Password Cracker Assignment

Skicka uppgift

Förfallodatum 11 maj före 21:00 **Poäng** 100 **Lämnar in** en filuppladdning
Filtyper zip **Tillgänglig** 27 apr kl 10:00–11 maj kl 21:00 14 dagar

Introduction

In this assignment, you are provided a traditional UNIX-style password file. The file contains salted and hashed passwords. Your job is to reveal as many of the passwords as possible, by performing a dictionary attack where you generate password guesses, hash them, and match the result with the entries in the password file.

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.

Password Cracking

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. A more sophisticated dictionary attack not only uses common words and phrases, but also attempts users' surnames, common pet names, "worst passwords" from lists published on the web, 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., 0string;
- 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 only need to consider characters that are letters (uppercase and lowercase) and numbers; so no special characters (such as "#", "¢", "?", etc) or control characters. (This goes against the general wisdom that says that a "good" password should consist of different kinds of characters, but it simplifies the assignment.)

There are several programs available to system administrators to test the guessability of user passwords, as well as by hackers to perform dictionary attacks, such as John the Ripper, Cain & Abel, and Crack.

The goal of this assignment is to implement a portion of those programs' 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; 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. 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:

- Extract the encrypted password and salt for that user (see format below);
- 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);
- With the salt and augmented wordlist, systematically encrypt words and compare against the stored encrypted password;
- Redo step 3, but using mangled versions of the words;
- 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. Also consider if you should try to break one password at a time, of if you should try to match each guess against all entries in the password file. The algorithm only considers the first eight 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.

Encryption Specifics

On traditional UNIX systems, 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".

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.

For example, if a user's plain text password is "amazing" and the salt is "lb", then JCrypt would return "lbUx9LiAcW8As". 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
```

The GCOS field is in free-text format and is often used for the full name (the origin of the label "GCOS" is historical). 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).

As a remark, newer systems make dictionary attacks more diffcult 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.

Implementation

In this assignment, a the cracker program is called **PasswordCrack**. It takes two arguments, and should be executed as follows:

```
$ javac PasswordCrack.java
$ java PasswordCrack <dictionary> <passwd>
```

The first argument <dictionary> is dictionary of words, with one word per line. The second argument <passwd> is the password file, containing the hashed passwords in the format described above. The cracked plaintext passwords are printed to the terminal (standard output), one per line. The passwords can be printed in any order, so as soon as you have cracked a password, just print it out. **Do not write anything else besides passwords to the terminal!** So no logging or debug messages. We will run your program and check its output, and everything your program writes will be taken for cracked passwords.

Environment

This assignment is a programming exercise, and you should use Java to solve it. You can use any Java development environment of your choice; however, it is a requirement that the code you finally submit runs on the the course virtual machine. Furthermore, your submitted code should be possible to execute directly in a Linux shell, as described above in section "Implementation", *so it should not depend on any IDE tool such NetBeans or Eclipse.*

Challenge

Each student gets a unique password file to crack – a *challenge*. So the first thing you should do is to fetch your challenge. We use a separate assignment in Canvas for this: Password Cracker Challenge. Request your challenge by making a submission there (it doesn't matter what you submit). After a while, you will get your individualized challenge as feedback to your submission.

Your challenge will be a zip archive "cracker-challenge.zip". There you can find the following files:

- passwd1.txt – password file with twenty entries
- passwd1-plain.txt – plaintext passwords for the entries in passwd1.txt
- passwd2.txt – password file with twenty entries (each student gets a different file)
- INSTRUCTIONS – a file with brief instructions for your submission
- Makefile – configuration file for "make" that you should use to create your submission

You can find a list of words that you can use as a dictionary here: dict.zip. Actually, we strongly recommend you use this dictionary, since it is the main dictionary we use for generating passwords.

Submission

For submission, you will turn in the following files:

- passwd2-plain.txt – plaintext passwords you have successfully cracked for passwd2.txt. Only the passwords, nothing else. One password per line, in no particular order.
- PasswordCrack.java – your Java source code.

To submit your solution, use the config file for "make" included in the challenge:

```
$ make cracker.zip
```

This will create a zip archive called "cracker.zip". To submit, upload exactly as is.

Important: the challenge contain authentication files, which are used to verify that your submission really is for your challenge (and not for someone else's). The "make" program will automatically include those files with your submission. Therefore, you must run "make" in the same directory where you extracted your challenge from the zip archive. Otherwise we will not be able to grade your assignment.

Testing

The grading is based on how many correct passwords you report from "passwd2.txt". In addition, your submission will be tested against two other /etc/passwd files, which will not be provided to you. You can find more information about the grading in Password cracker grading notes.

In addition, your implementation will be tested for basic input validation and error handling. If the user gives incorrect parameters, or if the input data does not match the parameters, this should be detected and an informative error message should be generated. (Java exceptions do not count as informative error messages.)

Here are some examples of tests your program should pass.

- Proper checking that the input files exist and are readable.
- Deal with input files of variable length (don't assume a certain size of dictionary and password file).

Acknowledgements

This assignment originally comes from CS361 Introduction to Computer Security at University of Texas at Austin by Dr. Bill Young. We are very thankful to Dr. Young for letting us use this assignment.