COSC 345 – Homework 1
Passwords, Cryptography, & First Blog Entry

Part 1: Writing a Password Cracker

Objective

This is a small programming assignment to get the feeling for how a password cracking program works. You also should learn some about one-way encryption and why choosing a good password is important. **You may use Python, Java, or C/C++ to complete this assignment.**

Background info

If you use C:

The Unix `crypt(3c)` function is used to encrypt a password. It is a function that implements a one-way encryption algorithm used for password encryption and requires `#include < unistd.h>`. It takes two `const char*` parameters and returns a `char*`. The first parameter is the password to encrypt (only the first 8 chars are significant). The second parameter is a "salt" used by the encryption algorithm. The salt is a two-character string chosen from the set [a-zA-Z0-9./]. On a successful encryption `crypt()` returns a pointer to the encrypted password. The string returned is 13 characters (from the set [a-zA-Z0-9./]) long and the first two chars are the salt. On an error it returns a null pointer. *(Linux note: you need to link with libcrypt, `-lcrypt`)*. Be sure to read the entire man page Linux: `man 3 crypt`.

If you use python:

15.1. hashlib — Secure hashes and message digests

https://docs.python.org/3/library/hashlib.html

Source code: Lib/hashlib.py

This module implements a common interface to many different secure hash and message digest algorithms. Included are the FIPS secure hash algorithms SHA1, SHA224, SHA256, SHA384, and SHA512 (defined in FIPS 180-2) as well as RSA's MD5 algorithm (defined in Internet **RFC 1321**). The terms "secure hash" and "message digest" are interchangeable. Older algorithms were called message digests. The modern term is secure hash.

Since `crypt()` or hashlib provide a one way encryption the password cannot simply be decrypted. So given an encrypted version of a password it seems that we cannot get the plaintext password. However, we can make guesses and see if one of our guesses encrypts to the same thing. If it is then chances are we have guessed the password (or at least something that encrypts to the same thing, which is all we really need). Knowing these things one can write a password cracker. A

brute force cracker would try all possible passwords. If we know properties of the password this could be feasible. Other crackers (hashcat, john the ripper) use rules and a dictionary of words to generate guesses (like capitalize the first char of a word and follow with a number).

Assignment
Now, finally to the assignment. You will write a password cracker that uses some rules and does a brute force attack on a set of passwords given those rules. Your program must be written in C++ or Python and work on any size password file that was encrypted using the **SHA-256** hashing. The format of the password file is (with one entry per line):

```
username:encyption[:otherstuff]
```

[:otherstuff] is optional (some password files have it, some do not, so your program needs to be able to handle both ways).

Rules to implement in your password cracker

- A seven char word from `/usr/share/dict/words (Linux or Mac)` which gets the first letter capitalized and a 1-digit number appended.
- A five digit password with at least one of the following special characters in the beginning: *, ~, !, #
- A five char word from `/usr/share/dict/words` with the letter 'a' in it which gets replaced with the special character @ and the character 'l' is substituted by the number '1'.
- Any word that is made with digits up to 7 digits length.
- Any number of chars single word from `/usr/share/dict/words (Linux or Mac)`

Your cracker should work as follows. In a loop generate guesses based on the rule you are implementing. When you have found a match print it out to standard output and to a file in the format encrypted:password. Your program should stop once it has cracked all the passwords in the file or you run out of combinations for that rule. You may want to generate test cases of your own as well.

Some samples to test if your cracker is working

```
   Puzzles4 -> homer:84b175349a3d5a8bcabda1ab8eb84e3c36139f27e3a04b17b47c497a3b577940:20:Homer
Simpson:/home/homer:/bin/tcsh
   ~0123 -> marge:fd479093306572e9a230caaafbf8d975b129909199befcb3c35b8c35dc29c2f4:351:Marge
Simpson:/home/marge:/bin/tcsh
   be@ch -> lisa:e77599c90db264dbe4b449565a03b5a26c989e975089fcc7dc3c55a720928e66:353:Lisa
Simpson:/home/lisa:/bin/tcsh
   programming ->
Maggie:ab2620f9b7154d9f9dc1b3c2d949d85d595fe77f45411b3dbe6e5b47da564177:42:20:Maggie
Simpson:/home/maggie:/bin/tcsh
```

Note that your code will also be tested with other passwords.

Think how you could improve the speed of your password cracker software and implement the solution or at least develop your thoughts in your report.

What to submit:
1. Your code main file named: <LastName1>_<LastName2>_HW1_passwd_cracking. Any additional files needed to run your code.
2. A set of tests for your code.
3. A README on how to run the code.
4. Add all these files in a directory named: <LastName1>_<LastName2>_HW1 and compress this directory. Submit the zipped directory to Oaks.