CS 5435 – Security and Privacy Concepts in the Wild

Homework 1: Docker version

Instructor: Vitaly Shmatikov, TA: Marina Bohuk *Fall 2022*

Note: Use this version of homework 1 if you are using Apple Silicon.

Instructions.

The deliverables in this homework are marked in **blue boxes** as exercises, and at the end of the document we list the files you should turn in. The task boxes in **purple boxes** are for your edification only, no deliverables are associated with these tasks. You should submit your homework in Canvas as a PDF (we encourage using the provided LaTeX template) along with any code files required by the assignment. You may complete the homework on your own or in a team of two, which you will indicate in Canvas and within the PDF.

Part 0: Getting Started

In this and subsequent homeworks, you will be interacting with a toy web server, finding and fixing vulnerabilities as we discuss them in class. Portions of this (and subsequent) assignments were inspired by assignments from Stanford's CS155 and MIT's 6.858.

This first part of the homework will step through getting the web server up and running locally using Docker on your computer. vulnerabilities.

Setting up Docker. Set up Docker by installing (Docker Desktop); or, if you use homebrew to manage your packages, you should be able to just run the command `brew install docker` in the command line.

Running the web server. The code for the web server is distributed via Github and uses the Git version control system, which you may find useful for tracking changes while you complete the lab. To get the code for homework 1, clone the git repo:

```
you@laptop:  git clone https://github.com/cs5435/cs5435-hw1.git
Cloning into 'cs5435-hw1'...
```

Now you can try running the web server. We will do this using a Docker image that already has the right version of python and the pip packages preinstalled; this Docker image should be automatically pulled when you run the following commands, if it has not been pulled or updated recently.

```
you@laptop:  cd cs5435-hw1/
you@laptop:  docker run -p 8080:8080 --platform linux/amd64 -it -v $PWD:/app
cs5435/cs5435 python3 -m app
Bottle v0.12.17 server starting up (using WSGIRefServer())...
Listening on http://0.0.0.0:8080/
Hit Ctrl-C to quit.
```

If you see a warning show up (`WARNING: The requested image's platform (linux/amd64) does not match the detected host platform (linux/arm64/v8) and no specific platform was requested`), you can either ignore it, or add the flag `--platform linux/amd64` to the `docker run` command to suppress the warning.

Now access the web server by opening your web browser and connecting to `http://localhost:8080/login`. You should now see a login page for the web application.

While completing the remainder of the lab, you should not need to install any new Python dependencies. If you feel like you need to, please first get permission from a TA, as we will need to update our grading environment.

Part 1: Credential stuffing attacks

In this section, you will mount credential stuffing attacks on the web server using account credentials from previous password breaches.

When you start running the server, a set of users are pre-registered using the registration script located at `app/scripts/registration.py`. It reads a file `app/scripts/registration.csv` that contains the usernames and plaintext passwords of pre-registered users. To get the full experience of this lab, it is recommended you do not look at the contents of this file. Your attacks will be graded on a different set of registered users.

In the `app/scripts/breaches/` directory, you will find `.csv` files representing leaked credential breaches of other web services. In credential stuffing attacks, attackers will attempt to login to a target web service using credentials leaked from other web services, with the assumption that users often share passwords across services.

Later in the lab, we will make use of cryptographic hash functions. Some examples of their use are given in `app/util/hash.py`.

Your first job will be to execute a credential stuffing attack on the web server using a breach of plaintext passwords at `plaintext_breach.csv`.

Exercise 1.1: Implement `credential_stuffing_attack` in `stuff.py`. It takes as input an iterable of username and plaintext password credential pairs and should output a list of credential pairs that successfully login to the web service. Use the following Docker command to run `stuff.py` when you test it:

```
you@laptop: docker run --add-host=localhost:172.17.0.1 -it -v
$PWD:/app cs5435/cs5435 python3 stuff.py
```

Task 1.2: Test your attack using the breach of plaintext passwords in `plaintext_breach.csv`. You should be able to successfully login to 3 user accounts.

To prevent breaches to be used for credential stuffing, it is typical for servers to not store user passwords in

plaintext. Instead, the server can use a cryptographic hash function and store the hash of the password. To authenticate users, the server can hash the input password and compare to the hash saved in the database.

A data breach of a web server that uses this approach would reveal the hashes of user passwords. However, this can also be sensitive information! If users used completely random passwords, then the revealed hashes would not provide useful information. However, recall from class we learned that the distribution of passwords used by real users is very different from random. In fact, there is a relatively small set of passwords that is used by a disproportionately large set of users. An attacker can precompute the hashes of common passwords and compare them to the breached hashes to learn user passwords. These precomputed tables of hashes are a common tool used to crack passwords (and can be made even more sophisticated using more advanced techniques as in rainbow tables).

Task 1.3: Create a hash lookup table using the passwords in `common_passwords.txt` or use an online lookup table such as `https://crackstation.net` to learn the plaintext passwords of users present in the breach of hashed passwords at `hashed_breach.csv`. The hashes in this breach are generated using `sha256`. Using your credential stuffing attack with the learned plaintext passwords, you should be able to successfully login to 3 more user accounts.

Looking up passwords with precomputed tables works because hash functions are deterministic: the hash of a password is always the same value. To prevent these types of lookups, we must introduce some randomness into the hashing process.

The standard approach for this is called password *salting*. In this approach, during user registration, a value called a salt is randomly generated (typically, a 128-bit string). Instead of storing the hash of the password as before, the server now stores the hash of the password concatenated with the salt, along with the salt. The server will use the stored salt to recompute the hash and authenticate users. Observe that now precomputed lookup tables are unhelpful; it is infeasible to precompute the hash of common passwords with all possible salts in order to do a lookup.

However, given a salted hash and a salt (as in a breach), an attacker can still attempt to perform an offline brute-force attack to guess the password associated with salted hash. If the attacker guesses the password correctly, the hash of the password concatenated with the salt will equal the salted hash. Again, such an attack would not be successful if passwords were chosen well, but can be quite effective when they are not.

Exercise 1.4: Implement `brute_force_attack` in `brute.py` using the passwords from `common_passwords.txt`. It takes as input a salted hash and salt and should output the matching password or `None` if no match is found. Assume the salted passwords use `pbkdf` with `sha256` and 100,000 iterations.
Use the following Docker command to run `brute.py` when you test it:

```
you@laptop:  docker run --add-host=localhost:172.17.0.1 -it -v
$PWD:/app cs5435/cs5435 python3 brute.py
```

Task 1.5: Test your brute-force attack using the breach of salted passwords at `salted_breach.csv`. Use your brute-force attack to learn the plaintext password of each user and then using your credential stuffing attack, you should be able to successfully login to 3 more user accounts.

Part 2: Countermeasures

Let's add features to our web server to help protect user accounts better. First we will prevent credential stuffing attacks, at least for known breaches. We will store any available plaintext, hashed, and salted breaches on the web server. When a user attempts to register with a username that exists in one of these previous breaches, we will check to see if the password they are attempting to register with matches an entry in a breach. If so, we will reject the registration attempt.

Exercise 2.1: Implement the credential stuffing countermeasure in `do_login` in `app/api/login.py`. Edit `app/scripts/breaches.py` to load the three breaches. An error should be returned and the registration rejected if a user attempts to register with a breached username and password pair. A data model for storing the breached passwords in the database is provided in `app/models/breaches.py`.

Now that you have fixed the web server to protect users against credential stuffing attacks from breaches of other web servers, let's also fix our web server so that if it is breached, it will be of limited usefulness for use in future credential stuffing attacks.

Exercise 2.2: Implement salted passwords by editing `do_login` in `app/api/login.py` and the user data model in `app/models/user.py`. Use a hash function suited for password hashing such as `pbkdf`. Note that to ensure the registration script continues to work even after your data model changes, you must keep the signature of `create_user` unchanged meaning your hashing and salting logic to create a user should go inside `create_user`.

Part 3: Short Questions

Exercise 3.1: Consider an attacker that wants to guess some user's password, where the user's password pw is chosen according to a distribution p, i.e., $p(pw)$ represents the probability that pw was chosen by the user. Let q be the number of guesses the attacker can make.

1. In an offline brute-force guessing attack, what limits the number of guesses q made by an attacker? In an online (also called remoted) brute-force guessing attack, what limits the number of guesses q made by an attacker?

2. Suppose the adversary makes some number of q guesses. What is the optimal guessing strategy and what is its probability of success (as a function of p)? This is called the q-success probability of the distribution p.

3. Define Shannon entropy and given an example of a distribution p whose q-success probability is high and whose Shannon entropy is also high. Explain why Shannon entropy is a misleading estimate of password strength.

Exercise 3.2: Salts for password hashing prevent use of rainbow tables and, more generally, ensure that cracking effort against one user's hash is not reusable for another user's hash. Some web services will additionally employ a pepper, which is a single random value also added to the password hashing input and stored separately from the password hashes. Explain the purpose of a pepper.

Exercise 3.3: Describe how you would modify the web service to limit the efficacy of online guessing attacks.

Exercise 3.4: A colleague suggests you might strengthen password selection by enforcing a password composition policy, such as requiring an upper case letter, lower case letter, and symbol. Provide an explanation of why this is a bad idea. What should you do instead to encourage stronger password selection?

Final deliverables. You will submit the following in Canvas:

- PDF of solutions to Exercises 3.*
- Python files: `stuff.py`, `brute.py`, `app/api/login.py`, `app/scripts/breaches.py`, `app/models/user.py`