The tools that I decided to go with was hashcat. That is because I wanted to use the GUI that comes with hashcat for the sake of simplicity and not having to use the command line for everything. I know that I personally would have had plenty of struggles getting either of these tools working exclusively through the command line, so hashcat was the perfect solution to this problem. Though it did take a good bit of effort to find a proper download link for the GUI, it made handling hashcat much easier.

I used my personal computer to look for passwords to hopefully decrease the long processing times. My machine has a Nvidia GeForce RTX 3060ti GPU which is used to increase the searching capabilities of hashcat. This gave me an average password check rate of roughly 9,000,000 passwords a second. This was a lot faster than when attempting to use my laptop with no GPU which could only guess around 15,000 passwords a second.

To try to be as efficient as possible, when using rules to search, I tried to limit the amount of rules to try to limit the amount of different possibilities to hopefully find some of the passwords quicker. Unfortunately, I ended up needing to run with all the leet speak rules all at once in order to find those password types. Another strategy I used was enabling hashcat to remove the found hashes from the shadow file to ensure that I was not repeating hashes.

Prior to giving a result of my found passwords, I should mention that I used the hashcat hash type “md5crypt, MD5 (Unix), Cisco-IOS 1 (MD5) (500)”. Also in order to try and shorten guess times, I had hashcat remove hashes from the shadow file that have already been found. Luckily, I was able to find all the passwords in the shadow file. Here is a list of all the found normal words salted and hashed in the shadow file: cincinnati, revolutionary, longitude, forwarding, increased. These were all found pretty much instantly by having hashcat run through the provided word list.

Here are the l33t speak passwords that I found: c0mp4ni35, 41t3rn4t3, 31iz4b3th, int3rn4ti0n411y, 14b0r4t0ri35. Initially, I tried to use the three l33t speak rules that came with hashcat, but none of them worked alone and it wouldn't work when I tried using all three together due to a lack of RAM. A fellow class mate instead recommended that I make a script that would make a new word list. What the script I made does is go through each word in the list, and make all possible combinations of the l33t speak version of that word (I will include the script and word list with my submission). Once I had the new word list, hashcat once again was able to find the l33t speak words pretty much instantly.

Here are all the alphanumeric passwords: ta, SJ, lsv, lr4, uddg, 31w4, lkyru, 2c8zc, 70gTHT, ugrknq. These passwords were found through brute-force, using the default mask ?1?1?1?1?1?1?1?1 with the character set that was recommended to me ?!?d?u. After running for an hour and 18 minutes, I was able to find all ten alphanumeric passwords.

To recap, here are all the passwords in the order of which they were found: cincinnati, revolutionary, longitude, forwarding, increased, c0mp4ni35, 41t3rn4t3, 31iz4b3th, int3rn4ti0n411y, 14b0r4t0ri35, ta, SJ, lsv, lr4, uddg, 31w4, lkyru, 2c8zc, 70gTHT, ugrknq.

1. Assuming that you used your setup for this project alone, how long do you calculate that it would take to crack a 6-character alphanumeric password? Eight characters? Ten characters? Twelve characters? Give exact estimates for either the average time needed to crack a password of the given length and composition, or the max time needed. Show your work.
    - My setup: 9,000,000 passwords a second. Possible characters 26 lowercase, 26 uppercase, 10 numbers = 62 possibilities per character.
    - 6 characters: 62^6 / 9,000,000 = 6,311 seconds or about 1.75 hours
    - 8 characters: 62^8 / 9,000,000 = 24,260,000 seconds or about 281 days
    - 10 characters: 62^10 / 9,000,000 = 9.32x10^10 seconds or about 2,956 years
    - 12 characters 62^12 / 9,000,000 = 3.59x10^14 seconds or about 11.4 million years
    - On average, you would find the password in half the time so here is the estimate for each size: 6 characters: 0.875 hours, 8 characters: 140.5 days, 10 characters: 1478 years, 12 characters: 5.7 million years
2. Recently, high-end GPUs have revolutionized password cracking. An RTX 3080 (a recent high-end GPU) is able to check 2.5 billion passwords a second using MD5 crypt. Using this guessing rate, estimate how long it would take to crack the passwords described in Question 1.
    - 6 characters: 62^6 / 2.5x10^9 = 22.7 seconds
    - 8 characters: 62^8 / 2.5x10^9 = 87,300 seconds or about 24.26 hours
    - 10 characters: 62^10 / 2.5x10^9 = 3.36x10^8 seconds or about 10.6 years
    - 12 characters: 62^12 / 2.5x10^9 = 1.29x10^12 seconds or about 40,800 years

- On average here is the time for each length. 6 characters: 11.35 seconds, 8 characters: 12.13 hours, 10 characters: 5.3 years, 12 characters: 20,400 years.

3. Do you think that the password meter is a good indication of actual password security? From the results of your experiment, what is your recommendation for minimum password length? Be creative in your response. Imagine what hardware and resources a potential attacker might have, and briefly justify your assessment of the attacker's capabilities.
    - Password meters only measure a passwords strength based off upper and lower, numbers and symbols. For example, P@ssw0rd might be considered a strong password but it can be solved in seconds because it is predictable.
    - I think a good password length would be at least 12 characters long. A GTX 3080 takes over 20,000 years to guess a password of that length, and there are far better GPUs available now. Passphrases like “GrandCanyonLibertySkateboard” are far better than any short passwords.
    - Attackers might also have multiple GPUs which could easily get them to well over a trillion guesses per second, so really any password that can be guessed under several years through brute-force should be considered weak.
4. Fedora 14 and other modern Linux distributions use SHA-512 (rather than MD5) for hashing passwords. Does the use of this hashing algorithm improve password security in some way? Why or why not?
    - Since SHA-512 is a far superior algorithm, it should theoretically slow down brute-force attacks. However, this only really matters if your password is considered safe.
5. Does the use of a salt increase password security? Why or why not?

- Salting is essential in password security since it prevents the use of precomputed rainbow tables, which makes it very easy to break hashes. It ensures that each password will be attacked individually instead of all at once.

6. Against any competent system, an online attack of this nature would not be possible due to network lag, timeouts, and throttling by the system administrator. Does this knowledge lessen the importance of offline password attack protection?
   - Protection against offline attacks is still crucial. Real data breaches happen often which gives attackers access to user databases with hashed passwords, and since it is offline, there is nothing on the network side that can stop attackers from guessing passwords.