Homework 2

COM402 - Information Security and Privacy 2017

- The homework is due **Thursday, 30th of March, at 23h00 on Moodle**. Submission instructions are on Moodle. Submissions sent after the deadline **WILL NOT** be graded.
- This homework is likely to take up more time and require more coding than the previous one. **Friendly advice**: please start well ahead the deadline.
- In this homework, you will submit solutions that will run in a Docker container. For the last exercise, our web server will connect directly to your docker. While we have taken some basic precautions, there might still be some vulnerabilities in our setup. **Please**, **do not try to break the system**. We will know who initiated the attacks ;). In the event that you find vulnerabilities, you are welcome to disclose them to us (can even have a bonus !)
- **Docker setup issues**: Hopefully most of you have a working docker setup on your machine. In case you still experience difficulties, please work in the VM that we provided on moodle, which has docker installed. Also, in case you're using the VPN and experience weird behavior, please run your scripts while connect to the epfl wifi network. **The only setup we provide support for is VM & epfl wifi network**. Also, there will be a few laptops available in BC263 that are already set up for you to solve the homework. If you prefer to work on these please send an e-mail to the TAs.
- Happy coding!

Exercise 1: Are Clients Better Authenticated In The West Or In The East v2?

In the first exercise set we all saw that password verification on the east-side is not the best idea, as tech-experiences users can derive their password by looking at the client-side javascript.

In this exercise you need to do "west-style" password verification. You need to write a python script that instantiates a web-server that can check the correctness of the password.
Your web server should listen on `127.0.0.1:5000`. It should expect POST queries to the URL `/hw2/ex1`. The POST content is a JSON of the following form:

```
{"user": user,"pass": password}
```

If the password is correct, the response must have the status code 200. If it is not correct, it should be 400. Your verification function must use the same algorithm as the one used in the first exercise of the first homework (`com402.epfl.ch/hw1/ex1`).

The library you must use is named Flask. You can find all the documentation at http://flask.pocoo.org/docs/0.12/ . You must use Python3, and must submit only one file **ex1.py**.

To get the token, you can upload your script to `com402.epfl.ch/hw2/` for exercise 1.

For testing, your web-server can serve the html of hw1/ex1. Note that our verification scripts won't look at the HTML that you send back.

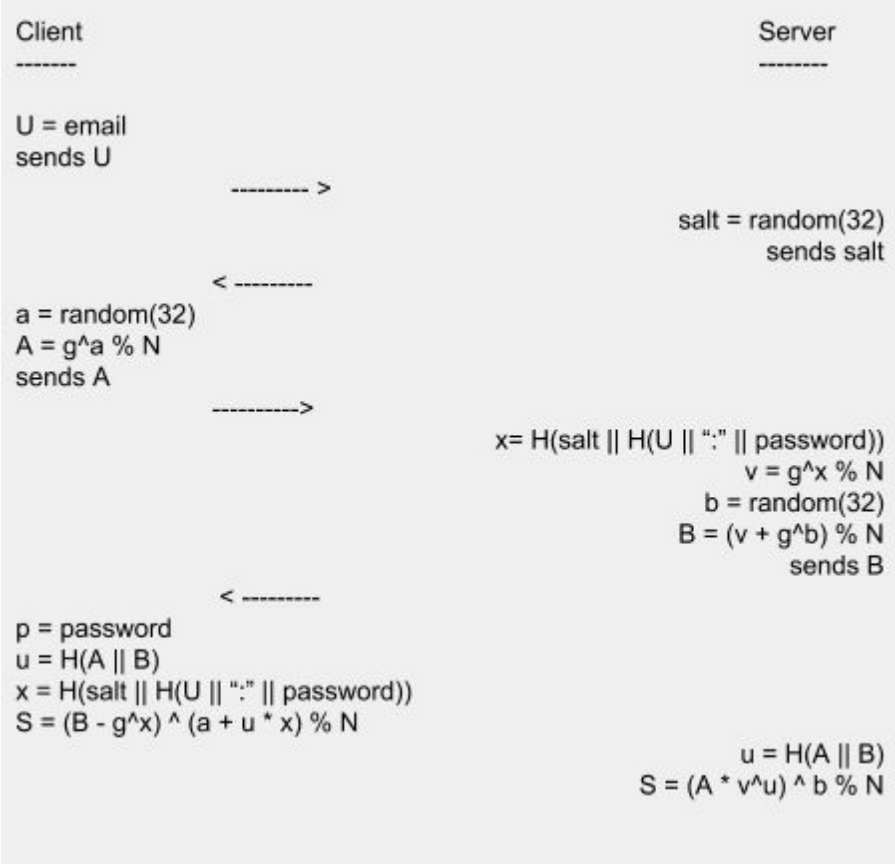
Exercise 2: Once you know our secret…

This exercise is about implementing a password-authenticated key agreement protocol, PAKE. PAKE serves two main purposes: derivation of a cryptographically secure shared key from a low entropy password and proving the knowledge of a secret password to each other *without* actual transmission of this password. This is useful for a lot of applications. For example, ProtonMail is an end-to-end encrypted email service provider which uses a PAKE protocol named Secure Remote Password protocol (SRP).

You have to implement the client part of the SRP protocol that interacts with our server using websockets at `ws://com402.epfl.ch/hw2/ws`. This client should perform the SRP exchange with the username being your EPFL EMAIL address and with the password that has been generated in both the previous exercise and homework 1 - ex1.

Protocol

The web server implements a simplified version of the SRP protocol described in the RFC.



At the end of the protocol, both parties should have the same shared secret S.
To get the token, send `H(A || B || S)` to the server at the end, and look at the response!
You should be convinced that both sides get the same secret at the end, we encourage you to take a look at the RFC for more details.

Parameters

H = sha256
N="EEAF0AB9ADB38DD69C33F80AFA8FC5E86072618775FF3C0B9EA2314C9C256576D674DF7496EA81D3383B4813D692C6E0E0D5D8E250B98BE48E495C1D6089DAD15DC7D7B46154D6B6CE8EF4AD69B15D4982559B297BCF1885C529F566660E57EC68EDBC3C05726CC02FD4CBF4976EAA9AFD5138FE8376435B9FC61D2FC0EB06E3"
g = 2

The modulus N is in hexadecimal form, make sure you transform it into a integer before using it!

Encoding

All strings are utf8 encoded strings (email and response). In Python3, you can use

`myString.encode()` to utf8-encode your strings.
All numbers (A, B, salt) are **first** encoded into bytes in big endian mode, **then** the array of bytes is encoded as a hexadecimal string and sent as is.
We provide you the sample code to encode and decode an integer in Python3 below:

encoding:

```
i = 123456789
buff = i.to_bytes((x.bit_length() + 7) // 8, 'big')
strToSend = binascii.hexlify(buff).decode()
```

Confused about hexlify followed by decode? Check out this explanation and look for the title 'Hexadecimal'.

decoding:

```
buff = binascii.unhexlify(msgReceived)
i = int.from_bytes(buff,'big')
```

Python3 supports arbitrary length integer operations, so there is no need to install an external library.

Network

All communication happens through a websocket channel. You can create this channel by connecting to `ws://com402.epf.ch/hw2/ws`.

Implementation

This exercise can be solved using any programming language you want. However, we do provide you with some python3 snippet code and libraries pointers. For python3, you can use this library. Creating a websocket client is very easy and you can just use this snippet. Make sure you have python >= 3.5 if you want to use the first snippet. Regardless of the language, the file's name should be **ex2** (e.g., for python3 **ex2.py**).

Exercise 3 - If it looks like a chocolate chip, there should not be raisins. Check before you eat!

In this exercise you are asked to create a web-server that serves cookies to clients based on whether they are simple users or administrators of the system. However, as you now know, malicious users might try to tamper with the cookie to get administrative access.
You decided to protect the authenticity of your cookies, so you are going to add a Keyed-Hash Message Authentication Code (HMAC) as an additional field. The secret key of the HMAC is **your password of hw1/ex1 encoded in utf8**.

You must provide a single python3 script **ex3.py** implementing a web server using the Flask library that listens on `127.0.0.1:5000`.

1. To login, POST a json at localhost:5000/ex3/login :

   {"user": user,"pass": password}

The response must contain a cookie. It must be an administrator cookie if the username is “administrator” and the password is “42”. Otherwise, it must be a regular cookie.
The cookie should have the same fields as in the exercise 2 of homework 1 with an additional field holding an HMAC. Example of a cookie:

Yoda,1489662453,com402,hw2,ex3,administrator,*HMAC*

2. localhost:5000/ex3/list must return a specific HTTP status code:
   - code 200 if the user is the administrator
   - code 201 if the user is a simple user
   - code 403 if the cookie is tampered

Once you have a working script, you can upload it to `com402.epfl.ch/hw2` to get the token.

Exercise 4 - HTTPS transmission

As you saw in the previous homework, it is quite easy to intercept and change traffic. One way to make this attack harder is to use HTTPS. HTTPS brings encryption and authentication to HTTP.
In this exercise, you have to set up an Nginx-server to use TLS for HTTP, also called HTTPS. You can find more information about ngninx at https://nginx.org/en/ .

In the first part of the exercise, you must set up a self-signed certificate. A self-signed certificate is a first-step protection for a website. However, the websites with such certificates are usually shown as “dangerous” in your browser because the certificates are not signed by

a Certificate Authority trusted by your browser. An attacker can create self-signed certificates on the fly and hope that users will click to accept it.
In the second part of the exercise, you must use our fake DEDIS-CA to have a 'correct' certificate signed by our CA.

4a - Use a self-signed certificate

You can download and start the docker-image with the pre-installed nginx and openssl:

```
docker run -ti --name hw2_ex4 -P dedis/com402_hw2_ex4
```

If, by any chance, you exit from the session (`exit`), you can later re-connect to it using:

```
docker start -i hw2_ex4
```

You will now be logged in a bash session where you can edit the configuration file `/etc/nginx/conf.d/default.conf` with nano, vim or emacs. For users not used to command line, we suggest you use nano:

```
nano myfile
```

There is a simple webpage available now at http://localhost - your goal is to add the necessary comments to nginx to have a self-signed certificate used to serve https://localhost.
Once you think your setup is correct, you can run the automatic script verification **inside the container**:

```
./verify.sh a
```

If all is well, you should see your token in the output. Copy the configuration file `default.conf` as `ex4a.conf` and submit it via moodle.

4b - Make your certificate trustworthy

The self-signed certificate is not secure - you must get it signed by a CA. We did set up a CA for the signature of your certificate. You can submit your *certificate signing request* to our website at http://com402.epfl.ch/hw2/

You should now adapt your configurations file `/etc/nginx/conf.d/default.conf` inside the docker container. Once you think your setup is correct, you can run the verification script **inside the container**:

```
./verify.sh b
```

You should see your token if everything is fine. Copy the configuration file `default.conf` as `ex4b.conf` and submit it via moodle. Congrats!

Good luck!