Lab 1: Password Cracking and Crypto Hash Functions

Wednesday sections submit by 2/11

Monday sections submit by 2/16

See useful instructions for all labs in Lab 0.

Passwords are the most well-known and widely used method for user authentication and have many advantages. Passwords are simple to use and implement; do not require hardware; are versatile; and more. Passwords are strings of characters. The number of **potential** passwords is large; for example, there $64^8 = 2^{64}$ passwords using eight characters chosen from the set of 52 alphabetic characters, ten numerals and two special symbols. Trying all these would be very challenging; a few more characters or options, and it would exceed the capacity of supercomputers. And typically, the login process introduces delays and limits to a reasonable, relatively small number of guesses.

However, passwords are often vulnerable. Many people choose passwords which are easily guessed, such as 123456. Large sets of commonly-used passwords are available, and referred to as a **dictionary**; research has repeatedly found that a very large percentage of the passwords in use, can be found in such dictionary of common passwords, e.g., half of the passwords in a dictionary of the hundred-thousand most common passwords.

In this lab, we will experiment with different ways for attackers to guess passwords – and with some of the basic defense mechanisms. The defenses we will learn will involve **cryptographic hash functions**. A cryptographic hash function receives a string of arbitrary length as input, and outputs a fixed length string. In this lab we'll focus on the SHA-2 standard hash function, with 256-bit length outputs; the input strings are usually much longer.

The two main security requirements from cryptographic hash functions are the **one-way** and **collision-resistant** properties**.** Basically:

- A hash function *h* is **one-way** when an attacker provided with the hash *h(x)* of some random input *x*, cannot, in reasonable time, find *x,* or any other input *x'* such that *h(x')=h(x).*
- A hash function *h* is **collision-resistant** if it is infeasible to find two different inputs *x, x'* such that *h(x)=h(x').*

Cryptographic hash functions are a very important and widely used cybersecurity tool, and, as we will see, very important for password security. This application uses the one-way property; in the next lab, we will use collision-resistance.

In this lab, use the SHA-256 hash function, which can be found in (1) the *hashlib* library, which should be installed on the VM, (2) the PyCryptodome library, or (3) the *cryptography* library. Use one or more of these crypto libraries for all relevant questions in this lab. See in *LabGen.*

An Important Note About Passwords in CSE 3140

The passwords assigned to fictitious users are pulled randomly from a real-life source of leaked passwords that cybersecurity specialists often come into contact with professionally. Unfortunately, many of these passwords contain extremely vulgar and offensive language, and we do not condone their use. We are happy to substitute passwords from artificial sources at your request.

Submitting Code for this Lab

This lab will require you to submit a zip file containing the scripts you created for each of the programming questions. Your code will be graded on correctness, efficiency, structure, and readability. At the beginning of this lab, you should create a **Solutions** directory within the Lab 1 directory and save all your .py scripts for this lab within it. After completing this assignment, you can then:

- Archive the Solution directory by running the following command in your Lab 1 directory:
    - zip -r lab1_sol.zip Solutions
- SCP (secure copy) the zip file to your local machine. On your local machine (after logging out of your VM) you can run the following command (you need to replace <vm_ip> and <local_path> with your own VM IP and destination file path on your local machine, respectively):
    - scp cse@<vm_ip>:/home/cse/Lab1/lab1_sol.zip <local_path>
- Upload the lab1_sol.zip file you saved to your machine on HuskyCT

Question 1 (10 points)

Many users select weak passwords; for example, in 2020, the password ***'123456'*** was used by over 2 million users and exposed over 23 million times, check this link. The file **MostCommonPWs** lists a few of the most common passwords.

As a cybersec researcher, you are asked to help find the password to the account of The Red Falcon, a notorious supervillain, in the darknet server of The Red Falcon's supervillain gang. The Red Falcon's username is **SkyRedFalcon914**. Since The Red Falcon didn't study cybersecurity and is known for villainy rather than intelligence, you decide he is likely to use one of these very common passwords. So, try common passwords against the login program of the gang's server, **Login.pyc**, to find out The Red Falcon's password. You can do this manually, easily, and it's not a bad idea to do so as a warm-up exercise. However, you must then write a simple Python script, **Break1.py,** that will find The Red Falcon's password by trying the different passwords against Login.pyc. Your program, Break1.py, should also print the time, at both the beginning and the end of the run; the runtime should be very short.

To execute another program within your Python script, you can import the **subprocess** module and use the **run** function that is defined within it. Here's the documentation for that function.

Note: the file **LoginTemplate.py** may help you understand how **Login.pyc** works; we created Login.pyc from it by replacing the string `##passwrd' with a random password from MostCommonPWs.

Submission Requirements

Submit in the submission webserver (IP: 10.13.4.8): The Red Falcon's password.

Upload all solution code alongside lab report in a zip file.

Question 2 (15 points)

After breaking into The Red Falcon's account, you found there a file, **gang**, that contains all usernames in the server of the supervillain gang. Surely, a few of these guys also use one of the common passwords in the file MostCommonPWs! Extend your code from Break1.py and create a new script, **Break2.py**, which will find different gang member that uses one of the passwords in MostCommonPWs, by using the login program **Login.pyc**.

Submission Requirements

Submit in the submission webserver: the name of the gang member and their password.

Upload all solution code alongside lab report in a zip file.

Also in lab report: Include a screenshot of your script's finished execution, printing out the name and password as well as the start/end time.

Question 3 (15 points)**

Even when users do not use one of the obviously weak passwords such as these in MostCommonPWs, they still often use commonly used passwords. We now want to find passwords of an additional supervillain gang member, whose password is from the `dictionary' of 100,000 common passwords, which you will find in the file **PwnedPWs100K** (downloaded from https://haveibeenPwned.com – a useful resource, visit it!).

Extend your code from Break2.py and create a new script, **Break3.py,** that will try out all of the passwords in PwnedPWs100K, for gang members whose password was not found yet. Run this modified program (Break3.py) and see how long it takes to find the password for *one additional* gang member using **Login.pyc**. Try to make your program efficient! Note: while in reality the passwords in the file are sorted by popularity, we pick them uniformly, i.e., each password is equally likely.

****IMPORTANT:** Even when programming with efficiency in mind, this script is checking a very large number of passwords and will most likely take MANY HOURS (up to 24hrs) to run and find the correct password. As a result, you will need to:

- run your script inside a **tmux** session so that your VM does not stop its execution when your SSH session ends. Refer to our tmux tutorial on HuskyCT in the Lab1 folder.
- test your script thoroughly before letting it run for many hours! You can even use simple print debugging statements to help with this. It's very easy for an insidious bug (e.g., forgetting to strip off newlines) to prevent the correct password from being found. I recommend making your own test version of Login.py where you hard code a chosen password from PwnedPWs100K (choose one very early on in the file) and test your script against that test login program first. Also, **make sure your script breaks as soon as it finds the correct password** to prevent any unnecessary runtime!

Submission Requirements

Submit in the submission webserver: the name and password of new gang member whose credentials you discovered in this question.

Upload all solution code alongside lab report in a zip file.

Also in lab report: Include a screenshot of your script's finished execution, printing out the name and password as well as the start/end time.

Question 4 (15 points)

Often, attackers get hold of a leaked file of passwords and use it to break into accounts of the user at other services where the users reused the same passwords; this is called `credential stuffing attack'. There were **billions** of passwords exposed in the recent years, and the numbers seem to even grow over time. Attackers often use a password file exposed from one service/site, to break into accounts of the same users in another service/site – since many users use the same password in multiple sites.

In this question, you will use this technique to find the passwords of additional gang members, where the techniques of previous questions failed. In fact, these will be quite random passwords. You receive an exposed passwords file, **PwnedPWfile**, and will write a program, **Break4.py,** that will search for gang member which have an account (and password) in the file**.** For any gang members whose passwords are listed in **PwnedPWfile,** Break4 will check if the password `works' against the login program **Login.py**.

Submission Requirements

Submit in the submission webserver: the name and password of new gang member whose credentials you discovered in this question.

Upload all solution code alongside lab report in a zip file.

Also in lab report: Include a screenshot of your script's finished execution, printing out the name and password as well as the start/end time.

Question 5 (15 points)

To reduce the exposure from a leaked passwords file, password files are usually kept in a form which makes it easy to check if a given string is the correct password (of a given user), but harder to find the passwords given only the file. In this question we discuss the more basic form of this defense, where the file contains the **hashed passwords**. You are given another exposed passwords file, **HashedPWs,** which contains, for each user *x*, the results of applying a **cryptographic hash function** *h(.)* to the password PW_x of user *x*, i.e., $h(PW_x)$. (In the next question we will see an improved defense.)

Cryptographic hash functions *h(.)* are efficient functions mapping from arbitrary-long strings into short, fixed-length strings, e.g., 160 bits. They have many applications, and several security requirements. The application of making it harder to abuse an exposed passwords file relies on the **one-way** property, which basically says that given *h(pw),* the hash of a password *pw*, should not help the attacker to find *pw* (or any other password *pw'* which will hash to the same value, i.e., *h(pw')=h(pw)*).

Write a new program, **Break5.py,** that uses the file **HashedPWs** to find, as quickly as possible, the passwords of these additional gang members. It will be infeasible to test all random passwords (why?); instead, focus on gang members who pick a random password from PwnedPWs100K, and concatenate to it **two random digits**. (Many users do such minor tweaks to their passwords, to bypass password-choice requirements, or in the incorrect hope that this suffices to prevent password guessing.). For gang members whose passwords you recover using **HashedPWs, use Login.pyc** to check if the gang member used the same password.

To compute hashes: The hashes for this questions were computed with a SHA256 hashing algorithm. In Python, you can compute the SHA256 hash of a string using the following process:

```
import hashlib
h = hashlib.sha256()
h.update(bytes(string_var, 'utf-8'))
hashed_guess = h.hexdigest()
```

Hint: think carefully how to do this question efficiently, or it may be quite slow.

Submission Requirements

Submit in the submission webserver: the name and password of the new gang member whose credentials you discovered in this question.

Upload all solution code alongside lab report in a zip file.

Also in lab report: Include a screenshot of your script's finished execution, printing out the name and password as well as the start/end time.

Question 6 (15 points)

To further improve security, password files usually do not contain the hash of the password PW_x of user *x*. Instead, the password file contains two values for each user *x*: a random value $salt_x$, called **salt**, and the result of hashing of a combination of the password PW_x and of the salt. In this question, you are given a file **SaltedPWs** which contains, for each user *x*, the pair $(salt_x , h(salt_x + PW_x))$, where $salt_x$ is a random value chosen for user *x.* Here we use the + sign to denote concatenation.

Write a new program, **Break6.py,** that uses the file **SaltedPWs** to find the passwords of as many additional gang members as possible that use passwords from PwnedPWs100K concatenated with **one** random digit, as quickly as possible. Break6 should also save a file containing the names and corresponding passwords. Confirm the exposures using **Login.pyc**.

Submission Requirements

Submit in the submission webserver: (only) the name and password of one gang member exposed (only) in this question.

Upload all solution code alongside lab report in a zip file.

Also in lab report:

- Include a screenshot of your script's finished execution, printing out the name and password as well as the start/end time.
- Explain why the SaltedPWs would (normally) be harder to attack, compared to HashedPWs.

Question 7 (5 points)

The https://haveibeenPwned.com service collects passwords from the many password-file exposures, and allows users to detect when their password was exposed; it also allows service providers to detect when a user is trying to use an already-exposed password. In this question, we ask you to use this service for at least your UConn email(s); do this from home or campus – you will need to be connected to the Web.

Submission Requirements

Submit in lab report (only): Screen shots showing your registration and results.

Question 8 (5 points)

One may expect every `serious' web service to use salted passwords file, no? Unfortunately, this is not the case; in fact, some services (maybe not that serious) did not even use hashing – their password files just contained passwords in the clear! Search for such incidents. You may be surprised!

Submission Requirements

Submit in lab report (only): Short description, in your own words, of the two `worst' exposures you could identify, one of plain passwords and one of hashed-but-unsalted passwords. Explain why you consider these two to be the worst exposures. Include a reference to at least one reliable source of information on each of the exposures.

Question 9 (5 points)

Password-based authentication relies on the user's knowledge of the password; in general, authentication methods relying on information known to the user are referred to as *something you know* authentication factor. The two other main authentication factors are *something you have* and *something you are*. Search these terms online to understand them better and to identify popular websites that support other authentication factors, instead of relying on passwords or in addition to passwords (*two-factor authentication,* aka *2FA*).

Submission Requirements

Submit in lab report (only): An example of a `significant' website which does NOT support 2FA, and an example (non-UConn) website which supports 2FA. Include screen shots and explain why you chose these particular websites. Do you personally use 2FA, in any accounts besides your UConn account? Do your parents?

Some further comments

Many clients use simple tweaking of passwords between different websites/services, instead of sharing them exactly as-is. While this is surely better than reusing passwords, attackers are aware of this practice and have been known to also attack using tweaks of exposed passwords. If interested, you may find the following paper from the Usenix Security conference interesting.