Login : tianchi.yu **Mot de passe :** •••••••••••
Connecter

0. Berfore starting

Download Lab8.zip and unpack it.

Exercises

The aim of this lab is to implement "secure" socket connections using the Station-to-Station (STS) protocol. Let us fix, once and for all, a 256-bit RSA public key `(N, e)`:

```
#RSA Public key (256 bits):
N = 0xbb88601851b3f1922ff4ba05c757b45c13c4ad411558cf2cbaa43b1c9c6397cb
e = 0x6dce838a29acd81ae76ee6f611025855c737355cd413893e1947c3000548417
```

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!

1. Building blocks

Diffie-Hellman

The overall objective of the Lab is ultimately to implement the *Station-to-Station* (STS) protocol (that you will study next week), which wraps basic Diffie-Hellman key exchange protocol with a layer of authentication. Our Diffie-Hellman component will be based on the group $(Z/pZ)^*$, where

$p := 2^{127}+29$ = `0x8000000000000000000000000000001D`.

We will use g := 2 as a generator for $(Z/pZ)^*$.

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

You 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 X workstations, a list of which is given as `data/hosts.3536`, even for the trial phase. (These machines can be accessed from outside the campus since the lock-down.) For instance:

```
slogin -X my_login@bentley.polytechnique.fr
```

The `-X` means that you can open a window on `bentley` that will be displayed on your laptop. This is handy when you use a text editor that uses a window (like `gedit`).

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. Still 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. On consoleA, enter

```
$ server=localhost; client=localhost
$ ./server $client 1789
```

Open another console (Bob's, hence consoleB) and type:

```
$ server=localhost; client=localhost
$ ./client $server 1717 try_send
```

In consoleA, messages will look something like

```
Nothing arrived!
...
Hello!
I am the client sending to the server.
```

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

As a matter of fact, server will listen on port 1717 and client will listen on port 1789.

Socket connection on the network

Of course, it is possible to replace `'localhost'` by an (IP address, port number) or a remote machine name.

A list of machine names is given in `data/hosts.3536`.

Exercise: basic connections

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

Nothing really changes, you need to replace `'localhost'` by the name of the machine, for instance `'rolls'` or `'ferrari'`:

```
ferrari$ ./server rolls 1789
```

```
rolls$ ./client ferrari 1717
```

As a matter of fact, server will listen on port 1717 and client will listen on port 1789 (hence the swap in names). Send an amusing message through your client socket by modifying function `try_send` in file `client.c`.

Caution. During your exchanges with your colleague, 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

Checking encryption/decryption

First try this:

```
./client $server 1717 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.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.

On the client side, you should see something like:

```
./client $server 1717 try_send_aes
Sending: AES
Sending: glUtscd8X26/ACFu03ekN4JVLbHHfF9uvwAhbjt3pDdMChSFB751LRzDjQ9uV6yKYnPXPCjV1Ijg7ngnxkCDIC8d1B/865usTnksK9PtpjEljF
```

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`.

Upload your file `client.c` here

Le nom du fichier à déposer 选择文件 未选择任何文件 Déposer
Il faut se connecter avant de pouvoir déposer

Of course, this way of setting a key is rather obsolete, so that 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 mod 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 mod 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 mod 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.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 $server 1717 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.c` here

Le nom du fichier à déposer 选择文件 未选择任何文件 Déposer
Il faut se connecter avant de pouvoir déposer

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 0x69d0fbd46ac6eca149aa774c56648c7e069f8a75c7d78abf60e157a4db7d7855 0x3
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_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>
```

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_2_pub.txt my_certif
```

verifies the certificate you obtained form authority_2. 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.

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`.

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

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

   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

   $$y = \mathrm{ENCRYPT}_k(\sigma_B)$$

   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

   $$\sigma_B = \mathrm{DECRYPT}_k(y)$$

   using k. Accepts k if

   $$\mathrm{VERIFY}_{PK_B}(\sigma_B,\ (g^b \bmod p,\ g^a \bmod p))$$

   returns `True`. Otherwise, she rejects the connection;
4. generates the signature

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

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

$z = \mathrm{ENCRYPT}_k(\sigma_A)$;

6. sends (z, C_A) to Bob. In our instantation of STS, he encodes his reply as the byte array

```
STS: BOB/ALICE 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

$$\mathrm{VERIFY}_{PK_A}(\mathrm{DECRYPT}_k(\sigma_A), (g^a \bmod p, g^b \bmod p)).$$

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 rolls 1789 server_certificate.txt server_sk.txt auth_pk.txt
```

and that for the client is

```
./client ferrari 1717 client_certificate.txt client_sk.txt auth_pk.txt
```

Once everything works on localhost, you may try with one of your tutors of your colleagues, trying the protocol and sending more messages.

Upload your file `client.c` here

Le nom du fichier à déposer 选择文件 未选择任何文件 Déposer

Il faut se connecter avant de pouvoir déposer