Laboratory 8. X509 CERTIFICATES AND HTTPS CONNECTIONS IN JAVA

In this laboratory we get used with X509 certificates and the establishment of secure HTTPS connections. The laboratory is written for Java but .NET support exists as well. We are interested in how certificates are encoded/stored but also on how to retrieve the certification chain and content of a webpage under HTTPS in Java. Nonetheless we also discuss how to add self-signed certificates to the Java **truststore** in order to establish HTTPS connection with **self-signed certificates**. Note however that self-signed certificates are vulnerable to man-in-the-middle attacks and thus they are not recommended for practical use.

8.1 CERTIFICATE SAVING AND ENCODING

X.509 is a standard that defines how to format public-key certificates. Digital certificates stay at the core of secure tunnelling protocols, i.e., protocols that allow the construction of an encrypted tunnel between clients and servers. One of the most commonly used applications is SSL/TLS which stays behind the Secure Hyper Text Transfer Protocol (HTTPS) protocol – a secure protocol for web-browsing.

Encoding X509 certificates. There are two ways to encode X509 certificates:

a. ***Binary*** encoding, also known as ***.DER*** coding,
b. ***ASCII*** enconding in ***Base64*** also known as ***.PEM*** encoding.

In case that you are not familiar with Base64 encoding, it means exactly what the name suggests. Each digit in a Base64 encoding represents a 6-bits of data, i.e., it encodes numbers $0\text{-}2^{64}$ to characters for providing a more convenient way to print them (as you know in case of binary representations not all bytes are printable). So Base64 provides a very handy encoding form, but with some data expansion, e.g., for storing 24 bits of data you will need 4 Base64 digits (4 bytes) rather than the usual 3 bytes. Generally, such data expansion is not relevant for general computer applications but it is more costly for embedded applications where using binary encoding should be the choice. Naturally, there are tools for converting between .DER and .PEM or vice-versa since they encode the same information and thus there is no problem with working with either formats. One tool that is commonly used to assure such conversions or generate certificates is *openssl*[1].

Storing X509 certificates. Of course, encoding refers only to how the data is represented inside the certificate. Further, the certificate needs to be stored in a file for which the most common extensions are: .CRT (usually in Unix), CER (usually in Windows) or .KEY. File with such extensions will store either .DER or .PEM encoded certificates. You can easily use your web-browser for saving the certificates of a website. In Figures 1 and 2 we show how to use Firefox for this purpose. Under the View Certificate button, the certificates is displayed and then you have an Export button to save the certificate. You can save it under either in binary or ASCII encoding and also include the certification chain.

[1] https://www.openssl.org/

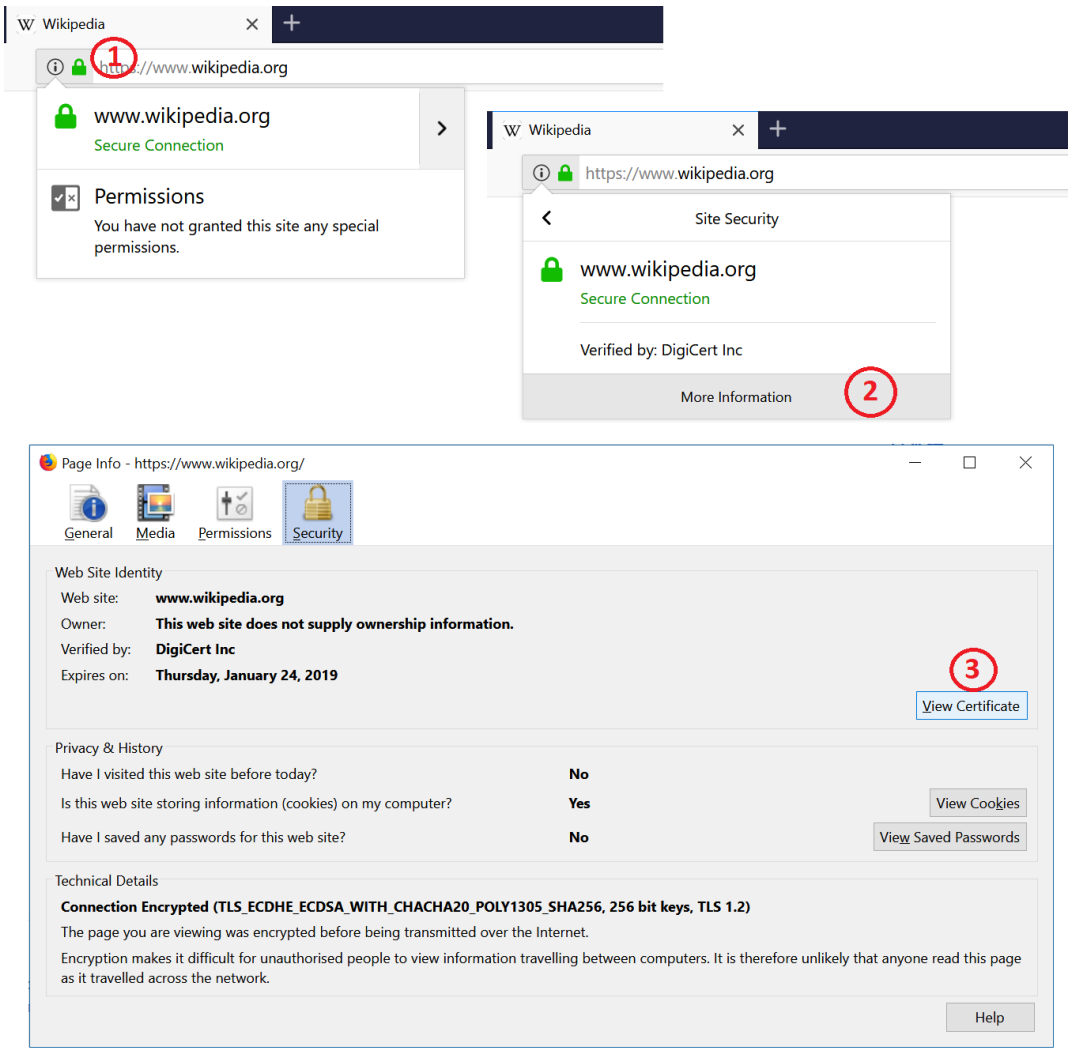

Figure 1. View certificate in Firefox

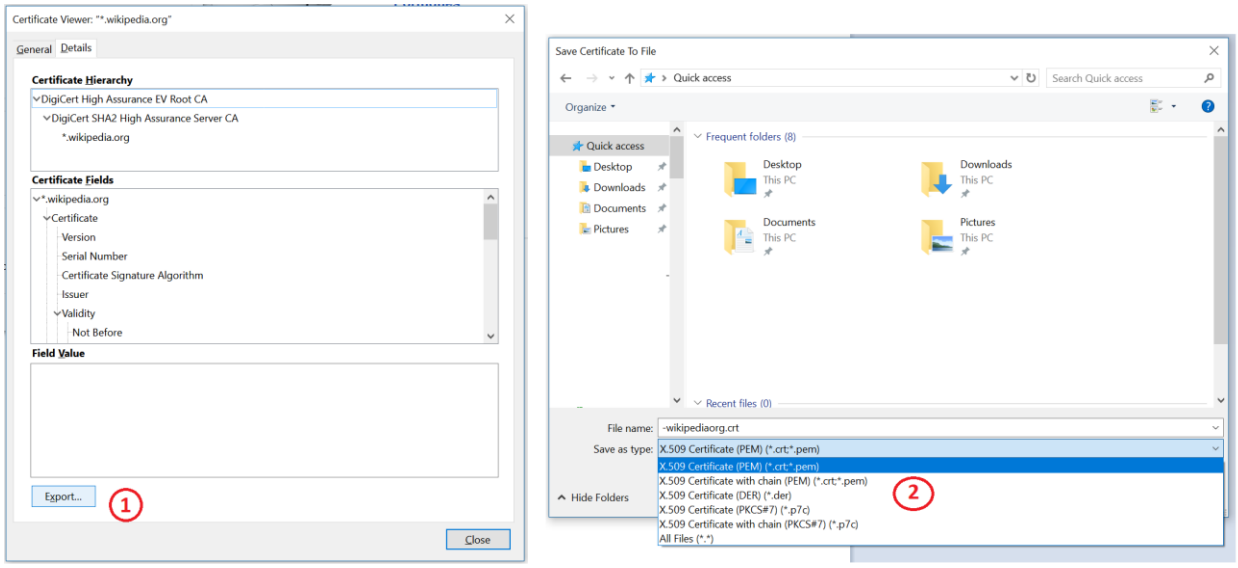

Figure 2. Export certificate in Firefox

8.2 ESTABLISHING AN HTTPS CONNECTION IN JAVA

Establishing the HTTPS connection. To establish the connection, first an *URL* object is needed. This object is instantiated with a string that holds the corresponding URL. Then the *openConnection* method returns the corresponding *HttpsURLConnection* object which we further use in our application. The complete source-code is available in Table 1. Once you have established the connection (this may return an exception so surrounding with try/catch is necessary) the *HttpsURLConnection* object provides the *getCipherSuite()* by which you can retrieve the cipher suite that was established with the server. In the current case the object returns TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256. Here GCM stands for the Galois Counter Mode, an encryption mode for block cipher and here AES in particular. The encoding of the cipher suite specifies the following: the key exchange protocol (RSA or ECDH), the signature (RSA or DSA), the symmetric encryption (AES usually), the hash for message authentication (usually SHA) and the particular elliptic curve (if elliptic curves are used or the curve is not implicit). This is shown in Figure 3. There are of course many other cipher-suites in use, examples include but are not limited to TLS_RSA_WITH_3DES_EDE_CBC_SHA, TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA, TLS_RSA_WITH_AES_128_CBC_SHA, etc. You should be familiar with the meaning of the previous acronyms since they all refer to cryptographic primitives that you have used in the previous laboratories.

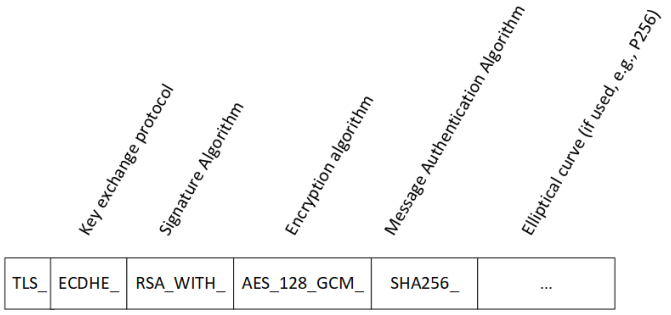

Figure 3. Structure of the cipher suite encoding

```
package javahttpstest;

import java.net.MalformedURLException;
import java.net.URL;
import java.security.cert.Certificate;
import java.security.cert.X509Certificate;
import java.io.*;

import javax.net.ssl.HttpsURLConnection;
import javax.net.ssl.SSLPeerUnverifiedException;
```

```
public class JavaHTTPSTest {

   public static void main(String[] args) {
    String https = "https://www.google.ro/";
    URL url;
    HttpsURLConnection con;
    try {

        // try to open connection
        url = new URL(https);
        con = (HttpsURLConnection)url.openConnection();
        InputStream conIS = con.getInputStream();
         // print SSL/TLS cipher-suite for this connection
         System.out.println("Cipher Suite : " + con.getCipherSuite());
         System.out.println("\n");

         // enumerate through the certification chain
         Certificate[] certs = con.getServerCertificates();
         for(Certificate cert : certs){
              System.out.println("Cert Public Key Algorithm : "
                        + cert.getPublicKey().getAlgorithm());
         System.out.println("\n");
         }
         } catch (SSLPeerUnverifiedException e) {
                    e.printStackTrace();
          } catch (IOException e){
                    e.printStackTrace();
     }
   }
```

Table 1. Code for establishing an HTTPS connection in Java

Retrieving the certification path. The Public-Key-Infrastructure is responsible for binding a particular name with a public-key. This mechanism is essential in circumventing man-in-the-middle attacks. Otherwise, anyone could generate a public-key certificate and claim a particular identity. The binding process is assured by certification path which links the certificate of a host (for example google.com) through one or more intermediate certificate authorities (CA), the last of which is bind to a Root CA which is trusted by the application. Figure 4 gives an example of certification path for google.com with the certificates that we retrieved with our *HttpsURLConnection* object in Java. To retrieve the complete certification path (the certification chain) you can use the *getServerCertificates()* which returns an ordered array of certificates with the host certificate followed by the certificate authorities in the order that they signed. The code in Table 1 enumerates through the certificates in order to print the public-key algorithm that was used.

Reading data from the connection. The *HttpsURLConnection* inherits the getInputStream method from the URLConnection class. This method returns the InputStream from which data can be further read. Note that encryption is transparent for the InputStream so data will be read in plaintext.

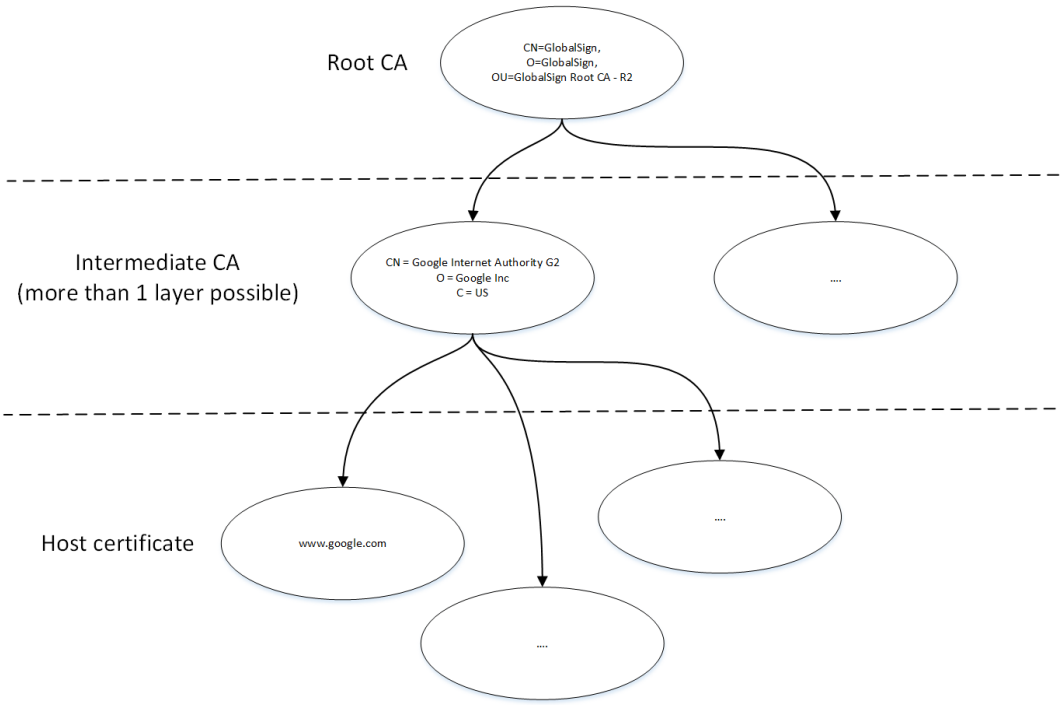

Figure 4. Example of certificate path, as retrieved for google.com

```
//print data from the url
BufferedReader br =new BufferedReader(new InputStreamReader(conIS));
String input;
while ((input = br.readLine()) != null){
      System.out.println(input);
 }
br.close();
```

Table 2. Code for reading data from the URL connection

8.3 Establishing an HTTPS Connection in Java with Self-Signed Certificates

Self-signed certificates are insecure since you cannot be sure that they are indeed the certificates of the end-point that claims them. Such certificates open road for man-in-the-middle attacks, an adversary can intervene in the middle of the channel and send his own certificate, but they are nonetheless common in practice. One way to surpass their insecurity is to use a secure off-line

channel to distribute them. This is of course contrary to the purpose of the PKI, but it is reasonable for a not so scalable setup.

If you are trying to establish a connection with a website that has a self-signed certificate, the previous program will end with the following exception: "*unable to find valid certification path to requested target*". This exception is shown in Table 3. Obviously, the Java run-time environment is not able to verify the certification path. The same exception is yield if you connect to a website that has a signed certificate but for which the root CA is unrecognized. To bypass this error the certificate must be added to the file were java store its trusted certificates, i.e., the Java Certificate Authority certificates file named **cacerts**. The file is usually located in C:\Program Files\Java\jdk1.8.0_151\jre\lib\security\cacerts but this is installation depended so you could simply perform a search for the cacerts file to find its location. The you can use the keytool executable to import a particular certificate. For this you also need to save the certificate of the corresponding website which was discussed in the previous paragraphs. The command line input and output is outlined in Table 4, you need administrator rights to modify the cacerts file.

```
javax.net.ssl.SSLHandshakeException: sun.security.validator.ValidatorException: PKIX path
    building failed: sun.security.provider.certpath.SunCertPathBuilderException: unable to
    find valid certification path to requested target
    at sun.security.ssl.Alerts.getSSLException(Alerts.java:192)
```

Table 3. Exception yield when connecting to a website with a self-signed certificate

```
C:\Program Files\Java\jdk1.8.0_151\jre\bin>keytool -importcert -keystore "C:\Program
Files\Java\jdk1.8.0_151\jre\lib\security\cacerts" -file
c:\Users\bogdan\Desktop\www.autuptro.crt

Enter keystore password:
Owner: EMAILADDRESS=webmaster.aut.upt.ro, CN=www.aut.upt.ro, O=" Automation and Applied
        Informatics Department", L=Timisoara, ST=Timis, C=RO
Issuer: EMAILADDRESS=webmaster.aut.upt.ro, CN=www.aut.upt.ro, O=" Automation and Applied
        Informatics Department", L=Timisoara, ST=Timis, C=RO
Serial number: 0
Valid from: Tue Jul 24 14:26:00 EEST 2007 until: Fri Jul 21 14:26:00 EEST 2017
Certificate fingerprints:
    MD5:  A9:68:2C:A5:97:D0:ED:39:44:96:66:86:5C:39:A1:B0
    SHA1: 00:25:33:98:73:41:FA:3C:06:31:11:99:FF:D0:78:DF:53:E1:EC:D5
    SHA256:
        2C:BB:97:1D:1F:5C:DC:E4:96:4B:38:7B:CD:F2:73:15:DB:85:67:D2:01:48:2A:D2:DE:46:3B
        :1C:29:32:AF:CA
Signature algorithm name: MD5withRSA (weak)
Subject Public Key Algorithm: 4096-bit RSA key
```

```
Version: 3

Extensions:

#1: ObjectId: 2.5.29.35 Criticality=false
AuthorityKeyIdentifier [
KeyIdentifier [
0000: B3 A5 9D 9B 40 D8 26 97   E1 CD E9 2E 4B C0 70 82  ....@.&.....K.p.
0010: F8 88 37 61                                        ..7a
]
[EMAILADDRESS=webmaster.aut.upt.ro, CN=www.aut.upt.ro, O=" Automation and Applied
              Informatics Department", L=Timisoara, ST=Timis, C=RO]
SerialNumber: [    00]
]

#2: ObjectId: 2.5.29.19 Criticality=false
BasicConstraints:[
  CA:true
  PathLen:2147483647
]

#3: ObjectId: 2.5.29.14 Criticality=false
SubjectKeyIdentifier [
KeyIdentifier [
0000: B3 A5 9D 9B 40 D8 26 97   E1 CD E9 2E 4B C0 70 82  ....@.&.....K.p.
0010: F8 88 37 61                                        ..7a
]
]


Warning:
The input uses the MD5withRSA signature algorithm which is considered a security risk.

Trust this certificate? [no]:  y
Certificate was added to keystore
```

Table 4. Adding the certificate www.autuptro.crt to the specified java keystore

8.4 EXERCISES

1. Write a Java program that establishes an HTTPS connection to a remote server and displays the entire certification chain.

2. Write a Java program that establishes an HTTPS connection to a remote server that uses a self-signed certificate. Add the self-signed certificate to the Java keystore.