Tutorial 2: Psychology and Usability

This tutorial covers the **fundamentals of passwords** and serves as a **warm-up exercise** to strengthen your *Python programming skills*.

🔐 Building a Password Cracker

> **Passwords** are intuitively understood by humans as a way to protect access to a resource.
> Just think of the fairy tale of **Ali Baba**!
>
> However, *memorising them while maintaining unpredictability* is the hard part — which is why **'Open Sesame'** is an exceptionally bad password.

🎯 What Are We Doing?

In this task, we will **create a password cracker** to get a feel for how passwords are constructed — and hacked.
This will also serve as preparation for the next lecture.

We'll work from **very naive approaches** ➡️ to **more sophisticated ones**.

🧩 How Passwords Are Stored

- 🔑 Passwords are usually stored as **hashes (with salt)**, not as plain text.
- ✅ Password matching happens by **comparing hashes**, rather than plain-text passwords.

📚 Your Tutor Will Explain:

- 🌀 *What is hashing?*
- 🔒 *Why are passwords stored as hashes?*
- 🧂 *Why is salt important?*
- 💡 *What is MD5 hashing?* (hashing will be covered in detail later)

📝 Your Task

You are given a set of **MD5 hashes** in `hashes.txt` for you to break.
These are encoded versions of correct passwords.

We provide **skeleton code** that allows you to compare a password against its corresponding MD5 hash.

a) Brute-force

The naive approach to cracking a password is to use **brute force**, trying all possible combinations of letters, digits, and special characters. For this task, we will limit ourselves to:

- All letters in the English alphabet (both lowercase and uppercase)
- Digits `0–9`
- Special characters `#`, `!`, and `$`

Complete the provided code template to brute force passwords up to a length of **4**. Also, count the number of guesses required.

Since the enumeration can take some time, we have included the MD5 hash of the password `"aA0#"` in the **hashes_test.txt** file. If your iteration loops are implemented correctly, you should find a match quite quickly. Once your code is working with this file, proceed to the **hashes.txt** file.

Expected outcome:
You should successfully crack a password of length 4. The number of guesses may vary depending on your implementation. If you follow the given code template, the brute-force process should take approximately **140–180 seconds** on the provided Azure VM.

In [1]:
```
from Crypto.Hash import MD5
import string
import sys
import time

### Start timing – Do Not Change
start_time = time.time()

### TODO: Read the hashes from the file into a list
hash_file = open("hashes.txt", "r")

### TODO: Read all lines from the file
hashes = hash_file.readlines()

### TODO: Remove the newline characters
hashes = [hashe.rstrip() for hashe in hashes]

### Function to obtain the MD5 hash of a given string – DO NOT Change
def break_password(password_attempt, md5_hash):
    return md5_hash == MD5.new(password_attempt.encode()).hexdigest()

### Building the alphabet
alphabet_list_1 = list(string.ascii_lowercase)  # TODO: Get the ASCII low
alphabet_list_2 = list(string.ascii_uppercase)  # TODO: Get the ASCII upp
alphabet_list_3 = list(string.digits)            # TODO: Get the digits a
symbols = ['#', '!', '$']                        # TODO: Get the symbols

### TODO: Build the final alphabet of all allowed characters
alphabet = alphabet_list_1 + alphabet_list_2 + alphabet_list_3 + symbols
```

```
counter = 0

### Nested for loops for enumeration
for a in alphabet:
    for b in alphabet:
        for c in alphabet:
            for d in alphabet:
                counter = counter + 1
                password = str(a + b + c + d)
                target_hash = MD5.new(password.encode()).hexdigest()
                if target_hash in hashes:
                    print(
                          "Match:", password, MD5.new(password.encode()).
                          "Attempt:", str(counter),
                          "Time (Seconds):", (time.time() - start_time))
```

```
Match: Uq4C 6295d31334a99b0cc63c67e6bf15cedb Attempt: 12704019 Time (Secon
ds): 44.182260274887085
```

Now try passwords of length 5.

- Do you still get a hit?
- How many guesses?
- How much time did this take?

Repeat the experiment for passwords of length 8.

Don't let the program run for more than a few minutes or so (stop the code using the STOP button above).

b) Using a Dictionary - 1

The above exercise should make you realise that brute force becomes very impractical very quickly. This is why almost all password cracking attempts try to leverage the weaknesses of the human mind: difficulties memorising random strings. Most users therefore tend to rely on words or common patterns they can remember.

- Write code to load the dictionary file (located in the Tutorial 2 folder) into a suitable Python data structure.
- What data structure should you use? Discuss this with your tutor if you are unsure!
- What are the trade-offs?
- Add code to try words from the dictionary as passwords. Do you get a hit?

> **Solution:**
>
> Lists or arrays are fine here, as in the context of this question we have to loop over all entries anyway.
> An optimisation would be to use a dictionary, with the keys being the first letters of each word.
> For example, all words starting with `a` could be stored as a list/array under the key `a` in the dictionary.

> There should be one more hit:
> **Match:** `tutorial` `0575c8d592fb7b088226750aceec2b4e`

In [2]:
```
from Crypto.Hash import MD5
import string
import sys
import time

### Start timing - Do Not Change
start_time = time.time()

#### TODO Open the dictionary file
password_file = open("dictionary.txt", "r")

#### TODO Read the password file into a list
passwords = password_file.readlines()

#### TODO Remove the newline characters
passwords = [password.rstrip() for password in passwords]

#### TODO Enumerate through each password, generate the hash, and check w
for password in passwords:
    target_hash = MD5.new(password.encode()).hexdigest()
    if target_hash in hashes:
        print("Match:", password, MD5.new(password.encode()).hexdigest(),
```

```
Match: tutorial 0575c8d592fb7b088226750aceec2b4e Time (Seconds): 0.0527617
9313659668
```

Now modify your code to be case-insensitive when checking passwords from the dictionary. Do you get more hits?

> **Solution:**
> One more hit:
> **Match:** `Tutorial` `368fe771261fcb18f7988833c9294a20`

In [3]:
```
### Start timing - Do Not Change
start_time = time.time()

### TODO Get all uppercase versions of the above passwords
pl_list = [pl.upper() for pl in passwords]

### TODO Get title case versions of the above passwords
pt_list = [pt.title() for pt in passwords]

### TODO Combine the three lists of passwords
passwords_case = passwords + pl_list + pt_list

#### Ignore case
for password in passwords_case:
    target_hash = MD5.new(password.encode()).hexdigest()
    if target_hash in hashes:
        print(
            "Match:", password,
```

```
            MD5.new(password.encode()).hexdigest(),
            "Time (Seconds):", (time.time() - start_time)
        )
```

```
Match: tutorial 0575c8d592fb7b088226750aceec2b4e Time (Seconds): 0.0507452
48794555664
Match: Tutorial 368fe771261fcb18f7988833c9294a20 Time (Seconds): 0.1076066
4939880371
```

b) Using a Dictionary - 2

You hopefully had a hit above, but you still haven't found all the passwords yet. Try a further approach!

- Add code to try combinations of words from the dictionary.
- You may assume there are no spaces between them.
- Do you get more hits? How much longer does it take?

> **Solution:**
> There are hits. Students can either generate the combinations on the fly or write them out to a new dictionary.
> Do not exceed available RAM. If you do, write the results out to disk.
> Also, write the code so that it exits after the first hit.
>
> **Match:** `actress granny 173e06cdeafc20ff5704f595dd7650d8`

In [4]:
```
### Start timing - Do Not Change
start_time = time.time()

### We need to use a loop break because the passwords list is really larg
loop_break = False

### TODO Write two nested loops to go through the passwords twice
for password1 in passwords:
    for password2 in passwords:
        password = password1 + password2
        target_hash = MD5.new(password.encode()).hexdigest()
        if target_hash in hashes:
            print(
                "Match:", password1, password2,
                MD5.new(password.encode()).hexdigest(),
                "Time (Seconds):", (time.time() - start_time)
            )
            loop_break = True
            break
    if loop_break:
        break
```

```
Match: actress granny 173e06cdeafc20ff5704f595dd7650d8 Time (Seconds): 1.4
428608417510986
```

c) Using a Dictionary - 3

Some people replace letters with digits or special characters.

- Add code to try words from the dictionary, but this time allow for letter substitutions (e.g., 4 for A or a).
- Do you get more hits? How much more time does it take?
- Now add code to also consider combinations of words and substitutions.
- Do you get more hits?
- How much time does it take?

> **Solution:**
> A good solution here is to perform the substitutions on the fly, i.e., have an extra method that is called to make substitutions.
>
> **Match:** `fin4lly` `68709d844aed1b2936a9fae66bc8b697`

In [5]:
```
### Start timing - Do Not Change
start_time = time.time()

for password in passwords:
    password = password.replace('a', '4')
    password = password.replace('A', '4')
    target_hash = MD5.new(password.encode()).hexdigest()
    if target_hash in hashes:
        print(
            "Match:", password,
            MD5.new(password.encode()).hexdigest(),
            "Time (Seconds):", (time.time() - start_time)
        )
```

```
Match: fin4lly 68709d844aed1b2936a9fae66bc8b697 Time (Seconds): 0.02870035
171508789
```

Some users append symbols such as "#" and "!" to common words.

- Extend your code to check for added special characters, but limit yourself to two (`'#'` and `'!'`) and the special characters from the brute-forcing section above.
- Do you get more hits? How much time does it take?

> **Solution:**
> There should be a hit.
> **Match:** `hacking!#` `e9e8144285eca2c050ab0643409e79c2`

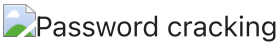
In [6]:
```
for password in passwords:
    password = password+"!#"
    target_hash = MD5.new(password.encode()).hexdigest()
    if target_hash in hashes:
        print("Match: ", password, MD5.new(password.encode()).hexdigest()
```

```
Match:  hacking!# e9e8144285eca2c050ab0643409e79c2
```

e) Know the password strengths

By now, you should have some idea of various password cracking methods and how long they take. The following table shows the times taken to crack different types of passwords. Note the following:

- These calculations are based on a specific hardware setup. Advances in hardware and CPUs will speed things up.
- These times are not guaranteed. Usually, there are defenses such as limits on the number of incorrect password attempts, restrictions on the number of requests from a single IP address, etc.
- Also, the attacker's options depend on the scenario. An attacker will have more time if they are trying to break into a hard disk to which they have physical access. Much longer and more complex passwords are required in such cases.


Password cracking

Source https://www.hivesystems.com/blog/are-your-passwords-in-the-green

f) Homework - More on storing passwords

Read more about salting, peppering, rainbow tables, and selecting password hashes by following this link.

> **Solution:**
>
> **Salting:**
> Salting is the process of adding a random string (called a **salt**) to a password before hashing it.
>
> - Ensures that even if two users have the same password, their hashes will be different.
> - Makes pre-computed attacks (like rainbow tables) ineffective.
> - **Storage:** Salts are typically stored in plaintext alongside the password hash because they are not meant to be secret. Their job is to make each hash unique.
>
> **Peppering:**
> Peppering is similar to salting, but the **pepper** is kept secret (at the application or hardware level) instead of being stored with the password.
>
> - Adds an extra layer of security if the database is compromised.
> - Unlike salts, the pepper is common across all passwords and not stored in the database.

> **Rainbow Tables:**
> Rainbow tables are large pre-computed lists that map possible passwords to their hash values.
>
> - Attackers use them to reverse hashes quickly.
> - Salting defeats rainbow tables because each unique salt requires a new table.
>
> **Selecting Password Hashes:**
> Use strong, slow password hashing functions such as:
>
> - **bcrypt**
> - **scrypt**
> - **Argon2**
>
> These are designed to be computationally expensive and resistant to brute-force attacks.
> Avoid using fast hash functions like MD5 or SHA-1 for password storage.

g) Homework - Calculation of password cracking times

A company uses 8-character passwords composed only of lowercase letters (`a–z`). This means there are **26 possible characters** per position and **8 positions**.

1. **Total password combinations:**
   Calculate the total number of possible passwords.

2. **Cracking rates:**
   Assume the following cracking speeds:

   - A CPU can attempt **100 million (1×10^8) hashes per second**
   - A single GPU can attempt **5 billion (5×10^9) hashes per second**
   - A multi-GPU rig with 8 GPUs can attempt **40 billion (4×10^{10}) hashes per second**

   (a) Calculate how long it would take (in seconds, minutes, hours, and days) to try every possible password in the worst-case scenario for each of the three setups.

3. **Impact of salting:**

   - Suppose the company adds a **unique 32-bit salt (4 bytes)** to each user's password.
   - There are **1,000 users**.

   (a) Explain how salting affects the attacker's work if they want to crack all user passwords.
   (b) Recalculate the total number of combinations the attacker must try if they have to crack all users' hashes individually.
   (c) How does this change the cracking time for each setup (CPU, GPU, multi-GPU)?

Hint:

- Total combinations for unsalted passwords = `26^8`
- When salting, the attacker must attack each user's hash individually because the salt is unique to each user.
- Convert large numbers of seconds into days to interpret your answers.

Solution:

1. Total Password Combinations (Unsalted)

Passwords use 8 lowercase letters (`a–z`), so:

`26^8 = 208,827,064,576`
or approximately **2.09×10^{11} possible passwords**.

---

2. Cracking Times Without Salting

CPU (100 million = 1×10^8 hashes/sec)

Time = $2.09 \times 10^{11} \div 1 \times 10^8 = 2{,}088$ seconds

- **Seconds:** 2,088 s
- **Minutes:** 2,088 ÷ 60 = 34.8 min
- **Hours:** 34.8 ÷ 60 ≈ **0.58 hours**

Single GPU (5×10^9 hashes/sec)

Time = $2.09 \times 10^{11} \div 5 \times 10^9 = 41.8$ seconds

- **Seconds:** 41.8 s
- **Minutes:** 41.8 ÷ 60 ≈ **0.70 min**

Multi-GPU Rig (8 GPUs = 4×10^{10} hashes/sec)

Time = $2.09 \times 10^{11} \div 4 \times 10^{10} = 5.22$ seconds

- **Seconds:** 5.2 s

Observation:
A multi-GPU rig is **~400× faster** than a CPU in this example.

---

3. Impact of Salting (Unique 32-bit salt per user)

With unique salts:

- Each user's hash must be cracked **individually**.
- For 1,000 users, total combinations =

$1000 \times 26^8 = 1000 \times 2.09 \times 10^{11} = 2.09 \times 10^{14}$

Now recalculate times:

CPU

Time = $2.09 \times 10^{14} \div 1 \times 10^{8} = 2.09 \times 10^{6}$ seconds

- **Seconds:** 2.09 × 10^6 s
- **Days:** 2.09 × 10^6 ÷ 86,400 ≈ **24.2 days**

Single GPU

Time = $2.09 \times 10^{14} \div 5 \times 10^{9} = 41{,}765$ seconds

- **Seconds:** 41,765 s
- **Hours:** 41,765 ÷ 3,600 ≈ **11.6 hours**

Multi-GPU Rig

Time = $2.09 \times 10^{14} \div 4 \times 10^{10} = 5{,}221$ seconds

- **Seconds:** 5,221 s
- **Hours:** 5,221 ÷ 3,600 ≈ **1.45 hours**

Key Takeaways

1. **Without salting:**
   - Multi-GPU: 5.2 seconds
   - Single GPU: 42 seconds
   - CPU: 35 minutes
2. **With unique salts:**
   - Multi-GPU: **1.45 hours**
   - Single GPU: **11.6 hours**
   - CPU: **24 days**
3. Salting **dramatically increases the attacker's workload**, because each user's hash must be attacked separately and precomputed tables (like rainbow tables) become useless.