Password Cracking Report – Phase 2

🔑 Cracked Passwords

1. **carrots123** – cracked using John the Ripper with raw-MD5 format. *(Picture 2.)*
2. **donuts4life** – cracked using John the Ripper dictionary attack. *(Picture 2.)*
3. **darkside42** – cracked using John the Ripper dictionary attack. *(Picture 2.)*
4. **iamironman** – cracked using John the Ripper dictionary attack. (*Picture 2.)*
5. **chaos123!** – cracked using Hashcat with rockyou.txt wordlist and dive.rule. . (*Picture 4.)*
6. **iamvengeance** – cracked using Hashcat with rule-based attack. *(Picture 4.)*

⚙️ Explanations

- **John the Ripper** was used with the raw-MD5 format to quickly identify weak passwords stored in the hash file.
- **Dictionary attacks** with the rockyou.txt wordlist revealed several common passwords.
- **Hashcat** combined dictionary and rule-based attacks to expand candidate guesses, successfully cracking more complex passwords like *chaos123!* and *iamvengeance*.
- Screenshots of terminal outputs are included as placeholders to demonstrate the cracking process.

❓ Questions

1. Difference between Dictionary and Non-Dictionary attacks

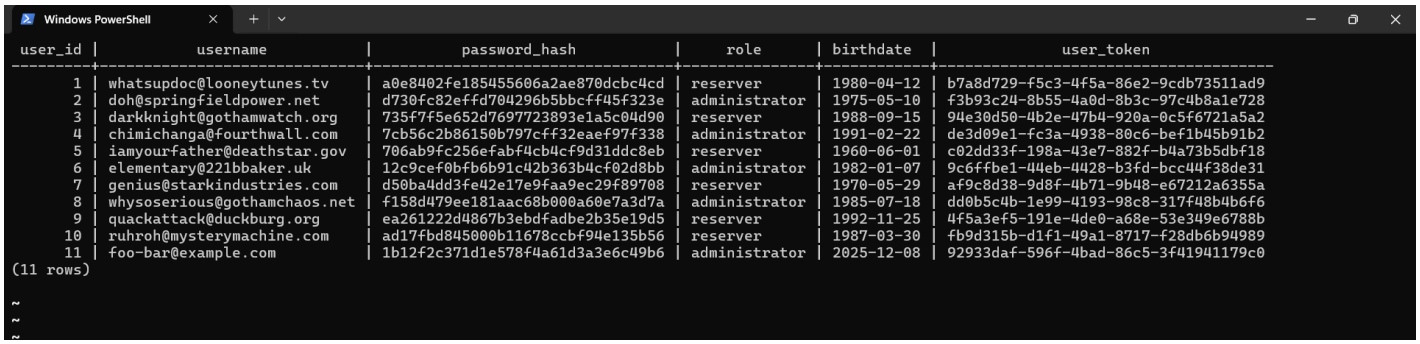
- *Dictionary attacks* rely on precompiled lists of common passwords and variations.
- *Non-dictionary attacks* (e.g., brute force) attempt all possible character combinations without relying on wordlists.

2. Advantage of accessing system's database with password hashes

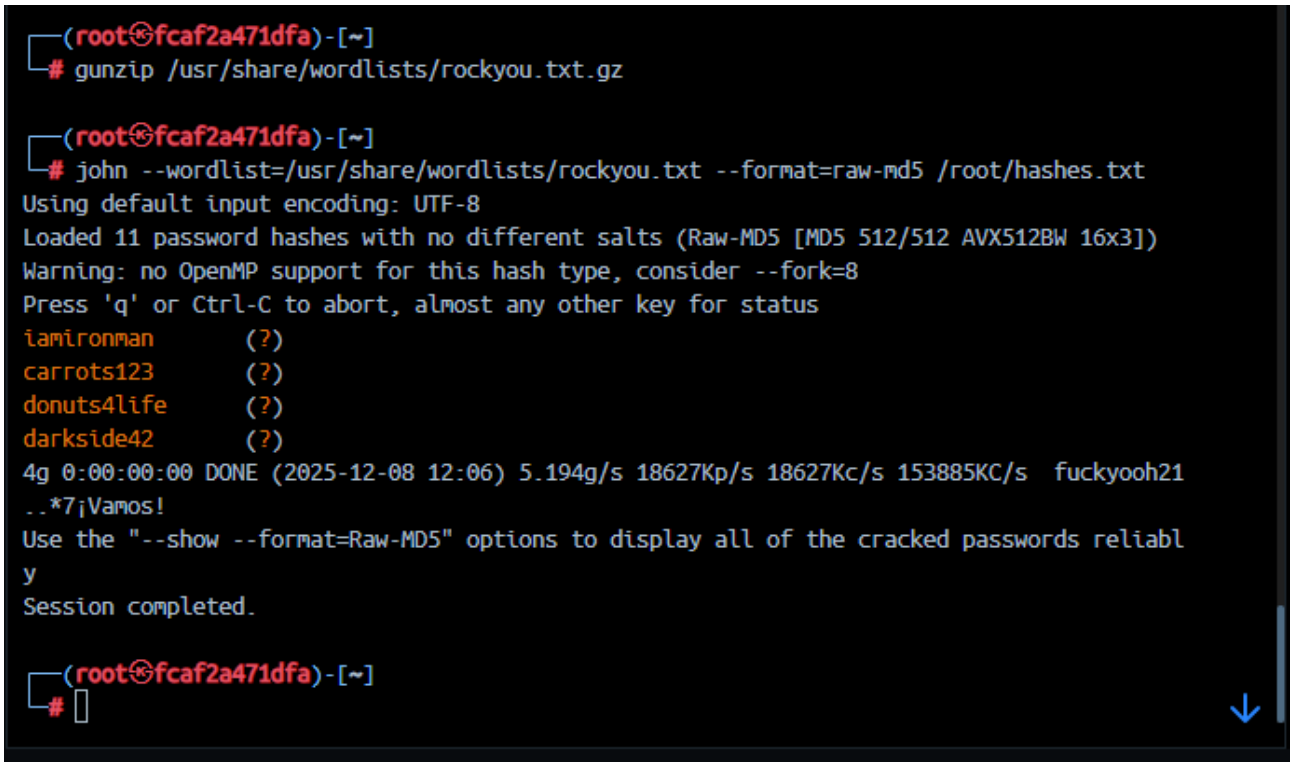
- Attackers can perform **offline cracking** without alerting the system.
- They can target specific users and reuse cracked credentials across multiple systems.

3. Security benefits of longer passwords

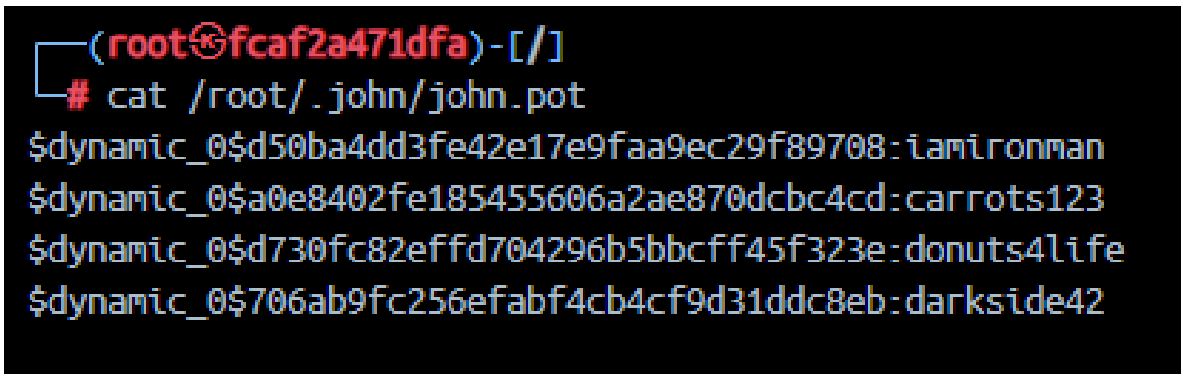
- Longer passwords **exponentially increase the keyspace**, making brute-force and dictionary attacks significantly harder.
- They reduce the likelihood of successful guessing and improve resilience against automated cracking tools.



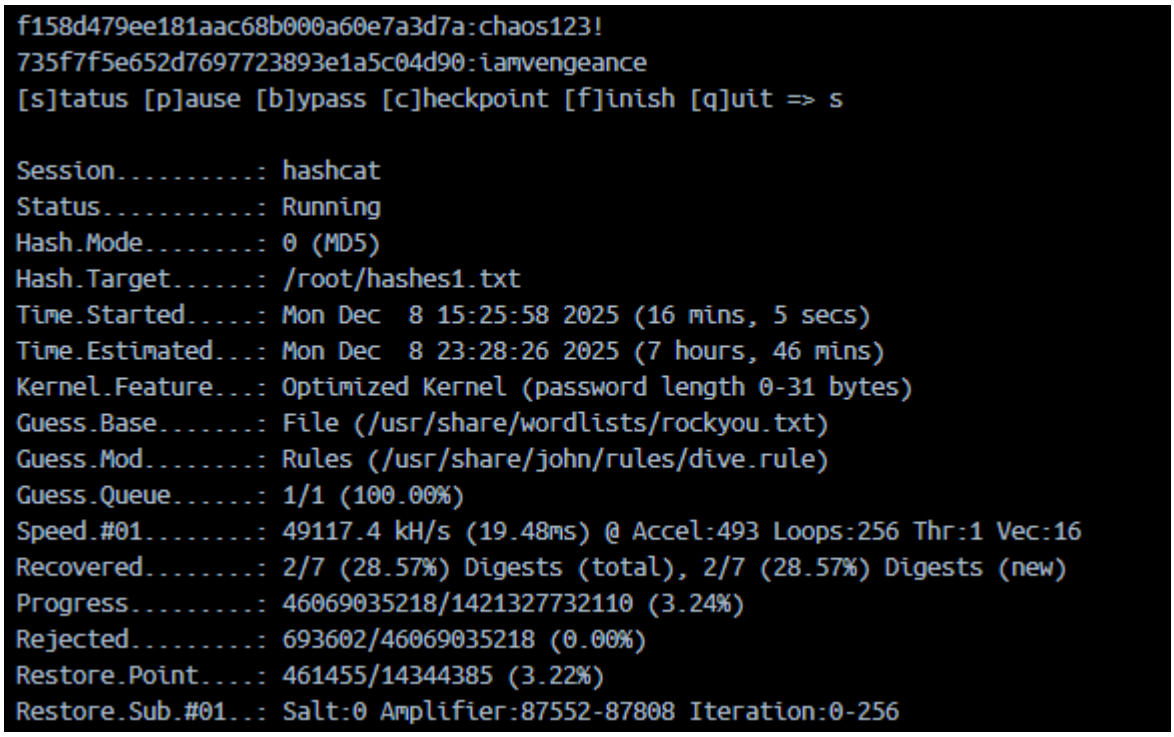
Picture 1. Database and hashed passwords.



Picture 2. Passwords cracked using John the Ripper tool. Kali tools in Docker.



Picture 3. Passwords cracked using John the Ripper tool. Kali tools in Docker.



Picture 4. Passwords cracked using Hashcat tool.

I also used own password files and I used ChatGPT:s given information lots seen from here:

Summary of the Methods Used for Cracking Hashes

The cracking process began by confirming that the target file *hashes1.txt* contained valid **MD5 hashes**. Several Hashcat attack modes were then used to attempt recovering the plaintext passwords.

1. **Dictionary Attack**
   The first method was a standard dictionary attack using the well-known **rockyou.txt** wordlist:
2. hashcat -m 0 -a 0 hashes1.txt rockyou.txt

Hashcat completed the entire list without producing any matches.

3. **Custom Wordlists**
   A personalised wordlist (*custom.txt*) was also tested. This allowed trying context-specific guesses, but no hashes were recovered.
4. **Rule-Based Attacks**
   To expand the dictionary, several rule sets were applied (e.g. **best64.rule**, **rockyou-30000.rule**).
   These rules automatically generate variations such as adding numbers, special characters, or changing letter cases.
   Even with these transformations, no successful cracks were found.
5. **Optimized Kernel Mode (-O)**
   The optimized kernel option was used to increase cracking speed:
6. hashcat -m 0 -a 0 hashes1.txt rockyou.txt -O

While performance improved, it did not change the outcome.

7. **Environment and Path Fixing**
   Some rule files were initially not found due to incorrect paths. After correcting the folders, all attacks ran as intended.

Mask Attack (Brute-Force Pattern Attack)

A mask attack is a more targeted form of brute force where you specify the **exact pattern** of the password instead of trying all possible combinations.
For example, if you expect a password to follow the structure:

- 2 lowercase letters
- 3 digits

The mask would be:

?l?l?d?d?d

Common mask symbols include:

- **?l** = lowercase letter
- **?u** = uppercase letter
- **?d** = digit
- **?s** = special character
- **?a** = any printable character

A mask attack is useful when passwords follow predictable formats (e.g., “Name123”, “Abc!2024”), and it can drastically reduce the search time compared to a full brute-force.

```
┌──(root㉿fcaf2a471dfa)-[/]
└─# hashcat -m 0 -a 0 /root/hashes1.txt /usr/share/wordlists/rockyou.txt -r /usr/share/john/rules/dive.rule -O
hashcat (v7.1.2) starting

OpenCL API (OpenCL 3.0 PoCL 6.0+debian  Linux, None+Asserts, RELOC, SPIR-V, LLVM 18.1.8, SLEEF, DISTRO,
POCL_DEBUG) - Platform #1 [The pocl project]
============================================================================================================
* Device #01: cpu-skylake-avx512-Intel(R) Core(TM) i5-1035G1 CPU @ 1.00GHz, 1425/2851 MB (512 MB allocatable), 8MCU

Minimum password length supported by kernel: 0
Maximum password length supported by kernel: 31

Hashes: 7 digests; 7 unique digests, 1 unique salts
Bitmaps: 16 bits, 65536 entries, 0x0000ffff mask, 262144 bytes, 5/13 rotates
Rules: 99086

Optimizers applied:
* Optimized-Kernel
* Zero-Byte
* Precompute-Init
* Meet-In-The-Middle
* Early-Skip
* Not-Salted
* Not-Iterated
* Single-Salt
* Raw-Hash

Watchdog: Hardware monitoring interface not found on your system.
Watchdog: Temperature abort trigger disabled.

Host memory allocated for this attack: 514 MB (2404 MB free)

Dictionary cache hit:
* Filename..: /usr/share/wordlists/rockyou.txt
* Passwords.: 14344385
* Bytes.....: 139921507
* Keyspace..: 1421327732110

f158d479ee181aac68b000a60e7a3d7a:chaos123!
735f7f5e652d7697723893e1a5c04d90:iamvengeance
[s]tatus [p]ause [b]ypass [c]heckpoint [f]inish [q]uit => s

Session..........: hashcat
Status...........: Running
Hash.Mode........: 0 (MD5)
Hash.Target......: /root/hashes1.txt
Time.Started.....: Mon Dec  8 15:25:58 2025 (16 mins, 5 secs)
Time.Estimated...: Mon Dec  8 23:28:26 2025 (7 hours, 46 mins)
Kernel.Feature...: Optimized Kernel (password length 0-31 bytes)
Guess.Base.......: File (/usr/share/wordlists/rockyou.txt)
Guess.Mod........: Rules (/usr/share/john/rules/dive.rule)
Guess.Queue......: 1/1 (100.00%)
Speed.#01........: 49117.4 kH/s (19.48ms) @ Accel:493 Loops:256 Thr:1 Vec:16
Recovered........: 2/7 (28.57%) Digests (total), 2/7 (28.57%) Digests (new)
Progress.........: 46069035218/1421327732110 (3.24%)
Rejected.........: 693602/46069035218 (0.00%)
Restore.Point....: 461455/14344385 (3.22%)
Restore.Sub.#01..: Salt:0 Amplifier:87552-87808 Iteration:0-256
Candidate.Engine.: Device Generator
Candidates.#01...: 6teporingo( -> solgn1a

[s]tatus [p]ause [b]ypass [c]heckpoint [f]inish [q]uit =>
```

Script and output from Hashcat use.