How good is a P@ssword?

Intro

How many passwords do you have? It could be dozens! You certainly use at least that many platforms... but you may reuse your passwords between sites.

Passwords give you access to your social circle through social media, information about your work and studies, and even your finances.

Have you ever used a password that was your name? What about a family member's name? Or something like P@ssword or 12345? If you're curious, you can look for your passwords on https://haveibeenpwned.com/Passwords to see if they have already leaked to networks of hackers.

(Kibo recommends using a password manager - you can revisit this section of the laptop setup guide in case you aren't using one already: laptop setup guide - password manager)

The aim of this project is to walk us through the logic behind generating passwords, develop intuition for how they are often attacked and what we can do about it, as well as ways to store them safely.

By the end of this project, you will have compared different password schemes, you will also calculate the efficacy of various attacks on those schemes, and you will apply your theoretical understanding of sets, probabilities, and relations to figure out how to best store these passwords.

Instructions

The project will have 3 parts:

- Part 1: Password Generation

- Part 2: Attacking Passwords

- Part 3: Storing Passwords

More information will be published here when the project is assigned.

- Part 1: Password Generation

- Part 2: Attacking Passwords

- Part 3: Storing Passwords

Each part will involve applying a different set of mathematical concepts you've learned.

Deadlines

This is a multi-week project.

You have until Nov 21 to finish all of the required parts of the project.

You are expected to submit your work by these deadlines:

- Nov 7: Part 1

- Nov 14: Part 2

- Nov 21: Finished Project (all three parts)

Your answers will only be scored after the finished project deadline. You are encouraged to revisit your answers to Part 1 and Part 2 if you think you can improve them.

You have access to the entire project now, but we have not have covered all the topics yet. Plan to revisit the project each week as you learn the skills required to solve each part of the challenge.

Submission

For each deadline, there will be a gradescope link to submit your work. You'll be able to revise and resubmit those parts for the final submission deadline. The final submission is what will be graded.

For each of the questions, create a pdf or scan an image to show your work. It may be easiest to write your solution by hand, then upload a photo of that instead of typesetting your work in a text editor; feel free to do either.

Notes

- For questions that require you to perform calculations, make sure to show your work.

- For questions that ask you for a hypothesis or opinion, we do not expect a ***correct*** answer. Feel free to share your genuine thoughts, and we will revisit them later through the exercise.

Part 1: Password generation

In the first part of the project, you'll think through different sets related to passwords, and calculate the sizes of those sets. The sizes of the sets of possible passwords will act as a lens for

thinking about the probabilities involved with guessing passwords, as well as to start building an intution for how 'secure' a password is.

A set of pins

Here's an example of the set of 2-digit pin numbers, using the digits 0-9.

{ 00, 01, 02, 03... 09, 10, 11, 12, 13... 19, 20, 21, 22, 23... 29, ... 90, 91, 92, 93... 99}

It's all the permutations of two digits. Since there are 10 options for each choice, and two choices, the total number of possibilities is 10^2, so 100 possible 2-digit pin numbers. With only 100 possible pin numbers, this probably isn't very secure.

Here's one way to think about the insecurity: if you and your classmate each chose a 2-digit pin code, you'd have a 1% chance of choosing the same code.

Now you try: 4-digit pin

1. Calculate the size of the set of 4-digit numeric pin codes.

2. Then, calculate the probability that your classmate would choose the same code as you, if you each chose randomly.

More password schemes

3. For each of the following password schemes, compute the size of the set generated by the scheme, and the probability of two randomly-chosen passwords colliding.

- S0: 2-digit numeric pin [0-9]

- S1: 4 digit numeric pin [0-9]

- S2: 8 digit numeric pin [0-9]

- S3: 8 character lowercase alphabetic password [a-z]

- S4: 8 character alphabetic password, upper- and lowercase [A-Z][a-z]

- S5: 8 character alphanumeric password [A-Z][a-z][0-9]

- S6: 8 to 12 character password, alphanumeric plus spaces and symbols [A-Z][a-z][0-9] [~`!@#$%^&*()-_+=,<.>/?:;'"[{]}]

- S7: 8 to 12 character password, same set as S6, but with restrictions:
  - must use at least one number

  - must use at least one uppercase and one lowercase character

  - must use at least one symbol

Name	Size	% chance of collision
S0	100	1%
S1		
S2		
S3		
S4		
S5		
S6		
S7		

4. Compare schemes S1 through S6 - which seems most secure? Explain your reasoning.

5. Compare S6 to S7 - which seems more secure? Which is more common? What is your hypothesis as to why S7 may be preferred to S6?

Part 2: Attacking passwords

In Part 1, you got a rough sense of the security of a password scheme by asking how likely it was that two random passwords from the same scheme would match. But... that's not actually what happens in the real world. In the real world, passwords have to be secure against an *attacker*.

In order to understand how secure different password schemes are, you'll step into the shoes of an attacker. Read the descriptions and complete the prompts below to think through password security from the point of view of breaking a password.

Brute force

The simplest method for attack is to guess all the possible passwords. This is called "brute force" because there's nothing clever about it. It's surprisingly effective against many password schemes.

The basic idea is to try everything, very quickly. Just how fast can a computer check a password? Quite fast, it turns out! The limiting factor depends a lot on what kind of passwords are being checked, and how they are stored, but the top results are usually measured in *billions*

of passwords per second. Those speeds are typically for 'offline' attacks, where the attacker can try passwords without needing to interact with the system they are attacking. For 'online' attacks, where the attacker has to interact with the service under attack, speeds are lower and techniques are more complicated (but still shockingly fast!)

Questions

1. Let's assume that a dedicated attacker can try 1 million passwords per second. Answer the following questions for each of the password schemes S1-S7 from Part 1.

- How quickly would every possible password be tried?

- How likely is it that the password would be broken within 1 minute of trying?

- How likely is it that the password would be broken within 1 hour of trying?

- How likely is it that the password would be broken within 1 day of trying?

(Give the probabilities as a % chance)

2. A common mechanism to mitigate brute force attacks is to limit login attempts. An attacker would have to wait a 'cooloff' period before they get to try more passwords. Answer the following questions for each of the password schemes S1-S7.

- How likely is an attacker to correctly guess the password in 3 attempts? (give your answer as a %)

- If after every 3 attempts, there was a 5 minute 'cool off' period, how quickly would the scheme be guaranteed to be broken? (give your answer as a number of seconds)

- If you were an attacker, what ideas do you have to get around such a scheme?

Dictionary attacks

Trying every possible combination is not the most effective strategy. Attackers are smart - they can try guessing passwords that are more likely! People are predictable. They need to choose passwords that they can remember, so they end up choosing some passwords much more frequently than others.

Consider the password scheme S1 - a 4-digit numeric pincode. What do you think are the most common 4-digit pincodes?

This post analyzed the most frequent pincodes from a database of leaked credit card information. A stunning 11% of pincodes were `1234`! As detailed in the post, lots of people seem to use their birth year as their pincode.

An attacker doesn't need to try every single password. They can try them in order of how likely people are to use that password.

This style of attack is called a *dictionary attack*. Instead of trying random passwords, it guesses from a 'dictionary' of frequently used passwords.

3. Assume an attacker is using a dictionary attack against the pin codes (scheme S1), starting with the top 20 passwords from that post. Assume that they can try 1 password / second. Given the % of users who would choose one of those as their 4 digit pin, what is the % chance that a password will be cracked within 1 minute? 1 hour? 1 day?Asumme that for there is a uniform distribution of frequencies for passwords between the 21st and the 9,979th.

Dictionary attacks for other passwords

What would an attacker need in order to conduct a dictionary attack for alphabetic passwords? Instead of a list of commonly used pins, the attacker could use lists of *words*.

Consider S3 (8-character lowercase alphabetic passwords). Many people will use a combination of *words* for their password.

Let's call weak_S3 the set of passwords that are made up entirely of known words - one word, or more than one.

4. The table below shows how many common words there are of each length. Calculate the number of possible passwords in the set weak_S3.

word length	# in dictionary
8	80148
7	78035
6	87151
5	158390
4	149165
3	15939
2	675

Note the difference in size between weak_S3 and S3.

This table seems a bit off, so for the purpose of Questions 5, 6 , and 7 just assume that |weak_S3| is 50,000.

Some large fraction of people use weak passwords. Depending on what you ask, between 30% and 80% of people have bad password habits. Let's assume for our purposes that bad habits mean picking an easy to remember password: an existing word or combination of words.

5. Assume that 60% of users would choose a weak password (a password from weak_S3), and that an attacker can try 10^6 passwords per second. How likely is it that a given password would be broken within 1 minute? 1 hour? 1 day?

6. weak_S3 is S3, but made only from common words. How you would you calculate the set weak_S4, which allows for capitalized characters?

As an optional bonus task: write a python function that takes one member of weak_S3 as an argument, and returns a list of related passwords in weak_S4.

7. Consider the set of passwords S5 (8 character alphanumeric passwords). What would be - in your opinion - a weak password in this scheme? What if we *forced* the password to have 2 numeric digits?

8. Given what you now know about dictionary attacks, what do you think about the distinction between sets S6 and S7?

As an optional bonus task: write a python function to return a set of passwords in weak_S6 from a member of weak_S7. (Assume that those sets are similar to the other weak_* sets we've defined above)

Part 3: Storing passwords

Even more effective in practice than brute force attacks and dictionary attacks are *credential stuffing attacks*.

Here's how it works from the attacker's perspective. Say some company has a data breach, which comprimises the passwords and emails of thousands of users. That company might fix their problem - get everyone to change their password, or something. But... people reuse passwords across sites. An attacker could buy that list of usernames and passwords, and try using them on other platforms.

The site HavelBeenPwned maintains a database of leaks, so that you can find out if a given username / password pair has been leaked. There are a *ton* of sites that have leaked passwords.

Credential stuffing attacks aren't really that interesting as users or as attackers. For users, there's no great answer. You should use different passwords on different sites, and you should try not to use sites that don't have strong security... but it's hard to tell which those are. As an attacker, credential stuffing is *effective*, but there's no cleverness to it - you just try the usernames and passwords in the list.

Preventing credential stuffing and mitigating data breaches is *really* interesting from the perspective of application developers. If we're trying to build secure systems, what are the lessons to learn? How can we avoid being added to the list of sites with data breaches?

Hashing passwords

One key problem in many of these breaches was that passwords were stored in plain text. In the database for these applications, there was a username column and password column. When a user logged in, the app would check that the password matched the username.

Attackers were able, through various means, to get access to that database. When they did, all the passwords were there to be stolen, just like that. Those passwords could be reused in credential stuffing attacks, as described above.

What if the application never stored the passwords in plain text? If there was some way to avoid keeping a *password* column in the database, then even if the attacker got access, they wouldn't be able to get the passwords.

The app still needs to be able to *verify* the password when someone logs in. We need a way to verify passwords, without storing them. It turns out, there is a way to do this, based on a clever mathematical trick called a *hash function*.

(We can also get some protection using *encryption functions*, which you will explore more in the next project)

A hash function is useful because it's *very hard to undo*. There's no easy-to-compute inverse function. Instead of storing the password in the database, we can store the hash of the password (the output of the hash function). Because the hash function is non-invertible, even if an attacker got ahold of the database, they wouldn't be able to see the passwords - just some random-looking noise.

A hash function

Run the code in hashing_demo, and provide it with a test password. What do you notice about the size of the hashed value? Try it again, several times, with the same password. What do you notice about the output? Finally, make a small change to your password - swap a letter, or change its casing, or add just one number. What do you notice about the output?

The algorithm in `hashing_demo` is called *md5* and it used to be a very popular hash function.

Here's the formal definition of a hash function:

> A hash function takes input of an arbitrary size, and generates output of a fixed size.

That... doesn't say much about non-invertibility, or other things that make a function a *good* hash function.

So, what else makes a function *good* for hashing passwords?

What makes a good hash function?

Domain needs to be the set of possible passwords, range should be a fixed-length output.

- **Deterministic**: output should be the same for the same input
- **Uniform**: should generate each output with the same probability
- **Avalanche Effect**: Small changes to the password should yield drastically different outputs

Task: Assessing hash functions

Let's try to assess some hashing functions now.

Consider the following hashing algorithm H1 for numeric pincodes in S2:

```
- Sum the last 4 digits
- Add the result to the number represented by the first 4 digits
- Output the result
```

1. Prove that for any number *N* in S2, applying the instructions above can not yield a result that is 6 digit large.

2. Consider the relation H1, with domain S2, and range N(5) (five digit numbers)

- Is H1 a function? Why or why not?

- Come up with two numbers, x and y, such that H1(x) = H1(y)

- Does H1 fit the formal definition of a hash function?

- Is H1 a *good* hash function? Why or why not?

3. Consider a hashing function H2. It's domain is S7, and it's range is N(6) (six digit numbers). What is the cardinality of the range of H2?

4. Consider the following statement

> *"If a function's range is smaller than its domain, then that function cannot be injective"*

- Prove or refute this statement.

- Based on your proof, what is the implication for H2?

- Could there be a function D1 that is the inverse of H2?

- What is the implication for using H2 to hide passwords? What might happen if hashed passwords leaked?

Conclusion

Hash functions are prone to *collisions*. Collisions happen when two different inputs lead to the same hash output. This means that while we are safer by storing **a hash of our passwords** instead of our plain passwords, we now run a new risk: there could be some other password with the same hash value as our password! A malicious user could gain access to an account using a that different password.

Let's add another property to the list for what makes a good cryptographic hash function:

- **Collision resistant**: difficult to find two different messages that have the same hash

Security is an arms race. When researchers invent new techniques to keep things secure, attackers discover new flaws, which in turn motivate new techniques. All of the techniques, for both attackers and defenders, are heavily reliant on mathematics and design: probability, sets, functions, and, as you'll see when you learn more about encryption, number systems and number theory.

On a lighter note, take a look at this comic.

5. (Optional) What do you think of the statement the comic makes? Is the password scheme from the comic (four 3-to-8 letter words) safe? What would you recommend to your friends and family, in terms of password management? -->