Homework 2

Never use your actual GASPAR password on COM-402 exercises (if the login form is not Tequila).

Exercise 1: [attack] Cookie tampering

Hello, Elliot. The Evil Corp is back with their evil plans; rumors say they can hack and spy on anyone. Fortunately for our cause, it seems that the authentication mechanism they use for identifying their admins is severely broken. Can you bypass their login page and defeat them?

The web server of Evil Corp is located at http://com402.epfl.ch/hw2/ex1. Your goal is to login as an administrator and "Spy & Hack" on someone.

Exercise 2: [defense] HMAC for cookies

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 have seen in the previous exercise, malicious users can easily tamper with cookies.

To protect their integrity and ensure only you can produce a valid cookie, you will add a Keyed-Hash Message Authentication Code (HMAC) as an additional field.

Implementation

Implement your solution using Python 3 and a Flask server.

Your server should accept a `POST` login request of the form:

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

on the URL `/login`. The response must contain a cookie (defined below).

Your server should have a second endpoint: it should accept `GET` requests on `/auth`. It can return a blank page, but the status codes should be:

- code 403 if no cookie is present

ex1

By looking at the loginpage, the encode method is base64 so we just need to decode the message and then, encode a new one by changing user to admin to attack the website

ex3

By looking more carefully at the script of the website, we just need to have password = hash(username,mySecureOneTimePad) to avoid the login page. we can find the value of mySecureOneTimePad in the script too = Never send a human to do a machine's job

Then we can you the console on ctrl+maj+i and use : hash('florian.genilloud@epfl.ch',"Never send a human to do a machine's job") to know what is the password with this login username

- code 200 if the user is the administrator
- code 201 if the user is a simple user
- code 403 if the cookie has been tampered

Cookie

The cookie should have the same fields as in Exercise 1 with an additional field containing the HMAC.

Example of admin cookie: `admin,1489662453,com402,hw2,ex2,admin,HMAC`

Example of user cookie: `Yoda,1563905697,com402,hw2,ex2,user,HMAC`

It must be an administrator cookie if the username is `admin` and the password is 42. Otherwise, it must be a regular (user) cookie. The name of the cookie is `LoginCookie`.

Testing your solution

Code your solution in a file named `server.py`. Here's a template:

```
from flask import Flask

app = Flask(__name__)

cookie_name = "LoginCookie"

@app.route("/login",methods=['POST'])
def login():
    # implement here

@app.route("/auth",methods=['GET'])
def auth():
    # implement here

if __name__ == '__main__':
    app.run()
```

To test your server, use the following command in the same folder as `server.py`:

`docker run --read-only -v $(pwd):/app/student com402/hw2ex2`

Note: for scalability reasons, we send you the verification script in a Docker. With a bit of work, you can have access to the verification script; **looking at the verification script is not the intended way of doing exercises in Com402**. In practice, imagine that this docker is running somewhere else, i.e., you cannot read the verification script.

Workbench lab

Server side

Testing it is also very simple! We provide a way for you to test the final code, but debugging simple mistakes will be painstakingly long if you have to invoke a docker instance for every test. Here are a few tips to make your life easier.

First of all, using a terminal, go to the location of your `server.py` file, and launch these two commands:

```
export FLASK_APP=server.py # file of your server
export FLASK_ENV=development # say it's for dev purpose
```

Note that these will last in the current terminal, and you will have to re-launch them every time you open a new one. Then, type the following:

```
flask run
```

This will launch a flask server, in *debug* mode. Each time you make a modification in the python file, the server will automatically restart to take into account the changes!

Client side

Now, to easily query the server, open a new terminal, and open a python interpreter (command: `python3`). You'll be using `requests`, a fairly standard wrapper for HTTP requests. requests' *sessions* are very useful to conduct a sequence of operations (`GET`, `POST`,...) while keeping cookies (among others). Here are a few useful commands:

```
import requests

# create a session
session = requests.session()

# perform a GET
session.get("http://url_or_ip_address:port")

# perform a POST with a payload
session.post("http://url_or_ip_address:port/some/path",
             data={"field1": "value1", "field2": "value2"})

# inspect your cookies
print(session.cookies)

# modify your cookies
session.cookies.update({"name1": "value1"})
```

With a server in debug mode, and this interpreter ready, you can simply do modifications to the server, save, go to the interpreter, and launch a test command. And the window in which you launched the server will inform you of any python errors, with the adequate stacktrace.

Exercise 3: [attack] Client-side password verification

You need to bypass the authentication on the Very Secure™ login page http://com402.epfl.ch/hw2/ex3/.

Exercise 4: [defense] PAKE implementation

Important disclaimer: In this exercise, we ask you to implement a cryptographic protocol. This is a challenging and interesting task, notably to get a good understanding of what is going on. However, in real life, implementing your own cryptographic primitives is generally a **bad idea**. Read more about this here or here.

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.

Protocol

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

```
Client                                          Server
-------                                         --------

U = email (string)
sends U (UTF-8 encoded string)
                 --------- >
                                                salt = randomInt(32)
                                                sends salt (UTF-8 encoded Hexadecimal String)
                            < ---------
a = randomInt(32)
A = g^a % N
sends A (UTF-8 encoded Hexadecimal String)
                            ---------->
                                                x= H(salt || H(U || ":" || PASSWORD))
                                                v = g^x % N
                                                b = randomInt(32)
                                                B = (v + g^b) % N
                                                sends B (UTF-8 Encoded Hexadecimal String)
                             < ---------
u = H(A || B)
x = H(salt || H(U || ":" || PASSWORD))
S = (B - g^x) ^ (a + u * x) % N
                                                u = H(A || B)
                                                S = (A * v^u) ^ b % N
```

At the end of the protocol, both parties should have the same shared secret `S`. To validate the exercise, send `H(A || B || S)` to the server at the end!

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.

Implementation

This exercise can in theory be solved using any programming language you want; however, we strongly recommend to use Python 3, for which you are given code snippets below.

In Python $>=$ 3.5, you can use this library for websockets; an example for creating a websocket is given here.

Network

All communication happens through a websocket channel on port 5000. You can create this channel by connecting to `ws://127.0.0.1:5000/`.

Encoding

String encoding: In Python3, all strings (`str`) are by default UTF-8 encoded. But some libraries (such as hashing) work with `bytes`. You can convert between the two easily: * `byte.decode() -> str` * `str.encode() -> byte`

You can specify arguments to `encode` and `decode` in order to decode to (or encode from) another encoding (say, ASCII for example), but that likely won't be needed there.

Number encoding: All numbers (A, B, salt) follow the same procedure: * To use in mathematical functions (pow, mod,...), keep as integers * Otherwise, first encode them to an UTF-8 hexadecimal string, then encode them to a bytes string (see above). Used for hashing or sending to the websocket.

Note that sending a string as bytes or utf-8 through a websocket is similar, the `websocket` library will decode the message to a `utf-8` string upon arrival.

Numbers you receive from the websocket (from our server) are encoded the same way (hexadecimal utf-8)

Here's a sample code in Python 3:

Encoding:

```
utf8_hex = format(12345, "x").encode()
await websocket.send(utf8_hex)
hash.update(utf8_hex)
```

Decoding:

```
# Number is received as UTF-8 encoded hexadecimal String
utf8_hex_message =  await websocket.recv()
# Hexadecimal string is converted back to an integer
integer_message = int(utf8_hex_message, 16)
```

Hashing

In this exercise, you have to hash (a mix of) strings and numbers. The encoding is consistent with what is described above; always use bytes-encoded hexadecimal strings.

```
import hashlib
h = hashlib.sha256()
h.update(format(12345, "x").encode())
h.update(some_other_bytes_hex_string)
# h.hexdigest() returns an hexadecimal representation of the hash
utf8_hex_result = h.hexdigest()
```

Exponentiations

In the second message sent by the client, you're supposed to compute `A = g^a % N`. Note that the python expression `g**a % N` will *not* work ! Due to the priority of `**`, this would first compute `g**a`, then apply `mod N` (think about the size of `a` in bits). However, this expression can be efficiently computed as `pow(g,a,N)`, which uses standard modular exponentiation techniques (such as computing `a mod \phi(N)` first, then applying the mod at every multiplication).

Randomness generation

The function `randomInt(32)` should generate a **number** from 32 random bytes. Look into `os.urandom` and `int.from_bytes`

Constants

```
EMAIL = "your.email@epfl.ch"
PASSWORD = "correct horse battery staple"
H = sha256
N = EEAF0AB9ADB38DD69C33F80AFA8FC5E86072618775FF3C0B9EA2314C9C256576D674DF7496EA81D3383B4813D
g = 2
```

Note: The modulus N is given here in hexadecimal form, make sure you transform it into an integer before using it.

To test your client, use the following command in the same folder as `client.py`:

```
docker run --read-only -v $(pwd):/app/student com402/hw2ex4
```