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
- 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,ex3,admin,HMAC`

Example of user cookie: `Yoda,1563905697,com402,hw2,ex3,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.

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
sends U
                ---------- >
                                           salt = randomInt(32)
                                           sends salt
                           < ---------
a = randomInt(32)
A = g^a % N
```

```
sends A
                           ---------->
                                            x= H(salt || H(U || ":" || PASSWORD))
                                            v = g^x % N
                                            b = randomInt(32)
                                            B = (v + g^b) % N
                                            sends B
                            < ---------
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: All strings are UTF-8 encoded strings (email and response). In Python 3, you can use `s.encode()` to utf8-encode a string.

Number encoding: All numbers (A, B, salt) are: 1. first encoded into bytes in big-endian mode, then 2. the array of bytes is encoded as a hexadecimal string, to be sent in the web socket.

Numbers you receive from the websocket (from our server) are encoded the same way.

Here's a sample code in Python 3: Encoding:

```python
import binascii

number = 123456789
bigendian_array = number.to_bytes((number.bit_length() + 7) // 8, 'big')
utf8_hexadecimalstring = binascii.hexlify(bigendian_array).decode()
await websocket.send(utf8_hexadecimalstring)
```

Decoding:

```python
import binascii

utf8_message = await websocket.recv()
bigendian_bytes = binascii.unhexlify(utf8_message)
number = int.from_bytes(bigendian_bytes, 'big')
```

A good reference about hexlify is here, section “Hexadecimal”.

Hashing

In this exercise, you have to hash (a mix of) strings and numbers. The encoding is consistent with what is described above; if you’re lost, `sha256(some_bigendian_bytes || some_utf8_string)` is expressed as:

```
import hashlib

h = hashlib.sha256()
h.update(some_bigendian_bytes)
h.update(some_utf8_string)
bigendian_bytes_result = h.digest()
```

Randomness generation

The function `randomInt(32)` should generate a **number** from 32 random 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
```