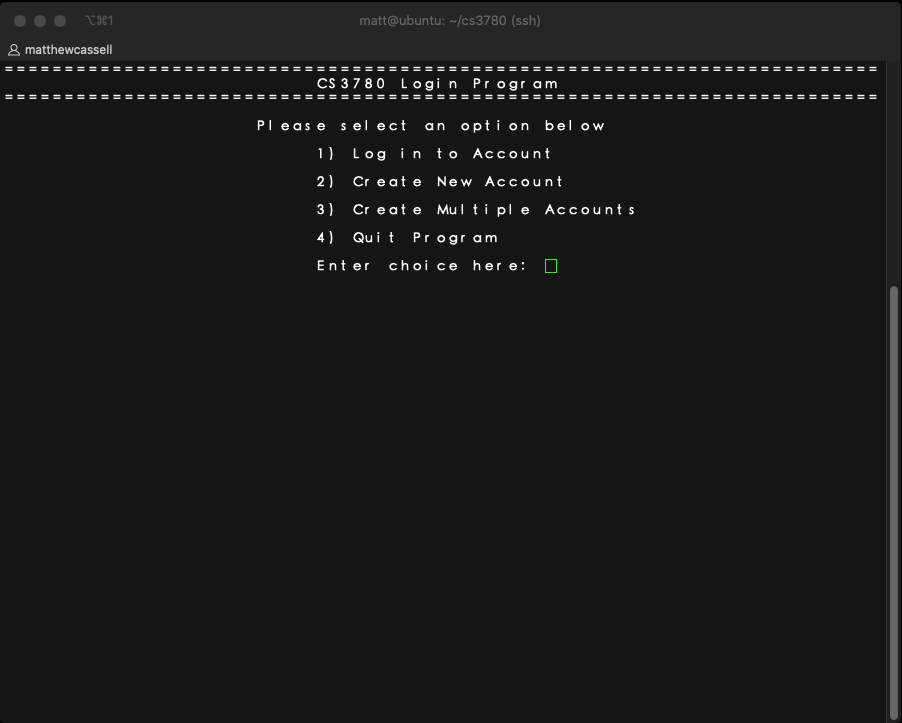
**CS 3780 Project 2**

**Overview:**

To complete all of the assigned tasks my partner and I created two programs; the first program was created using C++ and implemented OpenSSL (a cryptography toolkit) to hash our user's passwords. The second program was created using Python and was beneficial in documenting our password cracking efforts. The division of labor was that I created everything in C++ and my partner (Brandon London) created everything in Python. Before our analysis, I will show the framework of both of the above list programs:

**Task 1 and Task 2:**



The user runs a CPP file and brought to a menu, the menu serves to fulfill all required tasks as outlined in the assignment. First, let us create a user ourselves, and then create some random users.



Here, we create the user "markhaus" and give him a password of "linux". What happens behind the scenes is that we generate a random salt, and append it to the user's password before hashing and storing it into local text file. If the above procedure successfully occurs, we get the screen below:



We have now successfully created our first user. The user's name, password, and salt are now stored in a text file, similar to the notorious /etc/shadow file. What if we want to create one-hundred users with random password as required in task 2?



We ask the user the number of accounts they need to create, as well as the desired password length. Again, if the accounts are successfully written to our password file, we get this confirmation above. Let us now try logging into our program. When we attempt to login, we prompt the user to specify which password file they would like to authenticate against. Preferably, we would always want the authenticate against our hashed file with the salt included. However, we leave this option to are user as specified in the assignment.



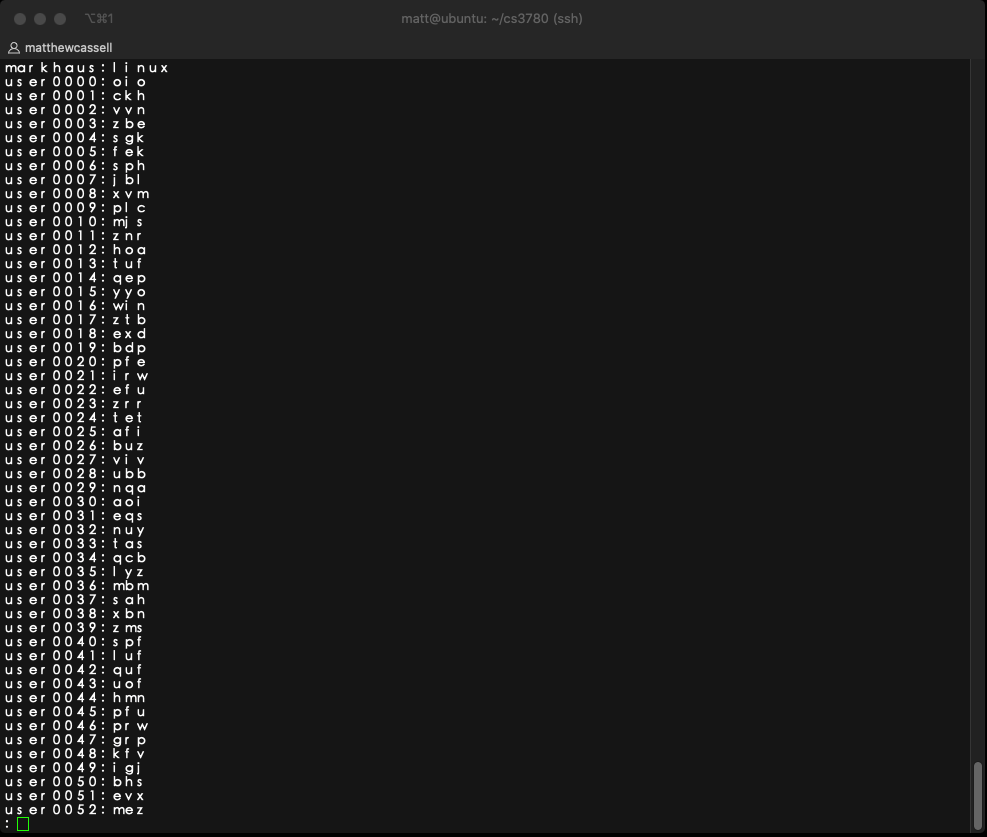
We choose to authenticate against our hashed password file that includes a random salt. For demonstration purposes let's a password we know will fail and see the results. We type in the username "markhaus" with a password of "apples":



Our program searches our password file for the username, if found it grabs the user's salt and hashes the attempted password along with the salt. It tries to see if the resulting hash matches what we have stored in our password file. If not we reply that either their username OR password is incorrect (from class we never tell them which value was invalid). Now let us run a successful login, with our user "markhaus" and his correct password of "linux":

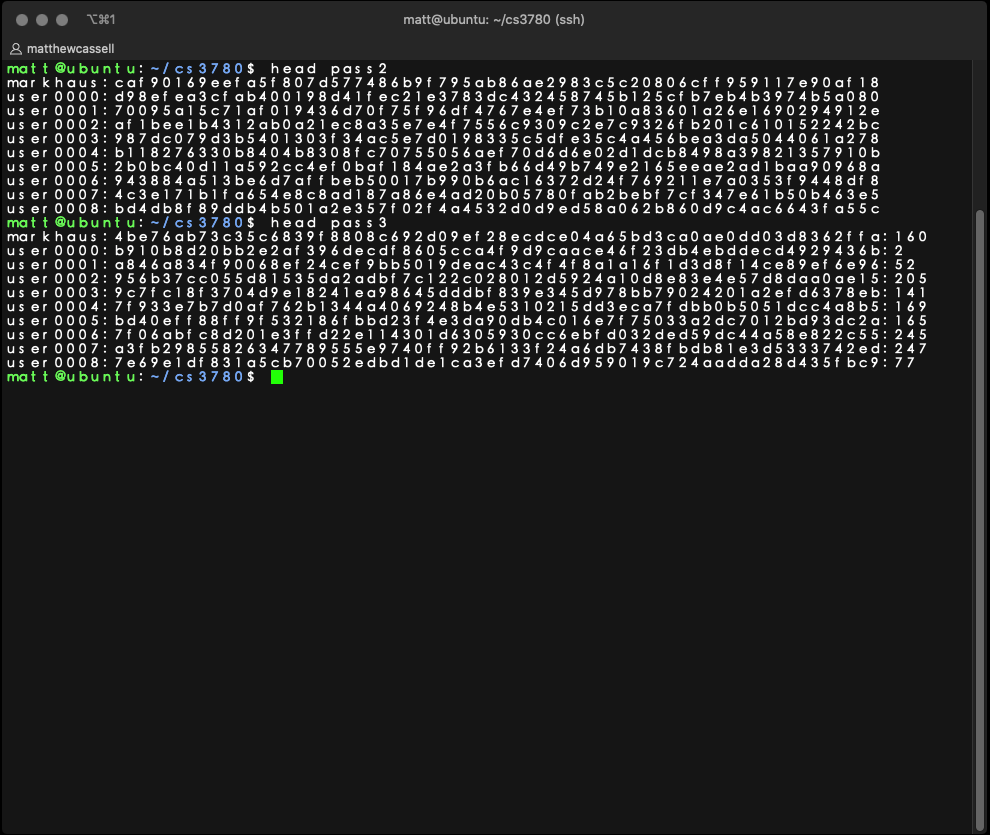


Our login functionality ends there. Let's inspect the password files we created as a result of running our program. Remember, we created 3 password files; a clear text file, a hashed file, and a hashed file that contains our user's password plus a random salt value. Here is our clear text password file:





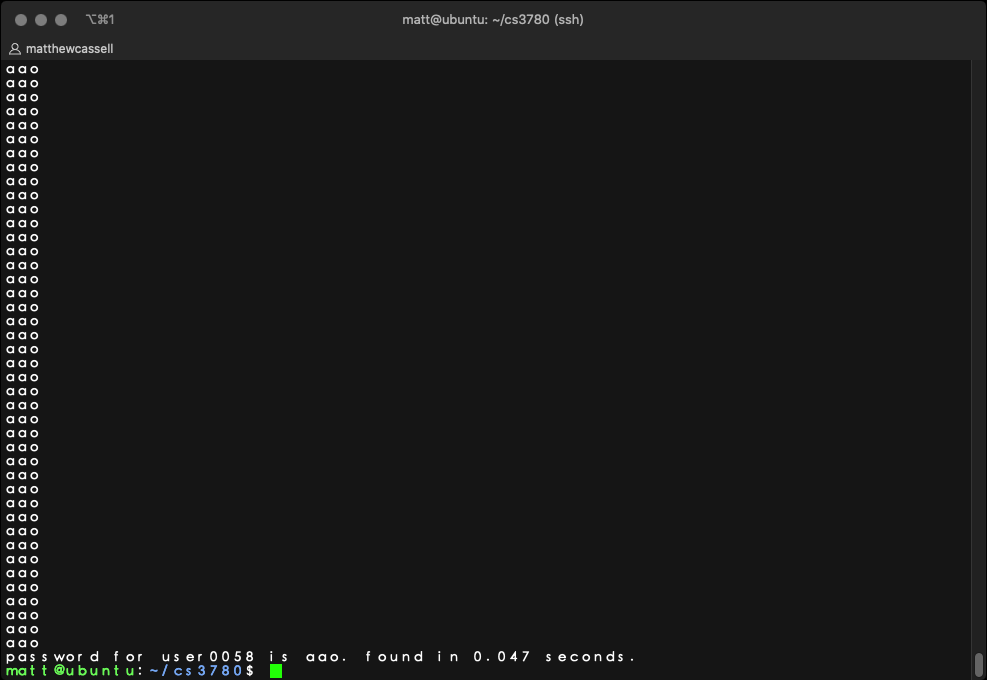
As promised, we have our initial user "markhaus" and our other 100 randomly created accounts. What about our hashed files? Here are the first 10 (for the sake of brevity) entries in our other two password files. Note our password files follow the username:hash:salt convention below:



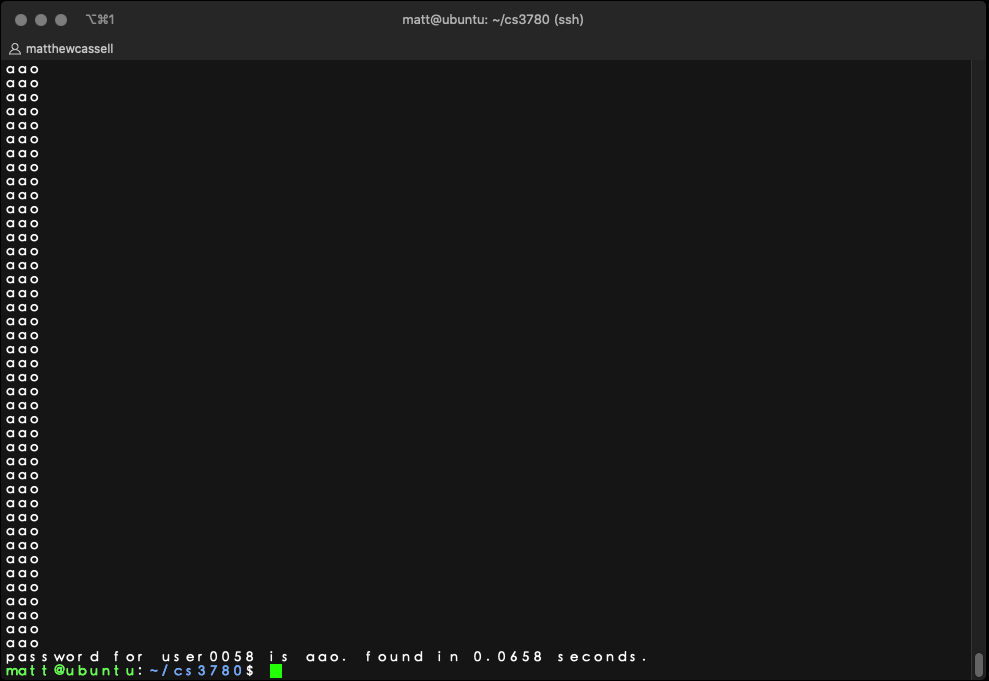
In conclusion, we have taken usernames and passwords, along with a randomly generated salt to create a hash using the OpenSSL SHA256 function shown in our sha256.cpp file. We've then stored the username, hash value, and salt value in a password file. We can then allow or deny access based on the user's login credentials, by authenticating them against our password file.

**Task 3 and Analysis**

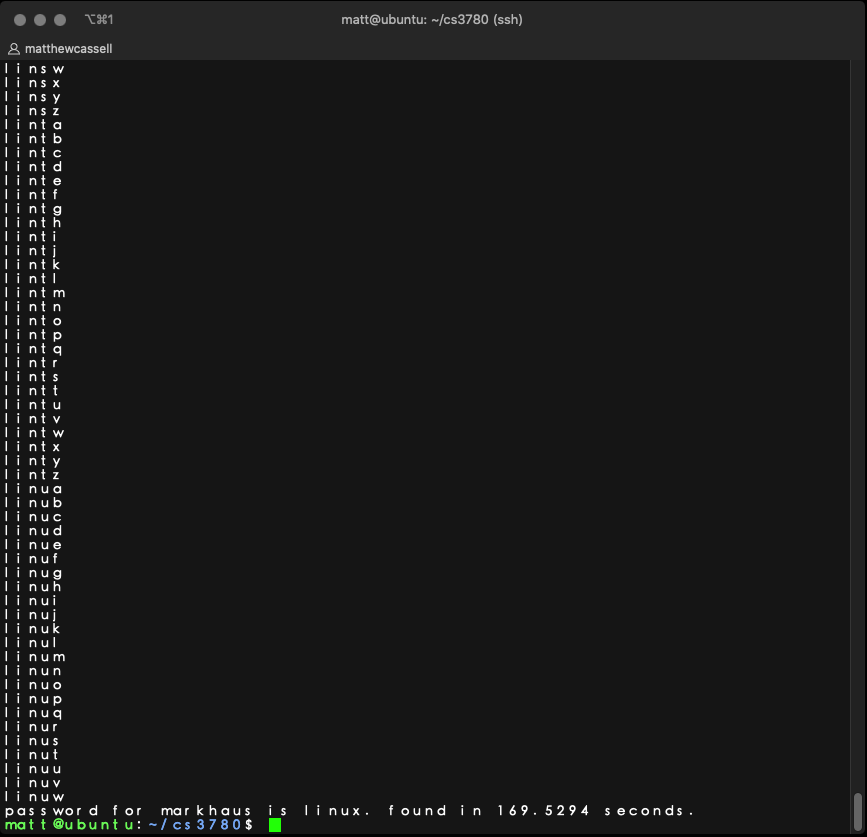
The next requirement is to try and "crack" the password file. We are under the circumstance that we have somehow managed to get read permissions on one of the hashed password files. To achieve this goal, we've created a program named crack.py, to try every possible password between 3 and 8 characters. Let's run it on our hashed password file and then on our hashed w/salt password file. With the command: python crack.py pass2 we get:



It takes our program 0.0457 seconds to crack its first password. That password is "aao" for user "user0058". Now let's try our crack.py against our hashed w/salt password file.



We get the same username and password, but it takes slightly longer at 0.0658 seconds. In our testing, all of our outcomes follow this trend. The password that are appended with a salt take longer to get cracked. The above examples are really best-case scenarios. Let's remove all of our simple 3 letter passwords and see how long it would take our program to find a more realistic password like the one our user "markhaus" created. When we run the password cracking program against our hashed password file it takes 169 seconds (about 3 minutes).



While the time taken to crack our hashed w/salt password file takes 174 seconds (roughly 3 minutes). It appears that adding our salt merely gives an attacker a 5 second speed bump. But this is because we already knew the salt, in reality, attacker won't have access to the hashes salt. To answer the question what is an acceptable format for passwords? It appears that the longer the password the longer it takes to get cracked. The salt really wasn't a detriment, but again, this is because we only had it up to 256 possible values and already gave it to the cracking program. Our program required that a user entered a password that was between 3 and 8 characters long. Which is not enough for an attacker with better performance than the machine we performed our analysis on. A combination of a long password with a minimum 12 characters: lower-case, upper-case, digits, and special characters will dramatically slow down an attacker's efforts.