PKI and TLS

Project 2, EITF55 Security, 2021

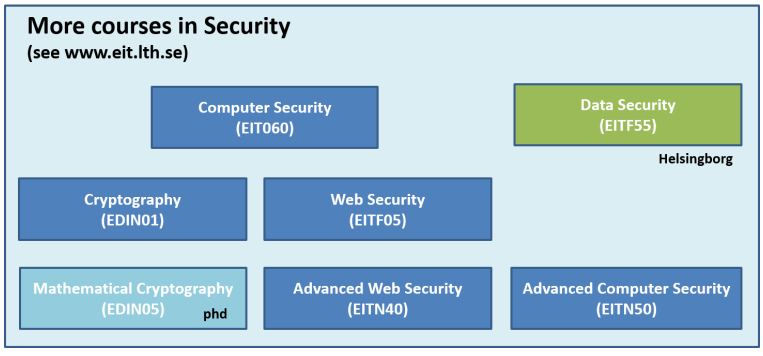
Ben Smeets
Dept. of Electrical and Information Technology, Lund University, Sweden

Last revised by Ben Smeets on
2021–01–05 at 17:42

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 assignments that guide your project work and you should use the assignments to structure your report.

- Give clear indications where you put your answers to the assignments.
- For convenience name the report Project2_eitf55, where xyz corresponds to your group id.
- You should submit the reports electronically in pdf or word format and use the subject "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 assignments. 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 keystores and truststores (with keytool -v option) included in the report.
3	Code of your Server and Client (as text file, not pdf) using server authentication only.
4 5	Code of your Server and Client (as text file, not pdf) using server and Client authentication AND specific cipher suite selection.

2 Introduction

TLS is a one of the most used secure communication protocols. Every modern engineer working with data communication, automatisation, 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 a Java 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 one has 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 organisation 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.

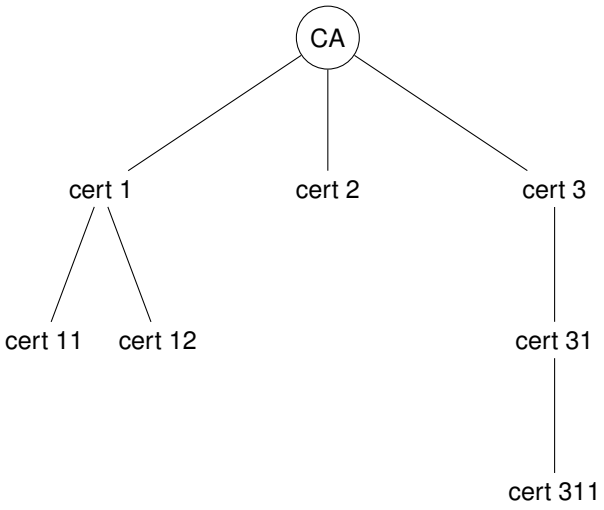


Figure 1: Example of PKI: tree organization with CA and other certificates.

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 others 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

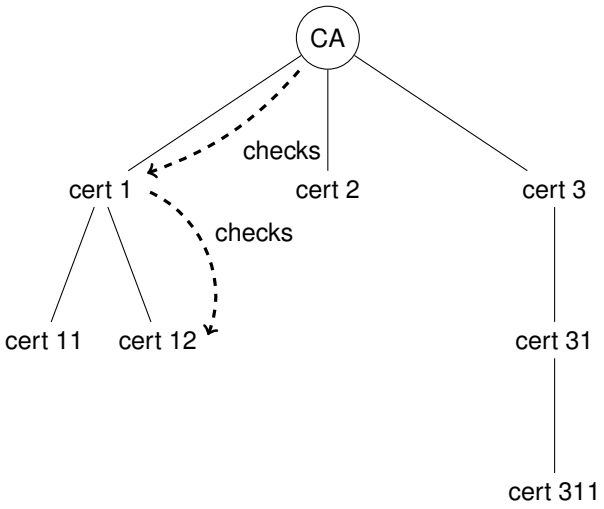


Figure 2: Complete certificate chain in PKI tree.

server and receives such a self-signed certificate cannot determine if it can trust this certificate unless it has been told beforehand that the server must have this certificate, that is the client must have stored this beforehand. 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 server certificate and traversing on the PKI tree down to the CA certificate, see Figure 2. The list of certificates that are processes is referred to as "a complete certificate chain".

When using Java the method to store the CA certificates that are used in the verification of other certificates is called the TrustStore. The server's private key and the certificate for the complete certificate chain must be stored securely and for this Java has a KeyStore that supports password protected access to the private key. The realisation of the TrustStore and KeyStore can done in varies ways and will have an impact on the actual level of security that is achieved by the realization. In this project we will ignore such issues and will assume that the implementations of the the TrustStore and KeyStore are sound.

Thus the task to setup a Java 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 its KeyStore
6. Client stores CA certificate in its TrustStore.

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 certificates to be certificates issued by a CA (or another entity in the PKI tree). The server can use the CA certificate to verify the client certificates. 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 in its KeyStore and the CA certificate in its TrustStore.
8. Client stores its private key in its KeyStore and the CA certificate in its TrustStore.

In the subsequent sections of the project assignment you will be guided through the above steps and the construction of a simple Java client and server that communicate via TLS. 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 a the signatures is a certificate chain is the most crucial step of verifying a certificate of a server or client that always must be preformed. But 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, can designate the DNS names for which the certificate is valid. That is via a SAN list you can create a server certificate that holds for a server that serves my.server1.com or for the server for my.server2.com is the SAN entries are `my.server1.com, my.server2.com`. However it is not always clear that these entries in a certificate are checked in an implementation. So one must be careful before relying on the use of these entries. The https specification is demanding that implementations should check dns names. In this project we use socket connections and there many additional verifications of https 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. Unfortunately we cannot directly do this with standard Java tools. 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.

Assignment 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 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 private RSA 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
```

Assignment 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 could continue to use OpenSSL to generate the Server and Client certificates. However we will use the Java `keytool` instead. The `keytool` command is a versatile command that can be used for many purposes. See the "How To" section in this document.

Below you see an example how to generate a server certificate for the server. In some setups the server's IP address or dns name, like myserver.demoland.se, must be specified for TLS to accept it as server certificate for your server. In our examples that is not necessary.

```
keytool -genkey -alias yourdomain -keystore keystore.jks -keyalg RSA -sigalg Sha1withRSA
What is your first and last name?
```

```
  [Unknown]:  Ben Smeets
What is the name of your organizational unit?
  [Unknown]:  Education
What is the name of your organization?
  [Unknown]:  LU
What is the name of your City or Locality?
  [Unknown]:  Lund
What is the name of your State or Province?
  [Unknown]:  Scania
What is the two-letter country code for this unit?
  [Unknown]:  SE
Is CN=Ben Smeets, OU=Education, O=LU, L=Lund, ST=Scania, C=SE correct?
  [no]:  yes
```

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. Here it does not matter.

In the above call to keytool we also created a keystore and places the private key in this keystore and used "123456" as password. It is useful to call the keystore file `serverKeyStore.jks` in case we work on the server keys and certificates. You can list the entries in the keystore by issuing the command `keytool -list alias yourdomain -keystore keystore.jks .` 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. This sign request can be generated by keytool and we give it file file name 'server.csr'. Next our CA has to process this request. Here we use again OpenSSL. Below we give the two steps[1]

```
keytool -certreq -alias yourdomain -keystore keystore.jks -file server.csr
openssl  x509  -req  -CA rootCA.pem -CAkey rootCA.key  -in server.csr
        -out server.cer  -days 365 -extfile server_v3.txt  -set_serial 1
```

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. For a server we typically have

```
authorityKeyIdentifier=keyid,issuer
basicConstraints=CA:FALSE
keyUsage = keyAgreement, keyEncipherment, digitalSignature
```

For a client certificate we typically have

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

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

[1] The second command in split over two lines. You should enter the command on one single line.

```
openssl x509 -text -inform pem -in server.cer
openssl req -text -noout -in server.csr
```

In the above the `-inform pem` argument is optional. If the certificate is in `der` (=binary) format one should enter `-inform der`.

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

Assignment 3 *Prepare a* `server_v3.txt` *and a* `client_v3.txt` *file containing the proper extensions as shown above. Generate RSA keys of size 1024 bits. The certificate that you generate should by 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. *Why should the entries* `CN=, OU=, O=, L=, ST=, C=` *for each certificate be unique?*
3. *Investigate the server certificate and identify the "X509v3 Authority Key Identifier". Where have you seen this value before?*
4. *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?*
5. *Make prints of the certificates and add them as appendices to your report.*
6. *Generate also an 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.*

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

Assignment 4 *Note that the entries* `CN=, OU=, O=, L=, ST=, C=` *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!*

3.3 Populating our KeyStore and TrustStore

You should by now have a clientKeyStore.jks file and a serverKeyStore.jks file containing the client key and server key, respectively. We add the certificates that we generated to the keystores. Adding a certificate, say cert.cer, to the keystore.jks can be done by the following keytool command

```
keytool -importcert -file cert.cer -keystore keystore.jks -alias youralias
```

Assignment 5 *Note that we give the certificate a friendly alias name: "youralias". if the certificate is a response of csr request you should use the same alias as for the key that the certificate request was initiated. The import should conclude with the message* `Certificate reply was installed in keystore`.

Add the certificates that you generated before to their respective key stores. List their contents and add a printout in the appendix of your report. Why must you add first the rootCA cert to the keystore beside the client/server certificate. Why?

Before calling help: *Dump the contents of the keystores in a file that you can show your project assistant.*

To summarize, the flow of commands for creating the server keystore serverKeyStore.jks is thus:

```
keytool -genkey -alias yourdomain -keystore serverKeyStore.jks -keyalg RSA
        -sigalg Sha1withRSA
keytool -certreq -alias yourdomain -keystore serverKeyStore.jks -file server.csr
openssl  x509  -req  -CA rootCA.pem -CAkey rootCA.key -in server.csr
         -out server.cer -days 365 -extfile server_v3.txt -set_serial 1
keytool -importcert -file rootCA.pem -keystore serverKeyStore.jks -alias myCA
keytool -importcert -file server.cer -keystore serverKeyStore.jks -alias yourdomain
```

Assignment 6 *Repeat the above steps to create the keystore* `test.jks`*. The difference in making this keystore is that at the end you import your server certificate with the command*

```
keytool -importcert -file server.cer -keystore serverKeyStore.jks -alias testdomain
```

that is you use not the same alias when importing as you used for the key generation and csr creation.

List the new keystore with the keytool using the `-v` *option to see more details of the entries.*

Describe the differences? What happened in this case compared to the case where the import was done with `yourdomain` *as alias.*

Now the Java TLS engine knows what keys and certificates to use for the authentication and key establishment. Well almost, the step of authentication involves not only that the keys and certificates are used and presented, it also involves the *verification* of the certificates. Towards this end Java maintains a TrustStore. The TrustStore should contain the CA certificate which can be created and added as follows:

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

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

```
keytool -list -v -keystore "C:\Program Files\Java\jdk-13.0.2\lib\security\cacerts"
```

You can import the CA certificate we generated to this truststore by

```
keytool -import -alias eda625test -file rootCA.pem -trustcacerts
        -keystore "C:\Program Files\Java\jdk-13.0.2\lib\security\cacerts"
```

On windows you need to run this command in an elevated (administrator) command window to have the permission to write the updated truststore file. We will **NOT** put our project CA certificate in the default TrustStore. This to avoid introducing problems with your usual Java Runtime environment.

Assignment 7 *1. How and who did place the certificates in the Java* ***default*** *TrustStore in the first place?*
2. How is the Truststore file secured against modifications?

Assignment 8 *When you list the keystore of, for example, the server. 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 a next task.

Assignment 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?*

4 The TLS Server and Client

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 tlsserver using sslsockets
import java.io.*;
import java.net.*;
import java.security.*;
import javax.net.ssl.*;
public class tlsserver {
// likely this port number is ok to use
    private static final int PORT = 8043;
    public static void main (String[] args)  throws Exception {
       //set necessary truststore properties - using JKS
       //System.setProperty("javax.net.ssl.trustStore","C:/pathtoyour/truststore.jks");
       //System.setProperty("javax.net.ssl.trustStorePassword","changeit");
       // set up key manager to do server authentication
       SSLContext context;
       KeyManagerFactory kmf;
       KeyStore ks;
 // First we need to load a keystore
       char[] passphrase = "123456".toCharArray();
       ks = KeyStore.getInstance("JKS");
       ks.load(new FileInputStream("C:/pathtoyour/serverKeyStore.jks"), passphrase);
// Initialize a KeyManagerFactory with the KeyStore
       kmf = KeyManagerFactory.getInstance("SunX509");
       kmf.init(ks, passphrase);
// Create an SSLContext to run TLS and initialize it with
// KeyManagers from the KeyManagerFactory
       context = SSLContext.getInstance("TLSv1.3");
       KeyManager[] keyManagers = kmf.getKeyManagers();
       context.init(keyManagers, null, null);
// Create a SocketFactory that will create SSL server sockets.
       SSLServerSocketFactory ssf = context.getServerSocketFactory();
// Create socket and Wait for a connection
       ServerSocket ss = ssf.createServerSocket(PORT);
//     Socket s = ss.accept();
// Get the input stream. En/Decryption happens transparently.
       BufferedReader in = new BufferedReader(new InputStreamReader(s.getInputStream()));

// Read through the input from the client and display it to the screen.
       String line = null;
       while (((line = in.readLine())!= null)) {
            System.out.println(line);
       }
       in.close();
       s.close();
      }
}
//=============================================================================
```

Note the use of port 8043, the password "123456", and that we first wait for a connection. The client

code is equally simple.

```
//=============================================================================
//Sample tlsclient using sslsockets
import java.io.*;
import java.net.*;
import java.security.*;

import javax.net.ssl.*;
public class tlsclient {
private static final String HOST = "localhost";
  private static final int PORT = 8043;
  public static void main(String[] args) throws Exception {
    // TrustStore
char[] passphrase_ts = "changeit".toCharArray();
    KeyStore ts = KeyStore.getInstance("JKS");
    ts.load(new FileInputStream("c:/pathtoyour/"truststore.jks"), passphrase_ts);
    TrustManagerFactory tmf = TrustManagerFactory.getInstance("SunX509");
    tmf.init(ts);
    // Keystore ?

    SSLContext context = SSLContext.getInstance("TLSv1.3");
    TrustManager[] trustManagers = tmf.getTrustManagers();
    KeyManager[] keyManagers   = NULL;

    context.init(keyManagers, trustManagers, new SecureRandom());
    SSLSocketFactory sf = context.getSocketFactory();
    Socket s = 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();
    while (inCharacter != '~')
    {
      toserver.write(inCharacter);
      toserver.flush();
        inCharacter = System.in.read();
    }
    toserver.close();
    s.close();
  }
}
//=============================================================================
```

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.

Assignment 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.

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`.

Assignment 11 *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?.*

Assignment 12 *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
```

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.

Assignment 13 *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.

Assignment 14 *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. Google 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.

Assignment 15 *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*

- *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 the cipher suite. This can be achieved by the* `setEnabledCipherSuites` *method of the SSLSocket.*

Now activate client authentication.

Assignment 16 *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 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

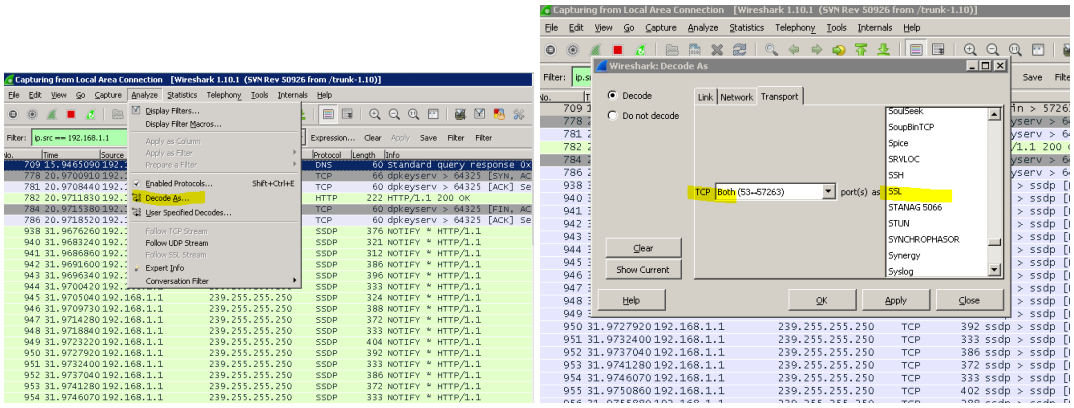


Figure 3: a) Analyze->Decode As, b) Set TCP and SSL.

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.

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.