Homework 4

COM402 — Information Security and Privacy 2019

- Submission instructions and the deadline are on Moodle. Submissions sent after the deadline **WILL NOT** be graded.
- In the event that you find vulnerabilities, you are welcome to disclose them to us (can even have a bonus !)

Exercise 1: Differential privacy (25p)

This time, you play the role of a company that collects movie ratings, e.g., IMDb, and you want to securely disclose them. You have a rating database in clear, where each row has the format `[user, movie, date, rating]`. By now you know well that creating anonymized datasets is very hard, so releasing a database with pseudonymized (e.g., hashed) identifiers is not an option. You still want to allow some researchers to learn some useful information from your database, but not at the cost of the privacy of your users. Hence, you decide to allow the researchers to send you queries that you can respond to in a differentially private way. This should help preserve the privacy of your users while still allowing the researchers to learn some insights about the data.

You allow counting queries of a certain format. The researchers should be able to get the number of people who have rated a given movie greater or equal than a certain level. If you were to express this in terms of SQL:

```
SELECT COUNT(*)
FROM db
WHERE movie = [queried movie name]
      AND rating >= [queried rating level]
```

This will allow them to learn which movies are rated well and which ones are not. You make the following assumptions:

1. A researcher can issue counting queries as many times as she wants, but the condition is that she has to send the queries all at once, in a batch. Your response has to satisfy differential privacy with level epsilon=**ln(2)** *for the full batch of a researcher's queries*.

   This means that if you removed or added one rating to the dataset, the maximum privacy loss incurred by responding to the full batch of queries should be smaller than $e^{\ln(2)} = 2$. This is the amount of leakage in the worst case, so your users have agreed that this level of privacy loss is acceptable to them.

   Hint:

2. In your database, a user can only rate one movie once.
   Hint:
3. You want to use the Laplace mechanism for ensuring differential privacy. You want to distribute the privacy leakage evenly across all queries in the batch, and you compute the required noise parameters as if there were no duplicate queries.
4. The researchers pass a thorough vetting process, so you can safely assume they do not collude—will not share outputs they receive with each other.

You will have to implement a Python function `count_ratings(db, movies, rating_levels)`. It takes as input your database db (as a list of tuples), and the batch of queries of a single researcher: `movies` is a list of movie names for each query, and `rating_levels` is a list of rating levels for each query. The function has to return the list of outputs to the queries, such that it satisfies differential privacy subject to the assumptions above.

You need to download the following file from our server:
http://com402.epfl.ch/download/com402_hw4_ex1.tgz
This will help to generate a database for your testing purposes. Put all three files in the same directory and write your code in `dp_query.py`. In the script, you can use the *scipy* and *numpy* Python packages.

Obtaining the token

The script `dp_query.py` is set up so that you can submit it on the com402 website and obtain either the token, or hints on what you're doing wrong. Keep in mind that our tester runs your script multiple times. A part of `if __name__ == "__main__"` takes care of all this, so don't change it!

Note that if you get an error saying that your noise doesn't satisfy the privacy level, try to submit it again—this can sometimes happen even if your solution is correct due to the probabilistic nature of some of our tests.

Exercise 2: bcrypt (25p)

This is a simple exercise where you need to write a minimalistic server (using Flask). The server should expect a POST request on `127.0.0.1:5000/hw4/ex2` with JSON data containing the `user` and `pass`. The only thing your server has to do then is to hash the password using **bcrypt** and send back the hash with status code 200. Make sure you UTF-8 encode the password **before computing its hash**! You should submit your script, named `hw4_ex2.py`, via the submission page.

Exercise 3: Defense of the data (25p)

You're being asked to build a super minimal website using some data stored in a MySQL database. Of course, you're starting to worry about the security nightmare this project might have.

Well not really, so many libraries do the job for us now.

Run the Docker image:

```
docker run -it -p 80:80 --rm --name ex3 dedis/com402_hw4_ex3 bash
```

You will find inside the skeleton script `site.py` where you have to fill in two endpoints `/messages` and `/users`. All information needed is included in `site.py`.
The goal is to make sure your script is not susceptible to SQLi attacks.

Once you have finished, you can upload your own `site.py` to the submission webpage.
Good luck !

Exercise 4: Towards a secure web server in vacuum (25p)

Being a hacker yourself, you know that just having an option for HTTPS on your webserver does not solve all the problems. Attackers can still try to downgrade users' connections or even make use of laid-back users who still access all the websites by typing http://…
You know that you cannot rely on users to take all the security precautions.

The goal of this exercise is to further learn how to configure the security parameters of an nginx server to properly protect connecting users!

To start with, run the Docker image:

```
docker run -it -p 80:80 -p 443:443 --name hw4_ex4 dedis/com402_hw4_ex4
```

Make sure that you bind both ports 80 and 443 of your container to localhost when you run the container (it's set with the -p flag in the command above).

In the container, you will find the same setup as in exercise 4 of HW2 where you had to build a barebone https server that used a certificate signed by our DEDIS-CA. You have to use this new container because it has a new ssh key written in which allows us to verify your setup.

You need to start with the same steps as in HW2 by adding HTTPS support to the `/etc/nginx/conf.d/default.conf` file. You can reuse your DEDIS-signed certificate from HW2 (no self-signed is needed). If you do not have it anymore, generate a new one.
For the sake of verification, make sure to type `localhost` in the field Common Name (CN) when generating a request for a certificate to be issued by our DEDIS-CA which you will then use in your nginx server.

Then, you need to implement **HTTP Strict Transport Security (HSTS)** in your nginx server. HSTS is a mechanism that allows a webserver to declare that it can only be accessed using HTTPS. The HSTS option should be set for all subdomains and with a max-age period of 1 year. The latter stands for how long browsers should store cached information about HSTS for this website.

Your next step to implement **explicit redirect from HTTP from HTTPS**. This means that if a user connects using HTTP, the HTTP access will be redirected to HTTPS. The redirect must be supplied with **"Moved Permanently"** status code.

After having experienced the downgrade attack that you carried out yourself, you decide that this must never happen on your server! So your sacrifice backward compatibility and configure your server to **only** connect using **TLSv1.2**.

You then think about the case where someone sends an HTTP DELETE request to your server and erases all your carefully collected COM402 course notes. That is not what you have been working so hard for! So, you decide to limit the types of **HTTP requests that your server will accept, allowing only GET, HEAD and POST**. If someone sends another type of request, e.g., DELETE, you should return **the status code 405** as your response.

Finally, you need to configure your nginx server to notify browsers that they need **to enable XSS protection for your website**. The protection is usually already enabled by default in browsers, but it can be manually disabled by users. If you configure it explicitly, it will have a priority over the user's configuration.

To summarise, your tasks are:

1. Running HTTPS server
2. HSTS
3. Redirect
4. TLSv1.2
5. Allow only HTTP GET, HEAD, and POST
6. XSS protection

Before testing, add your email address to the `/www/index.html` file in the form of

```
email = "firstname.lastname@epfl.ch"
```

so we know who you are. Once you think your setup is correct, you can run the verification script **inside the container**:

`./verify.sh`

You should see your token if everything is fine. Copy the configuration file `default.conf` as `ex4.conf` and submit it via Moodle.