Practica 4 – PPC

Amaury Fauvel

1/ Description of the solution :

The code of Practica 1 is almost the same, but now 4 executables are produced. 2 clients and 2 servers, each for HTTP and HTTPS. The HTTPS version uses the OpenSSL library for C++.

Client : takes server adress and port in argument. The user is asked to enter a file name, which will be requested to the server. Then the answer is displayed (header and content) and new cookies are displayed. Each client handles either HTTP or HTTPS but not both.
Both client have also been improved to be able to make requests to any server on the web (in practica 1, the requests were not recognized by other servers). The executables name are `client-http` and `client-https`

Server : takes port as argument. The HTTPS version loads its certificate from the file `mycert.pem` and then works the same as before. Now, the ports 80 (for HTTP) and 443 (for HTTPS) will be used if no port is specified (requires root permissions). The executables name are `server-http` and `server-https`

2/ How to use it :

If the OpenSSL development library is not installed, install it with :

```
sudo apt-get install libssl-dev
```

Then build with :

```
cmake --build cmake-build-debug
```

2.1/ client - server :

From `cmake-build-debug` , run :

```
./server-https 4433

./client-https localhost 4433

tcpdump -A -I any port 4433
```

From three different terminals.
From the client, type `testFiles/test.txt` You can see the answer header, and the content of the file. On the server side, you can see the name and size of the file.

In the tcp dumper, you can see that the packets are encrypted (iby trying with client-http and server-http we would see the request headers in plain text).

Each program can be ended with Ctrl+C.

2.2/ client – website :

Run `./client-https www.random.org 433` to connect to the random.org website (which has a well-known api for random number generation).
Request `randomness/` to see the source code of a web page.
Request `cgi-bin/randbyte?nbytes=32&format=h` to use their server API to generate random numbers.

This way, any web page can be requested using the good client (HTTP for port 80 and HTTPS for port 443).

2.3/ server – web browser :

Run `sudo ./server-https` to start the HTTPS server on port 443.

Open a web browser, open a console with F12 to display the network traffic, and type as a URL `https://localhost/testFiles/index.html` You will be able to see two requests (the html file and an image). You may have to handle a security warning, because the server's certificate wasn't delivered by a known certificate authority.
For more requests (including JavaScript HttpRequest), type as a URL :
`https://localhost/testFiles/game/game.html`
For a larger file, type as a URL :
`https://localhost/testFiles/test.pdf` (larger test files aren't provided, I tried up to a 1GB zip file)

2.4/ memory checks :

Any of these tests can be performed with the memory tool valgrind to notice that every allocated byte is freed accordingly.

Example :

```
sudo valgrind ./server-https

echo "/intl/en/about/" | valgrind ./client-https www.google.com 443
```

Even after a lot of requests, valgrind tells us that no memory leak is possible.

3/ Implementation :

Every communication function (sockets `read` and `write`, but also OpenSSL handshakes and context preparation) have been abstracted in the header file `CommMethods.h`.
This way, whether or not the preprocessor macroconstant `SECURE` is defined, the communication will use bacis BSP sockets or OpenSSL. All the source code is the same for HTTP and HTTPS and understandable without knowing the OpenSSL overlayer.

Each version uses a new reply socket which will be either a `struct {SSL *ssl, int socket}` or a simple socket (`int`).