NORWEGIAN UNIVERSITY OF SCIENCE AND TECHNOLOGY

TTM4135 APPLIED CRYPTOGRAPHY AND NETWORK SECURITY

Web Security Lab

March 3, 2023

Contents

1 Preliminaries and General Information

1.1 Objectives

This practical project ("Lab") is an assessed part of the course, counting for 20% of the final grade. The project focuses on hands-on experience with cryptographic security mechanisms in selected web security software.

The assignment tasks include investigation, configuration and programming related to relevant security mechanisms but, as in real engineering, there are other details to deal with as well. The purpose of this assignment is to experiment with practical configuration of TLS on a typical web server implementation and practice using PGP.

1.2 Logistics

This project will be performed in groups of three or four students. Groups are assigned through Blackboard.

The teaching assistants (TAs) will be available online and physically (at published times) for the duration of the lab period. A detailed timetable of availability of the Sahara Lab will be

published in Blackboard. You are welcome to work together as a group in the Sahara Lab where there will be TAs available during the published times.

For part II and III of the lab, a server is available. This server is a virtual machine (VM) that runs Ubuntu. Part II contains instructions for activating and authenticating to this VM. To login to the VM when it is set up you can work from your own computer. Mac or Linux users can use the built-in console program, Windows users can either use the unix subsystem to run the commands, or install an SSH-client, e.g. PuTTY [4].

1.3 If you are stuck

If you are unable to solve an issue by searching online for an answer, consider asking other groups for help. You could do that directly if you know students in other groups, but you are encouraged to make use of the Piazza discussion forum. It is likely that other students have had, or will have, the same problems as you. Therefore it is worthwhile to check whether your problem is already discussed there and, if not, to post it to the forum. The TAs will also monitor the forum.

If this does not solve your problem you can show up at the Sahara Lab and talk to a TA during the published times, or you can make an appointment for an online meeting with a TA by email via `ttm4135@item.ntnu.no`.

1.4 Evaluation

The assignment submission consists of two parts:

1. Demonstration of achieved milestones (see below).
2. Lab report.

We do expect that the group work is actually done as a group, with everyone in the group participating. Make sure to coordinate with the rest of your group when you are going to meet, who is going to be responsible for what and set deadlines for the intermediate parts. Ensure together that the final report is a product of joint effort between all group members.

1.4.1 Milestones

Throughout this document you will find certain activities to guide you through the lab. There are three categories.

Tasks: these are things you should do to make progress, but they are not checked individually.

Milestones: these have to be completed before the lab ends. The teaching assistants will verify that the milestones have been reached. In order to optimise the time of the TAs please have milestones checked by email wherever possible, via `ttm4135@item.ntnu.no`.

Questions: these are things that you need to discuss in your report. You do not have to write them up in detail during the lab, but you should at least make notes so that you have the information available when you come to write your report.

There are no intermediate deadlines, but we recommend to stick to the following schedule:

Part I. (PGP) Completed in the first lab week.

Part II. (Apache and certificates) Completed in the second lab week.

Part III. (Eavesdropping) Completed by the end of the final lab week.

The lab deadline before which all milestones must be completed is Friday March 24, 2023.

1.4.2 The Report

Your report will be assessed. The lab grade consists of completion of specified lab tasks and report assessment which together count as 50% of the pre-exam grade.

The report deadline for submission through Blackboard is Friday April 17, 2023.

⇒ **A template file for writing the report** along with instructions on how to complete the report is available in Blackboard.

2 Part I: Encrypted email

Email messages are by default insecure: the content of messages can easily be read by an adversary, and it is almost trivial to spoof the sender of a message.

In this first part of the lab we will learn how to secure email messages with PGP and GPG, and how the principles of cryptography you learned in the lectures are applied when using these tools.

2.1 Historical background

PGP (Pretty Good Privacy) is a system for encryption of messages. Its background provides some insight into the issues of the first days of internet cryptography.

The software was first released by Phil Zimmerman in 1991 as a command line tool capable of symmetric encryption. Asymmetric RSA encryption was added in version 2 and the software quickly became available outside the United States, but as a result of this, Zimmerman was put under investigation by the authorities since cryptographic software was at the time seen as a military technology and thus exporting it was illegal. In order to prevent criminalization, he published the source code to PGP in a book: the strong anti-censorship laws in the USA prevented this from being illegal. In 1996, Zimmerman and other key developers started a company and began to develop PGP3 as a commercial product.

PGP has since then become the *de facto* standard for encrypting email: relying on one software suite for this was thus undesirable. Zimmerman and his company thus moved the name PGP and the algorithms to the Internet Engineering Task Force in 1997, which has since worked on developing the OpenPGP standard since then. The current standard is documented in RFC4880.

2.2 PGP vs. GPG

As described above, PGP is the *standard* we will use for message encryption in this part of the course. We will now focus on GPG, which stands for GNU Privacy Guard and is a software suite that adheres to this PGP standard.

GPG consists of several parts: it provides the users with a key management system and allows them to sign, encrypt, decrypt, and verify the signatures of messages. GPG itself remains primarily a command-line tool, but has also been incorporated with other programs, particularly mail client extensions, to improve the user experience. GPG retains the ability to use strictly symmetric encryption, but the default mode of operation is PKI-based. After GPG has obtained the public keys for another user, it can verify messages from and encrypt messages to the other user. Once a digital identity consisting of a public/private key-pair has been generated, GPG can sign messages it wishes to send and decrypt messages it has received.

GPG gives users the ability to manually manage their so-called keyring (a library of public and private keys) by importing and exporting keys, but it is more common to use an online keyserver.

2.3 Task: generating a PGP keypair

We will now use GPG to generate a PGP public key and private key, which you can then use to encrypt and send emails.

Generating a PGP key using GPG can be done by running `gpg` with the `--gen-key` option. The script will ask your name and email address: please use your real name and NTNU email address (`@stud.ntnu.no`) for the benefit of this course.

By default, GPG generates an RSA public-private key pair used for signing messages, and an RSA subkey for encryption. Both are part of the PGP identity. Additional subkeys can be signed by the master RSA (or DSA) key pair, and will be distributed with the master public key as part of the PGP identity when shared. The subkey may only be used for either encryption (Elgamal or RSA) or signing (DSA or RSA).

Afterwards, you can verify that the key has been generated by using the option `--list-keys`. This is what you should see in the console after running both commands:

```
  $  gpg --gen-key
  ...
  $  gpg --list-keys
pub   rsa2048 2019-02-27 [SC] [expires: 2022-08-29]
      F76DE997E3D4C03AB01D759333E694E557EEB343
uid           [ unknown] Course TTM4135 <ttm4135@item.ntnu.no>
```

> **Task 1:** Ensure you have GPG working on your computer and take a moment to generate a keypair using your NTNU email address and name.

Remark: If you already have a PGP key associated with another identity on your computer, you may need to prefix the other commands we will use in this tutorial with `-u keyid` to make sure the correct one is used.

You can have a look at your new public key by using the options `armor` and `export`, with your own fingerprint:

```
$ gpg --armor --export F76DE997E3D4C03AB01D759333E694E557EEB343
-----BEGIN PGP PUBLIC KEY BLOCK-----
Version: GnuPG v1
mQENBFWPrS8BCAC20iXSGy3QGh8Bb9buVQZ4hX2OVrN594ndVtVdTEgFl8VRwwF
>clipped<
DifyBjZ0rt8lcE4KNOhNMihLkN0Lhtw2v2Y0LwyZSd/0v7yv1sR3NStSDZjV5AN
5g==
=dy
-----END PGP PUBLIC KEY BLOCK-----
```

2.4 Web of Trust

Using this new pair consisting of a public and private key we can now do some interesting things: we can sign messages using the private key (and others can verify that we did so using the public key), and others can encrypt messages using our public key (which can then only be decrypted by us using the private key).

This is all very nice, but it is only useful if others are able to verify that we are in fact the legitimate owner of the keypair.

To facilitate that, PGP uses the *web of trust* model. Instead of forcing a user to blindly trust one or more universal authorities, PGP asks the user to make his or her own evaluation of the validity of a new key. In addition, users have the option to validate and sign the keys of other users.

Example: Bob wants to send a message to Cathy. On a keyserver he finds a key that is labeled with Cathy's name. Bob does not know whether this is actually Cathy's key, but he has a keypair which he knows belongs to Alice, and he trusts Alice. Now if Bob can see that Alice trusts this is Cathy's key, he may choose to accept it as true. Bob also trusts Eve, who also trusts Cathy's key, so he can be even more sure.

This flexibility allows GPG to accurately model the trust relations present within a society. It also allows for regular hierarchical authorities (e.g. Bob trusts anything that Alice trusts to the same degree that he trusts Alice's key).

At hacker events like CCC in Germany or DEF CON in the US you will find so called key-signing parties: gatherings intended to facilitate the web of trust by encouraging individuals to verify the physical identity of an individual claiming to own a key, and if verified, sign that key. Key-signing and any kind of partying are unfortunately not covered in this lab.

2.5 Key servers

Key servers are servers that provide a public key repository and can facilitate key distribution. Using these systems users can search for keys associated with a specific user, or retrieve a full key based on only its fingerprint.

As the degree of trust is specifically left to the individual user, key servers do not need to keep track of this. They are often pooled together to provide a more replicated and resilient infrastructure not dependent on any one organization.

In this exercise we will upload our newly generated keypair to a key server so others can use it.

Uploading our key to a server Above we showed how you can export the public key to ASCII text using `armor` and `export`. Some servers allow you to upload the key through an HTML form: in that case you just copy-paste the text above (after you double-check that you're using the public and not the private key block).

Commonly it is also possible to send the key to the server directly using gpg's `send-key` interface, like shown below. In this case we specify with `--keyserver` that it should go to the server `keys.openpgp.org`. If you don't specify that, it will be added to the default server in your pgp installation.

```
$ gpg --send-key --keyserver keys.openpgp.org \
F76DE997E3D4C03AB01D759333E694E557EEB343
```

Remark: If you find that `keys.openpgp.org` is temporarily offline or not responding you can use another server, or omit `--keyserver` so it is sent to the server pool.

Downloading keys from servers If you want to send a message to another user, you can download their keys in a similar way. For instance, if we want to have the key belonging to `ttm4135@item.ntnu.no` and we know that that key has ID `F76DE997E3D4C03AB01D759333E694E557EEB343`, do this:

```
$ gpg --search-keys ttm4135@item.ntnu.no
```

... and validate that it is on your keyring now:

```
    $ gpg --list-keys
pub   rsa2048 2019-02-27 [SC] [expires: 2021-02-26]
      F76DE997E3D4C03AB01D759333E694E557EEB343
uid           [ unknown] Course TTM4135 <ttm4135@item.ntnu.no>
```

Remark: An alternative way to do this is by using the web interface of the key server. In this example, you can go to `keys.openpgp.org` in your browser and search for the key ID or email address.

Task 2: Using the keyserver.

1. Upload your PGP key to a key server of your choice, or the default one.
2. Download the PGP key of at least one group mate and verify that you have it on your system.
3. Download the PGP key with ID `F76DE997E3D4C03AB01D759333E694E557EEB343` belonging to the address `ttm4135@item.ntnu.no`.

2.6 Operations using GPG

Now that we have some feeling of how to manage keys using GPG, we will start using them for a number of useful operations.

PGP standardizes encryption and decryption, but also signing messages and signature validation. This is useful if you download a software package which has been signed by its developers, or of course for email. Encryption and decryption require that both parties use PGP in order to transmit any information at all, whereas signature validation isn't required in order to access the signed content.

Signing messages Signing a message or file with a private key allows any individual to verify that the message received matches the message sent by the claimed sender. PGP supports three forms of signatures:

- *Standard signatures* compress and sign the original content to produce a single file. This mode requires that the received message is verified by GPG.
- *Clearsigned signatures* are intended for text documents, and produce an ASCII-armoured (partially human-readable) signature concatenated to the message. This mode is frequently used with emails and other short text messages.
- *Detached signatures* do not incorporate the message being signed in the resultant file. This

is especially useful when the recipient may not use GPG to validate the message, especially when the message is a large ISO or binary download.

Clearsigned and detached signatures can be created as follows:

```
$ gpg --output messageC.sig --clearsign message
$ gpg --output messageD.sig --detach-sign message
```

Verify On the other end of message validation is verifying that a signature matches a sender and message (or file). Messages can be validated as follows.

```
$ gpg --verify  messageC.sig
```

If the message or file is accompanied by a detached signature, both the signature and message files must be provided:

```
$ gpg --verify messageD.sig message
```

Task 3: Exchange a signed message with one of your group mates: verify among yourselves that this actually works as it should.

Remark: Opening the generated encrypted files or signatures may cause problems, as some editors will automatically convert line endings and other whitespace. You are encouraged to take a peek, but do so in a copy of the file, or after you have submitted.

Encrypt Encryption requires the public key of the recipient, and thus that key must be imported into the GPG keyring as explained above. Then you can encrypt the message as follows:

```
$ gpg --output message.gpg --encrypt --recipient olanordm@stud.ntnu.no message
```

In reality, GPG will not encrypt the entire message using the recipients' public key p: instead a new symmetric key k is generated, that is used to encrypt the message and then both the message (encrypted with k) and k (encrypted with the public key p) are sent to the receiver.

Question 1: Generating a symmetric key k just for encrypting that one message seems like an unnecessarily complicated step. Why does GPG do that, instead of just encrypting the message with p?

Messages can be simultaneously signed and encrypted as well:

```
$ gpg --output message.gpg --sign --encrypt --recipient \
      olanordm@stud.ntnu.no message
```

Decrypt Decryption is simple, and requires only that the user have the private key that matches the public key used to encrypt the message sent.

```
$ gpg --output message --decrypt message.gpg
```

Task 4: Messages. Exchange a message which is encrypted *and* signed with one of your group mates: verify among yourselves that this actually works as it should.

Milestone 1: Create a plaintext file that contains your group number and all your names. Sign that file with your own private key, encrypt it with the public key of `ttm4135@item.ntnu.no` and send it to that email address. The TA will use that email to verify that you are able to encrypt and sign messages correctly. Make sure that you are sending the email from the same email address as is associated with the key that signed the message and that this key has been uploaded to the key server and has not been revoked previously.

2.6.1 PGP Certificate Revocation

The ability to revoke (declare the invalidity of) a key is important in a key management system: if your private key becomes known to the world, your public key should not be used anymore. PGP utilizes revocation certificates: certificates based on the same secret key that can be used to remove the associated public key from service.

Revocation certificates should be generated once a keypair has been created, and should be securely stored but can be generated as long as the secret key is available. To generate a revocation certificate for our previously generated key and store it as `revoke.asc`:

```
$ gpg --output revoke.asc --gen-revoke oladnorm@stud.ntnu.no
```

To revoke the key on the personal keyring, one simply needs to import the revocation key into the keyring:

```
  $  gpg --import revoke.asc
  ...
  $  gpg --list-keys
--------------------------------
pub   rsa2048 2019-02-27 [SC] [revoked: 2020-03-06]
uid                   Ola Nordmann <olanordm@stud.ntnu.no>
```

In order to declare to other users that the key is no longer valid, the revocation certificate is submitted to the relevant keyserver(s), at which point users who update their keyrings against a server will be notified that the key is no longer valid.

```
$ gpg --send-key --keyserver keys.openpgp.org \
F76DE997E3D4C03AB01D759333E694E557EEB343
```

Task 5: Generate a revocation certificate and store it in a safe place.

2.7 Comparing signature efficiency in PGP

There is a choice of using either RSA or DSA signatures with PGP, but of course you need to have the appropriate keys.

Task 6:

1. Generate two new signing keys, one for RSA signing with sizes 2048, and one for DSA signing also with sizes 2048 bits. Do not upload these keys to any server, they are just for experimental purposes.

2. Sign and verify both long (say 1MB) and short (say 1kB) messages for the different algorithms and keys

Question 2: How many bytes does PGP use to store the private signing and public verification keys for each of the two signature types?

Question 3: How many bytes does PGP use to store the signatures in each of the four cases (both long and short messages)?

Question 4: How long, on average, does PGP uses for signature generation and verification in each of the four cases?

Question 5: Discuss your results for the above three measurements. In particular, how well do they correspond to what you expect from what was studied in the lectures? Include an explanation of how the different elements of the keys are stored (such as modulus, exponents, generators).

2.8 Email with PGP

Email can be secured using PGP: this is one of the more common applications of PGP. Doing so requires manually using GPG like above, or an extension to a regular mail client. Of course the latter option is preferable for usability reasons.

Examples of tools that make using PGP easier are GPG Suite (for Apple Mail), Enigmail [3] (for Thunderbird, Postbox) and Mailvelope [5] (for Chrome, Firefox).

When using PGP in real life, keep in mind that that metadata (to, from, etc) is not protected, and that both participants must use PGP in order to have an encrypted conversation. Message signatures do not require all participants to use PGP, and can allow for an added measure of security as to the identity of the sender.

Task 7: Upload your key to the key server at: https://keys.openpgp.org/

Question 6: What assurances does someone receive who downloads your public key from the server where you have uploaded it? What is the difference between the role of the certification authority in X509 and the key server in PGP, with regard to the security guarantees they give?

3 Part II: Installing a web server and obtaining a certificate

Before you can do the steps in this lab, you will have to activate a virtual machine on the NTNU SkyHiGh system. Login at `https://skyhigh.iik.ntnu.no` using your normal NTNU username and password, and perform the following steps:

Task 8:

1. Under *Compute* → *Key Pairs*, click "Import public key" and add your personal RSA key, or create a new one using the system. This you will later use to authenticate to the server.
2. Under *Compute* → *Instances*, click "Launch Instance" to create the server you will use. Give it a name, add it to the Network `ttm4135_lab` and make sure to link it to the key pair you create in the previous step. For 'source' you can use a version of the Image 'Ubuntu Server' (not 20.x). You can choose any of the available flavours. Click the big blue button.
3. Go to *Network* → *Security Groups* and edit the default group. Add a rule for HTTP, for HTTPS and for SSH. You don't have to change any fields, just use "Add Rule", pick the correct one from the dropdown and click "Add".
4. Go to *Network* → *Floating IPs*. You'll have one IP address here. Click "Associate" and link it to the VM you created in the second step.

Remark

Do not 'release' the one IP-address you have under *Floating IPs*, since this one is linked to your group number. If you release it you might get a different one, and we will have to update our record to make sure your group URL still works.

Note: To access the SkyHiGh web interface when you are not on campus, use the VPN provided by NTNU.

What just happened?

In the previous task you have created a virtual machine *in the SkyHiGh system. This VM behaves just like a real computer on a real network — except both the computer and the network are simulated somewhere on NTNU's server cluster in Gjøvik. You have specified what your VM looks like (2), given it a network address so you can reach it (4) and you've told the system that some ports in the firewall should be open (3) to make sure you can actually access the server both via the web browser and the command line. We will use the keypair to authenticate: that means you can sign a message with your private key, and the system will use the public key you uploaded to verify that it's really you.*

The server domain you were assigned is present in the DNS record to give you a subdomain, so these are now your group's details:

Login details:		
Server IP:	`129.241.xxx.xxx`	(from step 4)
Server domain:	`ttm4135-v23-gruppeN.public.skyhigh.iik.ntnu.no`	(`N` the group number)
Server username:	`ubuntu`	

Log in using the command terminal on your laptop or on the computers in the computer lab. Use the correct IP and change the path to the private key if necessary:

```
$ ssh ubuntu@129.241.xxx.xxx -i ~/.ssh/id_rsa
```

3.1 Installing the Apache webserver

Now that you have a computer running, we will install Apache. Apache is a software suite that configures your server to host websites.

```
$ sudo apt-get install apache2
```

The apt-get command downloads apache, verifies that the package is signed and runs the installation.

Task 9: Install Apache using the instructions above.

Question 7: Who typically signs a software release like Apache? What do you gain by verifying such a signature?

If all is well, you can now navigate to either Server IP address or your Server domain in a web browser, and you will get a page that says "It works!"

3.2 The webserver configuration

The main configuration file for Apache is called `apache2.conf`, and found in the `/etc/apache2` directory of your server. Take a look at it and try to figure out the purpose of some of the different commands inside it. In that directory you will also find a file called `ports.conf`, which will tell you which ports the server is currently using. The default port for an unencrypted HTTP-connection is port 80. For an encrypted HTTPS-connection this is port 443.

Whenever you change something in the config file you need to restart Apache by using:

```
$ sudo apachectl restart
```

If you want to turn if off and make your site unaccessible, you can do so by using:

```
$ sudo apachectl stop
```

Remark: We're not using an encrypted connection, so you should for now use `http` and not `https`.

Remark: By default, users are able to view a list of the content of all directories via the browser. You can add `Options -Indexes` to your config file to disable that list — not doing this is bad security practice.

3.3 Certificates

3.3.1 Authority versus trust

In the previous part of the assignment we learned about how we can verify keys in a web of trust where users make personal decisions on the truthfulness of keys. For the next part of the lab, we will work with a different type of trust distribution, namely a trust hierarchy.

When you use your web browser to contact a web server over a secure connection, your browser validates the identity of the server by checking that it actually possesses a valid certificate. These certificates are issued by a *certificate authority (CA)*, which is in turn certified by a higher-level authority. On the top of this certificate tree or certificate hierarchy we find the *root CA*, whose keys are hardcoded into your web browser when you downloaded and installed it.

Although the technical workings are similar, we use the term *certificate* instead of *key* to signify that we are using a hierarchy instead. Intuitively this makes sense: certificates are issued by an authority in real life, too.

Buying a certificate and having it signed by a CA typically used to cost a fee, which is why you had so many websites using unencrypted `http` instead of `https` just a few years ago. After some big events like the Edward Snowden revelations made the public more aware of the issues that unencrypted connections pose, a few nonprofit organizations started the *Let's Encrypt* project in 2016: Let's Encrypt is a CA which provides certificates for free to everyone. It is also the CA we will use in this course.

3.3.2 Setting up a certificate

We will use the Let's Encrypt CA to provide us with certificates to authenticate our web server. After requesting and downloading that certificate, we should tell Apache to turn on the SSL protocol, so users can connect using HTTPS. Of course, Apache should also know where to find our certificate.

By now this entire process can be done in a mostly automated way using the *Certbot* tool provided by the Electronic Frontier Foundation. When running, Certbot will request your info to put on the certificate, then place a file on your webserver to verify that you actually control the domain, and then upload the certificates onto your system.

> **Task 10:** Obtain and install a certificate using the instructions on `https://certbot.eff.org/instructions`.

However, you can still choose to only obtain a certificate from Certbot and install them manually. In this case, follow the instructions so you obtain a certificate chain file and a private key. Both files have the extension `.pem`.

You do not have to move them, but we do need to tell Apache where they are by editing the config file.

We'll add three lines: the first line will enable SSL. Edit the second and third line to reflect the actual location of certificate and key with the locations Certbot gave you. Example:

```
SSLEngine on
SSLCertificateFile /long/path/given/to/you/by/certbot/fullchain.pem
SSLCertificateKeyFile /long/path/given/to/you/by/certbot/privkey.pem
```

If you then reboot Apache, you should be able to access the web site through `https` and obtain the little green padlock in your web browser.

Task 11: Obtain and install the certificate, edit the config file, restart Apache and verify that you can now connect to your site via a secure `https` connection.

Question 8: Why did you obtain a certificate from Let's Encrypt instead of generating a self-signed one yourself?

Question 9: What does Let's Encrypt have to verify, and how do they do it, before they issue a certificate?

Question 10: What steps does your browser take when verifying the authenticity of a web page served over `https`? Give a high-level answer.

Question 11: Have a look at the screenshot in Figure 1. What does the string `TLS_DHE_RSA_WITH_AES_128_CBC_SHA` in the bottom left say about the encryption? Address all eight parts of the string.

Question 12: The screenshot in Figure 1 is obviously from a few years ago. Do you think the encryption specified by this string is still secure right now? Motivate your answer.

Task 12: Check the rating of your server using SSL Labs: https://www.ssllabs.com/ssltest/. Make any changes necessary to obtain an A+ rating. (Hint: you probably need to enable HSTS (strict transport security) before this works.)
Note: For Part III, you will need to enable the ciphersuite `TLS_RSA_WITH_AES_128_CBC_SHA` in your server. Make sure you complete Milestone 2 before you enable this and move on to Part III.

Milestone 2: Have a TA verify that your site has an A+ rating. Send a notification to `ttm4135@item.ntnu.no` to have this checked.

Question 13: What restrictions on server TLS versions and ciphersuites are necessary in order to obtain an A rating at the SSL Labs site? Why do the majority of popular web servers not implement these restrictions?

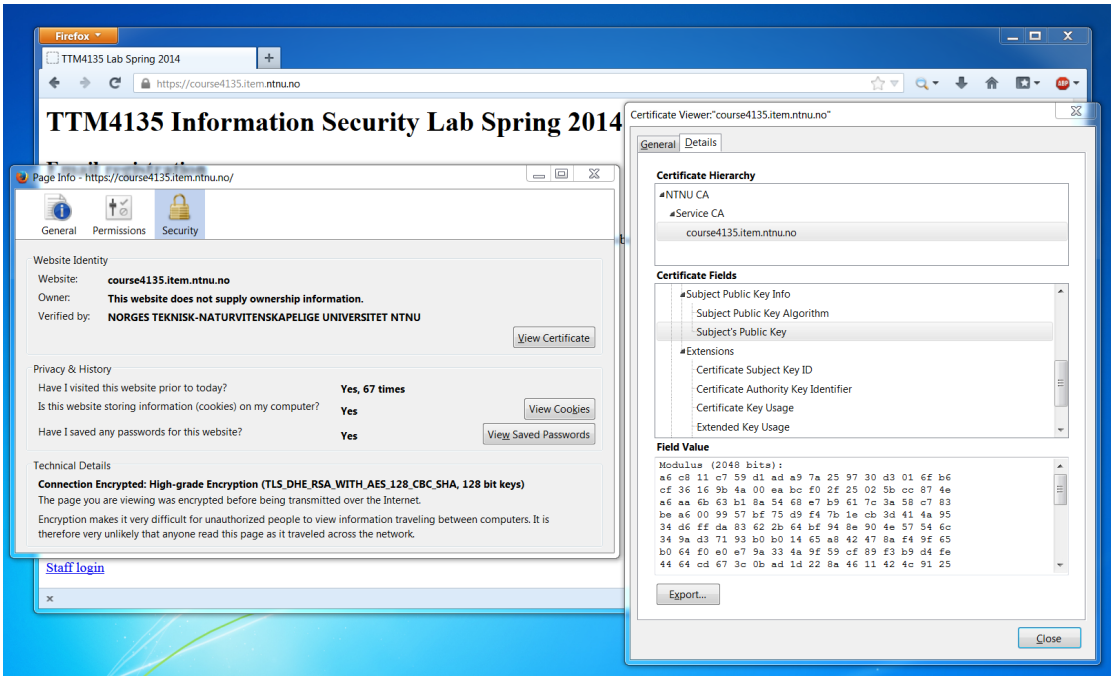

Figure 1: Information about a TLS connection to the former course site.

4 Part III: Examining TLS Traffic

In this final part you capture TLS traffic to see how it is structured and how it depends on the configuration in the server and client. You will also decrypt traffic directly from the intercepted traffic using the pre-master secret when using TLS with RSA (key transport) key exchange.

Task 13: Set up your web server so that it delivers a simple HTML file to a client.

Protecting the confidentiality of user application data is one fundamental property of a secure TLS connection. Many communication networks are easy to tap so we should assume that an attacker is able to observe the traffic. In addition, servers and clients may be compromised at certain times and keys, especially long-terms keys, may become known to the attacker.

In the following task you will capture traffic which you send yourself from a client web browser to your web server and analyse it to see the details of the TLS keys and the ciphertext of application data.

You should use the open-source network protocol analyzer Wireshark [1] to capture the traffic on the TLS connection that you set up. Wireshark is a powerful tool and has built-in support to allow you to filter out TLS traffic from your other network traffic. You do not need to use any complex features of Wireshark, but if you have not used it before then you should allow some time to get used to the interface. Wireshark comes with detailed documentation and support features.

The Illustrated TLS Connection [2] may be helpful in identifying some protocol fields. Note that there are two different versions of this resource: be sure that you use the one for TLS 1.2, not the one for TLS 1.3. However, you should also note that Illustrated TLS Connection shows a TLS 1.2 handshake with Diffie–Hellman, while we examine a connection using RSA

encryption in this assignment.

Task 14: In this task you will capture traffic from a TLS 1.2 session between a browser and your server. You can check that the traffic follows the TLS protocol specification.

1. Change the acceptable ciphersuites on a client web browser so that the ciphersuite `TLS_RSA_WITH_AES_128_CBC_SHA` is negotiated between the browser and your server. If you use the Firefox browser you can edit the acceptable cipher suites by typing `about:config` in the navigation bar and searching for "ssl". You may also have to disable TLS 1.3 by setting the maximum version to be "3". Note that you will get a warning about changing the configuration – if you are not confident about this you should not make the changes on your normal browser.
2. Use Wireshark [1] to capture the traffic in a session between the prepared client and your server, including both the handshake and application data. The application data should include a simple text string sent by you from the client to the server. Wireshark can be used to filter out the TLS traffic and it will also recognise the different TLS message types. You can also match the bytes with the examples used in the Illustrated TLS Connection.

Question 14: What are the values of the client and server nonces used in the handshake? How many bytes are they? Is this what you expect from the TLS 1.2 specification?

What is the value of the encrypted pre-master secret in the client key exchange field sent to the server? Is this the size that you would expect given the public key of your server?

Milestone 3: Demonstrate to a TA that you are able to set up a connection with your server with ciphersuite `TLS_RSA_WITH_AES_128_CBC_SHA`. For example, take a screenshot of the security info in Firefox for such a connection and send it to `ttm4135@item.ntnu.no`.

Task 15: Continue from Task 14 and now find the (pre)-master secret from which the traffic encryption keys are derived.

1. Find either the pre-master secret or the master secret used in the session. You are suggested to use the instructions on the Wireshark wiki to help you do to this: https://gitlab.com/wireshark/wireshark/-/wikis/TLS.
2. Decrypt the traffic using Wireshark and check that it is the same as was sent from the client.

Question 15: What was the value of the (pre)-master secret in the session that you captured? Write down the value, and whether it is the pre-master secret, or the master secret. Is this the size (in bytes) that you expect from the TLS specification? Explain how Task 15 shows that this session does not have forward secrecy.

References

[1] Wireshark download page. www.wireshark.org.

[2] Michael Driscoll. The illustrated TLS connection. https://tls.ulfheim.net.

[3] The Enigmail Project. Enigmail: A simple interface for OpenPGP email securit y. https://www.enigmail.net/home/index.php.

[4] Simon Tatham, Owen Dunn, Ben Harris, and Jacob Nevins. PuTTY download page. http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html.

[5] the Mailvelope Project. Mailvelope. https://www.mailvelope.com.

A Collaboration tools

Some students have previously struggled with collaborating while not being able to work on the same computer. The following is just a suggestion of tools one can use to ease this communication.

Video Chats: Zoom, Facebook Messenger, Discord, Slack, Microsoft Teams, Skype, TeamViewer. All of these (except for Messenger) also allow you to share your screens with each other, this might help if you choose to work on a single terminal. With TeamViewer, you could also allow others to input (like type or move the mouse).

Shared terminal: Teleconsole: Teleconsole is basically TeamViewer but only for the terminal. One person could run Teleconsole, then SSH into your server. All team members would then be able to join that one terminal and be able to type and edit.

Tmux: Tmux is TerminalMultiplexer, meaning it allows you to create virtual terminals. This only needs to be installed on the server. One person can then ssh into the server and write `tmux`, a new terminal is now created. All other members could then also SSH into the server and write `tmux attach` and would join the same terminal. All users are able to write to the console and run commands just like Teleconsole. But you can also create multiple 'tabs' by pressing CTRL-B then C for create. This will create a new tab. You can switch tabs by typing CTRL-B then <tab number>, e.g. 1. All tabs are listed at the bottom. To close a tab, just write the `exit` command inside the tab. Tmux closes when all tabs are closed. To leave tmux, but without closing the tabs (e.g. one person wants to leave) just press CTRL-B and D for Detach. Even if all members detach from tmux, it still continues to run in the background. You can reattach by running `tmux attach`, or simply `tmux a`. If you write just `tmux` while another tmux instance is running, then a new totally separate version of tmux is started. All tmux windows can be listed by `tmux ls`. You can then choose which tmux session you want to attach to with `tmux a -t <number>` where `-t` is for target and the number is the one first listed when running `tmux ls`.

B Apache Configs

Historically there have been quite a few questions related to Part II of the lab, especially with students not able to set flags for their configs and so on. This section is mainly aimed for those who want a small introduction into how Apache config files are structured and read. If you have completed Part II, then you probably don't need this, but it might cast some useful light over Apache nontheless. Depending on which version of Apache along with which OS you select for your servers, Apache might be installed as `apach`, `apache2`, `http` or `httpd`. In this section we will use just `apache2` from now on. All the Apache configs are should be inside `/etc/apache2` on your servers. Here you will find the main config called `pache2.conf`. This file is well documented, and I recommend you to read through it, at least the first part, where it describe most of the layout. The most important take-aways are the config layout. The first and only config to be read is the `apache2.conf` file. The file is read sequentially, meaning that if a value is set twice, then only the last value is used. There are also two keywords you can use to include additional config files: `Include` and `IncludeOptional`. If you scroll down through the main config file, you will notice some extra config files are included, in this order:

```
/etc/apache2/
|-- apache2.conf
|       `-- ports.conf
|-- mods-enabled
|       |-- *.load
|       `-- *.conf
|-- conf-enabled
|       `-- *.conf
`-- sites-enabled
        `-- *.conf
```

When apache includes a config, it is equivalent to copy-pasting the same config into the main config. So if two configs sets the same value, then only the last value is used. From the diagram above, we can see that inside the mods-enabled folder, all files ending in `.load` or `.conf` are included in our main config. We also notises that all files inside the conf-enabled folder, along with sites-enabled are also included. Also notice that sites-enabled are the last folder to be included, meaning these files are read last. Files are also read alphabetically. Meaning if you have a config named `A.conf` and another `B.conf` inside one of the included folders, then `A.conf` will be read before `B.conf`. You can see this if you scroll all the way down in your `apache2.conf` file, there should be two lines like:

```
--
# Include generic snippets of statements
IncludeOptional conf-enabled/*.conf
# Include the virtual host configurations:
IncludeOptional sites-enabled/*.conf
--
```

Also, if you put any lines after these, then those values will used instead of the ones inside the included config folders.

VHosts: VirtualHosts, aka VHosts, are as the name describes "virtual hosts" for the server. The structure is a bit similar to html/xml where you open a VHost tag, and only the values

inside the VHost tag are used for that VHost. VHost options also do not change values for other VHosts. For instance, you could have a config like `my-sites.conf` inside your `sites-enabled` config, including these two VHosts: `my-site.conf`:

```
---
<VirtualHost _default_:443>
   ServerName a.mysite.com
   DocumentRoot /var/www/html-site-a/
   ... more values ...
</VirtualHost>
<VirtualHost _default_:443>
   ServerName b.mysite.com
   DocumentRoot /var/www/html-site-b/
   ... more values ...
</VirtualHost>
---
```

Here we see the `ServerName` is set to `a.mysite.com` for one vhost, and `b.mysite.com` for the other vhost. If both domains point to the same ip, then we could serve different content and have different settings depending on the domain. Here we see that `DocumentRoot` is different, meaning we are serving files from two different directories, depending on which site you are visiting. We could also have split the two VHosts into two different files if we wanted. If you visit your `sites-enabled` on your server in the lab after setting up the certificate for the site with lets encrypt, and you ran certbot with the `-apache` flag, then lets encrypt might have created two configs for you in this folder. One of the files is probably an http to https redirect, usually called something like `*-le-redirect.conf`, and another called something. This other config is the main config you want to edit. You can also identify it as it has a VHost listening to port 443 (the default HTTPS port), like:

```
--
<VirtualHost _default_:443>
--
```

Inside this config, you will probably also find an include to a lets encrypt ssl config file. You do not need to change this config, as you can simply just override the values by entering the same values *after* you included the config. E.g. if you want to override which ciphers are used, just write the `SSLProtocols` keyword after the `include` of the letsencrypt `ssl.conf`.