CMSC 23200 HW 6

Ruolin "Lynn" Zheng

February 20 2020

Submission Files

1. shadow.txt - the users assigned to ruolinzheng in /etc/shadow, included in case TA's would like to run my .py files
2. ruolinzheng-problem1.potfile - generated by Hashcat
3. ruolinzheng-problem1.txt - parsed username-password pairs to be used in Problem 2 and 3
4. ruolinzheng-problem2.txt
5. ruolinzheng-problem3.txt
6. ruolinzheng-problem2.py - this needs to read `ruolinzheng-problem1.potfile` and `shadow.txt` and will write to `ruolinzheng-problem1.txt` and `ruolinzheng-problem2.txt`
7. ruolinzheng-problem3.py - this needs to read `ruolinzheng-problem1.txt` and `ruolinzheng-problem2.txt` and will write to `ruolinzheng-problem3.txt`

Problem 1

I cracked a total of `86` passwords. I ran Hashcat on CSIL machines. My first crack, 'beaver', came from brute-forcing all strings of 6 lowercase letters for 30 seconds. I also tried brute-forcing all strings of 7 lowercase letters for 10 minutes. Since these weren't very effective methods, I then used the wordlists **john, cain, facebook, phpbb**, both as raw wordlists and with Hashcat's Best64 rules. The results are summarized in the following table. (Note that the order in which different strategies are listed do not imply the order in which I applied different strategies.) Of all strategies, I ran `john` with no mangling first. The reason why it cracked the most despite a small wordlist size could be that users' passwords are relatively simple and that running it first gives it a lot of these low-hanging fruits. I also observed that running plain wordlists without rules is about as effective as running them with rules when the wordlist itself is large; the other benefit of running wordlists without rules is that it's significantly faster than running them with rules because of Hashcat's way of rotating rules. Overall, I had the most success when I started with several medium-sized (3MB) plain wordlists. Once they are exhausted, I ran them with Hashcat's Best64 rules. I also found that real-world leaks like **phpbb** were pretty effective: One explanation could be that they capture the password creation pattern of real-world users that might not be easily captured by Hashcat's rules (e.g. some combinations of numbers could be more likely than others, like 666, 999, etc.).

Strategy	Wordlist Size	# Cracked
6 lowercase brute force		1
7 lowercase brute force		4
john	3107	21
join + Best64		13
cain	306706	16
cain + Best64		13
phpbb	184389	15
facebook	2442	2
facebook + Best64		1

Problem 2

I used time as a side channel: If a query immediately returns (in 0.3 second), this means that the user doesn't have an account; If a query takes ¿ 1 second to return, then the user has an account.

When the server receives a query, it will fist look up the user in its database. If the user isn't in the database, the server can directly return 'Failure'. That's why the response in this case is fast. However, if the user is in the database, the server will attempt to verify the password associated with that specific username. As we've discussed in class, the hash function used in passwords are deliberately designed to be slow to thwart attackers. Therefore, it takes time for the server to hash the password and verify it, hence the slower response.

To fix this leak, we need to thwart attacks using time as a side channel: We can require that queries have uniform response time regardless of whether the user has an account or not by adding sleep time to the server. In the context of this problem, for example, we can make all queries return in about 2 seconds (if the user isn't in the database, the server can sleep for 1.7 second before returning; if the user is in the database, the server can hash and verify the password, then sleep for 0.5 second before returning).

Problem 3

I identified **38** users as having accounts in Problem 2. (Hence I need 12 users for this problem.) I got quite lucky in this problem: Out of these 38 username-password pairs, **11** directly returned 'Success.' To get the remaining one, I tried appending a '1' to each of the failed passwords, and got **5** more successes. For example, I got 'gordon11' (from the original 'gordon1'), 'chess11', 'atoato1', 'gatsby1', 'bullhorn11'. This could perhaps be my luck or that the distribution script that generated these passwords is skewed. It could also point to an interesting pattern in people's password creation: Password reuse is a widespread phenomenon, to the extent that 30% of passwords are reused across different sites. Excluding password reuse, to make a password on a new site different from the one on an old site, append a 1. If there is already a 1, append another.