Security and Networks

Goal

The aim of this Lab is to implement "secure" socket connections using the Station-to-Station (STS) protocol, and retrieve your personal secret flag (see below).

0. Before starting

To be sure to use the latest version, download Lib.zip, extract the files and build the library.

```
$ wget https://www.enseignement.polytechnique.fr/informatique/INF558/TD/Lib.zip
```

```
$ unzip Lib.zip; cd Lib; make clean; make; cd ..
```

Download the source files Lab8.zip, unzip them and move to the corresponding directory.

```
$ wget https://www.enseignement.polytechnique.fr/informatique/INF558/TD/td_8/Lab8.zip
```

```
$ unzip Lab8.zip; cd Lab8; make
```

You are now ready to code. Open your favorite source-code editor. If you are used to working with VS Code, this can be done with:

```
$ code .
```

Key

Let us fix, once and for all, a 256-bit RSA public key `(N, e)`:

```
#RSA Public key (256 bits):
N = 0xc08dab3922e53b9cedeb32d65094462ced63cc2b6d71e8b4fd08f2f93b0c9281
e = 0xc84b83bf8c216c93b9c461027ecfae2d90a4be3d990064cb38ad50c845c70af
```

This public key is in the file `authority_pub.txt`, which you can find in the folder `data`. In this lab, your tutor will play the role of the trustable authority and this key is his RSA public key. You will use it to verify signatures in certificates.

Obviously the use of 256-bit RSA means that the security level of this system is essentially zero!

Capture-The-Flag

We prepared one secret value for each and everyone of you, that we will call a `flag`. A server is listening on the port `31415` of `rolls.polytechnique.fr`, ready to give you your secret flag, should you ask nicely.

By asking nicely, I mean correctly implementing the STS protocol, and running the following command `On a Polytechnique machine`:

```
$ ./client --sendto rolls --port 31415 --hostname <machine_name> --name <your.name> \
      try_CTF <certificate> <your_sk> <authority_pk>
```

where

- `machine_name` is the name of the machine where you are running the client (e.g `bentley`).
- `your.name` is of the form `{name}.{lastname}`. If you have spaces in your name, replace them by a dot (e.g `john.smith.junior`). Beware to respect the format, otherwise you won't be able to get your secret flag!
- `certificate` is the certificate issued by a trusted authority (ourselves), signing your public key. You will need to get one during the lab.
- `secret_key` is your secret key, associated to the aformentioned public key.
- `authority_pk` is the authority's (that is to say ours) public key. You will use it to verify signatures.

(You can also edit the variables `DEFAULT_*` on `client.h` if you prefer shorter commands).

Everything should clear up by itself all along the lab, but do not hesitate to ask!

At the end of the lab, you should get an output that looks like this:

```
gab=0xa33123eb6ca96af1fc234f3faae682e
last message: OK
Message = 7ca14de056d5087399a1366d65f6fffa4d196fda509f4fd622176c1c8ed0cc6c
[CTF] Congratulations! You captured your flag!
Secret=0x7ca14de056d5087399a1366d65f6fffa4d196fda509f4fd622176c1c8ed0cc6c
```

The secret value is represented as a 256-bit integer in hexadecimal.

When you finally got it, write it down in a file `flag.txt` that should contain the following line

```
john.smith.junior:0x7ca14de056d5087399a1366d65f6fffa4d196fda509f4fd622176c1c8ed0cc6c
```

We insist, but please, respect the format `{name}.{lastname}:0x{value}`

Upload your file `flag.txt` on VPL.

1. Building blocks

Diffie-Hellman

The overall objective of the Lab is ultimately to implement the *Station-to-Station* (STS) protocol (see the slides), which wraps basic Diffie-Hellman key exchange protocol with a layer of authentication. Our Diffie-Hellman component will be based on the group $(\mathbb{Z}/p\mathbb{Z})^*$, where

`p := 2^127+29 = 0x80000000000000000000000000000001D`.

We will use `g := 2` as a generator for $(\mathbb{Z}/p\mathbb{Z})^*$.

Symmetric encryption

We will use symmetric encryption and decryption both in the key exchange protocol and in the subsequent data transfer. In this lab, we will use AES with 128-bit keys. Moreover:

- the encryption operation (with key k) is denoted as: $m \mapsto \text{ENCRYPT}_k(m)$;
- the decryption algorithm (with key k) is denoted as $c \mapsto \text{DECRYPT}_k(c)$.

We will use the AES implementation and wrappers given in accompagnying files.

Signatures

For signatures, we will use 256-bit plain RSA signatures using the hash function SHA-3.

2. Socket connections

In this lab, we will be communicating with each other over the network, using C *sockets*. It is highly recommanded to log in on the Polytechnique workstations, a list of which is provided here, even for the trial phase. For instance:

```
$ ssh my_login@bentley.polytechnique.fr
```

If you want to open a graphic window on the remote machine, for instance to open a graphic text editor such as `gedit`, you can use the `-X` ssh flag:

```
$ ssh -X my_login@bentley.polytechnique.fr
```

This terminal you logged in is considered to be **consoleA** (for Alice).

Connection with localhost

To create a server connection, we work as follows. In this first exercise we perform a connection between two consoles on your own machine. On consoleA, first compile `client` and `server` using the Makefile. Do not forget to copy the directory `Lib/` (at the same level as `Lab8`) recompile everything on the station, as explained in Section 0.

The server is fully implemented, as well as a library `network.c` that worksu the `C SOCKET` API. In this Lab, you will only need to implement various functions for the `client`. **No previous knowledge about networks is required.** The list of useful network functions can be checked in `network.h`.

As always, you are encouraged to read all the source files! You can ask for help from your tutors, and/or you can call the programs with the `--help` option.

```
$ ./client --help

Client for version 2022/11/27:2239 for td_8

Usage: ./client [--sendto SERVER_HOST (default localhost)]
[--port SERVER_PORT (default 31415)] [--hostname CLIENT_HOST
(default localhost)] [--listen CLIENT_PORT (default 1789)] [
try_send | try_aes | try_send_aes | try_DH | try_STS | try_CTF ]
[--help] [--name NAME] [ OPTIONAL FILES ]

Arguments: NAME: Of the form name.lastname, lowercase [Example:
john.smith].  try_send: Try to send a basic message. Modify and
play with this function.  try_aes: Try to encrypt a basic
message with AES. Don't send anything.  try_send_aes: Encrypt
and send a basic message with AES.  try_DH: Perform DH key
exchange and encrypt a message with the shared key.  try_STS:
Perform Station-To-Station protocol with the server.  try_CTF:
Try to retrieve the secret flag prepared for you. Needs your
name.

Optional files (Required for STS and CTF):

client_certificate.txt client_sk.txt auth_pk.txt
```

```
$ ./server --help

Server for version 2022/11/27:2239 for td_8

Usage: ./server [--hostname SERVER_HOST (default localhost)]
[--listen SERVER_PORT (default 31415)] [ OPTIONAL FILES ]

Optional files (Required for STS):
```

```
server_certificate.txt server_sk.txt auth_pk.txt
```

On consoleA, enter

```
$ ./server
```

Open another console **on the same machine** (Bob's, hence consoleB) and type:

```
$ ./client try_send
```

In consoleA, messages will look something like

```
$ ./server
Server for version 2022/11/17:1613 for td_8
Port = 31415

Nothing arrived!
...
Nothing arrived!
RECV: Hello! My name is
john.smith, calling from localhost.
RECV: I am the client sending to the server.

Nothing arrived!
...
Nothing arrived!
```

and in consoleB, you should get

```
$ ./client try_send
Client for version 2022/11/27:2239 for td_8
Received "Hello localhost!" from localhost:31415!
```

If for some reason you get a connection error on the client side, check the port number announced by the server, and enforce the server port with the `--port` option.

By default, `server` will listen on port `31415`, but will choose one at random if it is not available. Indeed, you can only have `one` application listening on a given port.

In this Lab, the client and the server both send and receive messages. The client listens on a random port by default, but it can be enforced with the `--listen` option.

Code checking

During this lab, you will manipulate a lot of `char *`. Don't forget to free them when you don't need them anymore! It is more than advisable to use tools such as `valgrind` to track memory leaks. If you encounter segmentation faults, you may use tools such as `gdb`. Don't hesitate to ask!

Remote connection

Of course, it is possible for the `client` to connect to a remote `server`. To do that, you can use the `--sendto` and `--port` options. You can specify both a hostname, or an IP address. You can choose any port number between `1024` and `65535`. Some are in theory reserved for a specifc usage, attributed by IANA (Internet Assigned Numbers Authority), but as long as no other application is listening on your choosen port, there should be no issue at all. For fun, try to use another port, you will see.

Note that since the client also listens on some port, it can sends a message to itself!

A list of machine names is provided here.

As a matter of fact, a server is listening on port `31415` on the machine `rolls`. Try it!

Exercise: basic connections

Working with one of your neighbours, try to communicate using C sockets.

For the `server` to be able to reply, it needs your `hostname` (or IP address).

For instance, on `bentley` or `ferrari`:

```
ferrari$ ./server --hostname ferrari --listen 4242
```

```
bentley$ ./client --hostname bentley --sendto ferrari --port 4242 try_send
```

Send funny messages through your client socket by modifying function `try_send` in file `client_aux.c`.

Caution. During your exchanges with your colleagues, do not exchange anything too personal: your connection is not encrypted! Do you really wish to say personal things? Fine! Let's encrypt!

3. Adding AES in the channel

Let us encrypt using AES with a shared key.

Checking encryption/decryption

First try this:

```
$ ./client try_aes
```

which should print

```
It's a long way to Tipperary
```

The code is given to help you next.

Strengthening the channel using AES with a fixed key

In file `client_aux.c`, using the function `try_aes` as an example, complete

```
void send_with_aes(const char *host, const int port, uchar *msg, mpz_t gab){ ... }
```

which encrypts the message `msg` using the number `gab` that will be used as an AES key.

To help you sending messages over the network, you can use the functions available from `network.h`. The most important are the following two:

- `void network_send(char *host, int port, char *client, int cport, char *mesg)` that allows to send `mesg` from `client` to `host`. The parameter `cport` is just here to tell the server where to address its reply.
- `char * network_recv(int timeout)` that waits during `timeout` seconds (or indefinitely is timeout=-1) listening on the socket for a message.
- The `char*` given by the previous function is a lower level representation of a message (try to display it!). In order to convert it to a mesage, you can use the function

  ```
  int parse_packet(char **client, int *cport, char **msg, char *packet)
  ```

  If you do not care about who sent you a message, you can discard this information by setting the first two arguments to `NULL`. See `server.c` for more examples.

Take a look at the code from `try_send` to see how they are used.

In this `send_with_aes` function, you just need to send something using `network_send`. No need to use `network_recv`: the client already handles the reply from the server (see `client_aux.c:589`).

You can test your function using the `try_aes` command which sends the message `AES` using `send_with_aes`. On the client side, you should see something like:

```
$ ./client try_send_aes
Client for version 2022/11/27:2239 for td_8
Sending: AES
Sending: Z8Zpc1H/Suwpzbqr8vvjRmfGaXNR/0rsKc26q/L740aZE3yKVqwFC7d6+aLZJqBh+KDmUx+Mmax5fcirWCtRPAy6F1G9m2hoi8+6YM7uVbxRNjW04qQ
Received "AES: [OK]" from localhost:31415!
```

meaning the client is sending an encrypted message that was coded using base64 to be able to transmit safely a binary text. See the corresponding function `buffer_to_base64`.

The lines `Sending: cleartext` and `Sending: ciphertext` should be printed by the `send_with_aes` function that you need to write. The line `Received ...` is handled automatically by the client.

On the server side, you should see something like:

```
RECV: AES
AES/RECV: Z8Zpc1H/Suwpzbqr8vvjRmfGaXNR/0rsKc26q/L740aZE3yKVqwFC7d6+aLZJqBh+KDmUx+Mmax5fcirWCtRPAy6F1G9m2hoi8+6YM7uVbxRNjW04q
It's a long way to Tipperary
```

Upload your file `client_aux.c` on VPL.

Of course, this way of using a shared key is rather obsolete, so a real key exchange protocol must be used.

4. A basic Diffie-Hellman implementation

Suppose Alice wants to establish a secure network connection with Bob. They can use the basic DH protocol to set up a secure connection, communicating using a symmetric cipher with a common secret key.

Step 1

Alice

1. generates a random integer `a`, and computes

   $ga = g^a \bmod p$;

2. sends `ga` to Bob. In our instantiation of DH, she encodes this as the string

   `"DH: ALICE/BOB CONNECT1 0x..."`

Step 2

Upon reception of the message from Alice, Bob

1. generates a random integer `b`, and computes

   $gb = g^b \bmod p$;

2. sends `gb` to Alice. In our instantiation of DH, he encodes this as the string

   `"DH: BOB/ALICE CONNECT2 0x..."`

Step 3

Alice

1. computes the shared secret key

   $k = (g^b)^a \bmod p$;

2. and uses it to send a message encrypted using AES to Bob encoded as

   `"DH: ALICE/BOB CONNECT3 ..."`

Step 4

Upon reception of the message from Alice, Bob

1. computes the shared secret key

$$k = (g^b)^a \bmod p;$$

2. and uses it to decrypt the message encrypted using AES.

You are given the `server.c` file and we want you to complete the `client_aux.c` file so that both programs agree. Of course, you may try to adapt the server side to the client side. Take your time, check file `server.c` and ask questions.

Test your function using

```
$ ./client try_DH
```

In case of problem, set the value of `DEBUG` to 1 and recompile.

To ease debug, it is advised to use very small values of *a* and *b* first.

Upload your file `client_aux.c` on VPL.

5. Certificates

A certificate contains a public key and a signature allowing us to check that the key is authentic. The certificates we use this week will be strings (comprising five lines) in the following form:

`NAME` *name*
`VALID-FROM` *date*
`VALID-TO` *date*
`ISSUER` *issuer*
`KEY` *modulus exponent*
`SIGNATURE` *sigma*

Here, *name* is a string, both *date*s are integers (represented in decimal), *issuer* is the name of the certification authority (a string), and *sigma*, *exponent* and *modulus* are integers (represented in hexadecimal). For example:

```
NAME Alice COOPER
VALID-FROM 1480006461
VALID-TO 1511546061
ISSUER certification_authority
KEY 0x69d0fbd46ac6eca149aa774c56648c7e069f8a75c7d78abf60e157a4db7d78550x3
SIGNATURE 0x824dd3601ed4d9fa04dff2b33d42e90a202398b00d9426b2c80cbc59bbc4b13
```

The integer dates represent the number of seconds since the *Unix epoch*, which was midnight on January the 1st, 1970. To get these times, we simply round the floating-point values returned by

```
#include <time.h>
{ time_t tt = time(NULL); }
```

Exercise: Obtaining a certificate

Get yourself a public-key certificate signed by one of the tutors. To do this, you need to generate yourself a 256-bit RSA public and secret key files. You can for instance open a terminal and type:

```
$ make -f MakefileGen
./key_gen 256 <my_name> 3
```

Recall that the name used should be your *login name* of the form `{name}.{lastname}` (this to ease our job in creating certificates). Two files are created:

```
{name}.{lastname}_pub.txt {name}.{lastname}_sec.txt
```

Then, send your ***public*** key file by email to your tutors, one of whom will reply and send a certificate signed with his/her private key. What is signed is the string obtained as the concatenation of

- The name (which is already a string);
- the date of beginning of validity converted as a **decimal** string;
- the date of ending validity converted as a **decimal** string;
- the name of the issuer (a string);
- the public modulus N_U represented as an **hexadecimal** string;
- the public exponent e_U represented as an **hexadecimal** string.

Extracting certificates and checking them can be done using the primitives in `certificate.c`; also `handle_certificate` can help doing some operations on the certificates. For instance:

```
$ ./handle_certificate verify ./data/authority_pub.txt my_certif
```

verifies the certificate you obtained form authority. In case you obtain an error, check that the file `my_certif` does not contain `\r` characters (a typical end-of-line problem on non-Unix systems). Be careful not to use a win* editor... Use `eol.c` to convert the file. After that, you may also try

```
$ ./test_certificate ./data/authority_pub.txt my_certif
```

6. The Station-to-Station protocol

Suppose Alice wants to establish a ***secure*** network connection with Bob. They can use the STS protocol to set up an authenticated secure connection, communicating using a symmetric cipher with a common secret key.

We assume that Alice and Bob have already generated public-private key pairs (PK_A, SK_A) and (PK_B, SK_B) and obtained certificates C_A and C_B for them.

Take some time to understand how many files are needed to program and check everything, and more importantly why. Your tutors are here to discuss every aspect of this protocol with you. We encourage you to put the files you create in folder `data`.

Complete the function `int CaseSTS` that implements the STS protocol on the client side. This function should return `1` if the protocol was successful, and `0` otherwise (This is important for the `CTF` to work, see `client_aux.c:406`).

To help programming, we give the server program and you have to adapt it to the client case.

Step 1 (the same as in DH, actually)
Alice generates a random integer `a`, and computes

$$n = g^a \bmod p;$$

Alice then sends `n` to Bob. In our instantiation of STS, she encodes this as:

```
STS: ALICE/BOB CONNECT1 0x...
```

Step 2
Bob

1. generates a random integer `b`;
2. computes

$$n = g^b \bmod p;$$

3. computes the shared secret key

$$k = (g^a)^b \bmod p;$$

4. generates the signature

$\sigma_B = SIGN_{SK_B}(g^b \bmod p, g^a \bmod p)$.

To do this, he signs the concatenation of the two byte arrays of length 24 representing respectively : $g^b \bmod p$ and $g^a \bmod p$

5. computes

$y = ENCRYPT_k(\sigma_B)$

with k;

6. sends (y, n, C_B) to Alice. In our instantation of STS, he encodes his reply as the byte array

```
STS: BOB/ALICE CONNECT2 ...
```

Step 3

Alice

1. verifies the certificate C_B, thus ensuring that Bob's public key PK_B is valid (if not, she rejects the connection);
2. computes the shared secret key

$k = (g^b)^a \bmod p$;

3. decrypts

$\sigma_B = DECRYPT_k(y)$

using k. Accepts k if

$VERIFY_{PK_B}(\sigma_B, (g^b \bmod p, g^a \bmod p))$

returns `True`. Otherwise, she rejects the connection;

4. generates the signature

$\sigma_A = SIGN_{SK_A}(g^a \bmod p, g^b \bmod p)$.

Here again, she signs a concatenation of two byte arrays of length 24.

5. computes

$z = ENCRYPT_k(\sigma_A)$;

6. sends (z, C_A) to Bob. In our instantation of STS, she encodes her message as the byte array

```
STS: ALICE/BOB CONNECT3 ...
```

Step 4

Bob

1. verifies the certificate C_A. If Alice's public key PK_A is invalid, then he rejects the connection.
2. Otherwise, he computes

$VERIFY_{PK_A}(DECRYPT_k(\sigma_A), (g^a \bmod p, g^b \bmod p))$.

If `False`, he rejects the connection. Otherwise, he accepts the shared private key k by transmitting the string `'OK'`.

Step 5

Alice and Bob have now established each other's identities and a shared secret key k, which they can now use to encrypt data over their connection.

The command for the server is

```
$ ./server server_certificate.txt server_sec.txt data/authority_pub.txt
```

and that for the client is

```
$ ./client try_STS client_certificate.txt client_sec.txt data/authority_pub.txt
```

Once everything works on localhost, you may try on `rolls` or with one of your tutors or your colleagues. You may also try sending more messages.

Upload your file `client_aux.c` on VPL.

7. Catch the flag!

Don't forget the CTF! The instructions are at the top of the page.

Upload your file `flag.txt` on VPL.