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`?
   Yes, lasagna.jpg has public permissions of r/w
2. Can user `garfield` read the contents of file `lasagna.jpg`?
   No, garfield does not have read permission for lasagna.jpg as owner
3. Can user `odie` modify the contents of file `lasagna.jpg`?
   No, group pets only have read permission to lasagna.jpg.
4. Can user `jon` list the names of the files within directory `snacks`?
   Yes, jon has read and execute permission to the snacks directory
5. Can user `jon` read the contents of file `icecream.txt` in directory `snacks`?
   No, jon cannot read icecream.txt. This is because he does not have execute permission of the snacks directory as owner to see details on the files.
6. Can user `garfield` delete the file `icecream.txt` in directory `snacks`?
   No, garfield cannot delete icecream.txt because he doesn't have execute permission on the file. Public group has full permission to directory.
7. Can user `garfield` modify the contents of file `bone.png` in directory `toys`?
   Yes, garfield has write permission on the file under group pets. All users/groups have read/execute permission to read and view directory contents.
8. Can user `odie` delete the file `bone.png` in directory `toys`?
   No, odie does not have execute permission on bone.png as owner.

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.

   Done. See 2-1.py . Password is 'propositionize5' and took 5.643858909606934 seconds to find.

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.

   Max time per user = ~8.749659776687622 seconds (Just commented out the break statement)
   With that in mind, on average, the average password crack time per user is ~4.37482988834 seconds and with 64849 users it should take 283703.343429 seconds.

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

[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.

random from lower- and upper-case letters and digits. Please include your calculation and result in your answer.

Lowercase = 26, uppdercase = 26, digits = 10
26 + 26 + 10 = 62 variations per characters
With total length of 8 characters
62^8 = 2.1834010558 x 10^14 password variations
Time varies from setup to setup, but 8 characters is not feasible for a password as of today with modern computer builds.

Will be using an arbitrary number for calculation speed in this calculation.
Since we already know that the password is 8 characters long, composed randomly of lowercase/uppercase letters and numbers we will not compute time spent of 7 characters of less and only focus on passwords of 8 characters.

Time_required_to_crack_per_second = total_combinations / calculations_per_second
i.e. seconds = total / speed
218.34010558 seconds = 2.1834010558 x 10^14 / 1,000,000,000
Though the birthday paradox still applies so the real answer is half of this. So the real answer would be an average of 109.17005279 seconds per user.

With there being 64849 users it would take an average total of 7079568.75338 seconds to crack all of the users passwords.

To dive further with a recent source I found from 2022. It states that these passwords can easily be cracked in 7 minutes or less for passwords containing 8 characters of numbers and upper/lowercase letters.
https://www.techrepublic.com/article/how-an-8-character-password-could-be-cracked-in-less-than-an-hour/#:~:text=Must%2Dread%20security%20coverage&text=As%20described%20in%20a%20recent,the%20latest%20graphics%20processing%20technology.

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-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.

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

Calculated time: dictionaries work close to O(1)
See 2-2.py for code for this problem.
For the entire program to compute including creating dictionary/keys, it took ~18.854063987731934
For the last 2 users Eden, Farine their passwords are forebridge1 and 9townsman respectively

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.

   Farine
   hash: 64c2b210358c8e27c8cbfcaa48f4adcc2c278f9c2d05ed546b533a26c72f3a94
   salt: Voc0sMDUUck20p2hJUsLqMKEcWB9aOrT
   salted hash:
   0bf88c448314f45ca241de98349e26641411ad4b50aaf561776bdf2c5740a970

   Shahla
   hash: 64c2b210358c8e27c8cbfcaa48f4adcc2c278f9c2d05ed546b533a26c72f3a94
   salt: 8W6Gr5ZKMqowmCba01ghjZdvT3W4lAE0
   salted hash:
   4228521b924eeff2c354510bd0dd5757fce2abd4fceeab1a36d7bde57d7aee11

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.

   Used IO redirect to txt file for 2-2.py to get passwords

   ```
   python3
   Import hashlib
   # Farine
   password = "9townsman"
   salt = "Voc0sMDUUck20p2hJUsLqMKEcWB9aOrT"
   saltedpass = salt + password
   print(hashlib.sha256(bytes(saltedpass, "UTF-8")).hexdigest())
   - 0bf88c448314f45ca241de98349e26641411ad4b50aaf561776bdf2c5740a970

   # Shahla
   password = "9townsman"
   salt = "8W6Gr5ZKMqowmCba01ghjZdvT3W4lAE0"
   ```

```
saltedpass = salt + password
print(hashlib.sha256(bytes(saltedpass, “UTF-8”)).hexdigest())
- 4228521b924eeff2c354510bd0dd5757fce2abd4fceeab1a36d7bde57d7aee11
```

Both salted hashwords are correct with the calculation given above