PKI and TLS

Project 2, EITF55 Security, 2022

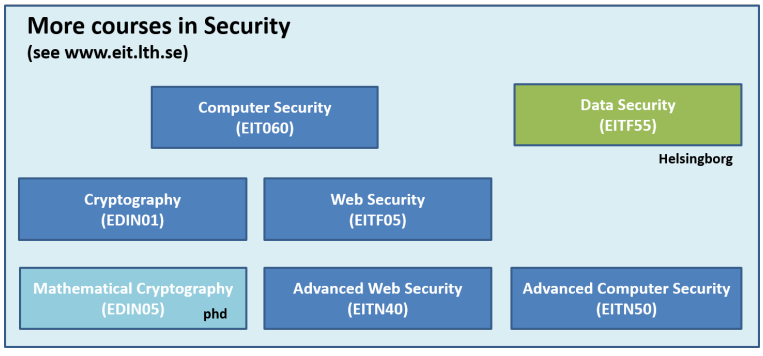
Ben Smeets
Dept. of Electrical and Information Technology, Lund University, Sweden

Last revised by Ben Smeets on
2022–01–06 at 11:56

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 authentication of HTTPS and TLS.
- Intercept and analyse TLS data traffic.



Contents

1 General instructions

This project has many small assignments that guide you through the project work. When writing your report you should use the assignments to structure the contents of your report so it is easy to check that you provided answers to the questions in the assignments.

- 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 your 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 the instructions you will find several commands that are useful for debugging your implementation, configuration and listing the contents of your certificates and 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. Just writing 'I cannot create the subCA certificate' does not tell us that much. We need/want more details before we can help you.

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 as appendix in the report.
3	Code of your Server and Client (as source/text file, not pdf) using server authentication only.
4 5	Code of your Server and Client (as source/text file, not pdf) using server and Client authentication AND specific cipher suite selection.
6	Dump of wireshark logs in an appendix of your report

2 Introduction

TLS is one of the most commonly used secure communication protocols. Web services use HTTPS which is HTTP over TLS. Every modern engineer working with data communication, automatisation, embedded systems, and web design should know how to setup and use TLS or HTTPS. 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.

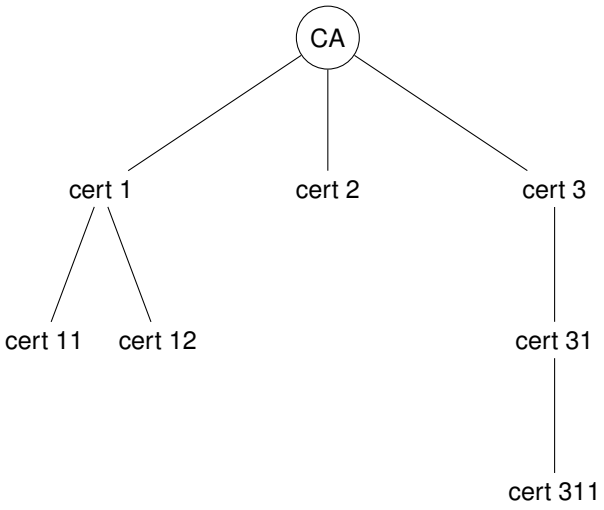


Figure 1: Example of PKI: tree organization with CA and other certificates.

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 other 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.

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

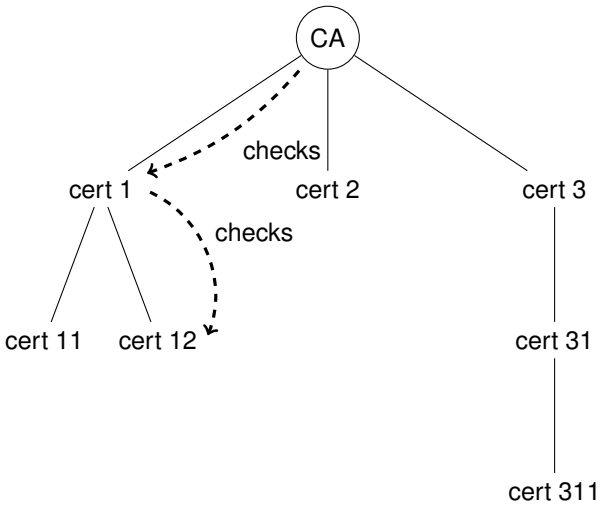


Figure 2: Complete certificate chain in PKI tree.

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 use and should verify. 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 place 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 subCA 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.

2.1 Project PKI

If the key of the CA is lost or misused this is devastating to the PKI and therefore in real-life systems server and client certificates are not created by the CA it itself. Instead the issuing is done via a subCA. This subCA is in the PKI tree a child of the CA and has its certificate signed by the CA. Thus the CA key has only to be used to issue the subCA certificate and, in real-life, the revocation orders that would

invalidate a subCA and all the certificates that subCA issued. That is a more safer setup than using the CA directly for the issuing of server and client certificates. Server and client certificates are often referred to as "end-entity" certificates as they occur at the end of the chain.

To generate a PKI tree for a server certificate we have to augment the previous steps as follows:

1. Generate RSA key pair for CA
2. Construct a (self-signed) CA certificate
3. Generate RSA key pair for a subCA
4. Request the CA issue a subCA certificate
5. Generate a RSA key pair for the server
6. Request subCA to issue a certificate for the server public key
7. Server stores its private key in its KeyStore
8. 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, as in our case, a subCA 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.

To make our project PKI even a bit more realistic we will use two subCAs. One subCA, referred to as the "subCA wolves", is used to issue certificates for the wolves' server and its clients. The other subCA, referred to as the "subCA sheep", is used for the sheep's server and its clients. Figure 3 illustrates our PKI that we use in this project.

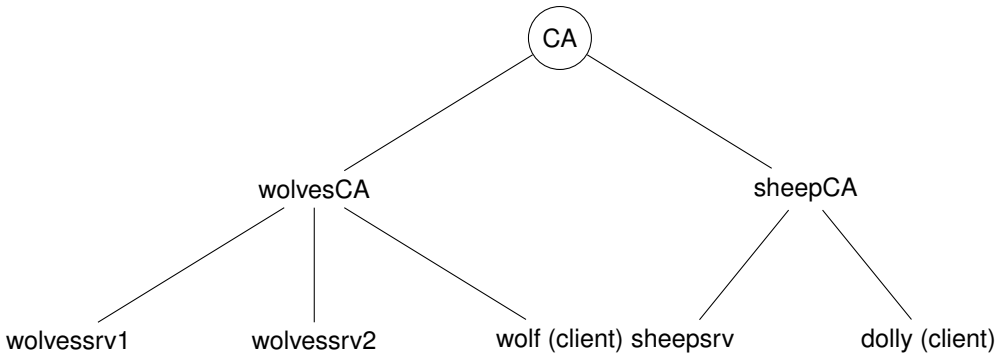


Figure 3: Project PKI with CA and two intermediate subCAs as issuing CAs which issues certificates to servers and clients.

3 Setting up the PKI

3.1 Before we start

In the subsequent sections of the project assignment you will be guided through the steps mentioned in the previous section to create the keys and certificates 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 1: The check of the certificate signatures by a certificate chain is the most crucial step of verifying a certificate of a server or client that always must be preformed. But also the validity dates written into the certificates need to be checked as well as attributes like 'KeyUsage' which tell for what purpose the key can be used. But a certificate contains in many cases more information that can be used when verifying the certificate. This verification task is up to the TLS stack as well as the application, in our case the TLS server or TLS client, to perform. Only the application knows what the purpose is of the certificate which makes the task of the verification a shared responsibility of the TLS stack and the application. Most common TLS stacks, like openSSL, follow the TLS and PKI RFC specifications which result in that the TLS stack will

- check certificate signatures in certificate chain
- check validity
- check keyUsage

What makes a certificate chain is not 100% clearly defined by the standards. The most common characterization is that a certificate chain is the sequence of certificates starting with the entity in question (e.g. server or client) up to the root CA. However is certain implementations the chain may stop at any issuing CA even if it is not the root CA. This is also referred to as *partial chain verification*.

If the certificates contain fields (called attributes) that are marked as critical and the TLS stack cannot handle these themselves, the TLS stack will try during the handshake to use the application to resolve this (e.g by using callbacks) and if that fails the usual procedure is to abort the TLS handshake.

Note 2: When a server or client certificate is validated as being correct there might still be certificate attributes that the server or client can use but that is highly application specific. A very common attribute in a server certificate is the so-called SAN-list. SAN stands for Subject Alternative Name and the SAN entries, for example, can designate the DNS names, or IP addresses for which the certificate is valid. That is via a SAN list you can create a server certificate that works for a server that serves `server1.wolves.com` respectively works for the server at `server1.sheep.com`. Then the SAN entry contains both `server1.wolves.com` and `server1.sheep.com`. However it is not always clear that these entries in a certificate are checked in a client implementation. So one must be careful before relying on the use of these entries. The HTTPS specification, [1] is demanding that HTTPS implementations should check DNS names if they appear in the SAN list. In this project we use socket connections and there many additional verifications of https are not carried out (you have to add them programmatically!). The lack of such checks in TLS may have import security consequences that you should be aware of.

In the below you should create an number of files and directories. Follow those instructions, failing to do so might lead to problems. You find many examples in the internet how you can create CA and server certificates but very often they do not cover PKI setups with an intermediate subCA and the use of SAN entries. If you use another approach to create certificates then is described here you are basically on your own and we most likely cannot or have the time to debug you procedure. Our approach may not be the best but it has been verified to work.

We use OpenSSL. The OpenSSL certificate commands use the `openssl.cnf` configuration file. The default file is almost good for us but in the next sections we need to make a few changes. For example to make the OpenSSL create certificates using so-called PrintableString encoding instead of UTF8

so they play together with the java keytool program. Instead of making the changes in the default file we make the changes in a copy of this file and use this modified file instead. The semantics of the openssl.cnf is somewhat complex. The file is split into sections where each section is tagged by a `[ a-tag ]` expression like `[ ca ]`.

Assignment 1 *Download the openssl.cnf file from the project 2 list of resource files. Open this file in your editor and look*

- *for the section* `[ CA_default ]` *and check if* `copy_extensions = copy` *is uncommented*
- *and that in section* `[ req ]` *we have set* `string_mask = pkix`

We will pass this file to the OpenSSL commands via the argument `-config openssl.cnf`.

3.2 CA certificates

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 2 *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. In the report you should mention the output of the command* `openssl version`.

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 use 4096 bit RSA keys for the CAs and 2048 bit RSA keys for the server and clients and the CA certificates 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 you can type (wait until you get the task to do this!)

```
openssl genrsa -aes128 -out ca.key 4096
```

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 ca.key -days 3560 -config openssl.cnf -out ca.pem
Enter pass phrase for ca.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]:  LTH
Organizational Unit Name (eg, section) []:.
Common Name (e.g. server FQDN or YOUR name) []: ANIMALS-CA
Email Address []: ca@animals.com
```

Note that we left the `Organizational Unit Name` empty. It is important that you use files names for the root CA as shown.

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

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

Assignment 3 *Create a directory where you can store your keys and intermediate results and open a command prompt in that directory. We called this directory 'tls' but the name is not important. In the 'tls' directory you have to create a new directory called 'demoCA'. 'cd' into the 'demoCA' directory and create two files called* `index.txt` *and* `serial` *and create the directory newcerts. The file serial should contain an integer number, for example 1. For example, in Windows you can do*

```
echo 01 > serial
del index.txt
notepad index.txt
mkdir newcerts
dir
 Directory of C:\Projects\tls\demoCA

2022-01-03  20:22    <DIR>          .
2022-01-03  20:22    <DIR>          ..
2022-01-03  20:22                 0 index.txt
2022-01-03  20:15    <DIR>          newcerts
2022-01-03  20:18                 4 serial
               2 File(s)             31 bytes
```

Note. The index file shall be empty (zero size) otherwise you will get problems later.

Generate in the 'demoCA' directory a 4096 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?*
7. *Include a printout of the subject name in root CA certificate in your report.*

Before calling help: *You should create your working directory not as a subdirectory of a system or OpenSSL installation directory. This might give problems with permissions. List the place from which you run your OpenSSL commands.*

Next we create the wolves' and sheep subCAs. The key can be created as before but the certificates has to be signed by the root CA. So each subCA has to generate certificate sign request. To do that

we use the generated subCA key to generate a csr (certificate sign request) message using OpenSSL.

Assignment 4 *Move out of the demoCA directory back into the 'tls' directory. See to it that you have also here the openssl.cnf file. Create the wolves and shéep subCAs and print their subject string in the report.*

To generate the subCA RSA key type

```
openssl genrsa -out wolvesCA.key 4096
```

Note this time, we omitted the "-aes128" argument in the command so there will be no password protection of the use of subCA private RSA key. This convenient here but of course really a bad thing in practice!

Next we create a certificate sign request (csr) message by typing

```
openssl req -new -key wolvesCA.key -config openssl.cnf -out wolfvesca.csr
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]:  LTH
Organizational Unit Name (eg, section) []:.
Common Name (e.g. server FQDN or YOUR name) []: wolvesCA
Email Address []: wolvesca@animals.com
```

This creates a csr message `wolvesca.csr` *(in pem format) which you can inspect by typing* `openssl req -in wolvesca.csr -text`. *Next we use the root CA to process the csr by the following command*

```
openssl x509 -req -days 365 -in wolvesca.csr -CA ./demoCA/ca.pem -CAkey ./demoCA/ca.key
       -CAcreateserial -next_serial -out wolvesca.crt -extfile openssl.cnf -extensions v3_ca
```

The output of this command is the subCA's certificate in the file `wolvesca.crt`.

3.3 Server and Client Certificate

We could continue to use OpenSSL to generate the Server and Client certificates. However we also use the Java `keytool`. The `keytool` command is a versatile command that can be used for many purposes. See the "How To" section in this document. We use it for creating the Keystores and Truststores used by the Java TLS stack.

Below you see an example how to generate a server certificate for the wolves' server. In some setups the server's IP address or dns name, like `wolvessrv.animals.com`, must be specified in the CN for TLS to accept it as server certificate for your server. However one should not use the CN for that any longer and instead use the SubjectAltName (san) attribute.

```
keytool -genkey -dname "CN=wolvessrv1,O=LTH,L=LUND,ST=SCANIA,C=SE" -alias wolfserver1
          -keystore wolvessrv1.jks -keyalg RSA -sigalg Sha256withRSA -storepass 123456
```

However, the entry `CN=wolvessrv1` can be `localhost` or the FQDN of the server e.g. `wolvessrv1.animals.com`.

In the above call to keytool we also created a keystore and places the private key in this keystore and used "123456" as password. This password we need from now on to make use of this keystore. For convenience we use, aj aj aj, this password for all our keystores. It is useful to call the keystore file `wolvessrv1.jks` in case we work on the server keys and certificates for the wolf's server. You can list the entries in the keystore by issuing the command `keytool -list -keystore wolvessrv1.jks -storepass 123456`.

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 the created key. First we must generate a certificate sign request that we send to an issuing CA. This sign request can be generated by keytool and we give it file file name 'server.csr'. Next one of our subCAs has to process this request. Here we use again OpenSSL. Below we give the two steps[1]

```
# csr creation
keytool -certreq -alias wolfserver1 -ext san=DNS:localhost
     -keystore wolvessrv1.jks -file server.csr -storepass 123456
# ca issuing
openssl ca -days 60 -create_serial -keyfile wolvesCA.key -cert wolvesCA.crt
     -extensions v3_req -config openssl.cnf -extfile server_v3.txt
     -notext -out wolvessrv1.crt -infiles server.csr
```

When using keytool to create the csr file we write into the request the desired CN and the desired SubjectAltName (as SAN). The SAN can be one of the following

- DNS FQDN entry: e.g. `DNS:wolfsrv.wolves.com`
- IP address entry: e.g. `IP:192.168.1.23`
- email address entry: e.g. `EMAIL:wolfsrv@wolves.com`

The OpenSSL `ca` command deserves some explanation. In particular the arguments `-extfile server_v3.txt` and `-create_serial`. The former instructs OpenSSL to read data from the file `server_v3.txt` that contains Certificate version 3 extensions and the latter secures we set 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 in the file:

```
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. Students 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

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

[1] The commands are split over two or three lines. You should enter the command on one single line at a time.

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 5 *Prepare a* `server_v3.txt` *and a* `client_v3.txt` *file containing the proper extensions as shown above. Generate RSA keys of size 2048 bits. The certificate that you generate should by valid for 60 days and should use SHA256 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. *Investigate the server certificate and identify the "X509v3 Authority Key Identifier". Where have you seen this value before?*
3. *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?*
4. *Make prints of the certificates and add them as appendices to your report.*
5. *Generate also an server certificate that will expire after one day. We will use it later to test if the TLS client really checks the expiry data.*
6. *Does the subCA 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 when asking for help.It is useful to make script/bat files for running the commands or at least have a text file where you write down the commands and use them for copy-and-paste into the command line.*

Note: when using the `ca` command to create certificates from the csr-files is handy as the content of the csr is partially copied into the certificate. But the use of this command has a downside. When a certificate has been created this command registers the certificates in the newcerts directory of the demoCA directory and updates the index.txt and serial files accordingly. The CA has per `openssl.cnf` definitions the policy to require unique entries `CN=, (OU=,) O=, L=, ST=, C=`. If an issued certificate is, for some reason, wrong, you cannot issue a new certificate with the same `CN=, (OU=,) O=, L=, ST=, C=` unless you manually remove them from the newcerts directory and correct the index.txt file. This is not really difficult but better is to avoid accepting the a wrong certificate into the demoCA PKI "database" in the first place.

Assignment 6 *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!*

3.4 Populating our KeyStore and TrustStore

You should by now have two keystores: e.g. the wolf.jks file respectively the wolvessrv1.jks file containing the client key and server key, respectively. You may have used other names. We add the certificates that we generated to these keystores. Adding a certificate, say cert.crt, to the keystore.jks having a key with alias `youralias` can be done by the following keytool command

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

However, if you try to import the issued server (or client) certificate into its corresponding keystore you get an error that there is no chain. That is the keytool cannot verify the issued certificate. So before

importing the issued certificate of, a server say, we create a chain as follows

```
copy wolvesca.crt chain.crt
type wolvessrv1.crt >> chain.crt
```

So we put first the subCA certificate and append the issued server certificate. After that we can do the import

```
keytool -noprompt -importcert -file chain.crt
          -keystore  wolvessrv1.jks -storepass 123456 -alias wolfserver1
```

Note the use of -noprompt option to avoid having to answer questions by the keytool.

Assignment 7 *Note that we give the certificate a friendly alias name: "wolfserver1". 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 (using `(keytool -list -v ...)` *their contents and add a printout in the appendix of your report.*

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

Assignment 8 *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 test.crt -keystore test.jks -alias other
```

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 `wolvessrv1` *as alias.*

Note: To do this task we on purpose generate a new key pair 'test' in order to avoid mixing up things.

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 demoCA/ca.pem -alias rootCA -trustcacerts -keystore truststore,jks -storepass
```

Normally a Java runtime 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. For example, to display the content of this keystore on a 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 eitf55ca -file demoCA/ca.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. HOWEVER, We will **NOT** put our project CA certificate in the default TrustStore. This to avoid introducing problems with your usual Java Runtime environment.

Assignment 9 *Answer the following questions.*
*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 by ordinary user (accounts)?

Assignment 10 *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 11 *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 [3], see also [4]. 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 java.security.cert.Certificate;
import java.security.cert.X509Certificate;
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 {
        // TrustStore
        /* char[] passphrase_ts = "654321".toCharArray();
        KeyStore ts = KeyStore.getInstance("JKS");
        ts.load(new FileInputStream("truststore.jks"), passphrase_ts);
        TrustManagerFactory tmf = TrustManagerFactory.getInstance("SunX509");
        tmf.init(ts); */
        TrustManager[] trustManagers = null; //tmf.getTrustManagers();

        // set up key manager to do server authentication
        KeyManagerFactory kmf;
        KeyStore ks;
        // First we need to load a keystore
        char[] passphrase = "123456".toCharArray();
        ks = KeyStore.getInstance("JKS");
        ks.load(new FileInputStream("srv-keystore.jks"), passphrase);
        // Initialize a KeyManagerFactory with the KeyStore
        kmf = KeyManagerFactory.getInstance("SunX509");
        kmf.init(ks, passphrase);
        KeyManager[] keyManagers = kmf.getKeyManagers();

        // Create an SSLContext to run TLS and initialize it with
        SSLContext context = SSLContext.getInstance("TLSv1.3");
        context.init(keyManagers, trustManagers, 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);
        System.out.println("TLS server running");
        SSLSocket s = (SSLSocket)ss.accept();
        //s.setNeedClientAuth(true);
        // below works only when client is authenticated
        //SSLSession session = ((SSLSocket) s).getSession();
        //Certificate[] cchain = session.getPeerCertificates();
        //System.out.println("The Certificates used by the client");
        //for (int i = 0; i < cchain.length; i++) {
        //    System.out.println(((X509Certificate) cchain[i]).getSubjectDN());
        //};
        // 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.
        System.out.println("TLS server waiting for input");
        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 java.security.cert.Certificate;
import java.security.cert.X509Certificate;
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 = "654321".toCharArray();
        KeyStore ts = KeyStore.getInstance("JKS");
        ts.load(new FileInputStream("cl-truststore.jks"), passphrase_ts);
        TrustManagerFactory tmf = TrustManagerFactory.getInstance("SunX509");
        tmf.init(ts);
        TrustManager[] trustManagers = tmf.getTrustManagers();

        SSLContext context = SSLContext.getInstance("TLSv1.3");

        KeyManager[] keyManagers = null;
        context.init(keyManagers, trustManagers, new SecureRandom());

        SSLSocketFactory ssf = context.getSocketFactory();
        System.out.println("TLS client running");
        SSLSocket s = (SSLSocket) ssf.createSocket(HOST, PORT);
        //s.socket.setEnabledCipherSuites(cipher_suites);
        OutputStream toserver = s.getOutputStream();
        SSLSession session = s.getSession();
        Certificate[] cchain = session.getPeerCertificates();
        System.out.println("The Certificates used by the server");
        for (int i = 0; i < cchain.length; i++) {
            System.out.println(((X509Certificate) cchain[i]).getSubjectDN());
        };
        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 client accepts the hostname information from the server in its certificate. This is so per default.

Assignment 12 *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. Start the server first and then the client.*

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 when you use a code developer tool like Eclipe 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 tlsclient.jar`. Of course you can just compile the java program `javac tlsclient.java` and then run it with `java tlsclient`.

Assignment 13 *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?.*

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 subca certificate. Hence the chain in the keystore is not complete. This works for our Java TLS because we have the root CA certificate in the trust store. This shows that we do not need the full chain in the keystore. This a Java specific behaviour!

Assignment 14 *Now create a new truststore,* `truststoreWolf.jks`*, for the client and import into it only the certificate of the wolvesCA (i.e. a subCA). Test if the tlsclient still is able to connect to the server or not. Explain what happens.*

Note: you might have already noticed that you have to restart the tlsserver every time the client stops or is restarted. That is normal.

In general, when using other SW stacks that implement TLS, the requirement of the root CA 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 the client you have to add the keystore and use it in the secure socket. This is similar to how this is done in the server. Instead of setting up the keystores within the programs, 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
```

Assignment 15 *Make the required modification in the TLS server and client code to have mutual TLS. Use the 'wolf' client certificate in the TLS client. Start the programs and explain what happens.*

One might wonder what happens if we create client credentials under our sheep subCA and use these instead of the wolf credentials which is an end entity under the wolfCA like the TLS server.

Assignment 16 *Create the client credentials (keystore and certificates) for a client 'dolly' under the sheepCA (subCA of the root CA) and configure you test tlsclient to connect to the wolf TLS server. Explain your findings.*

Assignment 17 *Repeat the previous task but use now the truststoreWolf.jks at the server Explain your findings.*

Assignment 18 *Restore the truststore of the server to the one that only contains the root CA. Replace the client truststore with the truststoreWolf.jks Test the client and server interaction. Explain your findings.*

5 TLS sockets and https

The experiments we have done use secure sockets that use TLS. On the internet, when using a browser to contact a secure server, you use the HTTPS protocol. Are secure sockets and HTTPS the same you might wonder. The simple 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 of TLS and as said before HTTPS is, for example, more restrictive how certificates are to be used.

Writing a real HTTPS server is too much for this course and even the configuration of such one like Apache or NGINX is a bit too much. Instead you can download a simple demo HTTPS server from the resources for this project. You configure it the same way as the tlsserver we used before. This demo web server when running on your local machine, implements a simple web resource `https://localhost:8043/test` that you can contact via a normal browser. Note the port number 8043.

Assignment 19 *Download the project HTTPS server called* `webserver.java` *and configure it with the wolf server credentials into its keystore. Start this webserver but now you use an browse(Firefox, Safari, Edge, or Chrome) to connect to your server, e.g. by using the url* `https://localhost:8043/test`. *Explain what happens.*

Note: If you try the above with our ordinary TLS server, the server may crash which is natural because a browser is not a type of client our TLS server can fully handle.

Assignment 20 *Add your CA certificate as trusted root certificate to your browser. Exit (all running instances) your browser and restart the broweser 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://support.securly.com/hc/en-us/articles/360008547993-How-do-I-Install-Securly-s-SSL-Certificate-in-Firefox-on-Windows-`

Assignment 21 *Create a new server certificate,* `wolvessrv2`*, under the wolvesCA and set the SAN field of the certificate as* `san=DNS:apes.animals.com`. *Restart your browser and repeat the experiment. Explain your findings.*

Assignment 22 *This assignment is just to check that you did all the required experiments and have them reported. Check the table*

task	*server type*	*client certificate*	*client truststore*	*server certificate*	*server truststore*	*connection works?*
14	*tlsserver*	*-*	*truststore*	*wolvessrv1*	*truststore*	*yes,no*
15	*tlsserver*	*wolf*	*„*	*„*	*truststore*	*yes,no*
16	*tlsserver*	*dolly*	*„*	*„*	*truststore*	*yes,no*
17	*tlsserver*	*dolly*	*„*	*„*	*truststoreWolf*	*yes,no*
18	*tlsserver*	*-*	*truststoreWolf*	*„*	*truststore*	*yes,no*
19	*webserver*	*-*	*browser default*	*wolvessrv1*	*truststore*	*yes,no*
20	*webserver*	*-*	*add your root*	*wolvessrv1*	*truststore*	*yes,no*
21	*webserver*	*-*	*„*	*wolvessrv2*	*truststore*	*yes,no*

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 23 *Let us first consider the server authentication only case. However, we force now the use of TLS1.2 instead of TLS1.3. This you can do by having*

```
SSLContext context = SSLContext.getInstance("TLSv1.2");
```

in the server and client code.

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 24 *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.*

Assignment 25 *Repeat the above (mutual TLS) with TLS set to be TLS1.3. What are the differences?*

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

Java is likely already present on your machine. If not you find instructions how to install Java on the web. With it follows Keytool. If you want more information on keytool you should consult [5]. Below we sample some of the information from `http://www.sslshopper.com/article-most-common-java-keytool-keystore-co` `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 versions, i.e. from `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 [6] and [7]

```
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 == 8043
```

Wireshark does understand the 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 TLS and port, see figures a) and b) in Figure 4.

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
```

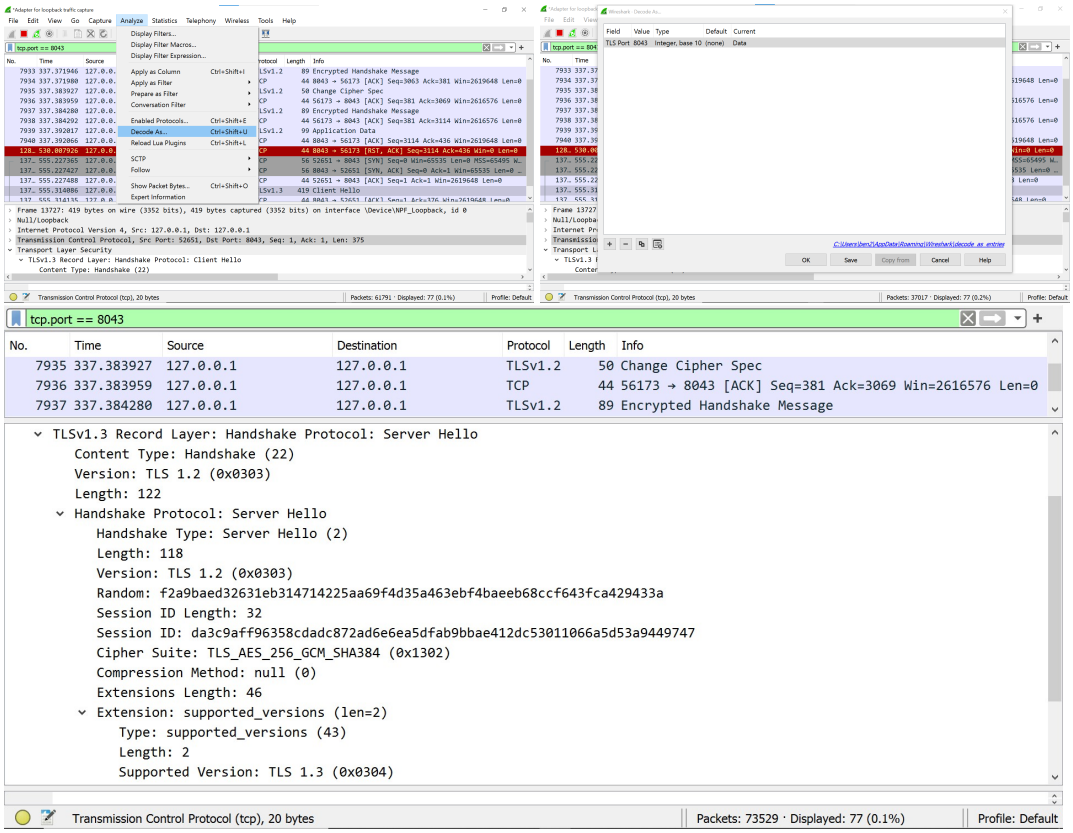


Figure 4: a) Analyze->Decode As, b) Set TLS and port, c) TLS1.3 server hello.

```
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 want help, we ask you to send us

- your code, you should send both client and server code (source code!, not jar or binary)
- the keystore(s) and truststore(s) your are using with their respective passwords
- your CA and subCA certificates and keys and their respective passwords (zip them together)

so we can try to repeat your problem at our side to replicate your setup to start with.

References

[1] HTTPS specification, IETF RFC2818: https://datatracker.ietf.org/doc/html/rfc2818

[2] 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.

[3] 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.

[4] 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.

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

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

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

9 appendix

9.1 used windows commandline commands

```
mkdir demoCA
cd demoCA

openssl genrsa -aes128 -out ca.key 4096
openssl req -x509 -new -key ca.key -days 3560 -config openssl.cnf -out ca.pem
openssl x509 -text -in ca.pem

echo 01 > serial
del index.txt
notepad index.txt
mkdir newcerts
cd ..
-------------
```

```
openssl genrsa -out wolvesCA.key 4096
openssl req -new -key wolvesCA.key -config openssl.cnf -out wolfvesca.csr
openssl x509 -req -days 365 -in wolvesca.csr -CA ./demoCA/ca.pem -CAkey ./demoCA/ca.key
       -CAcreateserial -next_serial -out wolvesca.crt -extfile openssl.cnf -extensions v3_ca
------------
keytool -genkey -dname "CN=wolvessrv1,O=LTH,L=LUND,ST=SCANIA,C=SE" -alias wolfserver1
          -keystore wolvessrv1.jks -keyalg RSA -sigalg Sha256withRSA -storepass 123456
# csr creation
keytool -certreq -alias wolfserver1 -ext san=DNS:localhost
     -keystore wolvessrv1.jks -file server.csr -storepass 123456
# ca issuing
openssl ca -days 60 -create_serial -keyfile wolvesCA.key -cert wolvesCA.crt
     -extensions v3_req -config openssl.cnf -extfile server_v3.txt
     -notext -out wolvessrv1.crt -infiles server.csr

copy wolvesca.crt chain.crt
type wolvessrv1.crt >> chain.crt
keytool -noprompt -importcert -file chain.crt
          -keystore  wolvessrv1.jks -storepass 123456 -alias wolfserver1
--------------------
keytool -importcert -file test.crt -keystore test.jks -alias other
-------------------
keytool -import -file demoCA/ca.pem -alias rootCA -trustcacerts -keystore truststore.jks -storepass
keytool -import -file wolvesCA.crt -alias subCA -trustcacerts -keystore truststoreWolf.jks -storepas
------------------
copy wolvessrv1,jks srv-keystore.jks
copy truststore.jks cl-truststore.jks
copy wolf.jks cl-keystore.jks
```