Security Lab – TLS Tester

VMware

This lab can be done with the **Ubuntu image**. The remainder of this document assumes you are working with this image.

You can do this lab also on your own system, provided you have an IDE and a current version of Java installed.

1 Introduction

In this lab, you'll develop a Java program that acts as a TLS[1] client and that can be used to analyze the configuration of any TLS server. The goal of this lab is to get familiar with the TLS functionality offered by Java and that you can apply this functionality correctly.

2 Basis for this Lab

As a basis, you get a Java project that contains two classes: a skeleton class for the program to be developed (*TLSTester*) and the helper class *CustomX509TrustManager* (see section 6). Do the following to set up and use this basis:

- Download *TLSTester.zip* from OLAT.
- Move the file to an appropriate location (e.g., into a directory *securitylabs* in the home directory */home/user*).
- Unzip the file. The resulting directory *TLSTester* contains a Java project based on Maven. This should be importable in any modern IDE. The remainder assumes you are using *NetBeans*, which is installed on the Ubuntu image.
- Start *NetBeans* and open the project.
- To build the project, right-click *TLSTester* in the *Projects* tab and select *Clean and Build*.
    - Whenever you do some changes, the project is rebuilt automatically, so it's usually not necessary to use *Clean and Build* again. However, if «something doesn't work as it should», do again a *Clean and Build*.
    - The executable code is placed in directory *TLSTester/target/classes*.
- In general, run the program on the command line (not in the IDE). Do this in a terminal as user *user*.

3 Task

You have to develop a program *TLSTester* in Java. The program can communicate with any TLS server and basically generates the following output:

- Which **SSL/TLS versions** are supported by the server and which are not (from SSL 3 to TLS 1.3).
- The **complete certificate chain** of the server, from the root certificate of the certification authority (CA) to the certificate of the server.
- The **secure and insecure cipher suites** that are supported by the server, depending on the used SSL/TLS version.

[1] When the term TLS (Transport Layer Security) is used in this lab, then this also includes the predecessor Protocol Secure Sockets Layer (SSL), so TLS is used to identify the entire SSL/TLS protocol family.

As a result, you get a small but quite useful tool to test the TLS configuration of arbitrary TLS servers.

4 Specifications

In the following, the detailed specifications of the program *TLSTester* are provided. Make sure that you follow the specifications as closely as possible.

- The program is a command line program that can be run in a terminal. The output is also written to the terminal (standard output).
- The program is used as follows with 2 or 4 parameters (assuming you are in directory *TLSTester/target/classes*):

  ```
  java ch.zhaw.securitylab.tlstester.TLSTester
       host port {truststore password}
  ```

  The parameters have the following meaning

  - *host* (required): The host name or the IP address of the server to test.
  - *port* (required): The port number of the server.
  - *truststore* (optional): The file name (or file path) of a truststore. *TLSTester* uses a truststore that contains trustworthy certificates (typically root CA certificates) that are used to verify the certificate(s) of the server and to build the certificate chain. If this parameter is not used, then the default Java truststore (located at *$JAVA_HOME/lib/security/cacerts*) is used. If the parameter is used, then the specified truststore file is used instead.
  - *password* (optional): The password to access the truststore if parameter *truststore* is used.
- Testing the supported SSL/TLS versions must include SSL 3, TLS 1.0, TLS 1.1, TLS 1.2 and TLS 1.3. These are all versions that are supported in the current Java version.
- The certificate chain should be included in the output from the «very top» (self-signed root CA certificate) to the «very bottom» (server certificate). For each certificate, the output should include subject, issuer, validity period, the algorithm used for the signature of the certificate (consisting of a hash algorithm and a public key algorithm), and the length of the RSA or EC public key (in bits) that is included in the certificate (with RSA, the key length corresponds to the length of the modulus). For instance, in the case of the web servers www.zhaw.ch:443 and www.google.com:443, the certificates of the servers should be included in the output as follows (note that the certificates may have changed in the meantime):

  Subject: CN=www.zhaw.ch,O=Zuercher Hochschule fuer Angewandte Wissenschaften,L=Winter thur,ST=Zuerich,C=CH,2.5.4.5=#130a323030372d30342d3032,2.5.4.15=#0c11476f7665726e6d 656e7420456e74697479,1.3.6.1.4.1.311.60.2.1.2=#0c075a756572696368,1.3.6.1.4.1.311.60.2.1.3 =#13024348
  Issuer: CN=QuoVadis EV SSL ICA G1,O=QuoVadis Limited,C=BM
  Validity: Mon Dec 18 16:51:36 CET 2017 - Wed Dec 18 17:01:00 CET 2019
  Algorithm: SHA256withRSA
  RSA public key length: 2048 bits

  Subject: CN=www.google.com,O=Google LLC,L=Mountain View,ST=California,C=US
  Issuer: CN=Google Internet Authority G3,O=Google Trust Services,C=US
  Validity: Tue Jun 18 10:22:58 CEST 2019 - Tue Sep 10 10:15:00 CEST 2019
  Algorithm: SHA256withRSA
  EC public key length: 256 bits

  As you can see, the certificate of www.zhaw.ch uses an RSA public key, while the one of www.google.com uses an EC public key. Almost all certificates used these days use RSA public keys, but there are a few servers that use EC certificates.
- If the used truststore does not contain the necessary certificates to construct the entire certificate chain, only the certificate(s) received from the server should be included in the output.

- You shouldn't hard-code any cipher suites in your program. Instead, your program should first learn what cipher suites are supported by Java and then it should test which of them are supported by the server. In addition, the program should determine which cipher suites are secure and which are insecure. For a cipher suite to be secure, the following should apply:
  - Length of the symmetric key is at least 128 bits.
  - The server must authenticate itself.
  - RC4 and 3DES should be considered as insecure ciphers.
  - MD5 should be considered as an insecure hash function.

Warning: Please don't test external servers right away. Start your first attempts with ZHAW servers such as www.zhaw.ch:443, intra.zhaw.ch:443 or mail.zhaw.ch:993. Take special care that your program does not flood the server with large amounts of messages (but establishing 100 or so TLS connections in a few seconds is certainly no problem). Start testing external servers once you are convinced your program works correctly.

5 Examples

The first example shows a test of www.google.com:443. Comments that are included with // are additional explanations and are not part of the actual program output. Your program does not need to produce exactly the same output, but it should contain the same information in a well-readable format (in particular the supported SSL/TLS versions, the certificate chain, and the secure and insecure cipher suites for each supported SSL/TLS version). Note that the configuration (e.g., the certificates or supported cipher suites) of www.google.com may have changed in the meantime, which may result in a (slightly) different output (this may also be the case with the further examples illustrated in this section).

```
$ java ch.zhaw.securitylab.tlstester.TLSTester www.google.com 443

// The first three lines are information messages that describe how many trusted certificates are read from the
// truststore (the default one in this case, which contains 134 certificates with the used Java version), whether the
// server can be reached at all, and whether the root CA that issued the certificate of the server is
// trusted (i.e., whether the root CA certificate is included in the trust store).
Use default truststore with 134 certificates

Check connectivity to www.google.com:443 - OK

The root CA is trusted

// The following lines show which SSL/TLS versions are supported by the server and which are not
// (from SSL 3 to TLS 1.3).
Supported SSL/TLS protocol versions:
SSLv3     No
TLSv1     Yes
TLSv1.1   Yes
TLSv1.2   Yes
TLSv1.3   Yes

// Print information about the certificates. The first certificate is from the local default truststore, The other two
// were received by the server during the TLS handshake. The first two certificates contain an RSA public key
// and the third contains an EC public key.
Information about certificates from www.google.com:443:

3 certificate(s) in chain

Certificate 1:
Subject: CN=GlobalSign,O=GlobalSign,OU=GlobalSign Root CA - R2
Issuer: CN=GlobalSign,O=GlobalSign,OU=GlobalSign Root CA - R2
Validity: Fri Dec 15 09:00:00 CET 2006 - Wed Dec 15 09:00:00 CET 2021
```

```
Algorithm: SHA1withRSA
RSA public key length: 2048 bits

Certificate 2:
Subject: CN=Google Internet Authority G3,O=Google Trust Services,C=US
Issuer: CN=GlobalSign,O=GlobalSign,OU=GlobalSign Root CA - R2
Validity: Thu Jun 15 02:00:42 CEST 2017 - Wed Dec 15 01:00:42 CET 2021
Algorithm: SHA256withRSA
RSA public key length: 2048 bits

Certificate 3:
Subject: CN=www.google.com,O=Google LLC,L=Mountain View,ST=California,C=US
Issuer: CN=Google Internet Authority G3,O=Google Trust Services,C=US
Validity: Tue Jun 18 10:22:58 CEST 2019 - Tue Sep 10 10:15:00 CEST 2019
Algorithm: SHA256withRSA
EC public key length: 256 bits
```

// The used version of Java supports 88 cipher suites, which are all tested.

```
Check supported cipher suites (test program supports 88 cipher suites)
..................................................................................
...... DONE, 88 cipher suites using 4 SSL/TLS protocol versions tested
```

// For each SSL/TLS protocol version that is supported by the server (here TLS 1.0, 1.1, 1.2 and 1.3), list the
// secure cipher suites that are supported by the server (here, 10 cipher suites overall).

```
The following SECURE cipher suites are supported by the server:

TLSv1: 6 cipher suites:
    TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA
    TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA
    TLS_RSA_WITH_AES_256_CBC_SHA
    TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA
    TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA
    TLS_RSA_WITH_AES_128_CBC_SHA

TLSv1.1: 6 cipher suites:
    TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA
    TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA
    TLS_RSA_WITH_AES_256_CBC_SHA
    TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA
    TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA
    TLS_RSA_WITH_AES_128_CBC_SHA

TLSv1.2: 4 cipher suites:
    TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384
    TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256
    TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA
    TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA

TLSv1.3: 2 cipher suites:
    TLS_AES_128_GCM_SHA256
    TLS_AES_256_GCM_SHA384

TOTAL UNIQUE SECURE cipher suites: 10
```

// For each SSL/TLS protocol version that is supported by the server (here TLS 1.0, 1.1, 1.2 and 1.3), list the
// insecure cipher suites that are supported by the server (here, 1 cipher suite overall).

```
The following INSECURE cipher suites are supported by the server:

TLSv1: 1 cipher suites:
    SSL_RSA_WITH_3DES_EDE_CBC_SHA

TLSv1.1: 1 cipher suites:
    SSL_RSA_WITH_3DES_EDE_CBC_SHA

TOTAL UNIQUE INSECURE cipher suites: 1
```

The next example shows a test of www.zhaw.ch:443. At the end, this example shows the desired output when no insecure cipher suites are supported.

```
$ java ch.zhaw.securitylab.tlstester.TLSTester www.zhaw.ch 443

Use default truststore with 134 certificates

Check connectivity to www.zhaw.ch:443 - OK

The root CA is trusted

Supported SSL/TLS protocol versions:
SSLv3     No
TLSv1     Yes
TLSv1.1   Yes
TLSv1.2   Yes
TLSv1.3   No

Information about certificates from www.zhaw.ch:443:

3 certificate(s) in chain

Certificate 1:
Subject: CN=QuoVadis Root CA 2,O=QuoVadis Limited,C=BM
Issuer: CN=QuoVadis Root CA 2,O=QuoVadis Limited,C=BM
Validity: Fri Nov 24 19:27:00 CET 2006 - Mon Nov 24 19:23:33 CET 2031
Algorithm: SHA1withRSA
RSA public key length: 4096 bits

Certificate 2:
Subject: CN=QuoVadis EV SSL ICA G1,O=QuoVadis Limited,C=BM
Issuer: CN=QuoVadis Root CA 2,O=QuoVadis Limited,C=BM
Validity: Tue Jan 13 17:42:15 CET 2015 - Mon Jan 13 17:42:15 CET 2025
Algorithm: SHA256withRSA
RSA public key length: 2048 bits

Certificate 3:
Subject: CN=www.zhaw.ch,O=Zuercher Hochschule fuer Angewandte Wissenschaften,
L=Winterthur,ST=Zuerich,C=CH,2.5.4.5=#130a323030372d30342d3032,2.5.4.15=
#0c11476f7665726e6d656e7420456e74697479,1.3.6.1.4.1.311.60.2.1.2=#0c075a7565726963
68,1.3.6.1.4.1.311.60.2.1.3=#13024348
Issuer: CN=QuoVadis EV SSL ICA G1,O=QuoVadis Limited,C=BM
Validity: Mon Dec 18 16:51:36 CET 2017 - Wed Dec 18 17:01:00 CET 2019
Algorithm: SHA256withRSA
RSA public key length: 2048 bits

Check supported cipher suites (test program supports 88 cipher suites)
................................................................................
...... DONE, 88 cipher suites using 3 SSL/TLS protocol versions tested

The following SECURE cipher suites are supported by the server:
```

```
TLSv1: 4 cipher suites:
    TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA
    TLS_RSA_WITH_AES_256_CBC_SHA
    TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA
    TLS_RSA_WITH_AES_128_CBC_SHA

TLSv1.1: 4 cipher suites:
    TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA
    TLS_RSA_WITH_AES_256_CBC_SHA
    TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA
    TLS_RSA_WITH_AES_128_CBC_SHA

TLSv1.2: 12 cipher suites:
    TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
    TLS_RSA_WITH_AES_256_GCM_SHA384
    TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
    TLS_RSA_WITH_AES_128_GCM_SHA256
    TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384
    TLS_RSA_WITH_AES_256_CBC_SHA256
    TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA
    TLS_RSA_WITH_AES_256_CBC_SHA
    TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256
    TLS_RSA_WITH_AES_128_CBC_SHA256
    TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA
    TLS_RSA_WITH_AES_128_CBC_SHA

TOTAL UNIQUE SECURE cipher suites: 12

The following INSECURE cipher suites are supported by the server:

No INSECURE cipher suites are supported by the server
```

The third example shows again a test of www.zhaw.ch:443, but this time, we use an own truststore that contains only the root CA certificate of www.zhaw.ch. The output should basically be the same as in the second example above with the difference that the used truststore contains only one certificate:

```
$ java ch.zhaw.securitylab.tlstester.TLSTester www.zhaw.ch 443 ts_local
secret

Use specified truststore (ts_local) with 1 certificates

Check connectivity to www.zhaw.ch:443 - OK

The root CA is trusted

...
```

The fourth and final example shows again a test of www.google.com:443, this time when using our own truststore that still contains only the root CA certificate of www.zhaw.ch. This means that the root CA that issued the certificate of www.google.com is not included in the used trustsore and consequently, TLSTester should report that it does not trust the root CA. In addition, only the certificates received from the server should be displayed (in this case, two certificates):

```
$ java ch.zhaw.securitylab.tlstester.TLSTester www.google.org 443 ts_local
secret

Use specified truststore (ts_local) with 1 certificates
```

```
Check connectivity to www.google.com:443 - OK

The root CA is not trusted, ignore root CA checks in this test

...

Information about certificates from www.google.com:443:

2 certificate(s) in chain

Certificate 1:
Subject: CN=Google Internet Authority G3,O=Google Trust Services,C=US
Issuer: CN=GlobalSign,O=GlobalSign,OU=GlobalSign Root CA - R2
Validity: Thu Jun 15 02:00:42 CEST 2017 - Wed Dec 15 01:00:42 CET 2021
Algorithm: SHA256withRSA
RSA public key length: 2048 bits

Certificate 2:
Subject: CN=www.google.com,O=Google LLC,L=Mountain View,ST=California,C=US
Issuer: CN=Google Internet Authority G3,O=Google Trust Services,C=US
Validity: Tue Jun 18 10:22:58 CEST 2019 - Tue Sep 10 10:15:00 CEST 2019
Algorithm: SHA256withRSA
EC public key length: 256 bits

...
```

Note that when using the default Java truststore and testing a server that does not own a certificate that was issued by one of the root CA certificates in this truststore (e.g., when using a self-signed certificate for the server), then the behavior should be similar as in the fourth example above in the sense that only the certificate(s) received from the server are included in the output.

6 Hints

In the following, you find some hints that should help you during the implementation. Before reading them, you should understand the JSSE examples discussed in the lecture, because the following hints are primarily meant as additional information to efficiently solve this lab. Further information can be found in the official Java API documentation.

- Per default, SSL 3 is not enabled in Java. To enable it (so you can test whether the server supports this version), the following line is used at the beginning of the main method (it's already included in the code):

  ```
  Security.setProperty("jdk.tls.disabledAlgorithms", "");
  ```

- Basically, you should proceed similar to the «elaborate example» from the lecture and work with *SSLContext*, *TrustManagerFactory*, and *SSLSocketFactory*.

  The *SSLContext* object is created as follows (using *"TLSv1.3"* guarantees that *SSLSocket* objects which are created based on this *SSLContext* support all versions from SSL 3 to TLS 1.3):

  ```
  SSLContext sslContext = SSLContext.getInstance("TLSv1.3");
  ```

  The *TrustManagerFactory* is created as follows:

  ```
  TrustManagerFactory tmf = TrustManagerFactory.getInstance("PKIX");
  ```

  When using the default truststore of Java, you must initialize the *TrustManagerFactory* as follows:

  ```
  tmf.init((KeyStore) null);
  ```

When using your own truststore, the content of the specified truststore file must first be loaded into a *KeyStore* object and this object must then be used to initialize the *TrustManagerFactory* (assume that *truststoreFile* contains the file name (or file path) of the specified truststore):

```
KeyStore truststore = KeyStore.getInstance("PKCS12");
truststore.load(new FileInputStream(truststoreFile),
  password.toCharArray());
tmf.init(truststore);
```

The *SSLContext* can now be initialized with the *TrustManager*s of the *TrustManagerFactory*:

```
sslContext.init(null, tmf.getTrustManagers(), null);
```

Note that the method *getTrustManagers* returns an array of *TrustManager*s. The reason is that if the used truststore contains different types of certificates, then one *TrustManager* per type is returned. In our case, the truststore contains only X.509 certificates and correspondingly, the returned array contains only one *TrustManager* (an *X509TrustManager*).

Finally the *SSLSocketFactory* can be created from *SSLContext*:

```
SSLSocketFactory sslSF =
  (SSLSocketFactory)sslContext.getSocketFactory();
```

This *SSLSocketFactory* can now be used to repeatedly create *SSLSocket* objects for the tests:

```
SSLSocket sslSocket = (SSLSocket)sslSF.createSocket(host, port);
```

Note that creating an *SSLSocket* establishes a TCP connection to the server, but the TLS handshake is not yet performed. This only takes place as soon as (1) *startHandshake()* is called, (2) *getSession()* is called, or (3) something is read from or written to the socket.

- If Java cannot build the certificate chain to a trusted certificate, the TLS handshake terminates with an *SSLHandshakeException*. To test servers that don't use certificates of a trusted CA, this exception must be prevented. This can be done by implementing and using an own *TrustManager* that does not perform any certificate checks. Below, the code of such a *TrustManager* (*CustomX509TrustManager*) is listed. The corresponding class file (*CustomX509TrustManager.java*) is included in the code base for this lab:

```
import javax.net.ssl.X509TrustManager;

/* Custom TrustManager that ignores all certificate errors */
public class CustomX509TrustManager implements X509TrustManager {

  public CustomX509TrustManager() {
  }

  public java.security.cert.X509Certificate[] getAcceptedIssuers() {
    return null;
  }

  public void checkClientTrusted(java.security.cert.X509Certificate[]
    certs, String authType) {
    // Empty as returning without throwing an exception means the
    // check succeeded
  }

  public void checkServerTrusted(java.security.cert.X509Certificate[]
    certs, String authType) {
    // Empty as returning without throwing an exception means the
```

```
      // check succeeded
    }
  }
```

If you want to use this *CustomX509TrustManager*, an instance of it must be used as the second argument when initializing the *SSLContext*. Note that the argument must by of type *TrustManager[]*, which contains one element of type *CustomX509TrustManager* in our case:

```
sslContext.init(null, new TrustManager[] {
                      new CustomX509TrustManager()}, null);
```

The best strategy is that you first use the default truststore or the one provided on the command line and try to perform a TLS handshake:

```
sslSocket.startHandshake();
```

If an *SSLHandshakeException* is thrown, initialize the *SSLContext* again as specified above. This will allow you to generate *SSLSockets* from the *SSLSocketFactory* that won't perform any certificate checks.

- During the TLS handshake, client and server communicate each other the highest SSL/TLS version they support, and they end up using the highest version that is supported by both endpoints (today, that's typically TLS 1.2 or TLS 1.3). To force usage of a specific version (to test which SSL/TLS versions are supported by the server and to test which cipher suites are supported for each supported SSL/TLS version), you have to use the method *setEnabledProtocols()* of *SSLSocket* and pass the desired version (here *version*) as a String array that contains one element:

  ```
  sslSocket.setEnabledProtocols(new String[] {version});
  ```

  The allowed values for the version are: *"SSLv3"*, *"TLSv1"*, *"TLSv1.1"*, *"TLSv1.2"*, *"TLSv1.3"*.

  Once the version has been set, *startHandshake()* can be used. If the TLS handshake can be completed, this means that the version is supported by the server. Otherwise, an exception is thrown.

- You can access the certificates that were received from the other communication endpoint (the server in our case) as follows:

  ```
  SSLSession session = sslSocket.getSession();
  X509Certificate[] certificates =
    (X509Certificate[])session.getPeerCertificates();
  ```

- To access the certificates in the used truststore (that has been used to initialize the *TrustManagerFactory*), you can use the *TrustManager* of the *TrustManagerFactory*:

  ```
  X509TrustManager tm =
    (X509TrustManager) tmf.getTrustManagers()[0];
  X509Certificate[] trustedCerts = tm.getAcceptedIssuers();
  ```

- The class *X509Certificate* provides several methods to get issuer, subject etc. of the certificate.

- To get the cipher suites that are supported by Java as a String array, you can use the method *getSupportedCipherSuites()* of the class *SSLSocket*:

  ```
  String[] cipherSuites = sslSocket.getSupportedCipherSuites();
  ```

- To test whether the server supports a specific cipher suite when using a specific SSL/TLS protocol version, you must specify both the protocol version and the cipher suite that should be offered by the client to the server during the handshake. Setting the protocol version can be done as explained above using *setEnabledProtocols()*. Setting the cipher suite can be done with the method *setEnabledCipherSuite()* of *SSLSocket* and by passing the desired cipher suite (here *cipherSuite*) as a String array that contains one element:

  ```
  sslSocket.setEnabledCipherSuites(new String[] {cipherSuite});
  ```

Once this has been done, *startHandshake()* can be used. If the TLS handshake can be completed, this means that the cipher suite is supported by the server when using the specified SSL/TLS protocol version. Otherwise, an exception is thrown. Overall, you have to test all cipher suites that are supported by Java (see *getSupportedCipherSuites()* above) with each of the protocol versions supported by the server. So if Java supports, e.g., 100 cipher suites and the server supports, e.g., 3 different SSL/TLS protocol versions, this will result in 300 handshake attempts in total.

- To create your own truststore, the *keytool* is used. The following command imports a certificate certificate.cer into a truststore with the name *ts_local* and that uses PKCS #12 format:

  ```
  keytool -importcert -keystore ts_local -storetype PKCS12
  -file certificate.cer -alias servercert
  ```

  The imported certificates can originate from different sources. For instance, it can be a self-signed certificate you generated with *openssl* or a CA certificate you exported from the certificate store of Firefox.

- Assuming a certificate that uses an EC public key is stored in the variable cert, the length of the public key can be determined as follows:

  ```
  ((ECPublicKey) cert.getPublicKey()).getParams().getOrder().bitLength();
  ```

- These days, it's difficult to find publicly reachable servers that still support SSL 3 or that are configured very poorly. Nevertheless, such servers are still available, and the servers listed on https://www.ssllabs.com/ssltest/ under *Recent Worst* may be good candidates.

Lab Points

In this lab, you can get **3 Lab Points**. To get them, you must demonstrate your results to the instructor:

- You get 1 point for demonstrating that when using the default truststore, printing the complete certificate chain works with any TLS server in the Internet that has a trusted certificate. If the server does not have a trusted certificate (i.e., the corresponding root CA certificate is not present in the truststore), only the certificates received from the server should be printed.
- You get 1 point for demonstrating that determining the SSL/TLS versions supported by the server and determining the secure and insecure cipher suites for each protocol version works for any TLS server in the Internet.
- You get 1 point for demonstrating that your program supports the optional two parameters, i.e., that the program is capable of performing the certificate check and printing the entire chain if an own truststore is used and passed to the program as a command line parameter. One possibility to test this is as follows:
  - Use the browser to connect to an HTTPS website and check which root CA certificate is used.
  - Export the root CA certificate from the certificate store of the browser and import it into a dedicated truststore.
  - Pass the file name of the truststore (and the password of the truststore) as command line parameters to the program and show that the entire certificate chain is printed.

In addition, you have to send your source code (*TLSTester.java*) by e-mail to the instructor. Use *SecLab - TLS Tester - group X - name1 name2* as the e-mail subject, corresponding to your group number and the names of the group members.

Plan B: In case you cannot demonstrate your solution in person: Make a video of a terminal that shows three runs of the *TLSTester* program and the generated program outputs:

- In the first run, use any TLS server of your choice and the default Java truststore.

- In the second run, use your own truststore that includes one root CA certificate and use a TLS server that uses a certificate issued by this root CA.
- In the third run, again use your own truststore that includes one root CA certificate (use the same as in the second run) and use a TLS server that uses a certificate that is *NOT* issued by this root CA.

Send the video and your source code (*TLSTester.java*) by e-mail to the instructor. Use *SecLab - TLSTester - group X - name1 name2* as the e-mail subject, corresponding to your group number and the names of the group members. If the video is too big to include in an e-mail, place it somewhere (e.g., SWITCHdrive) and include a link to it in the e-mail.