TLS and SSH

Laboratory for the class "Cybersecurity" (01UDR)
Politecnico di Torino – AA 2021/22
Prof. Antonio Lioy

prepared by:
Diana Berbecaru (diana.berbecaru@polito.it)
Andrea Atzeni (andrea.atzeni@polito.it)

v. 2.0 (08/10/2021)

Contents

Purpose of this laboratory

In this laboratory, you will perform exercises aimed to create secure (communication) channels among two nodes, to evaluate their security features, and to measure the performance of the system when the security features are modified. In practice, the exercises use two famous security protocols: TLS (Transport Layer Security) and SSH.

For this purpose, we will use the OpenSSL library (and some associated tools) offering in depth support for the configuration and analysis of SSL/TLS channels. Some of the exercises proposed use OpenSSL command line programs that provide various cryptographic functions via a specific shell, which you can start with the following command:

```
openssl command [ command_opts ] [ command_args ]
```

In particular, the command `s_client` implements the functionality of an SSL/TLS client (`man s_client`). The command `s_server` implements instead the functionality of a simple SSL/TLS server (`man s_server`).

To experiment SSH protocol we will use the OpenSSH command line client

Additional software tools

The tools listed below will be used as well throughout this laboratory:

Wireshark - open source tool (having a user-friendly graphical interface) which allows to capture network traffic. Available for Linux and Win32.
Home page = https://www.wireshark.org/

Additional (useful) commands

Some exercises may require to exchange messages between two computers. Consequently, you will have to start the `ssh` server:

```
systemctl start ssh
```

To copy a file (e.g. `mytest`) from the `ssh` client machine into the `root` directory on the machine running `ssh` server, you can use the command:

```
scp mytest root@IP_address_ssh_server:/root
```

Insert the password 'toor' when asked.

The exercise will require also to use the Apache web server. To start the Apache server use the command:

```
systemctl start apache2
```

The exercise will require also to use an SMTP mail server. To start it use the command:

To start, stop and restart the mail server, use the command:

```
systemctl start exim4
```

TLS

The OpenSSL library implements a simple SSL/TLS client and server, which can be used for testing the SSL/TLS protocol very easily.
To check the syntax of the related OpenSSL commands, we strongly suggest you to use the man pages, by running:

```
man s_client
```

```
man s_server
```

The OpenSSL command used to start an SSL/TLS client program is `s_client`, whose syntax is given below:

```
openssl s_client [command_opts]
```

For simplicity, we have selected below (only) some possible options, check out the man pages for the description of the other options as you will need them to perform the proposed exercise:

```
openssl s_client [-connect host:port] [-state] [-showcert] [-CAfile file_cert]
[-cipher cipherlist] [-reconnect]
```

where:

- `-connect` *host:port* specifies the host and optional port to connect to. If host and port are not specified then an attempt is made to connect to the local host on port 4433.
- `-state` prints out the SSL session states.
- `-showcerts` displays the whole server certificate chain: normally only the server certificate itself is displayed.
- `-CAfile` *file* indicates the file containing trusted certificates to use during server authentication and to use when attempting to build the client certificate chain.
- `-cipher` *cipherlist* allows to specify the cipher list sent by the client in the ClientHello message of the Handshake protocol. Although the server determines which cipher suite is used it should take the first supported cipher in the list sent by the client. See the OpenSSL `ciphers` command for more information.
- `-reconnect` allows the client to reconnect to the same server 5 times using the same session ID.

The OpenSSL command used to start an SSL/TLS server program is `s_server`, whose syntax is given below:

```
openssl s_server [command_opts]
```

For simplicity, we have selected below (only) some possible options, check out the man pages for the description of the other options as you will need them to perform the proposed exercise:

```
openssl s_server [-www] [-no_dhe] [-key server_pkey.pem] [-cert server_cert.pem]
[-CAfile file_cert] [-{vV}erify depth] [-cipher ciphersuite_list]
```

where:

- `-www` sends a status message back to the client when it connects. This includes lots of information about the ciphers used and various session parameters. The output is in HTML format so this option will normally be used with a web browser.
- `-key` *server_pkey.pem* indicates that *server_pkey.pem* contains the private key of the server.
- `-cert` *server_cert.pem* indicates that *server_cert.pem* contains the certificate of the server.
- `-CAfile` *file* indicates the file containing trusted certificates to use during client authentication and to use when attempting to build the server certificate chain. The list is also used in the list of acceptable client CAs passed to the client when a certificate is requested.

- `-verify` *depth*, `-Verify` *depth* indicates the verify depth to use. This specifies the maximum length of the client certificate chain and makes the server request a certificate from the client. With the -verify option a certificate is requested but the client does not have to send one, with the -Verify option the client must supply a certificate or an error occurs.
- `-cipher` *cipherlist*, this option allows modification of the cipher list used by the server. When the client sends a list of supported ciphers, the first client cipher also included in the server list is used. Because the client specifies the preference order, the order of the server cipherlist is irrelevant. See the OpenSSL `ciphers` command for more information.

1 TLS

1.1 Setting up a TLS channel

Try to connect with your browser on your physical machine to an SSL/TLS server, for example https://mail.polito.it.

In the browser, click on the lock in the navigation bar and check out some details of the TLS connection you have just established. Which TLS version was used in the TLS connection to the above server?

Analyse the content of the X.509 certificate sent by the server: which fields are used for the identification of the server? Which is the certification path? Which algorithms have been negotiated and used for protecting the data transferred?

1.2 Configuration of a TLS server

Try now to configure a TLS server. What do you need in the first place?

Prerequisites. For this purpose, you need a certificate for the TLS server. We have issued a server certificate by exploiting the demoCA in OpenSSL. The certificate contains the name "Server" in the field "Common Name" (the other X.509 fields have been set to "IT" for "Country", "Some-State" for "State" and "Polito" for "Organization). The password used to protect the private key is: "ciao".

In case you cannot access the material provided for this laboratory or if you simply want to recreate it on your own, we remind you the OpenSSL commands that you can use to generate a certificate for the TLS server:

1. create a test CA by exploiting OpenSSL:

   ```
   /usr/lib/ssl/misc/CA.pl -newca
   ```

   When asked, insert a password to protect the private key, e.g. "ciao". Remember the values you have inserted for "Country", "State", and "Organization", as you will have to use the same values in the following steps.

2. create a certificate request for the TLS server:

   ```
   openssl req -new -keyout server_pkey.pem -out server_creq.pem
   ```

3. issue the new certificate for the TLS server:

   ```
   openssl ca -in server_creq.pem -out server_cert.pem
   ```

1.2.1 Analysing the TLS 1.2 handshake messages

Start the OpenSSL `s_server` with the following command:

```
openssl s_server -www -key server_pkey.pem -cert server_cert.pem
```

By default, this server listens on port `4433/tcp`.

Try to connect with `s_client` and check out the result:

```
openssl s_client -connect 127.0.0.1:4433 -state -showcerts -CAfile
demoCA/cacert.pem -tls1_2
```

In a separate window, open a terminal (emulator) and start Wireshark:

```
wireshark
```

Select "eth0" and "Loopback:lo" and then click on the blue symbol (placed in the up, left position, under "File") corresponding to "Start capturing packets".

Run the `s_client` with the command above and intercept the TLS handshake communication messages exchanged between the `s_client` and the `s_server`. You should be able to view the captured messages exchanged between the client and the server in a dedicated window in Wireshark.

Now respond at the following questions:

How many RTTs do you see? Write them down in the following box:

→

Which TLS handshake messages are exchanged? Write them down in the following box:

→

Which ciphersuite has been negotiated?

→

Does it provide forward secrecy (if yes, explain briefly why in the following box)?

→

What do you see in the "Extensions" of the Client Hello message?

→

What is the purpose of the "New Session Ticket"?

→

What is the "Encrypted Handshake Message" and which is its purpose?

→

1.2.2 Session resumption

Resumption refers to starting a new connection based on the session details from a previous connection.

Which techniques are used for session resumption in TLS and which is the difference among them?

→

Execute the following `s_client` command, which opens up several consecutive connections by exploiting the session resumption:

```
openssl s_client -connect 127.0.0.1:4433 -state -CAfile demoCA/cacert.pem
-reconnect
```

In this mode, s_client will connect to the target server six times; it will create a new session on the first connection, then try to reuse the same session in the subsequent five connections.

Which parameters of the TLS session remain unchanged in successive connections and which ones change instead? (verify)

→

Now, stop the server and re-start it with the following command:

```
openssl s_server -www -key server_pkey.pem -cert server_cert.pem -no_ticket
-no_cache
```

Execute again the following `s_client` command, which opens up several consecutive connections by exploiting the session resumption:

```
openssl s_client -connect 127.0.0.1:4433 -state -CAfile demoCA/cacert.pem -tls1_2
-reconnect
```

Check out again the output on the `s_client`. What do you notice?

→

1.2.3 Analysing the TLS 1.3 handshake messages

Now, stop the server and re-start it with the following command:

```
openssl s_server -www -key server_pkey.pem -cert server_cert.pem -tls1_3
```

Try to connect with `s_client` and check out the result:

```
openssl s_client -connect 127.0.0.1:4433 -state -showcerts -CAfile
demoCA/cacert.pem -tls1_3
```

In a separate window, open a terminal (emulator) and start Wireshark:

```
wireshark
```

Select "eth0" and "Loopback:lo" and then click on the blu symbol (places up, left position, under "File") corresponding to "Start capturing packets".

Run the `s_client` with the command above and intercept the TLS handshake communication messages exchanged between the `s_client` and the `s_server`. You should be able to view the captured messages in a dedicated window in Wireshark.

Now respond at the following questions:

How many RTTs do you see? Write them down in the following box:

→

Which ciphersuite has been negotiated?

→

1.2.4 Client authentication in TLS

Now try to configure an SSL/TLS server with client authentication. What do you need in the first place?

Prerequisites In case you cannot access the material provided for this laboratory or if you simply want to recreate it on your own, we remind you can generate a client certificate with OpenSSL in the following way:

1. create a certificate request for the client certificate:
   ```
   openssl req -new -keyout client_pkey.pem -out client_creq.pem
   ```
2. issue a new certificate:
   ```
   openssl ca -in client_creq.pem -out client_cert.pem
   ```

Configure now `s_server` so that to request client authentication during the handshake phase of the SSL/TLS protocol:

```
openssl s_server -www -key server_pkey.pem -cert server_cert.pem -CAfile
demoCA/cacert.pem -Verify 0
```

Run the `s_client` command necessary to connect to the `s_server` started above:

```
openssl s_client -connect 127.0.0.1:4433 -state -showcerts -CAfile
demoCA/cacert.pem -cert client_cert.pem -key client_pkey.pem -tls1_2
```

By using wireshark, identify the TLS handshake messages that have changed in the handshake phase of the SSL/TLS protocol. Write them down in the following box:

→

In which TLS handshake message is placed the information about the CA accepted by the server (for the client certificates)?

→

1.3 Enabling TLS in Apache Web Server

The scope of this exercise is to allow you to configure the Apache HTTP server with support for TLS.

Prerequisites. First of all, we need a certificate for the HTTP server. We have issued a server certificate by the demoCA used in the previous exercises. The certificate contains the DNS name "myexample.com" in the field "Common Name". The password used to protect the private key is: "ciao". You need to configure the DNS name in the `/etc/hosts`. Open the file and insert the following association:

```
127.0.0.2 myexample.com
```

For simplicity, we provide also the Apache configuration file `default-ssl.conf` used in tests. You can find all this stuff in the archive `CYB_lab01_support.zip` available in the same folder of this text.

At this point, you can proceed as follows.

1. Activate the Apache web server, with server authentication:
   - run the following command to enable the SSL module of Apache:
     ```
     a2enmod ssl
     ```
   - use the following command to enable the Apache SSL site:
     ```
     a2ensite default-ssl
     ```
   - copy the files `myexample_cert.pem` and `demoCA/cacert.pem` in the directory `/etc/ssl/certs` and the file `myexample_pkey.pem` in the folder `/etc/ssl/private`. Note: `/etc/ssl` is the (default) Apache directory for the SSL configuration.
   - modify the following directives in the file `/etc/apache2/sites-enabled/default-ssl.conf`:
     ```
     SSLCertificateFile /etc/ssl/certs/myexample_cert.pem
     SSLCertificateKeyFile /etc/ssl/private/myexample_pkey.pem
     ```
     ```
     SSLCACertificateFile /etc/ssl/certs/cacert.pem
     ```
     ```
     SSLVerifyClient none
     ```
     Which certificates and key file have been configured?
   - restart the Apache web server, by using the command:
     ```
     systemctl restart apache2
     ```
     .
   - for testing purposes, you try to connect to the Apache web server with the OpenSSL s_client, by using the command:

```
openssl s_client -connect myexample.com:443
```

Check out the output of the s_client: which protocol version has been used?

Now connect again to the Apache web server with the OpenSSL s_client, by using the command:

```
openssl s_client -connect myexample.com:443 -tls1_2
```

Which are the differences you observe on the output of the s_client with respect to the output you have obtained in the previous command?

```
→
```

- next, try to connect with the Firefox browser to (your own) Apache web server. Open Firefox and insert in the address bar:

```
https://myexample.com:443
```

You should be able to see the "Apache2 Debian Default Page".

1.4 Enabling client authentication

To enable the client authentication, you need to perform two steps:

- import the (client) certificate in the browser. Additionally, if you use the OpenSSL s_client for testing the connection to the TLS web server, you need to specify the client certificate among the options (otherwise you'll see an error);
- configure the Apache server to ask for client authentication.

1.4.1 Configure the TLS server for client authentication

In the Apache server, you enable the client authentication in TLS by changing in the file

`/etc/apache2/sites-enabled/default-ssl.conf`

the directive:

```
SSLVerifyClient none
```

with the directives:

```
SSLVerifyClient require
SSLVerifyDepth 10
```

Next, restart Apache with the command:

```
systemctl restart apache2
```

For testing purposes, try to connect with the following command to the Apache web server:

```
openssl s_client -connect myexample.com:443 -state -showcerts -CAfile
demoCA/cacert.pem -cert client_cert.pem -key client_pkey.pem
```

You should be able to see the output providing information on the TLS connection that has been just established.

1.4.2 Importing the client certificate in the browser

To import the client certificate in the browser, you need the PKCS#12 file `client_cert.p12`.

It was generated by with the following command:

```
openssl pkcs12 -export -in client_cert.pem -inkey client_pkey.pem -name
client_certificate -certfile demoCA/cacert.pem -out client_cert.p12
```

Then, to import it into the browser, for example in Firefox, you need to select "Preferences" → "Privacy & Security" →. In the "Security" are, under "Certificates" you need to click "View Certificates", then select "Your Certificates", click "Import" and then select the "client_cert.p12". When asked, you need to insert the password you have used to protect the private key and the export bag ("ciao" in the provided material.

Next, open a browser and insert in the address bar:

```
https://myexample.com
```

You should be asked to select a browser certificate (of the client). After choosing the client certificate, you should be able to see the "Apache2 Debian Default Page".

1.4.3 Enabling certificate revocation checking in Apache web server

In this exercise, you will see how to enable certificate revocation checking with CRL in the Apache web server.

Prerequisites You need a revoked client certificate and the corresponding CRL. We have generated both of them with the demoCA, in the file `bob_cert.pem` you find the revoked certificate, and in the file `mycrl.pem` you find the CRL.

In case you don't access the material provided for this laboratory or if you want to recreate it, we remind you the steps below so that you can re-create them on your own (if needed).

To generate a certificate (for Bob):

```
openssl req -new -keyout bob_pkey.pem -out bob_creq.pem
```

```
openssl ca -in bob_creq.pem -out bob_cert.pem
```

To generate a CRL:

```
openssl ca -revoke demoCA/newcerts/serial_number_of_Bob_certificate.pem
```

```
openssl ca -gencrl -out mycrl.pem
```

Next, you need to create a directory named `ssl.crl` in /etc/apache2/ssl.crl and copy the CRL file `mycrl.pem` in the directory `ssl.crl`.

Finally, you need to configure the Apache web server to check the CRLs.

In the `/etc/apache2/sites-enabled/default-ssl.conf`, insert the following directives:

```
SSLCARevocationPath /etc/apache2/ssl.crl/
SSLCARevocationFile /etc/apache2/ssl.crl/mycrl.pem
SSLCARevocationCheck chain
```

Next, restart the Apache web server with the command:

```
systemctl restart apache2
```

Try to connect with the following command to the Apache web server:

```
openssl s_client -connect myexample.com:443 -state -showcerts -CAfile
demoCA/cacert.pem -cert bob_cert.pem -key bob_pkey.pem -tls1_3
```

You should see an error: "SSL3 alert read:fatal:certificate revoked".

1.5 Performance measurement

The scope of this exercise is to measure up the times required to transfer data over a channel protected with SSL/TLS.

Try to download first a file of small size, over a channel with no TLS protection. For example, you can use:

```
wget http://myexample.com/
```

Next, try to download the same file over a TLS protected channel, e.g. by using the command:

```
wget --ca-certificate demoCA/cacert.pem https://myexample.com
--secure-protocol=TLSv1_3
```

What do you note?

→

Now, generate different files of various sizes, i.e. 10 kB, 100 kB, 1 MB, 10 MB and 100 MB. For this purpose, you can use the following command of OpenSSL:

```
openssl rand -out r.txt size_in_bytes
```

Copy them in the directory `/var/www/html/` and then restart Apache.

Download them with wget as above, and note down differences in the speed and time. Change the TLS protocol version and use different ciphers and perform the same evaluations as above.

2 SSH

2.1 Connecting through a secure channel

SSH protocol is based on asymmetric cryptography, but for a basic usage complex operation related to the private/public cryptography can run behind the scene, and allow users a simple and user-friendly experience

One basic tool that takes advantage of SSH is the OpenSSH client

```
man ssh
```

The command `ssh` is a secure replacement for the ancient `telnet` and `rsh` program for logging into a remote machine and obtain a command shell. Contrarily to those old programs, SSH provides secure encrypted communication between two untrusted hosts over an insecure network.

The basic usage is through a username/password authentication.

To experiment this basic usage, form a couple (let's say Alice and Bob), on two different VMs.

Now we want to grant Alice access to Bob's machine. For doing so, first of all Bob creates a new user named `alice` on his machine (`adduser alice` and associate to the user a password). After that, what operations should they perform to allow Alice the possibility of having a secure command shell on Bob's machine? Make your hypothesis and write them down!

→

Then check your guesses you can adopt the following command on the Bob machine, an SSH server must be up and running, check with:

```
[B] systemctl status ssh
```

Now, Alice and Bob cooperate to check OOB the fingerprint of the Bob host key.

First, Alice ask for Bob's host key sha256 fingerprint:

```
[A] ssh -o FingerprintHash=sha256 Bob_IP
```

BEWARE

Alice must wait before accepting the key and adding it in her permanent store until she has checked the correspondence with the one on Bob machine (e.g. by comparing the fingerprint).

Meanwhile, Bob can display it on the command line:

```
[B] ssh-keygen -l -E sha256 -f ssh_key_file
```

NOTE

By default, the SSH server keys are located in `/etc/ssh`, and in particular the default one for Kali is in `/etc/ssh/ssh_host_ecdsa_key`. You can change it modifying the ssh server configuration in `/etc/ssh/sshd_config`, particularly the `HostKey` directive

After that, Bob can send the sha256 fingerprint to Alice OOB (e.g. on a private chat among them, or through a physical meeting)

If the two fingerprints correspond, then Alice can store it permanently in her file system (answering yes to the previously suspended question)

NOTE

After the positive answer, the public key of Bob's ssh server will be stored in `$HOME/.ssh/known_hosts` file. If a public key would become no longer trusted, it can be simply removed from this file

.

Finally, Alice can connect to Bob's machine through the ssh client:

```
[A] ssh alice@Bob_IP
```

To better understand the exchange described before, before emitting the different operations, run `wireshark` and use it to familiarise with the ssh protocol and identify different parameters of the secure channel that has been created by ssh. In particular:

- identify the client and server software version;
- identify the available encryption and mac algorithms;
- identify the chosen algorithms;
- identify the password used by Alice to connect to the Bob's host.

```
→
```

NOTE

openssh allows for visualisation of the host key through ASCII art. This could be used to make more "human-friendly" the comparison of the fingerprint. In order to do so you can use the following commands:

```
[A] ssh -o VisualHostKey=yes -o FingerprintHash=sha256 Bob_IP
```

```
[B] ssh-keygen -l -v -E sha256 -f ssh_key_file
```

2.2 Passwordless access

We are going to repeat the same scenario (Alice wants to run a command shell on the Bob machine) but removing the need to send to Bob a password to authenticate. This will be substituted by using the Alice private key for authentication.

Since SSH protocol is based on asymmetric cryptography, Alice needs a key pair, a private key (for the client machine) and a public key (for the server machine). SSH allows this creation using the `ssh-keygen` tool to produce it. For example, it can create an RSA key with the following command:

```
[A] ssh-keygen -t rsa
```

Keep note of the key position (by default it is in the $HOME directory in the file $HOME/.ssh/id_rsa)

Have a look at the files in the mentioned directory, what can you find in there?

→

NOTE

If you emit again the same command as before (and do not change any default) and answer 'yes', what would be the result? In particular, would you still be capable of authenticating with the previous key?

→

After the creation of the key pair, Alice needs to move the public key to Bob's host, and she needs to place it in the `/.ssh/authorized_keys`

First, she copies the public key on Bob's host

```
[A] scp ~/.ssh/id_rsa.pub alice@Bob_IP:~/.ssh/id_rsa_alice.pub
```

Then, she opens securely a command shell on Bob's host (as seen previously) and then she appends the key with the following command

```
[B] cat ~/.ssh/id_rsa_alice.pub >> ~/.ssh/authorized_keys
```

And then can check if it works (to do so, she may exit from the command shell on Bob's machine, and then try to re-open one)

In modern ssh installation, it is possible to take advantage of the `ssh-copy-id` command, which copies the public key and stores it in the appropriate place in just one command

```
[A] ssh-copy-id Bob_IP
```

Why passwordless access improves overall security?

→

Do you know any remaining issue for this kind of configuration?

→

2.3 Tunnelling

SSH provides different forms of tunnelling, i.e. direct, local, and remote, to transport higher-layer protocols inside a secure channel.

2.3.1 Direct tunnelling

SSH can carry a higher layer protocol inside a secure channel. This is pretty intriguing in the case of the X11 protocol, which is a communication protocol adopted under Linux to transfer from one host to another window and graphic components. By itself, the X11 protocol provides no channel protection.

The purpose now is that Alice connects to the Bob machine, and then run a graphical application (e.g. a browser), and navigate on the internet, exploiting a web browser running on Bob machine.

First, Alice connect to the Bob machine, enabling the tunnelling of X11 protocol with the option `-X`

```
[A] ssh -X alice@Bob_IP
```

then, through the command shell on Bob machine run an available browser (like `firefox` or `google-chrome`) and so the browser will be executed on the Bob machine, but the graphic result will be transferred and displayed on the Alice machine through the X11 protocol, in a secure way since the X11 protocol will be and all the traffic transferred from Bob's machine to Alice's host (through the X11 protocol).

Alice now can navigate on the Internet.

Running Wireshark on the Bob machine, what kind of traffic do you expect to see?

→

2.3.2 Local tunnelling

Now we will establish a secure tunnel towards a mail server. For doing so we exploit the `exim4` SMTP server, `telnet`, a command-line tool that allows a bidirectional and interactive text-oriented communication facility over a non-protected channel, and of course SSH to create a SSH tunnel

So first of all Bob has to configure the SMTP server:

```
[B] dpkg-reconfigure exim4-config
```

then Bob selects the parameters in the following manner:

1. General type of mail configuration: Internet site; mail is sent and received directly using SMTP.
2. System mail name: kali
3. IP-addresses to listen on for incoming SMTP connections: *// leave blank (delete data if present)*

4. Other destinations for which mail is accepted: kali
5. Domains to relay mail for: *// leave blank (delete data if present)*
6. Machines to relay mail for: *// leave blank (delete data if present)*
7. Keep number of DNS-queries minimal (Dial-on-Demand) ?: No
8. Delivery method for local mail: mbox format in /var/mail
9. Split configuration into small files ? : No
10. Root and postmaster mail recipient: *// leave blank (delete data if present)*

Subsequently, he starts the server with the command:

```
systemctl start exim4
```

Then Alice can open an ssh tunnel to the ssh server

```
[A] ssh -L localhost:1234:Bob_IP:25 alice@Bob_IP
```

> **NOTE**
>
> the above command starts a listening port (1234) on the localhost, the connection attempted to this port, if originating from localhost, will be forwarded to the *Bob_IP* address (thanks to the `alice@`*Bob_IP* part of the command) and there will be forwarded to the *Bob_IP* to the port 25 (thanks to the *Bob_IP*`:25` part)

.

> **NOTE**
>
> as consequence of the command, Alice will open a command shell on Bob's host. This is not relevant for the exercise, but leave it open to permit to the tunnel to persist (e.g. open another command shell to test the tunnel)

Then, Bob sniff the network thanks to Wireshark on Bob's machine, while Alice can try to connect to the SMTP server directly

```
[A] telnet Bob_IP 25
```

and just type `HELP` (which is an SMTP command that lists the available SMTP command from the mail server perspective)

In another command shell, Alice take advantage of telnet to connect to the SMTP server through the SSH tunnel

```
[A] telnet localhost 1234
```

and again just type `HELP`

What can you notice on the sniffed traffic in Wireshark?

```
→
```

2.3.3 Remote tunnelling

Now we will establish a secure tunnel towards a web server. For doing so we exploit the Apache HTTP server, a browser, and of course SSH to create a secure tunnel.

Bob starts the HTTP server with the command:

```
[B] systemctl start apache2
```

Then Bob can open a tunnel to the SSH server

```
[B] ssh -R 8000:Bob_IP:80 bob@Bob_IP
```

.

However, in order to work, the SSH server must have the remote port forwarding enabled, and this is done by setting the `GatewayPorts` directive in the `/etc/ssh/sshd_config` file to `yes` (otherwise, it will forward only packets coming from localhost). Remember to restart the SSH server after the configuration change.

Also, Bob activates Wireshark to see the ongoing traffic.

Then, Alice can starts a web browser (e.g. Firefox) and then connect directly to the webserver (to Bob's host, on the default port 80) and then to the web server through the SSH tunnel.

Which differences can be appreciated in the network traffic?

→