CS361 Assignment 2

Due: Tuesday, June 20, 2017, by midnight.

Deliverables:

You will be submitting your code electronically to the TA. Make sure that your code is commented and follows good coding style. Clearly identify the members of your team (one or two people). Along with your code, submit a README file explaining your algorithm and listing which passwords you were and were not able to crack from the two sample **/etc/passwd** files. Include timing results showing how long it took to crack the passwords. You can invent your own format for this information, but here's a suggestion: start a timer when you start your program. For each password you discover, print out the current value of the timer. That way, we'll be able to see the trend. E.g., did you find a bunch right away and then have to start working considerable harder, etc.

Make sure that all members of your group understand the code and contribute equally, as much as possible, to its development. Specifically:

1. The primary file name for this assignment is `PasswordCrack.java`. The main method must exist in `PasswordCrack.java`. Students can add more files as they wish but all must be submitted.

2. The assignment must be compiled by `javac *.java`.

3. The assignment must be executed by `java PasswordCrack inputFile1 inputFile2` where `inputFile1` is the dictionary and `inputFile2` is a list of passwords to crack.

4. Students must use standard output to print cracked passwords.

Don't forget that you'll be zipping your Java implementation files and README.txt file together and submitting a single zip file for the main program, and a separate zip file if you also do the extra credit. You should include either or both of the extra credit steps in one file. Don't forget to name your zip files according to the instructiions in General instructions.

Grading README file: Your zip file must contain a README.txt file. We're also using a program to grade your README.txt file. Any other names or formats of REAMDE file won't be accepted, e.g. readme.txt, readme, README, README.c, README.java. The README.txt file worths 4 (of the 10 points). If you use the incorrect REAMDE.txt file, you have to email TA to handle this issue. Here is an example README.txt for this assignment. Note that template is slightly different from earlier assignments. Please download it and modify it directly. Don't use your own template!!!

For the README.txt file, the maximum point is 4.

1. First Five Lines [0.25 point] have to match the format that has been explained in General instructions..

2. Good description of your code [0.25 point]

   You have to write some sentences (at least 100 words) to explain high-level idea of program, some short description of the key/important function in your code.

3. You need to tell us how much you've finished [0.5 point]

   (1) Finished all requirements (2) Only finished parts of the requirements.

If you're in the case (2), you have to list the unfinished things and write down the reasons for each, either no time or have some bugs. If grader thought you explanation is very good and detailed, you won't lose point(the 0.5 point here) even if you didn't finish many requirements of program.

In this time, you do need to write this sentence "We found XX/20 on passwd1 in time XX seconds. We found XX/20 on passwd2 in time XX seconds." in README.txt, otherwise you won't get 0.5 points.

4. Write down the number/list of cracked/uncracked cases [3 point]

   Here is an example of README.txt for this program assignment 2. There are two files: passwd1 and passwd2. For each file, you need to write one sentence in README.txt to tell us "the number of cracked passwords", "the list of uncracked ones", and "the list of cracked ones". If program works well, and you didn't write down the number and list in README.txt, you will lose 3 points directly. Each files contain 20 cases. [1.5] If you can crack >=15 cases; [1.0] If you can crack >=10 cases; [0.5] If you can crack>=5 cases.

Grading Program: For the program, the maximum point is 6.

1. You will get [2 points] if the program can be compiled. Before submission, make sure that your code can be compiled and executed on the Linux Machine (in our CS Lab). Even if your code is working on your own machine (Windows, ...), but not working on CS Lab machine, you won't get any points!

2. You will get [4 points] If the program can be complied and run.

3. You will get [6 points] If your output is reasonable.

Password Cracking: Dictionary Attacks

On a traditional Unix system, passwords are stored in encrypted form in a world-readable file **/etc/passwd**. Moreover the encryption algorithm is widely known. This means that an attacker can attempt to discover one or more passwords on the system by encrypting a sequence of strings and comparing the results against the stored encrypted passwords of the system. If any of the trial encryptions match stored encrypted passwords, the attacker will know the corresponding cleartext password for that user and can then use it to access the user's account. This is a classic *dictionary attack* and explains why many systems enforce rules to ensure that user-generated passwords are not easily guessed words.

Systematic password guessing involves both cleverness and brute force. Dictionary attacks are so named because a word list, or dictionary, is used to generate password guesses. For example, the public Linux systems at UT have such a dictionary file at **/usr/share/dict/words**. A more sophisticated dictionary attack, not only uses common words and phrases, but also attempts users' surnames, common pet names, etc. Such words and phrases may be prepended to the dictionary and then become available in the attack.

A user may attempt to render his or her password unguessable by "mangling" the plaintext password in some algorithmic way. Some common "mangles" (ways to take a password and make it less easily guessable) are listed below. Assume the plaintext password is "string". You might:

- prepend a character to the string, e.g., @string;
- append a character to the string, e.g., string9;
- delete the first character from the string, e.g., tring;
- delete the last character from the string, e.g., strin;
- reverse the string, e.g., gnirts;
- duplicate the string, e.g., stringstring;
- reflect the string, e.g., stringgnirts or gnirtsstring;
- uppercase the string, e.g., STRING;
- lowercase the string, e.g., string;

- capitalize the string, e.g., String;
- ncapitalize the string, e.g., sTRING;
- toggle case of the string, e.g., StRiNg or sTrInG;

You need only consider characters that you could type from your keyboard. Weird control characters don't usually occur in passwords.

A program called Crack is available to system administrators to test the guessability of user passwords, as well as by hackers to perform dictionary attacks. You can view the documentation for Crack at Crack Documentation.

The goal of this assignment is to implement a portion of Crack's functionality and attempt to guess one or more passwords. Input to your program will be a "captured" **/etc/passwd** file from a system with 20 users. Your aim is to crack as many passwords as possible. *But don't expect to crack them all*; there are a few passwords included generated from random strings. If you get 15 or so passwords, you're doing just fine.

How do you know when to stop? You don't! Write the program to run until it finds all of the passwords. The TA will stop it when he gets tired of waiting. Realistically, your program should find a majority of the passwords (12 or so) in just a few minutes. Make sure that you print out the passwords as they are found and that you code your program reasonably efficiently.

To do this for a specific user, you might take the following steps:

1. Extract the encrypted password and salt for that user (see format below);
2. Seed the word list with words that the user might have utilized in constructing his or her password (e.g., his first and last name);
3. With the salt and augmented wordlist, systematically encrypt words and compare against the stored encrypted password;
4. Redo step 3, but using mangled versions of the words;
5. Redo step 4, attempting to apply two mangles to each word.

Design your program in such a way as to be as efficient as possible. For example, your program should stop searching with respect to a given user if you have cracked that password. Consider whether to use a breadth-first or depth-first search. The algorithm only considers the first 8 characters of a password, but the user might or might not take that into account. You do not have to break all passwords, but you should break at least the simple passwords (generated from words in the dictionary using one mangle). In general, if you can't break most of the passwords, you're not trying hard enough.

For testing purposes, you will be provided with the following files:

- passwd1 - twenty accounts with plain text passwords provided passwd1 answers
- passwd2 - twenty accounts without plain text passwords provided
- words - a list of words that you can use as a dictionary (alternatively, you could use the Unix system dictionary at **/usr/share/dict/words** of ~100,000 words, but expect things to run pretty slowly).

After you turn in your program, it will be run against a third **/etc/passwd** file, which will not be provided before the turn-in date. This is to keep everyone honest, since it is completely possible to just run Crack on the provided files.

Encryption Specifics

On traditional UNIX system, passwords are encrypted and stored in the file **/etc/passwd**. The stored value is actually the result of encrypting a string of zeros with a key formed from the first eight characters of your password and a two-character "salt". **Notice: When you're encrypting, it's a total waste of time to consider more than eight characters. Always truncate your prospective passwords to 8 characters.**

The "salt" is a two-character string stored with a user's login information. Salt is used so that anyone guessing passwords has to guess on a per-user basis rather than a per-system basis. Also, in the case that two users have the same password, as long as they have different salt, their encrypted passwords will not be the same.

All of the passwords for this project have been encrypted using JCrypt which can be found on-line at: JCrypt. JCrypt is a Java implementation of the UNIX Crypt function. JCrypt includes a method **crypt(String salt, String password)** which will return the encrypted result of a given salt and password. Be careful--some implementations seem to take the args in the opposite order: **crypt(String password, String salt)**.

For example, if a user's plain text password is "amazing" and the salt is "(b", then JCrypt would return "(bUx9LiAcW8As". Use JCrypt in your program when checking your password guesses.

Lines in /etc/passwd have the following format, with fields separated by colons:
account:encrypted password data:uid:gid:GCOS-field:homedir:shell

For example, this line represents the account for Tyler Jones. The salt is "<q".
tyler:<qt0.GlIrXuKs:503:503:Tyler Jones:/home/tyler:/bin/tcsh

The encrypted password data field is thirteen characters long. The first two characters are the salt, and the next eleven characters are the encrypted password (actually, a string of zeros encrypted with the salt and the password).

Newer systems make dictionary attacks more difficult by employing "shadow passwords." In a shadow password system, the password field in /etc/passwd is replaced with an 'x'. Actual encrypted passwords are stored in a file /etc/shadow which is not world-readable.

The following are some on-line resources:

Password Security: article

Crack readme: ftp://coast.cs.purdue.edu/pub/tools/unix/pwdutils/crack/crack5.0.README