Homework Assignment 2

Cybersecurity

COSC 3371 / 2022 Spring

Please solve the following problems by answering each question based on the information that is provided here as well as what you learned in class. Please submit two files on Blackboard:

- text file containing all the numbered questions for Problems 1 and 2 with your answers inline (you can write your answers into this file and submit it as a PDF or Word file);
- source code file(s) for Problem 2.

Problem 1: Unix Access Control [3 points]

In this problem, you will evaluate file access permissions on a shared Unix server. The following users are present on this server: `garfield`, `odie`, and `jon`. There are two groups on this server: `pets` and `humans`. Users `garfield` and `odie` are members of `pets`, while user `jon` is a member of `humans`.

The terminal output below shows the Unix permission bits for the files and directories in the working directory, which you will need to use to answer the questions for this problem:

```
server:/files/$ ls -l
dr--r-xrwt 2 jon      humans    4096 Mar 23 snacks
dr-xr-xr-x 2 jon      humans    4096 Mar 24 toys
----r--rw- 1 garfield pets     16384 Mar 12 lasagna.jpg
```

The output below shows a file within the directory `snacks` and its permission bits:

```
server:/files/$ ls -l snacks/
-rw-rw-r-- 1 odie     pets      4096 Mar 20 icecream.txt
```

The output below shows a file within the directory `toys` and its permission bits:

```
server:/files/$ ls -l toys/
-rw-rw-r-- 1 odie     pets      8192 Mar 18 bone.png
```

Note that

- `d` character in front of the permission bits denotes a directory;
- `t` character within the permission bits means that both the sticky and the others execute bit are enabled;
- questions assume that the users do not change any permissions before trying to access the files or directories.

Please answer the following questions. If your answer is yes, **please list which permission bit(s) of which file (and directory) authorize the access**; if your answer is no, **please list which permission bit(s) deny the access**.

1. Can user `jon` read the contents of file `lasagna.jpg`?
2. Can user `garfield` read the contents of file `lasagna.jpg`?
3. Can user `odie` modify the contents of file `lasagna.jpg`?
4. Can user `jon` list the names of the files within directory `snacks`?
5. Can user `jon` read the contents of file `icecream.txt` in directory `snacks`?
6. Can user `garfield` delete the file `icecream.txt` in directory `snacks`?
7. Can user `garfield` modify the contents of file `bone.png` in directory `toys`?
8. Can user `odie` delete the file `bone.png` in directory `toys`?

Problem 2: Password Cracking [3 points]

In this problem, you will experiment with offline password cracking. First, download and unzip the attachment `Problem2.zip`, in which you will find three files:

- `password_dictionary`: list of possible passwords[1];
- `users`: list of users with their usernames and hashed passwords;
- `users_salted`: list of users with their usernames, randomly chosen salt values (strings), and salted & hashed passwords.

These users were not very imaginative (or careful), so they chose simple variations of English words as their passwords, which are all included in the attached dictionary. Hash values were computed using standard SHA-256 from the ASCII (or UTF-8) encoded password strings, and they are represented in the files `users` and `users_salted` in hexadecimal format. Salt values are mixed into the passwords by simply concatenating the salt string and the password, i.e., a salted & hashed password is `SHA-256(salt | password)`.

Problem 2.1: You've got to crack a few eggs to make an omelet

Suppose that an attacker has stolen the password storage file `users` and would like to find the users' passwords.

1. Write a Java or Python program that brute-force searches for the first user's password using the list `password_dictionary`, and measure how long it takes to find the correct password on your computer. Please include the measured running time in your answer to this question.
2. Calculate how long it would take to find each and every user's password if you performed the same brute-force search for every user. Please include your calculation and result in your answer.
3. Calculate how long it would take to find each and every user's password with brute-force search if the users' passwords were 8-characters long and chosen uniformly at random from lower- and upper-case letters and digits. Please include your calculation and result in your answer.

Problem 2.2: All your passwords are belong to us

Since performing a brute-force search for every password takes a lot of time, attackers will try to pre-compute the hashes if possible.

4. Write a Java or Python program (or extend the previous program) that computes the hash value of every password in the list `password_dictionary`, stores the hash-

[1] Note that this is a small dictionary for demonstration purposes only, which consists of less than 5 million variants of English words. Real attacks use larger dictionaries, which often include commonly used passwords and passwords that have been stolen in prior attacks.

password pairs in a Java `HashMap` or Python dictionary (or similar data structure)[2], and finds each and every user's password by simply looking up this data structure. Measure how long it takes to find every user's password on your computer using this program. Please include the measured running time in your answer to this question.

Problem 2.3: Why so salty?

Attacks based on pre-computed hashes can be prevented using password salting.

5. Select two users from `users` who have the same hash values (i.e., same passwords), and verify that their hash values are different in `users_salted`. Please include the selected users' usernames, password (recovered in the previous step), and hash values from `users_salted` in your answer.
6. Compute these users' salted & hashed passwords using the salt values included in `users_salted`, and verify that the hash values in `users_salted` are correct.

[2] In practice, attackers may use more efficient data structures for pre-computing hashes, such as rainbow tables.