Security Lab – TLS in Java

VMware

- You will work with the **Ubuntu image**, which you should start in networking-mode **Nat**.

1 Introduction

SSL / TLS (in the following, we use „TLS“ to identify the entire SSL / TLS protocol family) is the dominating protocol to secure the communication in the Internet. As a result, the likelihood that you'll use TLS when developing distributed applications is high.

In this lab, you'll develop a Java program that acts as a TLS client and that can interact with an arbitrary TLS server. The goal of this lab is to get familiar with using the TLS functionality offered by Java and that you can use technologies such as digital certificates and truststores correctly in a Java program.

2 Basis for this Lab

The Ubuntu image contains an Eclipse project that contains two classes: a skeleton class for the program to be developed (TLSTester) and the helper class CustomX509TrustManager (see section 6).

- To start eclipse, open a terminal, change to */home/user/eclipse*, and enter *./eclipse*.
- The project directory is */home/user/workspace/TLSTester*. Two directories *src* and *bin* are used for source and byte code.
- Run the program in a terminal in the *bin* directory as *user*.

3 Task

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

- The **highest TLS version** supported by the server.
- The **complete certificate chain** of the server (from the root certificate of the certification authority (CA) to the certificate of the server) is printed.
- The program tests which cipher suites are supported by the server and a corresponding list – separated in **secure and insecure cipher suites** – is produced.

A positive side effect of this lab is that this program provides you with a small but quite practical tool to test the TLS configuration of arbitrary TLS servers.

4 Specifications

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

- The program can be run in a terminal (on the command line) and the output is also written to the terminal (standard output). If you like, you can build a GUI around this, but providing a command line program is definitely "good enough" (and much better suited if the output should be processed further by another program).
- The program TLSTester is used as follows with 2 or 4 parameters:

```
java TLSTester host port {truststore password}
```

  - *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 truststore contains the trustworthy certificates. If this parameter is not used, the default truststore $JAVA_HOME/jre/lib/security/cacerts is used. If the parameter is used, then the specified file is used as truststore instead.
  - *password* (optional): The password to access the truststore that is specified with the parameter *truststore*.

  Supporting own truststores is not mandatory to get all 3 lab points, but it allows you to get one extra point. Details are found at the end of this document.

- The certificate chain should be printed from the "very top" (self-signed root CA certificate) to the "very bottom" (server certificate). For each certificate you should print subject, issuer, validity period, the algorithm used for signing (consisting of a hash algorithm and a public key algorithm), and – if RSA is used for the signature – the length of the modulus in bits. For instance, in the case of the certificate of www.google.ch:443, the output should look as follows:

  Subject: CN=.google.ch,O=Google Inc,L=Mountain View,ST=California,C=US*
  Issuer: CN=Google Internet Authority,O=Google Inc,C=US
  Validity: Wed Mar 27 14:27:06 CET 2013 - Tue Dec 31 16:58:50 CET 2013
  Algorithm: SHA1withRSA
  Public key length (modulus): 1024 bits

- 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 printed.

- You shouldn't hard-code any cipher suites in your program. Instead, your program should first learn what cipher suites are supported by Java. 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 (3DES is OK as well).
  - The server must authenticate itself.
  - As using MD5 for the MAC computation in TLS is not too security critical, you can decide for yourself whether cipher suites that use MD5 are considered secure or not.

Warning: Please don't test external servers right away. Star your first attempts with ZHAW servers such as 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 the test of von www.google.ch: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 certificate chain und analysis of the cipher suites).

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

// The first four lines are information messages that describe how many trusted certificates are read from the
// truststore (the default one in this case, which contains 155 certificates with the used Java version), whether the
// server can be reached at all, whether the root CA is trusted (i.e. is included in the trust store) and the highest
// TLS version supported by the server.

```
Use default truststore with 155 certificates

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

The root CA is trusted

Highest TLS version supported by server: TLSv1.2
```

// Print information about the certificates. The first certificate is from the local default truststore, The other three
// were received by the server during the TLS handshake

```
Information about certificates from www.google.ch:443:

4 certificate(s) in chain

Certificate 1:
Subject: OU=Equifax Secure Certificate Authority,O=Equifax,C=US
Issuer: OU=Equifax Secure Certificate Authority,O=Equifax,C=US
Validity: Sat Aug 22 18:41:51 CEST 1998 - Wed Aug 22 18:41:51 CEST 2018
Algorithm: SHA1withRSA
Public key length (modulus): 1024 bits

Certificate 2:
Subject: CN=GeoTrust Global CA,O=GeoTrust Inc.,C=US
Issuer: OU=Equifax Secure Certificate Authority,O=Equifax,C=US
Validity: Tue May 21 06:00:00 CEST 2002 - Tue Aug 21 06:00:00 CEST 2018
Algorithm: SHA1withRSA
Public key length (modulus): 2048 bits

Certificate 3:
Subject: CN=Google Internet Authority G2,O=Google Inc,C=US
Issuer: CN=GeoTrust Global CA,O=GeoTrust Inc.,C=US
Validity: Fri Apr 05 17:15:55 CEST 2013 - Sat Apr 04 17:15:55 CEST 2015
Algorithm: SHA1withRSA
Public key length (modulus): 2048 bits

Certificate 4:
Subject: CN=*.google.ch,O=Google Inc,L=Mountain View,ST=California,C=US
Issuer: CN=Google Internet Authority G2,O=Google Inc,C=US
Validity: Wed Sep 11 12:45:54 CEST 2013 - Thu Sep 11 12:45:54 CEST 2014
Algorithm: SHA1withRSA
Public key length (modulus): 2048 bits
```

// The used version of Java supports 80 cipher suites, which are all tested. The server supports 13 of them, and
// all are classified as SECURE

```
Check supported cipher suites (test program supports 80 cipher suites)
...........................................................................
..... DONE, 80 cipher suites tested

The following 13 SECURE cipher suites are supported by the server:
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
TLS_ECDHE_RSA_WITH_RC4_128_SHA
SSL_RSA_WITH_RC4_128_SHA
TLS_ECDHE_RSA_WITH_3DES_EDE_CBC_SHA
SSL_RSA_WITH_3DES_EDE_CBC_SHA
SSL_RSA_WITH_RC4_128_MD5

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

The next example shows a test of intra.zhaw.ch:443. In this case, only TLS 1.0 is supported by the server and two insecure cipher suites are supported – a clear indication that the server is not well configured:

```
$ java ch.zhaw.securitylab.TLSTester intra.zhaw.ch 443

...

Highest TLS version supported by server: TLSv1

...

Check supported cipher suites (test program supports 80 cipher suites)
...........................................................................
..... DONE, 80 cipher suites tested

The following 3 SECURE cipher suites are supported by the server:
SSL_RSA_WITH_RC4_128_SHA
SSL_RSA_WITH_3DES_EDE_CBC_SHA
SSL_RSA_WITH_RC4_128_MD5

The following 2 INSECURE cipher suites are supported by the server:
SSL_RSA_WITH_DES_CBC_SHA
SSL_RSA_EXPORT_WITH_RC4_40_MD5
```

The third example shows a test of www.rennhard.org:993. In this case, the root CA certificate is not included in the default truststore, which means that only the certificate(s) (in this case just one) that are received from the server are displayed:

```
$ java ch.zhaw.securitylab.TLSTester www.rennhard.org 993

Use default truststore with 155 certificates

Check connectivity to www.rennhard.org:995 - OK

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

Highest TLS version supported by server: TLSv1

Information about certificates from www.rennhard.org:995:

1 certificate(s) in chain

Certificate 1:
Subject: CN=*.rennhard.org,C=CH
Issuer: 1.2.840.113549.1.9.1=#16116d6172634072656e6e686172642e6f7267,
CN=Marc Rennhard,O=Marc Rennhard Private CA,C=CH
```

```
Validity: Fri Apr 12 15:00:07 CEST 2013 - Sun Apr 12 15:00:07 CEST 2015
Algorithm: SHA1withRSA
Public key length (modulus): 2048 bits

...
```

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

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

Use specified truststore (ts_local) with 1 certificates

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

The root CA is trusted

...
```

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.

- Basically, you should proceed similar to the „elaborate“ example from the lecture and work with `SSLContext`, `TrustManagerFactory` , and `SSLSocketFactory`.

  The SSLContext object (use TLS 1.2, which guarantees the server will use the highest version it supports) is created as follows:

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

  The TrustManagerFactory is created as follows:

  ```
  TrustManagerFactory tmf = TrustManagerFactory.getInstance("SunX509");
  ```

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

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

  When using your own truststore, it must first be read and must then be used to initialize the TrustManagerFactory;

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

  The SSLContext can now be initialized with the TrustManagers of the TrustManagerFactory:

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

  Note that the getTrustManagers method returns an array of TrustManagers. 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 only contains X.509 certificates and correspondingly, the returned array also only contains 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 produce SSLSocket objects for the tests:

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

Note that creating an SSLSocket establishes a TPC 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 or written to the socket.

- If Java cannot build the certificate chain to a trusted certificate, the TLS handshake throws 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. In the following, such a TrustManager is provided; it is also already included in the Eclipse project:

```
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 to initialize the SSLContext:

```
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 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.

- You can access the certificates that were received from the other communication endpoint (the peer) as follows:

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

- The SSLSession class also provides a method getProtocol() that returns the used TLS version.
- The certificates in the truststore can be accessed via TrustManagerFactory and TrustManager:

```
509TrustManager 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 set on the client-side a specific cipher suite that should be offered to the server during the handshake, you can use the method setEnabledCipherSuite() of SSLSocket and pass the desired cipher suite (here cipherSuite) as a String array that contains one element:

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

- To create your own truststore, the keytool is used. The following command imports a certificate server.cer into a truststore with the name ts_local:

```
keytool -importcert -keystore ts_local -alias servercert
-file server.cer
```

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

Lab Points

For **3 Lab Points** you must demonstrate your results to the instructor:

- You demonstrate that when using the default truststore, printing the complete certificate chain works with any TLS server in the Internet. If the certificate chain to a trusted certificate in the truststore cannot be built (i.e. the root CA certificate is not present in the truststore), only the certificates received from the server should be printed. (2 points)
- You demonstrate that determining the highest TLS version supported by the server and determining the secure and insecure cipher suites works for any TLS server in the Internet. (1 point)

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

Extra Point: You can get a fourth point if your program supports the optional two parameters, i.e. if the program is capable of performing the certificate check and printing the entire chain if an own truststore is used, the file name of which is passed to the program as a command line parameter. One possibility to demonstrate this is as follows:

- Use your 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 in in a dedicated truststore.
- Pass the file name of the truststore via command line parameter to the program and show that the entire certificate chain is still printed.