Applied Information Security and Cryptography

Lab 05: Certificates and Transport Layer Security

In this lab you will create a self-signed certificate for your HTTPS server, then you will try to negotiate a secure session with the server under different choices of parameters. Finally, you will establish a secure session with a real server and you will analyse the handshake messages.

This lab requires at least Python version 3.7. Moreover, you will need the Python "cryptography" module that you used in Lab 03 to create the self-signed certificates. This can be installed with: pip install cryptography (or pip3 install cryptography according to your configuration). See instructions here: https://cryptography.io/en/latest/ . In order to run your own minimal web server you will also need Flask (https://flask.palletsprojects.com/en/1.1.x/installation/). You can install this with: pip install Flask (or pip3 install Flask). A virtual environment is not strictly required but can avoid conflicts with dependencies if you have many Python projects on your machine.

In addition, to analyse handshake messages you have to install Wireshark (https://www.wireshark.org/), a popular and free network protocol analyser available for most platforms (Windows, Linux, macOS, and many others).

Creating and using certificates

The Python script create_cert.py has a function `gen_self_signed_cert()` showing how to create a self-signed certificate using the cryptography library. The function allows you to choose the type of key that will be contained in the certificate (either RSA or ECDSA) and the hash function used to sign the certificate (either SHA1 or SHA256). The subject of the certificate will be your hostname (i.e., the name of your PC). The issuer of the certificate will be the same as the subject, since this is a self-signed certificate. The certificate also reports localhost as an alternative name, so that the certificate can be used by a server running on localhost (127.0.0.1). The function returns the signed certificate and the private key corresponding to the certificate's public key, both encoded in PEM format (This is basically the binary certificate base64-encoded and saved in ASCII characters, with some human readable separation lines. You can open the files with a text editor to see how they look like.).

Use the script create_cert.py to create different self-signed certificates, using either RSA or ECDSA and either SHA1 or SHA256. Look at the size in bytes of the different certificates and report them.

Now, run your own HTTPS server using one of the self-signed certificates. This can be done using the Python script start_server.py. You have to pass to the function `context.load_cert_chain('cert.pem', 'cert.key')` the correct files for the certificate and the corresponding private key. If everything went fine, you should look something like this:

```
* Serving Flask app "start_server" (lazy loading)
```

```
 * Environment: production
   WARNING: This is a development server. Do not use it in a
production deployment.
   Use a production WSGI server instead.
 * Debug mode: off
 * Running on https://127.0.0.1:5000/ (Press CTRL+C to quit)
```

Your server is responding at the URL https://127.0.0.1:5000 (this is port 5000 on localhost address, this means that this server is only accessible from your machine). To give it a try, open a web browser and visit the above URL. The web browser should not be happy with your server. Why is that?

The web browser should allow you to inspect the server's certificate (by clicking on the icon next to the address or clicking a button Show Details). Verify that the certificate corresponds to your self-signed certificate. If you trust your own certificate, then you can visit the website.

TLS handshake

Now, we will make some experiments trying to negotiate TLS session parameters with your local server. For this, you will need the Python script test_client.py. For connecting to the local server, you have to set the following variables in the script (which should be already set):

```
hostname = '127.0.0.1'
portnumber = 5000
```

The script creates a basic SSL context using the Python ssl module:

```
context = ssl.SSLContext(ssl.PROTOCOL_TLS)
```

this is basically a wrapper for OpenSSL, which is a popular cryptographic library for managing TLS connections. You can decide whether to skip certificate validation (not secure, you will connect even to untrusted servers):

```
context.verify_mode = ssl.CERT_NONE
```

or to validate the server certificate using trusted certificates:

```
context.verify_mode = ssl.CERT_REQUIRED
```

To load the default trusted certificates stored by the operating system you can use:

```
context.load_default_certs()
```

To load a specific trusted certificate (e.g., the local server's certificate):

```
context.load_verify_locations('cert.pem')
```

You can decide which TLS versions the client is willing to negotiate by setting the variables:

```
context.minimum_version
context.maximum_version
```

With the default setting the client will accept TLSv1 as minimum version and TLSv1.2 as maximum version. Upon a successful connection, the script will report the negotiated TLS version, the cipher suite, and information on server's certificate.

Try the following experiments:

1. Try to connect to the local server using the default values. You should get an error, report it. Then, load the server's certificate as a trusted certificate (`context.load_verify_locations`) and try again. Verify that the received certificate is the correct one. Report the TLS version negotiated between client and server and the chosen cipher suite. Report the cryptographic algorithms used in the different parts of the handshake protocol: key-exchange, server authentication, data encryption and authentication, hash function.
2. Set the maximum TLS version negotiated by the client to TLSv1.3 and connect again. Report the TLS version and cryptographic algorithms. Did they change? Comment on this.
3. Set the maximum TLS version negotiated by the client to TLSv1.1 and connect again. Report the TLS version and cryptographic algorithms. What are the main changes? Do you see any possible vulnerabilities?
4. Remove ECDHE from the cryptographic algorithm negotiated by the client. This can be obtained with the instruction `context.set_ciphers('RSA:!ECDHE')`. Check that the server has a certificate with an RSA key. Connect again and report the cryptographic algorithms. What are the main changes? Do you see any additional vulnerabilities with respect to case 3?
5. Set the maximum and minimum TLS version negotiated by the client to TLSv1.3 and connect again. Did it work? Please comment.
6. Set the maximum TLS version negotiated by the server to TLSv1.3 (edit start_server.py) and restart the server (close the running server with Ctrl-C and run again start_server). Connect again and report the cryptographic algorithms. Did the removal of ECDHE has any effect in this case? Please comment. Which vulnerabilities are removed in this case?

Inspecting TLS handshake messages

Start Wireshark. You should see a window like in Figure 1. Select the interface which is currently connected to your home network (Typically Wi-Fi or Ethernet. You should see some activity on the trace shown to the right of the interface name.).

In …using this filter box write: **host www.polito.it**
This will filter all traffic to and from www.polito.it. Without a filter you would see a lot of traffic during a capture!

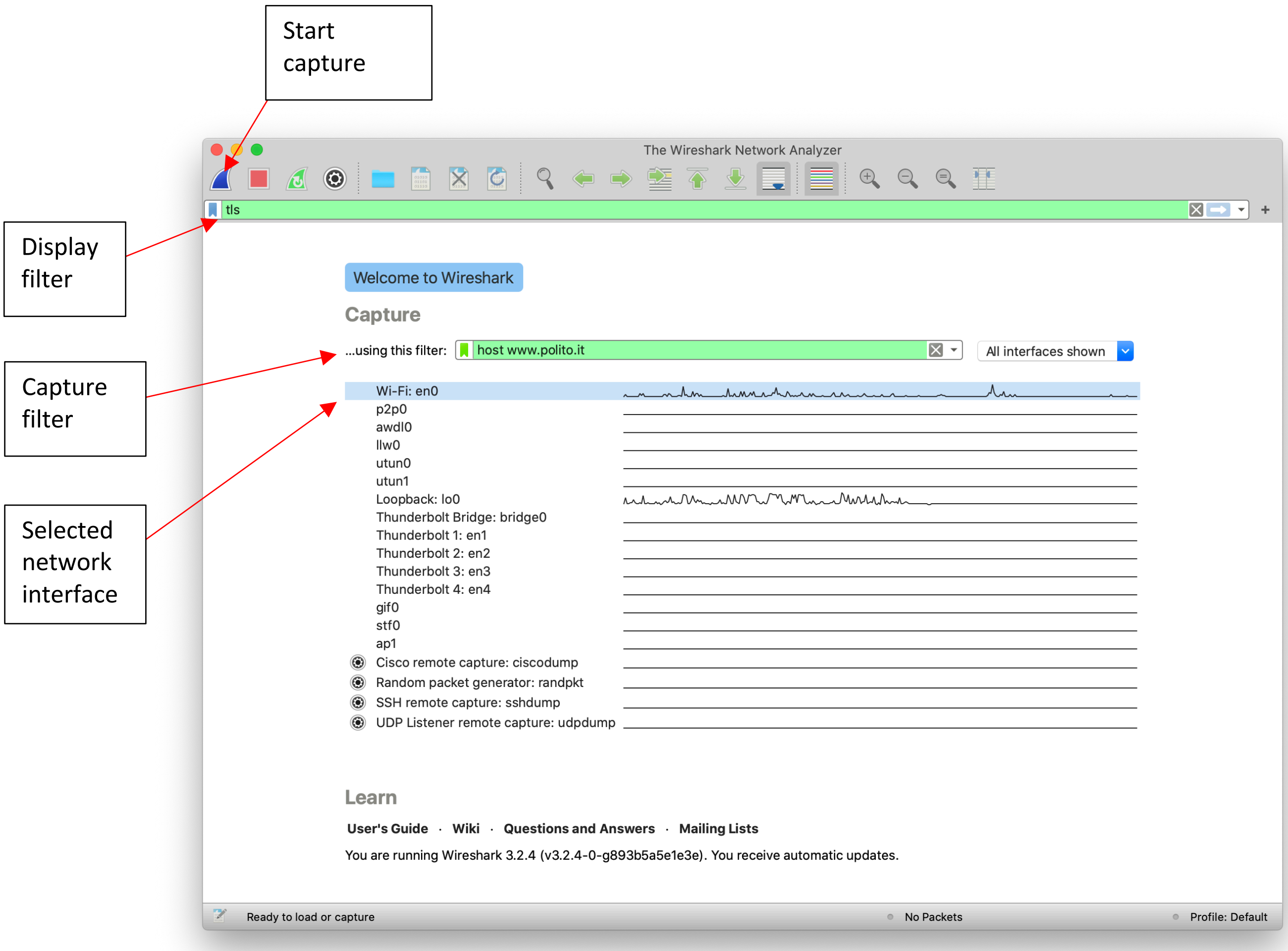
In the display filter box above write: **tls**
Click on the small arrow at the right of the display filter box to activate the filter. This will display only packets related to the TLS protocol.

Then start packet capture by clicking on the blue fin in the left corner of the menu.

Modify the hostname and portnumber in the script test_client.py:

```
hostname = 'www.polito.it'
portnumber = 443
```

and comment `context.set_ciphers('RSA:!ECDHE')`.



The Wireshark Network Analyzer

tls

Welcome to Wireshark

Capture

...using this filter: host www.polito.it

All interfaces shown

Wi-Fi: en0
p2p0
awdl0
llw0
utun0
utun1
Loopback: lo0
Thunderbolt Bridge: bridge0
Thunderbolt 1: en1
Thunderbolt 2: en2
Thunderbolt 3: en3
Thunderbolt 4: en4
gif0
stf0
ap1
Cisco remote capture: ciscodump
Random packet generator: randpkt
SSH remote capture: sshdump
UDP Listener remote capture: udpdump

Learn

User's Guide · Wiki · Questions and Answers · Mailing Lists

You are running Wireshark 3.2.4 (v3.2.4-0-g893b5a5e1e3e). You receive automatic updates.

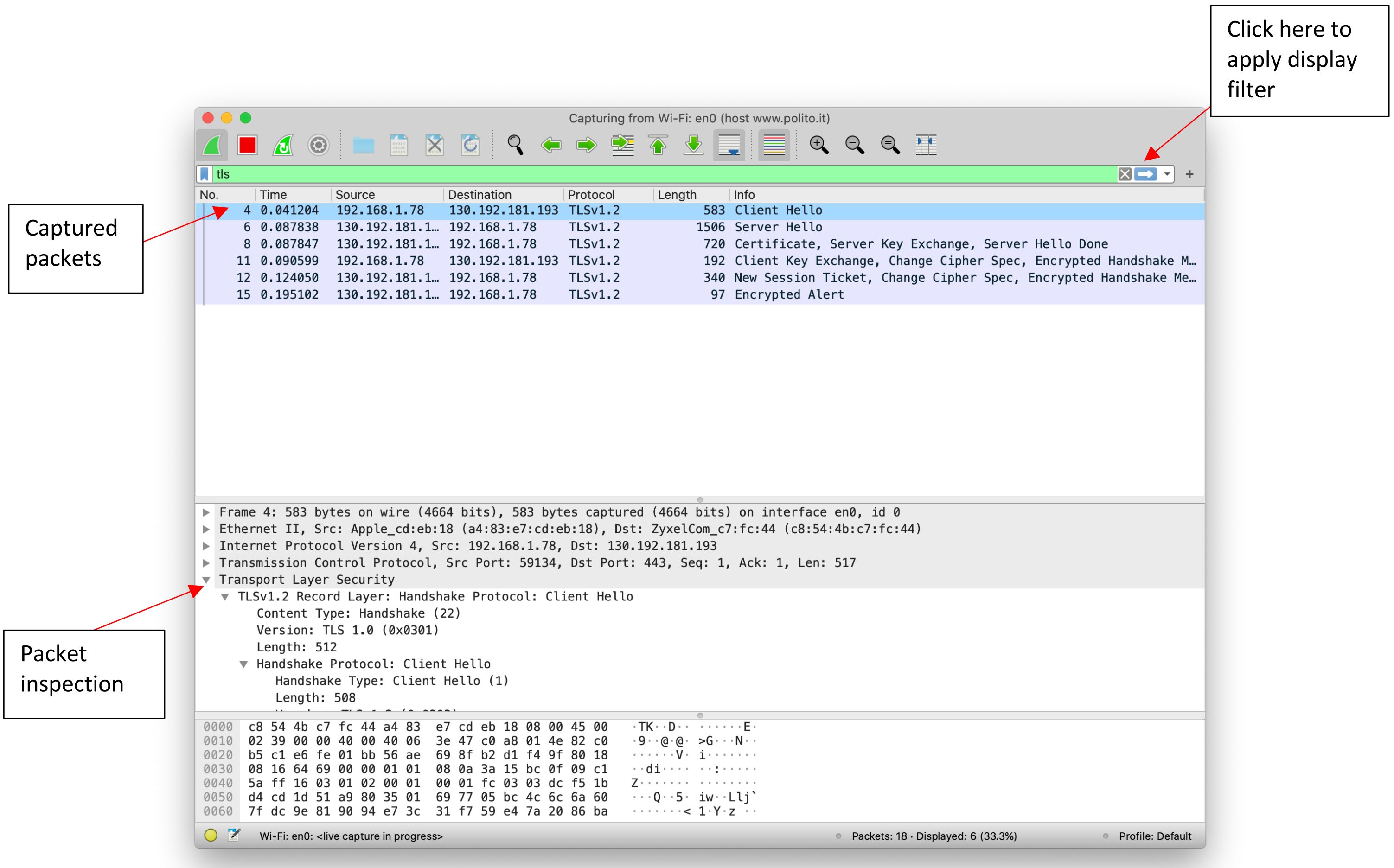
Ready to load or capture No Packets Profile: Default

Figure 1 Wireshark welcome screen

Now, perform the following experiments:

7. Set the maximum TLS version negotiated by the client to TLSv1.3 and the minimum TLS version to TLSv1. Run the script test_client.py to connect to www.polito.it on port 443. In Wireshark, you should see a complete handshake like in Figure 2. Inspect the different packets to see the content of the handshake messages. You can select the captured packets in the upper part of the window and inspect the content of each packet in the lower part of the window. Which TLS version is chosen by the server? What cryptographic algorithms? Report the content and meaning of the Server Key Exchange and Client Key Exchange messages.

8. Set the maximum TLS version negotiated by the client to TLSv1.1 and connect to the server. Report the handshake messages. What is the meaning of the server's message?
9. Close the current capture on Wireshark, in …using this filter box write: **host www.google.com** and start a new capture. Set the hostname variable in test_client.py to www.google.com. Set the maximum TLS version negotiated by the client to TLSv1.3 and the minimum TLS version to TLSv1. Connect to the server and report the handshake messages. Which TLS version is chosen by the server? What cryptographic algorithms? Can you see the server's certificate in the handshake messages? Why?
10. Set the maximum TLS version negotiated by the client to TLSv1.1. Connect to the server and report the handshake messages. What are the differences with respect to the previous point? And the differences with respect to point 8?


Click here to apply display filter
Captured packets
Packet inspection
Capturing from Wi-Fi: en0 (host www.polito.it)
tls

```
No.  Time      Source          Destination     Protocol  Length Info
   4 0.041204  192.168.1.78    130.192.181.193 TLSv1.2      583 Client Hello
   6 0.087838  130.192.181.1…  192.168.1.78    TLSv1.2     1506 Server Hello
   8 0.087847  130.192.181.1…  192.168.1.78    TLSv1.2      720 Certificate, Server Key Exchange, Server Hello Done
  11 0.090599  192.168.1.78    130.192.181.193 TLSv1.2      192 Client Key Exchange, Change Cipher Spec, Encrypted Handshake M…
  12 0.124050  130.192.181.1…  192.168.1.78    TLSv1.2      340 New Session Ticket, Change Cipher Spec, Encrypted Handshake Me…
  15 0.195102  130.192.181.1…  192.168.1.78    TLSv1.2       97 Encrypted Alert
```

```
▶ Frame 4: 583 bytes on wire (4664 bits), 583 bytes captured (4664 bits) on interface en0, id 0
▶ Ethernet II, Src: Apple_cd:eb:18 (a4:83:e7:cd:eb:18), Dst: ZyxelCom_c7:fc:44 (c8:54:4b:c7:fc:44)
▶ Internet Protocol Version 4, Src: 192.168.1.78, Dst: 130.192.181.193
▶ Transmission Control Protocol, Src Port: 59134, Dst Port: 443, Seq: 1, Ack: 1, Len: 517
▼ Transport Layer Security
  ▼ TLSv1.2 Record Layer: Handshake Protocol: Client Hello
      Content Type: Handshake (22)
      Version: TLS 1.0 (0x0301)
      Length: 512
    ▼ Handshake Protocol: Client Hello
        Handshake Type: Client Hello (1)
        Length: 508
```

```
0000  c8 54 4b c7 fc 44 a4 83  e7 cd eb 18 08 00 45 00   ·TK··D·· ······E·
0010  02 39 00 00 40 00 40 06  3e 47 c0 a8 01 4e 82 c0   ·9··@·@· >G···N··
0020  b5 c1 e6 fe 01 bb 56 ae  69 8f b2 d1 f4 9f 80 18   ······V· i·······
0030  08 16 64 69 00 00 01 01  08 0a 3a 15 bc 0f 09 c1   ··di···· ··:·····
0040  5a ff 16 03 01 02 00 01  00 01 fc 03 03 dc f5 1b   Z······· ········
0050  d4 cd 1d 51 a9 80 35 01  69 77 05 bc 4c 6c 6a 60   ···Q··5· iw··Llj`
0060  7f dc 9e 81 90 94 e7 3c  31 f7 59 e4 7a 20 86 ba   ·······< 1·Y·z ··
```

Wi-Fi: en0: <live capture in progress>   Packets: 18 · Displayed: 6 (33.3%)   Profile: Default

Figure 2 Wireshark capture screen

Questions (Answer these in your report)

Report the size of the different self-signed certificates that you obtained and comment on the differences.

Report the results of experiments 1-6, comment on them and answer the related questions.

Bonus task: What happens in point 4 if the server certificate has an ECDSA key? Please comment and explain the outcome of the handshake.

Report the results of experiments 7-10, comment on them and answer the related questions.

Bonus task: What happens in point 7 if the client removes ECDHE from the offered cipher suites (uncomment `context.set_ciphers('RSA:!ECDHE').`). Report the handshake messages. What is the meaning of the Server's message in this case?

Bonus task: Can you find the key exchange messages in point 9? Report the parameters used for key exchange in this case.

Bonus task: What happens if the client removes ECDHE from the offered cipher suites in point 10? Report the handshake messages, the selected cipher suite, and explain in which message there is a key exchange and which algorithm is used.