CMPS 122 Winter 2019

Lab #1: Password cracker

Due: 23:59 Thursday, January 24, 2019
This lab is 5% of your final grade

In this lab you will develop a password cracker. While dictionary based attacks are typically more successful against real-world passwords, it is recommended you use a brute force "key search" attack where every possible combination in the key set is checked against supplied encrypted passwords.

Background

Early UNIX systems stored users' one-way encrypted passwords in `/etc/passwd`, a publically readable file.

If users had adopted 8 character passwords (these early UNIX systems would often truncate passwords longer than that to 8 characters anyway) and had selected from all 90+ characters on the typical keyboards of the day, to crack a single password via a brute force attack on a single, single CPU machine would have taken between ten and one hundred years.

As computers became faster, that "password recovery" time decreased rapidly and the one-way encrypted passwords were moved from `/etc/passwd` to `/etc/shadow`, a file readable only by the super user.

Today, an 8 character password selected from the 96 keys on a standard US keyboard can be cracked in a matter of hours either by a super computer or, more likely, a distributed system where the key set is divided and the recovery/crack attempt carried out in parallel.

However, for such an attempt to be made, access to `/etc/shadow` must have been established. To do that super user privilege must have been gained and password cracking of non-privileged user accounts is now superfluous as the super user can do whatever they like, including creating a new account, potentially with it's own super user privileges.

The `crypt` function

On modern UNIX systems, the `/etc/passwd` entry for Joe Smith would look something like this:

```
smithj:x:561:561:Joe Smith:/home/smithj:/bin/bash
```

The matching /etc/shadow entry something like this:

```
smithj:Ep6mckrOLChF.:10063:0:99999:7:::
```

Where `smithj` is the user name and `Ep6mckrOLChF.` is the encrypted password.

The 13 byte encrypted password is generated by the `crypt()` function from the supplied plain text password and a two-character string (known as `salt`) used to perturb the algorithm in one of 4,096 different ways. The salt is included as the first two bytes of the encrypted password. http://man7.org/linux/man- pages/man3/crypt.3.html.

If the encrypted password is known, then a brute force attack can be developed. If we assume 96 keyboard characters are available and passwords must be between 6 and 12 characters inclusive; the search space for a brute force attack is:

6 characters: $96^6 = 7.8 * 1011$

7 characters: $96^7 = 7.5 * 1013$

8 characters: $96^8 = 7.2 * 1015$

9 characters: $96^9 = 6.9 * 1017$

10 characters: $96^{10} = 6.6 * 1019$

11 characters: $96^{11} = 6.4 * 1021$

12 characters: $96^{12} = 6.1 * 1023$

Total: 619,159,333,722,704,000,000,000 (0.6 septillion)

Scenario

You have obtained a copy of an old-style `/etc/passwd` file containing passwords you are reliably informed are exactly four bytes long and use only upper and lower case characters, and numeric digits.

```
alice:aldFq8L.QWpSY:alice:/home/a1ice:/bin/sh
bob:boXQXgsXdii8I:bob:/home/bob:/bin/sh
eve:evxATEyXE1n1Q:eve:/home/eve:/bin/sh
rivest:riOinhnROr.u2:rivest:/home/rivest:/bin/sh
shamir:shEadhBNYHfKk:shamir:/home/shamir:/bin/sh
adleman:adkTnZqlx/WwQ:adleman:/home/adleman:/bin/sh
```

Some things to note:

- You can easily see from the encrypted passwords that the `crypt()` salt used was the first two characters of the user name.
- The password search space contains $62^4 (14, 776, 336)$ unique combinations.

Setup

SSH in to the CMPS122 teaching server. Use Putty (`http://www.putty.org/`) if on Windows:

```
$ ssh <cruzid>@grunhilda.soe.ucsc.edu
```

Create a suitable place to work: (only do this the first time you log in)

```
$ mkdir -p CMPS122/Lab1
```

Install the lab environment: (only do this the first time you log in)

```
$ cd ~/CMPS122/Lab1
$ tar xvf /var/classes/CMPS122/Winter19/CMPS122-Lab1.tar.gz
```

Build the skeleton code:

```
$ cd ~/CMPS122/Lab1        (always work in this directory)
$ make
```

Now create a sample `users3` (for 3-character passwords) and `users4` (for 4-character passwords) file to test your implementation. Each line in these files has the format `<username>:<password>`. To simplify the development process, the password is stored as cleartext in these files. The grading harness will call `crypt()` on these passwords and pass the result to your code. For example, the contents of your `user3` file might include:

```
alice:god
bob:sec
charlie:123
```

Also try:

```
$ make check               (runs the required functional tests - see below)
$ make grade               (tells you what grade you will get - see below)
```

To run individual tests:

```
$ make single
$ make multiple
$ make speedy
$ make stealthy
```

To run individual tests with 3 character passwords (they will run significantly faster):

```
$ ./test -single users3 3
$ ./test -multiple users3 3
```

Note that initially, all tests will fail.

Requirements

Basic:

- Crack a single four character password
- Implement `crackSingle()` in `crack.c`

Advanced:

- Crack six, four character passwords in an old-style `/etc/passwd` formatted file
- Implement `crackMultiple()` in `crack.c`

Stretch:

- Crack six, four character passwords in an old-style `/etc/passwd` formatted file
- Do so in under 15 minutes on grunhilda.soe.ucsc.edu
- Implement `crackSpeedy()` in `crack.c`

Extreme:

- Crack a single four character password
- Do so without ever using more than 15% of available CPU resource.
- Do so in under 60 seconds on grunhilda.soe.ucsc.edu
- Implement `crackStealthy()` in `crack.c`

Notes

You can create as many additional `.c` and `.h` files as you like. If you need to use additional system libraries, modify `Makefile.libs`. You CAN NOT link to any third party libraries. DO NOT MODIFY `test.c`, `grade.sh`, or `Makefile`. Fresh copies of these files will be installed by the automated grading system so any changes you make will be ignored and you may consequently fail the lab if you relied on the changes made. The passwords used by the automated test system ARE NOT THE SAME as the ones used during your development process.

What to submit

In a command prompt:

```
$ cd ~/CMPS122/Lab1
$ make submit
```

This creates a gzipped tar archive named `CMPS122-Lab1.tar.gz` in your home directory.

UPLOAD THIS FILE TO THE APPROPRIATE CANVAS ASSIGNMENT.

In addition to submitting modified and new source files, you are required to write a short report (no more than two pages) on your work.

This report should contain at least:

- A defense of the rationale behind your design
- Details of tests your submission fails and what investigations you undertook to try and find out why

If you keep a simple journal as you work your way through this lab, writing the report will be easy - it's essentially a tidied up version of your journal.

SUBMIT YOUR REPORT TO CANVAS IN THE SAME ASSIGNMENT AS YOUR CODE ARCHIVE. Note that the report WILL NOT BE READ unless plagiarism is detected in your submission.

What steps should I take to tackle this?

Come to the sections and ask.

How much code will I need to write?

A model solution that satisfies all requirements has approximately 200 lines of executable code in it.

Grading scheme

The following aspects will be assessed:

1. (100%) Does it work?
   a) Basic Requirements (50%)
   b) Advanced Requirements (30%)
   c) Stretch Requirements (10%)
   d) Extreme Requirements (10%)
2. (-100%) Did you give credit where credit is due?
   a) You submission is found to contain code segments copied from on-line resources, but you did not give clear and unambiguous credit to the original author(s) in your source code (-100%)
   b) You submission is found to be a copy of another CMPS122 student's submission (-100%)
   c) Your submission is found to contain code segments copied from on-line resources that you did give a clear an unambiguous credit to in your source code, but the copied code constitutes a significant percentage of your submission:

< 50% copied code	No deduction
50% to 75% copied code	(-50%)
75% to 90% copied code	(-75%)
> 90%	(-100%)

Appendix 1: Running on your own computer

The home directories on the teaching server are AFS mounts of your UCSC home directory, so you should be able to access your files on any UCSC unix machine. To start the lab on a personal Linux machine, just copy the lab environment:

```
$ scp <cruzid>@grunhilda.soe.ucsc.edu:/var/classes/CMPS122/Winter19/Lab1.tar.gz .
```

Alternatively, if you have already set up a working directory on the teaching server, you can make a local copy like this:

```
$ scp -r <cruzid>@grunhilda.soe.ucsc.edu:./CMPS122/Lab1 .
```

If you have a Mac or a Windows machine, you should download Virtual Box `https://www.virtualbox.org/` and install a 64bit version of Ubuntu `https://www.ubuntu.com/` into it, then install the lab environment into the Ubuntu virtual appliance.

If you are running Windows 10, you can try and build the code inside an embedded Linux instance, but we can offer no support if you attempt this. `https://docs.microsoft.com/en-us/windows/wsl/about`.

If you do run on your own computer, you must remember to build your code on `grunhilda.soe.ucsc.edu` before submitting, to make sure everything works.