Crack

Next

The Shadow Knows

On most systems running Linux these days is a file called `/etc/shadow` that contains usernames and passwords. Fortunately, the passwords therein aren't stored "in the clear" but are instead encrypted using a "one-way hash function." When a user logs into these systems by typing a username and password, the latter is encrypted with the very same hash function, and the result is compared against the username's entry in `/etc/shadow`. If the two hashes match, the user is allowed in. If you've ever forgotten some password, you might have been told that tech support can't look up your password but can change it for you. Odds are that's because tech support can only see, if anything at all, a hash of your password, not your password itself. But they can create a new hash for you.

Even though passwords in `/etc/shadow` are hashed, the hash function is not always that strong. Quite often are adversaries, upon obtaining that file somehow, able to guess (and check) users' passwords or crack them using brute force (i.e., trying all possible passwords). Below is what `/etc/shadow` might look like, albeit simplified, wherein each line is formatted as `username:hash`.

```
brian:51.xJagtPnb6s
bjbrown:50GApilQSG3E2
emc:502sDZxA/ybHs
greg:50C6B0oz0HWzo
jana:50WUNAFdX/yjA
lloyd:50n0AAUD.pL8g
malan:50CcfIk1QrPr6
natmelo:50JIIyhDORqMU
rob:51v3Nh6ZWGHOQ
veronica:61v1CDwwP95bY
walker:508ny6Rw0aRio
zamyla:50cI2vYkF0YU2
```

Next

Safecracker

Your task is to design and implement a program, `crack`, that cracks passwords. We're not going to give too many hints on this one, but to get started you may want to read up on how the `crypt` function works on Unix/Linux systems, such as this lab environment. To do so, type:

```
man crypt
```

in the terminal. Take particular note of that program's mention of "salt".

In order to declare function `crypt` for use in your solution, you'll want to put

```
#include <crypt.h>
```

near the top of your file. Use `pseudocode.txt` as a notepad for ideas as to how you should organize your program!

Specification

- Your program should accept one and only one command-line argument: a hashed password.
- If your program is executed without any command-line arguments or with more than one command-line argument, your program should print an error (of your choice) and exit immediately, with `main` returning `1` (thereby signifying an error).

- Otherwise, your program must proceed to crack the given password, ideally as quickly as possible, ultimately printing the password in the clear followed by `\n`, nothing more, nothing less, with `main` returning `0`.
- Assume that each password has been hashed with C's DES-based (not MD5-based) `crypt` function.
- Assume that each password is no longer than five (5) characters. Gasp!
- Assume that each password is composed entirely of alphabetical characters (uppercase and/or lowercase).

Below, then, is some example behavior.

```
$ ./crack
Usage: ./crack hash
```

```
$ ./crack 50cI2vYkF0YU2
LOL
```

Hints

- Recall that `argc` and `argv` give us information about what was typed at the command line.
- Recall that a string is just an array of characters (`char`s).
- Recall that we can access individual elements of an array using square brackets (`[ ]`).
- Recall that the salt is the first two characters of the hash.
- Recall that sometimes, people use passwords that are actual words. Perhaps there's an optimization that can be employed?
- Brute force algorithms aren't the fastest, and that's okay! Recall that shorter passwords are usually easier to crack than longer ones.

Next

How to Submit

Execute the below, logging in with your GitHub username and password when prompted. For security, you'll see asterisks (`*`) instead of the actual characters in your password.

```
submit50 cs50/problems/2019/x/crack
```

You can then go to https://cs50.me/cs50x to view your current scores!