02233 Network Security
Assignments: Transport Layer Security (TLS)

2024-02-20

Today we will experiment with TLS certificates using the `openssl` toolkit, and have a brief introduction to Wireshark. The `openssl` toolkit is commonly used to generate and check certificates, which we use to encrypt communications and authenticate both ends. In this lab, we will:

1. generate a private RSA key and a certificate signing request (CSR),
2. generate an `x.509` certificate and learn to inspect the certificate,
3. setup a simple TCP server with TLS, and
4. use Wireshark to inspect the traffic between the server and the client.

If you do not have a Kali VM yet, you can download it from here (you can choose your favorite virtualization platform). Lastly, make sure that `openssl` and Wireshark are installed on your Kali VM (they should be pre-installed):

```
# Check if openssl is installed already
openssl version
# Check if wireshark is installed
wireshark --version
# OTHERWISE, install openssl and wireshark using your package manager
sudo apt-get update \
    && sudo apt-get upgrade -y \
    && sudo apt install openssl wireshark -y --fix-missing
```

Note: while it is not strictly necessary to use a Kali VM for this lab, we recommend it to avoid further issues. Virtual Machines are disposable, so you can play with them and then delete them without worrying about making changes to your operative system or losing anything important. However, you should keep your Kali VM for the following weeks.

Exercise 1

First, use the `genrsa` command of `openssl` to generate an **RSA private key** of 1024 bits and save it in a file. Then, use the `req` command of `openssl` to generate a **certificate signing request** (CSR) using the RSA key we just created. You can populate various fields of the certificate with information regarding location, company, etc.

A typical private key looks like this:

```
-----BEGIN PRIVATE KEY-----
MIICdQIBADANBgkqhkiG9w0BAQEFAASCAl8...
-----END PRIVATE KEY-----
```

And a CSR looks like this:

```
-----BEGIN CERTIFICATE REQUEST-----
MIIBhDCB7gIBADBFMQswCQYDVQQGEw...
-----END CERTIFICATE REQUEST-----
```

Solution
o generate a RSA private key with use:
`openssl genrsa -out rsa_1024 1024`
To create a Certificate Signing Request (CSR) use:
`openssl req -new -key rsa_1024 -out rsa_1024.csr`

Spend a few moments answering the following questions:

1. We just used RSA with a key length of 1024. Can you see any potential issues already?
2. We created our CSR from a private key. What would happen if we shared this key?

Exercise 2

In general, we refer to two types of certificates: those signed by a trusted Certificate Authority (CA), and *self-signed* certificates. In this exercise, we will generate a self-signed certificate by signing our certificate with our private key. We will use the common `x.509` standard to generate a public key certificate, which will provide TLS encryption for our HTTPS server in the next exercises. For this, we use the `x509` command from `openssl` to sign the CSR we previously generated. Lastly, inspect the certificate we just created using `openssl`.

Solution
o generate a self-signed certificate we could use:
`openssl x509 -req -in rsa_1024.csr -signkey rsa_1024 -out rsa_1024.pem`
To inspect the certificate use:
`openssl x509 -text -in rsa_1024.pem`

These days, `x.509` certificates have a default expiration date of 13 months, but this can be easily modified! spend a few minutes to answer the following questions (in groups?).

1. What would happen if we connected to a server with expired certificates?
2. Can you see any potential issue with extending the validity of the certificate?
3. What about connecting to a server with a certificate generated using weak cryptographic functions?
4. **Bonus:** What issues do you see in a certificate that has been valid for the last 20 years?

Exercise 3

Use the tool `openssl s_server` to set up a test TCP server listening on port 1443 on your machine, using the certificate and private key we just generated in the previous exercises.

You will likely get an error!!! specifying that the length of our private key is too short (as you have seen in the first exercise, this is a severe security issue).

Generate a new certificate using a longer key, for example RSA 4096 (this one is valid!), and start the server. Connect to the server from your preferred browser (still in the VM) on the address `https://localhost:1443`. What do you notice?

Solution
o start the server use:
`openssl s_server -accept localhost:1443 -cert rsa_4096.pem -www -key rsa_4096`

Note: you could use any port (for example 443, which is the default for HTTPS, so you wouldn't have to specify it in the browser), but port numbers below 1000 may require elevated permissions.

Exercise 4

Now, we will use Wireshark to inspect the traffic we generate to our TCP server. Start capturing traffic on the `lo` interface (adapter for loopback traffic), where you will see all the requests we are sending to our server (reload the page on your browser once or twice to see Wireshark getting populated with traffic traces). Then, spend some time to answer the following questions:

1. Which version of TLS are we using?

2. Which symmetric encryption/authentication algorithm do we use during the TLS negotiation between the server and our browser?

3. Navigate to `http://www2.compute.dtu.dk/courses/02234` and compare the packet headers with the ones from our server. Can you read the website content from the packet?

4. Use the keys generated during handshakes to decrypt TLS traffic in Wireshark (check out this article).

Solution
uring the TSL handshake, we exchanged keys and selected our ciphers to communicate with the server. The TSL version and encryption algorithms can be seen in the TLS headers. In my setting, I can see the following:
`Version: TLS 1.2 (0x0303)`
Cipher Suite: TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256 (0xc02f)

Bonus: Exercise 5

It is time to put everything together and mount our attack. You have learned that TLS encrypts communications between client and server. You have also learned how this happens, and at which specific moment: the handshake. In addition, you now know that if we can capture these keys we can decrypt the live traffic. Now, you are ready to mount a Man-in-the-middle SSL strip attack:

1. Intercept the communications between a client and a server (google "ARP poisoning").

2. Force the client to re-establish the communication channel, but this time, bridge the connection through your server. You will force the client to establish an insecure connection with your bridge, and your bridge will create a secure connection with the server. Now you can eavesdrop on the communication in plain text!

3. You can simplify the process using the tool `ettercap` (comes pre-installed in Kali). It requires a bit of manual configuration, but there are plenty of tutorials on the Internet. See the original repository of this attack here.

Note: there will be no solutions for this exercise!