THE UNIVERSITY OF BRITISH COLUMBIA
Electrical and Computer Engineering Department

CPEN 442 **Introduction to Cybersecurity** **Winter Term 1, 2024**

Assignment 4: Authentication

Due date: **Friday, October 18, 2024, 11:59 pm**
Hard deadline: **Sunday, October 20, 2024, 11:59 pm**

Note: check the due date; you have more time to do this assignment than any of the previous ones.

Please use Piazza for questions and clarifications. Instructor and TA help will be provided only until 2pm on the same day as the *due date* above. This assignment has a 48-hour automatic deadline extension to account for unexpected issues when uploading the assignment. You are allowed to submit the assignment until the *hard deadline* date above, with no penalization. No exceptions after this date will be made!! If you have a long-term issue that will prevent you, or your group, from submitting the assignment, let the instructor know *well before the due date*.

Total marks: 100 pts
TA: Mohammed Elnawawy `[mnawawy@student.ubc.ca]`

Doing the assignment: This assignment can be done either **individually** or in **groups of two**. Regardless of what you do, you must join a group set in Canvas (check the `People` section, then `A4 group`). The group can be just yourself.

What to submit:

- A PDF file called `a4.pdf`, with the answer to all the questions below.
- A Python file `a4.py`, with your solution to Question 3.1; please see the provided template and the evaluation script in `grade.py`.

In this assignment, you will learn more about the password cracking attacks that we saw in the classroom. As with the rest of the assignment, you will submit this as a group, but make sure that you (individually) understand the answer to these questions, as it is likely that you will be evaluated on this content during the midterm and/or final exams.

1 Offline attacks: dictionary attacks

For this part of the question, it is highly recommended that you use the free password cracking tool John the Ripper, as that will save you a lot of programming time. John the Ripper is available for many platforms, including Windows, Linux, and macOS. If you decide to install this, make sure you install the "jumbo" version, as it contains support for many more types of hash algorithms. Please check the John the Ripper homepage (`https://www.openwall.com/john/`) for installation help. Note that, for Windows, the binary is already available, and you do not need to install it. For macOS, you can install it using the command-line package manager "Homebrew" (e.g., `brew install john-jumbo`). All the passwords that you will crack in this part of the assignment can be cracked using the dictionary attack that John the Ripper runs by default. If you prefer to program the dictionary attack by yourself, all the passwords in this part of the assignment can be found in the file `password.lst` available with the assignment (this is the same default file that John the Ripper uses for dictionary attacks).[1]

The following is an example to test that John the Ripper is working correctly for you. Consider the following hash:

```
0b14d501a594442a01c6859541bcb3e8164d183d32937b851835442f69d5c94e
```

This is the digest that SHA256 produces for the password `password1`. Save this digest to a text file, e.g., `digest.txt`, and then run

```
john --format=Raw-SHA256 digest.txt
```

The password cracking should print, among other things, the cracked password (`password1`) as part of the output. If you try running the command again, you will get a prompt telling you there are no passwords left to crack in the file. This is because John keeps a cache with the already-cracked passwords, that you can check by running

```
john --format=Raw-SHA256 --show digest.txt
```

[1]The passwords in this assignment have been randomly sampled from this file of popular passwords (`password.lst`); we did not sanitize the sampling, and we apologize for any "offensive" passwords that might have been selected.

You can also check the file `john.pot` (the location of this file depends on the OS in particular) to display information about the cracked passwords, and deleting this file resets John's cache.

Now that you got John working for one example, you are ready for the following questions. If anything is not working: check the documentation, etc.

[9 pts] Question 1.1: grab the file `Q11.txt`. Inside this file, there are three password digests. Using John the Ripper, as well as your deduction skills, indicate:

a) Which hashing algorithm has been used to generate each hash and how you deduced this.

b) What the passwords that generated these digests are.

All the following questions can be done with the default files `Q12.txt`, `Q13.txt`, and `Q14.txt`. If your computer cracks them too fast for you to see noticeable timing differences, you can try *all* of the following questions with the corresponding files ending with `_HARD.txt`. If your computer is taking too long to crack the default files, you can try to do *all* of the following questions with the corresponding files ending in `_EASY.txt`.

[4 pts] Question 1.2: inside `Q12.txt` you can find ten digests. These digests are using the Modular Crypt Format, which indicates the hashing algorithm, as well as other parameters. You can find more about this format in the following link: `https://passlib.readthedocs.io/en/stable/modular_crypt_format.html` As answer to this question, indicate the hashing algorithm that was used for the hashes under `Q12.txt`, as well as any other information that you can get from the Modular Crypt Format.

[4 pts] Question 1.3: Use John the Ripper to crack all the passwords in `Q12.txt`. Then, do the same for `Q13.txt`. You do not need to provide the cracked passwords in your submission, but take a look at the runtime in each case. Explain which file took longer to crack and *why*.

[4 pts] Question 1.4: Now, consider the digests under `Q14.txt`. Crack them, and take a look at the runtimes. (You do not need to provide the cracked passwords in the submission.) Comparing the time that it took to crack `Q13.txt` vs. `Q14.txt`: *which* took longer to crack, and *why*?

[9 pts] Question 1.5: consider again the digests under `Q12.txt`. Imagine you have been hired to provide a security assessment of the password policies of a company, and these are the digests stored in the authentication server at that company. Using what you have learned in the previous questions so far, indicate *three* weaknesses about this password policy that you should indicate in your report, and *how to fix* them.

2 Offline attacks: brute-forcing

Consider a system where the users choose their passwords uniformly at random, following one of the password generation strategies below:

A) Pick *six* random lower-case words from the diceware list (chosen by rolling five 6-face dice)[2], and concatenate them, separating them with `-` characters. Then append a random digit `[0-9]`.

B) Pick a random 12-character password, where each character is from the set of all ASCII printable characters (95 possible values for each character).

[10 pts] Question 2.1: for *each* of these password generation strategies, determine *how many* possible different passwords can be generated.

[6 pts] Question 2.2: assume that the SHA-1 hash of a password is leaked. The adversary has a machine that can compute 10^{12} SHA-1 digests per second. For *each of the two strategies* above, what is the *maximum* amount of time that would take the adversary to break the password?

[6 pts] Question 2.3: Next, consider the case in which the server picks a random "salt" that consists of 6 characters from the set of all printable ASCII characters (95 possible characters). The server concatenates the password and this salt, and then computes the SHA-1 digest of this. Assuming the adversary gets hold of the digest *and* the salt, for *each of the two strategies* above, what is the *maximum* amount of time that would take the adversary to break the password?

3 Online attacks

As part of the assignment files, you can find a python script (`authentication_server.py`) that simulates an authentication server.

```
class Server():

    def __init__(self):
        """This simulates the user's registration in the server with a certain
            password, stored in self.password"""
        L = random.randint(5, 10) # random length
        random_bytes = random.getrandbits(80).to_bytes(10, 'little') # random
            bytes
```

[2] https://www.eff.org/files/2016/07/18/eff_large_wordlist.txt

```
        self.password = base64.b64encode(random_bytes).decode('ascii')[:L]

    def check_password(self, password):
        """This simulates the server checking if the provided password matches
           the one stored in self.password"""
        if len(password) != len(self.password):
            return False
        for i, char in enumerate(password):
            time.sleep(0.001) # protect against brute force
            if char != self.password[i]:
                return False
        return True
```

The server creates a random password when initialized (stored as a string of base64 characters). To deter brute-force attacks, the password checker function has a call to `time.sleep`.

Without modifying this server, you must write a python function that cracks the server's password. Inside `a4.py`, complete the function `online_attack`, which receives the `check_password` method from the server and returns the cracked password as a string. We will evaluate this using `grade.py`. You can run this script yourself after changing the second line in `grade.py` to import your implementation. Your score will be determined by your success rate and the average time it takes your script to crack one password, as depicted in Figure 1. We will grade this by running it in the ECE Linux servers, so you can check how your script does in these servers to get an idea of what your grade will be.

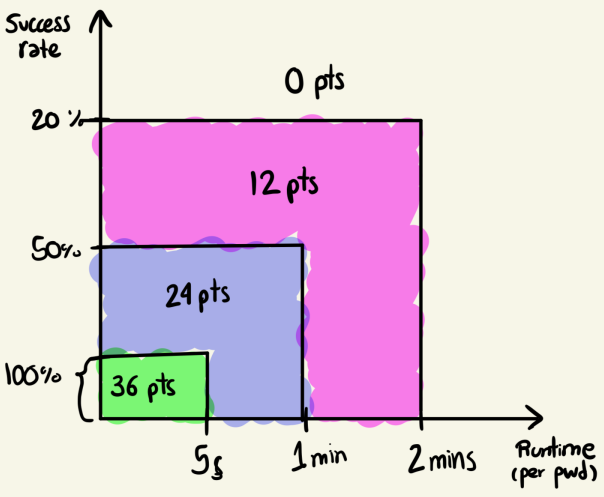

Figure 1: Grading for Question 3.1. The x-ticks are at 5 seconds, 1 minute, and 2 minutes. The y-ticks are at 20%, 50%, and 100%.

Note: Your function cannot use Python tricks to grab the variable `self.password` directly. Also, getting points in this exercise is not hard, and the easiest way to get a zero (in the whole assignment) is by copying this exercise.

[36 pts] Question 3.1: provide your script `a4.py`. Grading will be done using `grade.py` and following the chart in Figure 1 in the Linux ECE servers. In `a3.pdf`, for this question, simply mention which success rate and runtime you got when you tested your program.

4 Two-factor Authentication

The following link contains a list of services that support 2-factor authentication: `https://2fa.directory/`

For every member of you group: go to the link and pick a service from that list that you use.

[12 pts] Question 4.1: What service did you pick, and what forms of two-factor authentication does it support? What would it take for an attacker to circumvent that service's two-factor authentication? (Give an example in which the adversary would bypass the authentication.)