Week 2 Lab: Breached Passwords

Background:

We often hear about companies having their passwords compromised. Ideally, when a website stores your password, it is encrypted as a hash, as that reduces the vulnerabilities if an attacker was to get the list of encrypted passwords. If stored with a symmetric key, the attacker could simply search the website to find the key (or a theoretical list of keys) to the passwords and simply decrypt them.

However, you may have heard of "salting" a password. Does adding a string, that is stored in plaintext next to the password anyway, actually prevent successful attacks? To answer this question, you will be performing a dictionary attack on the salted data.

Files:

The three files you have for this are:

breached_data.txt

breached_data_salted.txt

passes_real.txt

The first two files are collections of data, containing a username, salt (in the salted file), and the hash of the password stored for this username, with each line having the information for each account. The file passes_real.txt contains the top 10,000 most common passwords, pulled from Wikipedia.

You can write your code to analyze these files in Python or C/C++.

Lab Steps:

Step 1:

Make a file called generate_unsalted_table to generate the hash table for the unsalted data, so you can compare the hashes of passwords. The format for the salt in this table is:

hash(password) using SHA256 hashing.

Step 2:

Make a file to compare the generated hashes to those in breached_data.txt. Put the results in cracked_passwords.txt. Keep count of the number of passwords you crack and take note of the time it takes.

Side Note:

Let's do some math before going to Step 3. Dictionary attacks require comprehensive hash tables, because you don't know what the . In breached_data_salted.txt, there are 1000 salted, hashed passwords; they don't all have different salts, but let's assume they do. We also have 10,000 possible passwords. If we made an entry for each possible password combined with each possible salt, how many lines would our file have?

Step 3:

Make a file to generate the hash table for the salted data, similar to Step 1, called generate_salted_table. The format for the salted hashed passwords is:

hash(salt+hash(password))

For feasibility and for you all to turn in, **limit the number of salts in your dictionary to 100**. If you have space on your computer and you are curious, I highly encourage you to generate a larger file to get a better idea of the time it takes, but **do not turn in that version.**

Step 4:

Similar to Step 2, compare the salted hash table to breached_data_salted.txt and store any cracked passwords in cracked_passwords_salted.txt. Keep count of the number of passwords you crack and take note of the time it takes (even at 100 salts, but try at more).

A Few Questions

1. What is your result from the Side Note?
2. If attempting to crack every password, which table for attacks would be bigger, salted or unsalted?
3. If attempting to crack every password, which dictionary attack takes longer to complete, salted or unsalted?
4. Would salting help against targeted attacks (i.e. you want to hack one, specific account), assuming you have the breached data?
5. If your password is salted and stored properly, do you still need a unique, complex password? Why?

To Turn In:

Code to generate salted and unsalted tables

Code to compare salted and unsalted tables to breached data

A document with the answers to the questions.

Please zip these files together and turn into Camino by the end of your lab Week 3.

Mon: 10/5 at 8pm

Tues: 10/6 at 5pm