Nick Alexander

February 7, 2015

Assignment 3 – Password Cracking

The purpose of this assignment was to crack a Linux style password file using a brute force dictionary attack. Linux stores a record of all users who have an account on the system in a file called passwd.txt located in /etc. The system implements a very unique, open-source encryption algorithm to encrypt the passwords using an 8 character password and 2 character salt against a string of 11 zeroes. Amazingly, it is mathematically impossible to reverse the encryption; this means that one would have to brute force the passwords to crack it. In the assignment ecosystem, a passwd with 20 users is supplied which is guaranteed to contain passwords created from a set of 390 words. While the environment is elementary and unrealistic, it exemplifies how safe the passwd file truly is.

Similar to the Cryptomatic assignment, my algorithm constructs a HashMap of the word list and then systematically applies common password mangles. These mangles include operations such as capitalization, reflection, and append. Once the word is mangled, it is encrypted using the JCypt application using the salt extracted from the encrypted password. The resulting encrypted string is compared against the real encrypted string for match. The algorithm is able to crack 18 out of the 20 (90%) user passwords in approximately 12 seconds. After running some empirical tests, I chose to remove the character prepend and character append double mangles since there was a lot of computation time for no results. In a real-world situation, this choice would most likely need to be reversed or at least optional.

The most immediate challenge was attempting to find a solution that systematically applied all double mangles without using a dozen if-else statements or a large set of for loops. In the beginning, my intention was to use something like a function pointer, but Java does not have support for function pointers. A similar concept was lambda statements; however, I was not able to see how lambdas could effectively solve the problem. At that point, I chose to abstract the methods into the Mangler class with a switch statement; I also created an enumeration of all possible mangles which allowed me to run a foreach loop across all values. This was the beginning of the solution, but the biggest hurdle was still to come.

The ultimate challenge in developing the algorithm was trying to handle the append and prepend double mangles. Too many times I would get stuck in an infinite loop or have the run time be exponential. In fact, I changed the entire architecture of the program upwards of 3 times in order to accommodate these two mangles. Finally, I was able to successfully apply the mangles to find that the prepend double mangle was largely fruitless. Based on this empirical data, I opted to remove the implementation to maintain a quick run time.

In terms of data structures, the algorithm utilizes three HashMaps each containing a String key and a String value. The dictionary map simply contains the words from the word list in both the key and value positions. While redundant, it still allows the value to be retrieved in constant time which is very important. Next, the accounts map stores a user name as a key and the corresponding encrypted password as the value. Finally, the crackedPaswords map stores an account name as a key and the cracked password as the value. Again, the maps are important because they allow for constant time retrieval and insertion.

As a conclusion, I will discuss my results again. Using single mangles, my algorithm is capable of cracking 13 passwords in roughly 2 seconds. Once the double mangles are applied, the run time relatively skyrockets to 13 seconds but cracks 18 of the 20 passwords. It is my opinion that the remaining two passwords are either nonsensical or contain a mix of numbers and special characters (l1k3th!s). The final case would be extremely hard to crack in a real world situation given the randomness combined with the 200,000+ word dictionary as found in the UNIX /usr/share/dict/words.txt file. Overall, I am very satisfied with the evolution and results of my algorithm. I am hopeful that I am at the very least a contender for the speed bonus points.