine if it can trust this certificate unless it has been told beforehand, that is the client must have stored the certificate in some manner in its trusted registry. A more clever way to let the client check if it can trust a server certificate is to use server certificates that are signed by a CA. Then the client only needs to store the CA certificate and can use this CA certificate to verify the server certificate sent in the TLS protocol. During the TLS handshake the server will present to the client the certificates the client needs to perform the verification. If the CA signs the server certificate then the verification is simple. Check the server certificate with the CA certificate. In general the verification involves a chain of verifications starting with the CA certificate and traversing on the PKI tree down to the server certificate, see Figure 2. The list of certificates that are processed is referred to as "a complete certificate chain".

CA certificates are used in the verification of other certificates. The CA, client, and server's private key and the certificate for the complete certificate chain must be stored securely with password protection against access to the private key. We will be using PKCS12 format for the storage of server and client keys and certificates.

Thus the task to setup a TLS server-client implementation and a PKI consists of the following steps:

1. Generate RSA key pair for CA
2. Construct a (self-signed) CA certificate
3. Generate a RSA key pair for the server
4. Request CA to issue a certificate for the server public key
5. Server stores its private key in a PKCS12 file format
6. Client stores its private key, client certificate, and the CA certificate in PKCS12 file format

Now if TLS is used in a manner that that the server can authenticate the client, then the client must have a so-called client certificate. Similar to server certificates it is rational to have client certificate to be certificated and issued by a CA (or another entity in the PKI tree). The server can use the CA certificate to verify the client certificate. To support, this the previous steps have to be modified slightly.

1. Generate RSA key pair for CA
2. Construct (self-signed) CA certificate
3. Generate RSA key pair the server
4. Request CA to issue certificate for the server public key
5. Generate RSA key pair the client

6. Request CA to issue certificate for the client public key
7. Server stores its private key, client certificate, and the CA certificate in PKCS12 file format
8. Client stores its private key, client certificate, and the CA certificate in PKCS12 file format

In the subsequent sections of the project exercises you will be guided through the above steps and the construction of a simple client and server that communicate via TLS. You have the choice of implementing the TLS server and client in Python or Java. You will study the certificates that are generated so you will get a better understanding of their content. Furthermore, by intercepting the data traffic between the client and server you will see what kind of packages the TLS protocol sends.

Note: The check of the signatures in a certificate chain is the most crucial step of verifying a certificate of a server or client, this must always be preformed in PKI. However, a certificate contains in most cases more information. A very common attribute in a server certificate is a so-called SAN-list. SAN stands for Subject Alternative Name and the SAN entries, for example, one can designate the DNS names for which the certificate is valid. via a SAN list one can create a server certificate that holds for a server that serves my.server1.com or for my.server2.com, in the SAN entries are `my.server1.com, my.server2.com`. One must be careful before relying on the use of these entries as it is not always clear that these entries in a certificate are checked in an implementation. The HTTPS specification demands that implementations should check DNS names. In this project we use socket connections and there are many additional verifications of HTTPS that are not carried out (you have to add them programmatically!).

3 Setting up the PKI

3.1 CA certificate

First we must create our CA. Here we use a program library/tool called OpenSSL. OpenSSL consists of a many parts and here we will use those parts for generating RSA keys and the creation and dumping (in text form) of certificates.

Exercise 1 *Check that you have OpenSSL installed. See the "How To" section. Determine the speed of your machine by typing in a command prompt "openssl speed". OpenSSL starts to run the algorithms it currently has and shows how fast it goes. You might want to just interrupt this process as it will really take a long while to complete.*

Before calling help: *If the openssl command does not function make a dump of your computer's environment variables, e.g. in a command prompt under windows you write set > env.txt, which allows you to inspect the environment variables in the file env.txt. Of course you should run this in a directory where you have write permissions. In a UNIX type machine use `env` instead of `set`.*

We are now ready to create a CA key and certificate. OpenSSL offers various ways to do that. We use a method where we first create the RSA and afterwards the certificate. In this project we are happy with a 2048 bit RSA key and a CA certificate that lasts 3650 days (about 10 years). As you will see in the instructions you have to provide some information that will be embedded into the certificate. You are rather free to change the input to what you would like it to be.

To generate the RSA key type

```
openssl genrsa -aes128 -out rootCA.key 2048
```

You will be asked to provide a password to protect your RSA private key. The protection is through the AES algorithm in 128 bit mode using CBC. If you omit the "-aes128" argument in the command there will be no password protection of the RSA private key.

Next we create the self-signed certificate

```
openssl req -x509 -new -key rootCA.key -days 3560 -out rootCA.pem
Enter pass phrase for rootCA.key: <enter the one you used before when creating the key>
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]: SE
State or Province Name (full name) [Some-State]: Scania
Locality Name (eg, city) []:  Lund
Organization Name (eg, company) [Internet Widgits Pty Ltd]:  LU
Organizational Unit Name (eg, section) []: Education
Common Name (e.g. server FQDN or YOUR name) []: Demo CA
Email Address []: ca@demoland.se
```

The certificate is encoded in a text format called PEM. You can open the rootCA.pem file in your favourite editor. Another way to look at the content of the rootCA.pem file is by using the openssl command

```
openssl x509 -text -in rootCA.pem
```

Exercise 2 *Create a directory where you can store your keys and intermediate results and open a command prompt in that directory. Generate a 2048 bit RSA key and construct a CA certificate for your CA using the previously mentioned procedures. Use the openssl x509 -text -in <yourCA pem file name> to list the contents of your certificate*

1. *What is the serial number of your certificate ?*
2. *Who is subject and who was the issuer of the certificate?*
3. *What algorithm is used for signing ?*
4. *What algorithm is used for hashing ?*
5. *What is the public exponent (as decimal number)?*
6. *What values do appear as X509v3 extensions? What is the basic constraint?*

Before calling help: *It is best you created your working directory not as a sub directory of a system or OpenSSL installation directory. List the place from which you run your OpenSSL commands.*

3.2 Server and Client Certificate

We will continue to use OpenSSL to generate the Server and Client certificates. However we will use the Java `keytool` to view the PKCS121 certificates. For implementation in Java we will use `keytool` to store the CA certificate in the default location for the JVM, while this is not necessary and one can create its own TrustStore location, it is common practice to store trusted CA's in cacerts. See the "How To" section in this document.

Below you see an example how to generate a server certificate for the server. At this point it is recommended to create a directory for the storage of the server cryptographic files, you should create a separate directory when you are creating the client's cryptographic file as well.

In some setups the server's IP address or dns name, must be specified for TLS to accept it as server certificate for your server. The entry `yourdomain` can be `localhost` or the FQDN of the server, e.g. `www.mywebserver.se`. The entry is important when using https as the implementation might check that this entry matches the connection. In our examples that is not necessary.

```
openssl genpkey -aes128 -algorithm RSA -out server_key.pem -pkeyopt
rsa_keygen_bits:2048
```

In the above call to openssl we also created a server private RSA key used "server" as password. We store the keys in a PEM file. PEM stands for "Privacy Enhanced Mail," is a file format that is widely used for encoding cryptographic keys and X.509 digital certificates. You can see the textual presentation of the key via `openssl rsa -in server_key.pem -text` with `-text` options specifies the output to be in text representation. But as we know what we have done thus far is not enough to make TLS work. We need also to get a certificate for this key. First we must generate a certificate sign request that we send to the CA. Next our CA has to process this request. Here we use again OpenSSL. Below we give the two steps

```
openssl req -new -key server_key.pem -out server_csr.pem
----------
Enter pass phrase for server_key.pem:
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:SE
State or Province Name (full name) [Some-State]:Scania
Locality Name (eg, city) []:Lund
Organization Name (eg, company) [Internet Widgits Pty Ltd]:LU
Organizational Unit Name (eg, section) []:Education
Common Name (e.g. server FQDN or YOUR name) []:server.demoland.se
Email Address []:server@demoland.se

Please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []:
An optional company name []:
```

Now we shall ask openssl to sign the created certificate with the rootCA certificate and key.

The openssl command deserves some explanation. In particular the arguments `-extfile server_v3.txt` and `-set_serial 1`. The former instructs openssl to read data from the file `server_v3.txt` that contains Certificate version 3 extensions and the latter sets the serial number of the certificate. Omitting the `-extfile` argument will result in a version 1 certificate even if the CA certificate itself is a version 3 certificate. The entries in the extension file have to reflect the purpose of the certificate expressed in the basic constraints. You need to create a file with the name`server_v3.txt` and `client_v3.txt` inside each respective directory.For a server you need to enter the following in `server_v3.txt`

```
authorityKeyIdentifier=keyid,issuer
basicConstraints=CA:FALSE
keyUsage = keyAgreement, keyEncipherment, digitalSignature
subjectAltName = @alt_names
[ alt_names ]
DNS.1 = localhost
```

For a client certificate you need to enter the following in `client_v3.txt`

```
authorityKeyIdentifier=keyid,issuer
basicConstraints=CA:FALSE
keyUsage = digitalSignature, nonRepudiation, dataEncipherment
subjectAltName = @alt_names
```

```
[ alt_names ]
DNS.1 = localhost
```

The use of extensions is a science by itself and their usefulness depends on the certificate verification engine that is used when implementing TLS. The latter implies that different TLS implementations may react differently on the same certificate, so be warned. People frequently forget the extension file and thus get a v1 certificate instead of a v3. As an alternative to the `-extfile` argument one can use the openssl.cfg file to force that the processing of the certificate sign request results in a v3 certificate. We will not study this here and we do not recommend you to go this way unless you know what you are doing. We conclude by showing two useful commands; the first one to print certificates and the second to print a certificate sign request

Note you need to specify a location of the stored rootCA files[1]

```
openssl x509 -req -CA ../rootCA.pem -CAkey ../rootCA.key -in server_csr.pem
-out server_cert.pem -days 365 -extfile server_v3.txt -ser_serial 1
```

After this we will create a PKCS12 file that would include the server private key, the server certificate and the CA certificate. You will be prompted to enter the pass phrase for the server key, as well enter a pass phrase for the sever PKCS12 file, we will use the same password 'server'[2].

Note: We only need the CA certificate inside the PKCS12 file if the server is intended to to mutual TLS authentication of the client certificate. Usually it is the client application that includes the CA certificate in order to verify the server during the TLS handshake.

```
openssl pkcs12 -export -out server.p12 -inkey server_key.pem -in server_cert.pem
-certfile ../rootCA.pem
```

To view the content of the PKCS12 file and the certificate file, you can enter these command. Inspect the file and see how many certificate are included.

```
openssl x509 -in server_cert.pem -text
openssl pkcs12 -in server.p12 -nodes
```

Having gone through all these steps it is time to actually create your server and client certificate

Exercise 3 *Create a directories for the cryptographic files. Prepare a* `server_v3.txt` *and a* `client_v3.txt` *file containing the proper extensions as shown above. Generate RSA private keys of size 2048 bits for the server and client. The certificate that you generate should be valid for 365 days. Be mindful to enumerate the serial of the certificates.*

1. *Use the commands detailed before to generate your server and client certificate. Store these certificates and their keys so you know where they are. Do not forget to increment the serial number of the certificates. We do this here by hand (but one could let OpenSSL do this for you by managing a file where the serial number is stored).*
2. *Discuss the purpose and features of PKCS12 format in the context of PKI? Discuss it's advantages and why it is widely used?*
3. *Why should the entries* `CN=` *for each certificate be unique?*
4. *Investigate the server certificate* `server_cert.pem` *and identify the "X509v3 Authority Key Identifier". Where have you seen this value before?*
5. *Repeat the generation of the server certificate so you will get a v1 certificate instead of a v3. Do you need to generate a new private key for this?*
6. *Make prints of the certificates and add them as appendices to your report.*
7. *Generate also a server certificate that will expire after one day. We will use it later to test if the client really checks the expiry data.*

[1] The CA signing command in split over two lines. You should enter the command on one single line.
[2] The PKCS12 command in split over two lines. You should enter the command on one single line.

8. *Does the CA for the client and server certificates have to be the same? Motivate your answer.*

Before calling help: *Store the printouts in a file that you can show your project assistant.*

Exercise 4 *Note that the entries* `CN` *for the CA and server certificates differ.*

Why must all certificates in the PKI have a different entry?

Before calling help: *Remember this task when you proceed!*

Exercise 5 *In the previous tasks you likely used passwords at several occasions. Reflect on their purpose and your choices. What are the consequences of using all these passwords when operating a TLS server and client?*

3.3 Secure Storage for Keys and Certificates

Private key files (*.key) certificates (*.cer), and PKCS12 (.p12) files should be stored in a secure location. This location should not be publicly accessible and should be protected by appropriate operating system-level security measures.

Usually CA cryptographic files are stored in a separate infrastructure that is protected by many measures. A compromise of CA key or certificate means that every certificate that is generated by the CA is invalid or untrusted.

Exercise 6 *For the purpose of illustration in this course, we have created the CA locally to generate signed certificates for the server and client. Illustrate what are the best practice for securing the CA keys and certificates?*

For the server, those file should be stored in a directory that's only accessible by the server application or user running the server. rootCA.pem should be stored if the server is also verifying client certificates.

The client should store its keys and certificates in a secure location, not accessible by any user except the client application. rootCA.pem should be stored in a location accessible by the client to verify the server certificate when it connects to it.

For file permissions in Unix/Linux system, file permissions are set by chmod. permissions 400 or 600 could be used to set permissions for read access rights, or read/write access rights.

Keys are the most sensitive information therefore they should be only allowed to be read and no write or execute privilege allowed. certificates could be accessed by anyone as they are not private. However, no one should be able to modify them except their owner.

```
chmod 600 server.key
chmod 644 server.cer
chmod 444 rootCA.cer
```

The directory where the files are stored, should also be protected with limited access. If the client application is a web application, a choice of a directoty outside the web directory is prefarrable. Set permission level on sever and client directoty as follows.

```
chmod 700 /path/to/secure_dir
```

For Windows OS, the OS handles file permissions differently, using Access Control Lists (ACLs). You can set these using the File Explorer GUI or the icacls command in the command prompt.

```
Right-click on the key file (e.g., server.key) and select Properties.
Go to the Security tab.
Click on Edit to change permissions.
Set groups and users that allowed to access and view the file.
```

Exercise 7 *Apply the given instruction to set correct privileges. Set file and directory permissions for CA, server, and client Keys and certificate. Motivate your choice for permissions. Include file and folder permissions to your report.*

4 The TLS Server and Client

In this section you will implement a TLS server-client application. You have the choice to do that either in Java or python. Each section has its own set of exercises, you should only attempt to solve the exercises in the section related to the programming language you decided to do the implementation in. In your Assignment report, include only the answers related to section you chose.

4.1 Implementation in Java

Java TLS engine will use the PKCS12 file to extract the keys and certificate to use for the authentication and key establishment. The next step involves not only that the keys and certificates, it also involves the *verification* of the certificates. Towards this end Java maintains a TrustStore. The TrustStore should contain the CA certificate.

Normally a Java run time has already a truststore. The default location for this file is:
`<jre location>\lib\security\cacerts`.

The default keystore password for the cacerts file is "changeit". While system administrators should change the access rights and the password for this cacerts file but the password changeit will probably work on developer or testing machines.

On windows you need to run this command in an elevated (administrator) command window to have the permission to write the updated truststore file.

Keytool is a Java tool used to create, view and import keys into the JVM environment. Keytool is usually located inside `<jre location>\bin\`.

You can import the CA certificate we generated to this truststore using keytool as follows:

```
keytool -import -alias eda625test -file rootCA.pem -trustcacerts
        -keystore "+<jre location>\lib\security\cacerts"
```

To display the content of this keystore on a 64bit windows machine:

```
keytool -list -v -keystore "+<jre location>\lib\security\cacerts"
```

Verify the Certificate is added on On Unix/Linux Machine:

```
keytool -list -cacerts -storepass changeit | grep myCA
```

On Windows machine:

```
keytool -list -cacerts -storepass changeit | Select-String 'myca'
```

You can also create your custom TrustStore, in such case your Java code need to know the path for the custom TrustStore to extract the CA certificate.

You can create a custom TrustStore as follows:

```
keytool -import -file rootCA.pem -alias myCA -trustcacerts
        -keystore truststore.jks -storepass TrustPass
```

Exercise 8 *How is the Truststore file secured against modifications?*

Keytool could also be used to display the content of our PKCS12 file by this command:

```
keytool -list -v -keystore client.p12 -storetype PKCS12
```

Exercise 9 *When you list the content of, for example, the server PKCS12 file. You see an entry indicating the certificate chain length of the server key, e.g.*

```
Entry type: PrivateKeyEntry
Certificate chain length: 2
```

Why must this chain length be larger than 1 as this point?

We come back to this chain length question in another task.

In this section you construct a small server and client using secure sockets in Java. You have to implement a server that writes the text input sent by the client on the screen and echoes it back to the client which prints the text received back from the server. Look through the JSSE reference document and the example code at the bottom [2], see also [3]. As an starting example of a simple server and client you can use

```
//=============================================================================
//Sample server using sslsockets
import java.io.*;
import java.net.*;
import java.security.*;
import javax.net.ssl.*;
public class server {
private static final int PORT = 8043; // likely this port number is ok to use
// Server PKCS12 file path
private static final String PKCS12Location = "<path>/server.p12";
private static final String PKCS12Password = "server"; // Update if password changed
public static void main (String[] args) throws Exception {
boolean keepRunning = true;
// First we need to load a keystore
char[] passphrase = PKCS12Password.toCharArray();
KeyStore ks = KeyStore.getInstance("PKCS12");
try (FileInputStream fis = new FileInputStream(PKCS12Location)) {
    ks.load(fis, passphrase);
}
// Initialize a KeyManagerFactory with the KeyStore
KeyManagerFactory kmf = KeyManagerFactory.getInstance("SunX509");
kmf.init(ks, passphrase);
// KeyManagers from the KeyManagerFactory
KeyManager[] keyManagers = kmf.getKeyManagers();
//Adding custom TrustStore
//System.setProperty("javax.net.ssl.trustStore","<pathtoyour>/truststore.p12");
//System.setProperty("javax.net.ssl.trustStorePassword","changeit");
//tmf.init(ts);
// Obtain the default TrustManagers for the system's truststore (cacerts)
        TrustManagerFactory tmf = TrustManagerFactory.getInstance("SunX509");
        tmf.init((KeyStore) null); // Use the system's truststore 'cacerts'
        TrustManager[] trustManagers = tmf.getTrustManagers();
// Create an SSLContext to run TLSv1.3 and initialize it with
SSLContext context = SSLContext.getInstance("TLSv1.3");
context.init(keyManagers, null, new SecureRandom());
SSLServerSocketFactory ssf = context.getServerSocketFactory();
        // Create server socket
        try (SSLServerSocket ss = (SSLServerSocket) ssf.createServerSocket(PORT)) {
        //ss.setNeedClientAuth(true); // Require client authentication
        System.out.println("Server started and waiting for connections...");
            // Continuously accept new connections
            while (keepRunning) {
                try (SSLSocket s = (SSLSocket) ss.accept();
                    BufferedReader in = new
                    BufferedReader(new InputStreamReader(s.getInputStream()))) {
                    System.out.println("Client connected.");
                    // Read and process input from the client
                    String line;
                    while ((line = in.readLine()) != null) {
                        System.out.println(line);
                    }
```

```
                } catch (SocketException | EOFException e) {
                    System.out.println("Client disconnected abruptly.");
                    keepRunning = false;
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            System.out.println("Server stopped.");
        }
    }
}
//============================================================================
```

Note the use of port 8043, the password "server", and that we first wait for a connection. The client code is equally simple. We chose the use of 'client' as the pass phrase to the client PKCS12 file.

```
//================================================================================
//Sample client using sslsockets
import java.io.*;
import java.net.*;
import java.security.*;
import javax.net.ssl.*;
public class client {
private static final String HOST = "localhost";
private static final int PORT = 8043;
// Client PKCS12 file path
private static final String PKCS12Location = "<path>/client.p12";
private static final String PKCS12Password = "client"; // Update if password changed
//Add custom TrustStore password if not using cacerts
    //private static final String TSPassword = "changeit";
public static void main(String[] args) throws Exception {
char[] passphrase_ks = PKCS12Password.toCharArray();
//Adding custom TrustStore
        //char[] passphrase_ts = CACERTSPassword.toCharArray();
//KeyStore ts = KeyStore.getInstance("PKCS12");
//ts.load(new FileInputStream("/pathtoyour/"truststore.p12"), passphrase_ts);
//TrustManagerFactory tmf = TrustManagerFactory.getInstance("SunX509");
//tmf.init(ts);
//TrustManager[] trustManagers = tmf.getTrustManagers();
// Add code for Keystore
        // ??
SSLContext context = SSLContext.getInstance("TLSv1.3");
//TrustManager (2nd argu) is null to use the default trust manager cacerts
//To use custom TrustStore, 2nd argument changes to 'trustManagers'
//context.init(??, ??, ??); //Add correct arguments
SSLSocketFactory sf = context.getSocketFactory();
try (SSLSocket s = (SSLSocket) sf.createSocket(HOST,PORT)) {
OutputStream toserver = s.getOutputStream();
toserver.write("\nConnection established.\n\n".getBytes());
System.out.print("\nConnection established.\n\n");
int inCharacter=0;
inCharacter = System.in.read();
try {
while (inCharacter != '~')
{
toserver.write(inCharacter);
toserver.flush();
inCharacter = System.in.read();
}
}catch (SocketException | EOFException e) {
System.out.print("\nClient Closing.\n\n");
e.printStackTrace();
toserver.close();
s.shutdownOutput();
        s.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
}catch (IOException e) {
System.out.println("Cannot estabilish connection to server.");
            e.printStackTrace();
```

```
        } finally {
            System.out.println("client stopped.");
        }
    }
}
//=============================================================================
```

Observe that we assume that we run the server on the same machine as the client program is running. This is convenient but you can of course place the server at another location. However you must be sure that the the client accepts the hostname information from the server in its certificate. This is so per default.

Exercise 10 *Construct a TLS echo server and a matching client where the server receives text input from the client sent via TLS, prints the received data on the screen and then echos the received data back to the client. The client collects all the data the server returns in a buffer and prints this buffer after it closes the connection with the server. Test your client and server.*

Before calling help: *Make the programs run from the command line. This works for Windows, OSx, and Linux operating systems. Run the commands from two distinct directories that are not sub directories of system resources or OpenSSL installation files.*
It is convenient to construct bat/shell script files for running the server and client. In that way you reduce the typing you have to do. Especially if want to enter arguments, for example needed for debugging.

Exercise 11 *Show how the server would do mutual TLS authentication of the client certificate? You can do this by changing the 2nd argument for* `context.init(keyManagers, null, new SecureRandom());` *from null to* `trustManagers`

Note: To run the client and server programs it might be a good idea to export your programs as jar files using, for example, the Eclipse export function. After the export you can start the program on the command line, e.g., `java -jar myjarfile.jar`.

Exercise 12 *What happens if you in the previous assignment use a server certificate that has expired? What error codes you will see at the server and the client?.*

Exercise 13 *We previously noted that the keystore of the server had chain length 2 indicating that in the keystore we have both the server certificate and the ca certificate. One might wonder is it really needed for the server to have this CA certificate in its keystore after we have imported its server certificate into it? It is the client that will need to use the CA root certificate in the chain validation and there is no security gain in having the server send it too in the TLS handshake. We try our the remove the CA cert from the keystore and see if we still get a TLS connection. To remove the CA certificate enter:*

```
keytool -delete -alias myCA -keystore serverKeystore.jks
```

Start the server again and try to connect with your client.

In general, when using other SW stacks that implement TLS, the requirement of the rootCA cert to be included in the TLS handshake is sometimes a must. So Java and python solutions may work differently here!

Now we have a setup where only the client checks the server certificate. But we can add support for

working with client certificates by the following modifications to the server code

```
// in server: force use of client auth and add truststore
SSLSocket  s  =  (SSLSocket)ss.accept();
s.setNeedClientAuth(true);
// in client: Add keystore for its private key
```

We must use more keystores and truststores. One can also indicate what keystores/trustores to use on the command line. For example

```
java  -Davax.net.ssl.keyStore=serverKeyStore.jks
      -Djavax.net.ssl.keyStorePassword=123456
      -Djavax.net.ssl.trustStore=truststore.jks -jar tlsserver.jar
```

4.2 Implementation in Python

In this section you construct a small server and client using secure sockets in python. You have to implement a server that writes the text input sent by the client on the screen and echoes it back to the client which prints the text received back from the server. As an starting example of a simple server and client you can use

```
import os
import ssl
import socket
import tempfile
import threading
from cryptography.hazmat.primitives import serialization
from cryptography.hazmat.primitives.serialization.pkcs12 import \
                    load_key_and_certificates
# Configuration
SERVER_ADDRESS = 'localhost'
KEY_ALIAS = 'serverdomain'
SERVER_PORT = 8043
PKCS12_PATH = '<path>/server.p12'  # Update the path to PKCS12 file
PKCS12_PASSWORD = 'server'
def start_tls_server(address, port, pkcs12_path, pkcs12_password):
    cert_path, key_path, ca_path = None, None, None
    try:
        p12_password_bytes = pkcs12_password.encode('utf-8')
        with open(pkcs12_path, 'rb') as f:
            private_key, certificate, additional_certificates = \
                    load_key_and_certificates(f.read(), p12_password_bytes)
        # Extract the private key and certificate in PEM format
        server_key = private_key.private_bytes(
            encoding=serialization.Encoding.PEM,
            format=serialization.PrivateFormat.PKCS8, #Format for private key in pem
            encryption_algorithm=serialization.NoEncryption()
        )
        server_cert = certificate.public_bytes(serialization.Encoding.PEM)
        ca_cert = additional_certificates[0].public_bytes(serialization.Encoding.PEM) \
                if additional_certificates else None
        # Create a temporary file for the server certificate and key
        with (tempfile.NamedTemporaryFile(delete=False) as cert_file,
              tempfile.NamedTemporaryFile(delete=False) as key_file,
              tempfile.NamedTemporaryFile(delete=False) as ca_file):
            cert_file.write(server_cert)
            cert_path = cert_file.name
            key_file.write(server_key)
            key_path = key_file.name
            # Load the CA certificate for client authentication, when needed
            if ca_cert:
                    ca_file.write(ca_cert)
                    ca_path = ca_file.name

        # Create an SSL context
        context = ssl.create_default_context(ssl.Purpose.CLIENT_AUTH)
        context.load_cert_chain(certfile=cert_path, keyfile=key_path)
        if ca_cert:
            context.load_verify_locations(cafile=ca_path)
        else:
            raise RuntimeError("CA certificate not found")
        #Change this to CERT_REQUIRED to enable mutual TLS
```

```
        context.verify_mode = ssl.CERT_NONE
        with socket.socket(socket.AF_INET, socket.SOCK_STREAM, 0) as sock:
            sock.bind((address, port))
            sock.listen(1)
            print(f"Server listening on {address}:{port}")
            with context.wrap_socket(sock, server_side=True) as ssock:
                conn, addr = ssock.accept()
                with conn:
                    print(f"Connected by {addr}")
                    while True:
                        data = conn.recv(1024)
                        if not data:
                            break
                        message = data.decode()
                        print(f"Received message: {message}")
                        conn.sendall(data)  # Echoing back the received message
    except Exception as e:
        print(f"An error occurred: {e}")
    finally:
        # Clean up the temporary files
        for path in [cert_path, key_path, ca_path]:
            if path and os.path.exists(path):
                try:
                    os.remove(path)
                except Exception as e:
                    print(f"Error deleting temporary file {path}: {e}")
def run_server():
    start_tls_server(SERVER_ADDRESS, SERVER_PORT, PKCS12_PATH, PKCS12_PASSWORD)
# Run the server in a background thread
thread = threading.Thread(target=run_server, daemon=True)
thread.start()
thread.join()
```

Client code

```
import os
import ssl
import socket
import tempfile
from cryptography.hazmat.primitives import serialization
from cryptography.hazmat.primitives.serialization.pkcs12 import \
                    load_key_and_certificates
# Configuration
SERVER_ADDRESS = 'localhost'
SERVER_PORT = 8043
PKCS12_PATH = '<path>/client.p12'  # Update the path to PKCS12 file
PKCS12_PASSWORD = 'client'
def start_tls_client(server_address, port, pkcs12_path, pkcs12_password):
    cert_file, key_file, ca_file = None, None, None
    try:
        p12_password_bytes = pkcs12_password.encode('utf-8')
        with open(pkcs12_path, 'rb') as f:
            private_key, certificate, additional_certificates = \
                load_key_and_certificates(f.read(), p12_password_bytes)
        # Extract the private key and certificate in PEM format
        # add code here ??
        client_cert = certificate.public_bytes(serialization.Encoding.PEM)
        # Process additional certificates (usually includes the CA certificate)
        ca_cert = additional_certificates[0].public_bytes(serialization.Encoding.PEM) \
            if additional_certificates else None
        # Write the client certificate and key to temporary files
        with (tempfile.NamedTemporaryFile(delete=False) as cert_file,
              tempfile.NamedTemporaryFile(delete=False) as key_file,
              tempfile.NamedTemporaryFile(delete=False) as ca_file):
            cert_file.write(client_cert)
            cert_path = cert_file.name
            key_file.write(client_key)
            key_path = key_file.name
            if ca_cert:
                ca_file.write(ca_cert)
                ca_path = ca_file.name
        # Create an SSL context for a TLS client
        context = ssl.create_default_context(ssl.Purpose.SERVER_AUTH)
        context.load_cert_chain(certfile=cert_path, keyfile=key_path)
        if ca_cert:
            context.load_verify_locations(cafile=ca_path)
        else:
            raise RuntimeError("CA certificate not found")
        context.check_hostname = True
        received_messages = []  # List to store received messages
        with socket.create_connection((server_address, port)) as sock:
            with context.wrap_socket(sock, server_hostname=server_address) as ssock:
                print("Client connected to server")
                while True:
                    message = input("Enter message to send (or 'exit' to quit): ")
                    if message.lower() == 'exit':
                        break
                    ssock.sendall(message.encode())
                    response = ssock.recv(1024)
                    received_messages.append(response.decode())# Store received message
        print("Received messages during the session:")
```

```
        for msg in received_messages:
            print(msg)
    except Exception as e:
        print(f"An error occurred: {e}")
    finally:
        # Clean up the temporary files
        for path in [cert_path, key_path, ca_path]:
            if path and os.path.exists(path):
                try:
                    os.remove(path)
                except Exception as e:
                    print(f"Error deleting temporary file {path}: {e}")
start_tls_client(SERVER_ADDRESS, SERVER_PORT, PKCS12_PATH, PKCS12_PASSWORD)
```

Observe that we assume that we run the server on the same machine as the client program is running. This is convenient but you can of course place the server at another location. However you must be sure that the the client accepts the hostname information from the server in its certificate. This is so per default

Exercise 14 *Construct a TLS echo server and a matching client where the server receives text input from the client sent via TLS, prints the received data on the screen and then echos the received data back to the client. The client collects all the data the server returns in a buffer and prints this buffer after it closes the connection with the server. Test your client and server.*

Before calling help: *Make the programs run from the command line. This works for Windows, OSx, and Linux operating systems. Run the commands from two distinct directories that are not sub directories of system resources or OpenSSL installation files.*
It is convenient to construct bat/shell script files for running the server and client. In that way you reduce the typing you have to do. Especially if want to enter arguments, for example needed for debugging.

Exercise 15 *What happens if you in the previous assignment use a server certificate that has expired? What error codes you will see at the server and the client?.*

Exercise 16 *Go through the following tasks and add your answers in the report. All code need to be added to the report.*

1. *You might have noticed that the server and client pass phrases are hard coded in the code sample. This is not security best practice. Suggest two or more methods that are more efficient and highlight which of the two that should be most preferable and under which conditions.*
2. `openssl offers the curl option with s_client -connect "server addr".` *Explain how this could used to examine the server certificate chain length.*
3. *Create a new server certificate as shown earlier, sign it by the root CA. Add it to the server code to use instead. Re-check the chain length, did it change? Explain your finding and illustrate why or why not it has not changed. If the chain length does not change, suggest a scenario where the chain length would increase (you don't need to do any implementation, search online to understand the concept of intermediateCA and what are the purpose)*
4. *Show how the server would do mutual TLS authentication of the client certificate? You can do this by changing the argument for* `context.verify_mode`
5. *Create a new PKCS12 file for the server, this time don't include the CA certificate. With mutual TLS authentication set in the server code, what behaviour do you get? Add your steps for*

creating the new PKCS12 file, add a dump of the file, add screen shots of the code output and highlight what parts of the code you changed to create mutual TLS authentication.

5 TLS sockets and https

The experiments we have done use sockets and specifically secure sockets that use TLS. When using a browser to contact you back you also use TLS in the https protocol. Are secure sockets and https the same you might wonder. The simple strict answer is no. The more complicated answer is that secure sockets lie at the bottom of https so in essence it is TLS we are using but https is an specific application that adds things on top and as said before https is, for example, more restrictive how certificates are to be used.

Exercise 17 *Start your server again but now you use an browse(Firefox, Safari, Edge, or Chrome) to connect to your server, e.g. https://localhost:8043. Explain what happens.*

Note: Our server will likely crash here which is natural because the browser is not the client our server can fully handle. But something you see more than just the server crashing.

Exercise 18 *Add you CA certificate as trusted root certificate to your browser and repeat the previous task.*

The procedure how to add a root certificate depends on your browser. Search the internet for the right steps. For example for Firefox these are `https://docs.vmware.com/en/VMware-Adapter-for-SAP-Landscape-Management/2.0/Installation-and-Administration-Guide-for-VLA-Administrators/GUID-0CED691F-79D3-43A4-B90D-CD97650C13A0.html`

6 Analysis of TLS traffic

The TLS protocol runs on top of the TCP layer meaning that the TLS data is sent as TCP packets. We can look at these packets using a network analyser. To see what is sent via the network interface of your computer you can use a tool called Wireshark. Wireshark understands many protocols and by using its filtering capabilities we can zoom in on only the TCP packages, see the "How To" section.

Exercise 19 *Let us first consider the server authentication-only case. For TLS1.3, TLS handshake traffic is encrypted, and we would need to use the TLS session key to decrypt the messages in Wireshark. To do this, we need to set an environment variable for the session key at the server side, start the server application, and then add the session key to Wireshark.*

Steps:

- For Linux OS do:

```
export SSLKEYLOGFILE=/path/to/server/folder/sslkeylog.log
```

- For Windows OS do this in command line:

```
set SSLKEYLOGFILE=C:\path\to\server\folder\sslkeylog.log
```

- Start server application

- Use Wireshark to Decrypt Traffic: Open Wireshark and start capturing traffic of the local interface. Go to Edit > Preferences > Protocols > TLS and specify the path to your sslkeylog.log file. Wireshark will use the session keys from this file to decrypt TLS traffic for analysis.

Start the client aplication on your machine. Trace the TCP packages on the server's ports. Identify the following

- *Key exchange method and packets*
- *The certificate information the server presents to the client. In what order do the certificates appear?*
- *Rerun the above but with the server modified so it picks one specific cipher suite. This can be achieved by the* `setEnabledCipherSuites` *method of the SSLSocket.*

Now activate client authentication.

Exercise 20 *Start Wireshark and activate capturing of the local interface, and then start the server and the client on your machine. Trace the TCP packages on the servers ports. Identify the following*

- *Key exchange method and packets*
- *The certificate information the client presents to the server. In what order do the certificates appear?*

Before calling help: *If something here does not work you should read the sections below on debugging and provide printouts of the programs when debugging is enabled and you should also present a printout of the keystores that you are using. Check that the certificate chains in your keystore make sense.*

7 How To

7.1 Install OpenSSL

First check if OpenSSL is not already installed on your computer. Open a terminal/command window and type "openssl". If it gives a new shell prompt where you can enter commands to OpenSSL you are set. Otherwise you have to install OpenSSL. To install OpenSSL do

Under Windows visit `http://slproweb.com/products/Win32OpenSSL.html` and read the information there (also on additional packages that might be needed) and download the 32bit installer (Win32 OpenSSL v1.1.1i Light) or the 64bit equivalent and install it on your computer. After installation you need to add the path to the OpenSSL bin directory to you path. After that your path should look something similar like `PATH=C:\Program Files (x86)\MiKTeX 2.9\miktex\bin;` C:\OpenSSL-Win64\bin

Under OS X Nothing to do, it is already there.

Under Ubuntu/Mint sudo apt-get install libssl.

There are also development packages for OpenSSL but we do not need those.

7.2 Java keytool

If you want more information on keytool you should consult [4]. Below we sample some of the information from `http://www.sslshopper.com/article-most-common-java-keytool-keystore-commands.`

`html`. On Windows a path needs to be added to the keytool bin directory similar to openssl. Probably something like: `PATH=C:\Program Files\Java\jdk-13.0.2\bin` Most often your have JDK installed when you also develop java code. Otherwise the runtime JRE suffices.

```
Java Keytool Commands for creation of keys/certs

    Generate a Java keystore and key pair:
    keytool -genkey -alias mydomain -keyalg RSA -keystore keystore.jks -keysize 2048

    Generate a certificate signing request (CSR) for an existing Java keystore:
    keytool -certreq -alias mydomain -keystore keystore.jks -file mydomain.csr

    Import a root or intermediate CA certificate to an existing Java keystore:
    keytool -import -trustcacerts -alias root -file myCA.crt -keystore keystore.jks

    Import a signed primary certificate to an existing Java keystore:
    keytool -import -trustcacerts -alias mydomain -file mydomain.crt -keystore keystore.jks


Java Keytool Commands for Checking

If you need to check the information within a certificate, or Java keystore,
use these commands.

    Check a stand-alone certificate:
    keytool -printcert -v -file mydomain.crt

    Check which certificates are in a Java keystore:
    keytool -list -v -keystore keystore.jks

    Check a particular keystore entry using an alias:
    keytool -list -v -keystore keystore.jks -alias mydomain

Other Java Keytool Commands

    Delete a certificate from a Java Keytool keystore:
    keytool -delete -alias mydomain -keystore keystore.jks

    Change a Java keystore password:
    keytool -storepasswd -new new_storepass -keystore keystore.jks

    Export a certificate from a keystore:
    keytool -export -alias mydomain -file mydomain.crt -keystore keystore.jks

    List Trusted CA Certs:
    keytool -list -v -keystore $JAVA_HOME/jre/lib/security/cacerts
```

7.3 Install and use Wireshark

Installing is rather simple

Windows and OS X Goto the download section of `http://www.wireshark.org/` there you find images for your Windows and OS X.

Ubuntu/Mint Depends on what version you are running. It is in the latest app repository. YOU

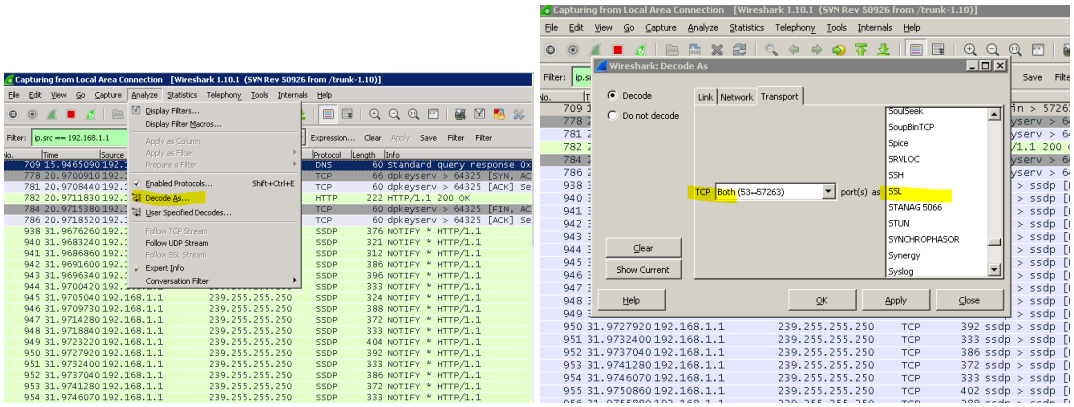


Figure 3: a) Analyze->Decode As, b) Set TCP and SSL.

> SHOULD RUN Wireshark FROM A TERMINAL WINDOW using sudo. If you want to run wireshark without sudo do the following: sudo dpkg-reconfigure wireshark-common press the right arrow and enter for yes. sudo chmod +x /usr/bin/dumpcap you should now be able to run wireshark without root but I (=Ben) did not test this well.

It might be a good idea also to download the User's Guide.

On Windows older versions of Wireshark cannot capture from the loopback interface. Hence you cannot do a live capture of the data sent between the server and the client if you run both on the same Windows machine. For our purpose we can partly solve that by using a loopback sniffer called `RawCap.exe` which you can download from `http://www.netresec.com/?page=RawCap`. Place it in your work directory and just double klick on it and the program will start and asks you the interface to perform the capture on. The data that it captures is stored in a file that can be opened by Wireshark. However the most recent Wireshark version `3.4.2` with its plugins should allow you to monitor the loopback interface directly.

If you not have worked with Wireshark or similar program before you should play around a bit with it. For example you could try to log the data when browsing to the Lund University site `www.lu.se`. Especially you should learn how to filter for specific data. An overview of the capture filter syntax can be found in the Wireshark User's Guide. Below we show some filters, see also [5] and [6]

```
Filter only traffic to or from IP address 192.168.1.1:
ip == 192.168.1.1

Capture only from
ip.src == 192.168.1.1

Capture only to
ip.dst == 192.168.1.1

Filter on port(s) and tcp
tcp.port == 8080
```

Wireshark does understand the SSL/TLS protocol. So it is very handy to let Wireshark do the decoding of the TCP data for you. Towards this end you select from the menu:Analyze->decode as and then set the protocol (under Network tab) to TCP and then to SSL, see figures a) and b) in Figure 3 [3].

[3] As you are using TLS 1.2 and 1.3 you need to adjust some setting in wireshark to be able to decrypt the traffic, look at the Using the (Pre)-Master-Secret section in this link

7.4 Debug

When you get an execution error JSSE will print some error messages. Normally you will not easily understand these. Luckily you can run your program in a debug mode which gives more detailed information. Do as follows:

```
    java -Djavax.net.debug=all -jar myjar.jar
To get a hexadecimal print of the Handshake messages one can use
    java -Djavax.net.debug=ssl:handshake:data  -jar myjar.jar

The other options are
-     all
-     ssl
        o record            (activate per-record tracing)
            plaintext        (print hexadecimal content)
        o handshake         (print Handshake Messages)
            data             (print hexadecimal content Handshake messages)
            verbose          (more info than data)
        o keygen            (show key generation)
        o session           (show Session activity)
        o defaultctx        (Standard SSL initialisation)
        o sslctx            (trace SSLContext )
        o sessioncache      (trace Session Cache)
        o keymanager        (trace Keymanager)
        o trustmanager      (trace Trustmanager)
```

7.5 Force use of specific cipher suites

The code below can be added to (server or client ??) to force one or several predetermined cipher suites.

```
SSLSocket s = (SSLSocket)ss.accept();
String pickedCiphers[] = {"TLS_AES_128_GCM_SHA256", "TLS_AES_128_CCM_SHA256"};
s.setEnabledCipherSuites(pickedCiphers);
```

Note: The TLSv1.3 specification mentions other cipher suites but they are not all supported in Java 13.

8 If everything fails

If for some reasons you get completely stuck and we ask you to send your code you should send both client and server code (text not jar or binary) and the keystore(s) and truststore(s) your are using with their respective passwords so we can try to repeat your problem at our side.

References

[1] Java SE 13 Security Overview, https://docs.oracle.com/en/java/javase/13/security/java-security-overview1.html#GUID-2EF91196-D468-4D0F-8FDC-DA2BEA165D10, last accessed on 2020-03-09.

[2] Java Secure Socket Extension (JSSE) Reference Guide, https://docs.oracle.com/en/java/javase/13/security/java-secure-socket-extension-jsse-reference-guide.html#GUID-93DEEE16-0B70-40E5-BBE7-55C3FD432345, last accessed on 2020-03-09.

[3] JSSE Samples, https://docs.oracle.com/en/java/javase/13/security/java-secure-socket-extension-jsse-reference-guide.html#GUID-0573BCE4-05C4-429C-8ECC-3D3D8CA807F4, last accessed on 2020-03-09.

[4] keytool documentation, https://docs.oracle.com/en/java/javase/13/docs/specs/man/keytool.html, last accessed on 2020-03-09.

[5] Wireshark filtering packets, https://www.wireshark.org/docs/wsug_html_chunked/ChWorkDisplayFilterSection.html, last accessed on 2020-03-09.

[6] Wireshark capture filters, https://wiki.wireshark.org/CaptureFilters, last accessed on 2020-03-09.

9 Assignment questions

It is mandatory for you to answer *all the questions* in this section for the assignment. All values shall be properly documented and the code used to answer the questions shall be submitted together with the report and output values you have obtained when solving the assignment questions.

Refer to Section 3.1 for submission instructions.

9.1 Question A

Generate a 2048 bit RSA key and construct a CA certificate for your CA. Use the openssl x509 -text -in <yourCA pem file name> to list the contents of your certificate.

1. What is the serial number of your certificate ?
2. Who is subject and who was the issuer of the certificate?
3. What algorithm is used for signing ?
4. What algorithm is used for hashing ?
5. What is the public exponent (as decimal number)?
6. What values do appear as X509v3 extensions? What is the basic constraint?

Refer to section 3.1 for instructions.

9.2 Question B

Prepare a `server_v3.txt` and a `client_v3.txt` file containing the proper extensions.

Generate server and client RSA keys of size 2048 bits.

Generate your server and client certificate. Store these certificates and their keys so you know where they are. Do not forget to increment the serial number of the certificates. We do this here by hand (but one could let OpenSSL do this for you by managing a file where the serial number is stored). The certificate that you generate should be valid for 365 days and should use SHA1 as hash algorithm.

1. Use the commands detailed before to generate your server and client certificate. Store these certificates and their keys so you know where they are. Do not forget to increment the serial number of the certificates. We do this here by hand (but one could let OpenSSL do this for you by managing a file where the serial number is stored).
2. Discuss the purpose and features of PKCS12 format in the context of PKI? Discuss it's advantages and why it has been widely used?
3. Why should the entries `CN=` for each certificate be unique?
4. Investigate the server certificate `server_cert.pem` and identify the "X509v3 Authority Key Identifier". Where have you seen this value before?
5. Repeat the generation of the server certificate so you will get a v1 certificate instead of a v3. Do you need to generate a new private key for this?
6. Generate also a server certificate that will expire after one day. We will use it later to test if the client really checks the expiry data.
7. Does the CA for the client and server certificates have to be the same? Motivate your answer.
8. given that you have inspected the CA, server, and client certificates, Why must all certificates in the PKI have a different entry?
9. In the previous tasks you likely used passwords at several occasions. Reflect on their purpose and your choices. What are the consequences of using all these passwords when operating a TLS server and client?
10. Make prints of the certificates and add them as appendices to your report.

Refer to section 3.2 for instructions. Make sure you correctly add the server and client certificates in the correct file.

9.3 Question C

In this question we will look into the secure storage of keys and certificates

1. For the purpose of illustration in this course, we have created the CA locally to generate signed certificates for the server and client. Illustrate what are the best practice for securing the CA keys and certificates?
2. Set file and directory permissions for CA, server, and client Keys and certificate. Motivate your choice for permissions. Include the steps you done to apply file and folder permissions to your report as well as screenshots of the resulted permissions.

Refer to section 3.3.

9.4 Question D

Construct a TLS echo server and a matching client where the server receives text input from the client sent via TLS, prints the received data on the screen and then echos the received data back to the client. The client collects all the data the server returns in a buffer and prints this buffer after it closes the connection with the server. Test your client and server.

You can choose to do this question in either Java or Python, depending on your choice you can answer questions in section 9.4.1, or 9.4.2

9.4.1 Java Implementation Questions

1. Add your code to the Assignment report. You can add screenshots of your test outputs.
2. Explain How is the Truststore file (cacerts) secured against modifications?
3. You might have noticed that the server and client pass phrases are hard coded in the code sample. This is not security best practice. Suggest two or more methods that are more efficient and highlight which of the two that should be most preferable and under which conditions. *Hint. If a password is stored in clear text, say in a code file that is accessible by anyone then it is not really secure, think how do you enter your passwords lets say to mail application. Think of another method for password extraction based on secure storage.*
4. When you list the content of, for example, the server PKCS12 file. You see an entry indicating the certificate chain length of the server key, e.g.

   ```
   Entry type: PrivateKeyEntry
   Certificate chain length: 2
   ```

   Why must this chain length be larger than 1 as this point?
5. What happens if you in the previous assignment use a server certificate that has expired? What error codes you will see at the server and the client?
6. Show how the server would do mutual TLS authentication of the client certificate? You can do this bychanging the 2nd argument for
   `context.init(keyManagers, null, new SecureRandom());` from null to `trustManagers`
7. We previously noted that the keystore of the server had chain length 2 indicating that in the keystore we have both the server certificate and the ca certificate. One might wonder is it really needed for the server to have this CA certificate in its keystore after we have imported its server certificate into it? It is the client that will need to use the CA root certificate in the chain validation and there is no security gain in having the server send it too in the TLS handshake. Try to remove the CA cert from the keystore and see if you still get a TLS connection. To remove the CA certificate enter: `keytool -delete -alias myCA -keystore serverKeystore.jks` Start the server again and try to connect with your client. What behaviour does your client and server experiance. Explain in your own words, add screenshots to motivate your answer

Refer to section 4.1 for instructions.

9.4.2 Python Implementation Questions

1. Add your code to the Assignment report. You can add screenshots of your test outputs.
2. What happens if you in the previous assignment use a server certificate that has expired? What error codes you will see at the server and the client?
3. You might have noticed that the server and client pass phrases are hard coded in the code sample. This is not security best practice. Suggest two or more methods that are more efficient and highlight which of the two that should be most preferable and under which conditions. *Hint. If a password is stored in clear text, say in a code file that is accessible by anyone then it is not really secure, think how do you enter your passwords lets say to mail application. Think of another method for password extraction based on secure storage.*
4. `openssl offers the curl option with s_client -connect "server addr"`. Explain how this could used to examine the server certificate chain length.
5. Create a new server certificate as shown earlier, sign it by the root CA. Add it to the server code to use instead. Re-check the chain length, did it change? Explain your finding and illustrate why or why not it has not changed. If the chain length does not change, suggest a scenario where the chain length would increase (you don't need to do any implementation, search online to understand the concept of intermediateCA and what are the purpose)
6. Show how the server would do mutual TLS authentication of the client certificate? You can do this by changing the argument for `context.verify_mode`
7. Create a new PKCS12 file for the server, this time don't include the CA certificate. With mutual TLS authentication set in the server code, what behaviour do you get? Add your steps for creating the new PKCS12 file, add a dump of the file, add screen shots of the code output and highlight what parts of the code you changed to create mutual TLS authentication.

Refer to section 4.2 for instructions.

9.5 Question E

1. Start your server again but now you use an browse(Firefox, Safari, Edge, or Chrome) to connect to your server, e.g. https://localhost:8043. Explain what happens.

   Note: Before you start with this question you need to disable TLS mutual authentication from the server side.

   Note: Our server will likely crash here which is natural because the browser is not the client our server can fully handle. But something you see more than just the server crashing.
2. Add you CA certificate as trusted root certificate to your browser and repeat the previous task. Explain what happens.

The procedure how to add a root certificate depends on your browser. Search the internet for the right steps. For example for Firefox these are
`https://docs.vmware.com/en/VMware-Adapter-for-SAP-Landscape-Management/2.0/Installation-and-Administration-Guide-for-VLA-Administrators/GUID-0CED691F-79D3-43A4-B90D-CD97650C13A0.html`

Refer to section 5 for instructions.

9.6 Question F.1

Let us first consider the server authentication only case. Start Wireshark and activate capturing of the local interface, and then start the server and the client on your machine. Trace the TCP packages on the servers ports. Identify the following

1. Key exchange method and packets

2. The certificate information the server presents to the client. In what order do the certificates appear?
3. Rerun the above but with the server modified so it picks one specific cipher suite. This can be achieved by the `setEnabledCipherSuites` method of the SSLSocket.

Highlight your finding in your assignment report by using screenshots of the captured traffic. Explain each step and what you noticed.

Refer to section 6 for instructions.

9.7 Question F.2

Now activate client authentication with mutual TLS authentication.

Start Wireshark and activate capturing of the local interface, and then start the server and the client on your machine. Trace the TCP packages on the servers ports. Identify the following

1. Key exchange method and packets
2. The certificate information the client presents to the server. In what order do the certificates appear?

Highlight your finding in your assignment report by using screenshots of the captured traffic. Explain each step and what you noticed.

Refer to section 6 for instructions.