

For this project, I shot myself in the foot. I initially tried using Hashcat and I couldn't get it to work. Then life happened... I only just started on this project on December 1st. Honestly, I wish I had done this project earlier because I really enjoyed doing this project, and I wonder how much more I could have cracked if I had more time.

I chose Hashcat mainly because the Windows UI and status output made it easier for me to see what was happening in real time and understand how different attacks were behaving. I initially made things harder on myself by downloading the source instead of the precompiled binary, I think that is why it didn't work when I originally try to start on this project. I also ran into trouble familiarizing myself with how the commands were structured in Hashcat, and I had to clean up the provided shadow file (removing the trailing ":::::" fields) before Hashcat would accept it. I ran attacks concurrently on both my laptop with RTX 2000 Ada 8GB and my desktop with a RTX 3070 8GB.

Before attacking in earnest, I did some setup work. I found a kaggle database that had the top one million commonly used passwords, so I downloaded that and used it as my base wordlist. I also started paying attention to Hashcat's keyspace estimates and time-to-complete numbers and noticed quickly that brute force gets exponentially slower as the maximum password length grows. That observation ended up driving most of my strategy changes over the course of the assignment.

My first real experiment was a pure brute-force mask on my laptop: eight lowercase characters against MD5-crypt hashes with Hashcat mode 500. After three hours, Hashcat had only covered about 2.3% of the keyspace and estimated more than five days total to finish, with no passwords recovered. The combined speed across my devices in this and later runs ranged from roughly 9.0M to 17.4M candidate passwords per second, depending on the attack and GPU. Even at those rates, it was obvious that naïve brute force over a large keyspace is not practical for long passwords. This first attack was more of a proof of concept that I understood the program.

The first time I actually cracked anything was when I switched to a straight wordlist attack on my desktop, using a 10-million-entry password list. That attack finished in about two seconds and immediately recovered two passwords: "cincinnati" and "longitude". This matched what I expected from human behavior: these are real words that appear in the kaggle password datasets, so they fell almost instantly once I tried a realistic wordlist instead of exploring a huge brute-force space. Although, I was expecting a more commonly used password such as "password" or "qwerty." That run was also extremely fast (around 12.9M checks per second), which made the advantage of dictionary attacks very obvious.

Next, I stayed on the desktop and layered in Hashcat's "best66.rule" rule set on top of the same wordlist. This "wordlist + rules" attack applies common mangling operations such as truncation, capitalization changes, and simple substitutions. Over about a minute, that attack cracked four additional passwords: "lsv", "ta", "lr4", and "SJ". These all look like short or modified forms of more meaningful words, which is exactly what rule-based mangling is designed to target. I also

tried a hybrid attack that combined the wordlist with a three-digit numeric suffix mask (`?d?d?d`) to model patterns like `password123`, but in this dataset it did not find any new passwords beyond the ones I had already cracked.

I then experimented with more structured brute force. On my laptop I tested a “PIN style” attack over 2–8 digits, which exhausted the entire numeric keyspace in a few minutes without finding anything. That run mostly shows how weak purely numeric passwords are. Back on the desktop, I ran an incremental lowercase brute-force attack from 2 to 8 characters, which ran for a bit over two and a half hours before I stopped it. That attack recovered three more passwords: “uddg”, “lkyru”, and “ugrknq”. These look more random than previously found passwords, which explains why they only appeared when I brute-forced the smaller lowercase keyspace.

Finally, I tried a mixed-case brute-force attack on my laptop using a custom charset of upper and lowercase letters. That run lasted about four hours and, on the laptop, “newly” cracked five passwords, but all of them were ones I had already found earlier on the desktop “ta”, “SJ”, “lsv”, “uddg”, and “lkyru”. In other words, it confirmed that those shorter passwords are vulnerable under many different guessing strategies, but it didn’t add anything new to the overall list.

Across all of these attacks, Hashcat typically reported speeds between about 9M and 17M candidate passwords per second, averaging roughly 1.2×10^7 checks per second across my main runs. Using that speed, the effective strategies were the ones that cut down the search space: large wordlists, rule-based mangling, and small-keyspace brute force over shorter lowercase strings. In chronological order of first discovery, the passwords I cracked and how I got them were: “cincinnati” (wordlist attack), “longitude” (wordlist attack), “lsv” (wordlist + rules), “ta” (wordlist + rules), “lr4” (wordlist + rules), “SJ” (wordlist + rules), and then “uddg”, “lkyru”, and “ugrknq” (incremental lowercase brute-force). Seeing how quickly simple, common, and short passwords fell, compared to how long the full brute-force masks would have taken, gave me a much more concrete feel for why long, unique passwords are so important in real systems.

Kaggle dataset:

<https://www.kaggle.com/datasets/joebeachcapital/top-10-million-passwords?resource=download>

1. From my Hashcat runs, my GPU speeds for MD5-crypt were in the ballpark of: ~9.0M c/s for brute-force masks, ~12.9M–14.5M c/s for wordlist and wordlist+rules, and ~17.4M c/s for an optimized lowercase brute-force. It averages out to about 12.6 million guesses a second. There are 62 characters in an alphanumeric password if you are using both upper and lower case. (I have rounded results)
 - a. Therefore, for a 6 character long password, there are a total of 62^6 possible combinations. Dividing that by 12.6 million guesses a second is 4507 seconds or 75 minutes.
 - b. For a 8 character long password, it will take 17328579 seconds, 288809 minutes, 4813 hours, or 200 days.
 - c. For a 10 character long password, it will take 666110607832 seconds, 11101843464 minutes, 185030724 hours, 7709614 days, or 21122 years.
 - d. For a 12 character long password, it will take roughly 8.1 million years.
2. Having a guess rate of 2.5 Billion per second will result in:
 - a. 6 Character: 22.7 seconds
 - b. 8 Character: 24 hours
 - c. 10 Character: 10.6 years
 - d. 12 Character: 40900 years
3. No, modern password meters typically require a minimum of 8 characters. A modern GPU can crack those passwords fairly easily. I would recommend for the time being, at least a 10 character minimum due to the greatly increased timeframe to crack a password. As technology gets better, we will have to move the goal post further and further however.
4. It does increase security a bit, but in the context of brute force attacks like we have been talking about, it does nothing.
5. Yes, a salt prevents users with the same password from having the same hash.
6. No, passwords get leaked all the time, even if they are hashed passwords, an attacker can download those leaks and perform an offline attack.