Problem Set 2: Crypto

This is CS50. Harvard University. Fall 2014.

Table of Contents

This is the Hacker Edition of Problem Set 2. It cannot be submitted for credit.

Objectives

- Become better acquainted with functions and libraries.
- Dabble in cryptanalysis.

Recommended Reading

- Pages 11 – 14 and 39 of http://www.howstuffworks.com/c.htm.
- Chapters 7, 8, and 10 of *Programming in C*.

diff pset2 hacker2

- Hacker Edition challenges you to handle extra whitespace in inputs.
- Hacker Edition challenges you to crack actual passwords.

Academic Honesty

This course’s philosophy on academic honesty is best stated as "be reasonable." The course recognizes that interactions with classmates and others can facilitate mastery of the course’s material. However, there remains a line between enlisting the help of another and submitting the work of another. This policy characterizes both sides of that line.

The essence of all work that you submit to this course must be your own. Collaboration on problem sets is not permitted except to the extent that you may ask classmates and others for help so long as that help does not reduce to another doing your work for you. Generally speaking, when asking for help, you may show your code to others, but you may not view theirs, so long as you and they respect this policy’s other constraints. Collaboration on the course’s final project is permitted to the extent prescribed by its specification.

Below are rules of thumb that (inexhaustively) characterize acts that the course considers reasonable and not reasonable. If in doubt as to whether some act is reasonable, do not commit it until you solicit and receive approval in writing from the course’s heads. Acts considered not reasonable by the course are handled harshly.

Reasonable

- Communicating with classmates about problem sets' problems in English (or some other spoken language).
- Discussing the course’s material with others in order to understand it better.
- Helping a classmate identify a bug in his or her code at Office Hours, elsewhere, or even online, as by viewing, compiling, or running his or her code, even on your own computer.
- Incorporating snippets of code that you find online or elsewhere into your own code, provided that those snippets are not themselves solutions to assigned problems and that you cite the snippets' origins.
- Reviewing past semesters' quizzes and solutions thereto.
- Sending or showing code that you’ve written to someone, possibly a classmate, so that he or she might help you identify and fix a bug.
- Sharing snippets of your own code online so that others might help you identify and fix a bug.

- Turning to the web or elsewhere for instruction beyond the course's own, for references, and for solutions to technical difficulties, but not for outright solutions to problem set's problems or your own final project.
- Whiteboarding solutions to problem sets with others using diagrams or pseudocode but not actual code.
- Working with (and even paying) a tutor to help you with the course, provided the tutor does not do your work for you.

Not Reasonable

- Accessing a solution in CS50 Vault to some problem prior to (re-)submitting your own.
- Asking a classmate to see his or her solution to a problem set's problem before (re-)submitting your own.
- Decompiling, deobfuscating, or disassembling the staff's solutions to problem sets.
- Failing to cite (as with comments) the origins of code or techniques that you discover outside of the course's own lessons and integrate into your own work, even while respecting this policy's other constraints.
- Giving or showing to a classmate a solution to a problem set's problem when it is he or she, and not you, who is struggling to solve it.
- Looking at another individual's work during a quiz.
- Paying or offering to pay an individual for work that you may submit as (part of) your own.
- Providing or making available solutions to problem sets to individuals who might take this course in the future.
- Searching for, soliciting, or viewing a quiz's questions or answers prior to taking the quiz.
- Searching for or soliciting outright solutions to problem sets online or elsewhere.
- Splitting a problem set's workload with another individual and combining your work.
- Submitting (after possibly modifying) the work of another individual beyond allowed snippets.

- Submitting the same or similar work to this course that you have submitted or will submit to another.
- Submitting work to this course that you intend to use outside of the course (e.g., for a job) without prior approval from the course's heads.
- Using resources during a quiz beyond those explicitly allowed in the quiz's instructions.
- Viewing another's solution to a problem set's problem and basing your own solution on it.

Getting Ready

First, curl up with these shorts on loops, functions, Caesar's cipher, and command-line arguments:

http://www.youtube.com/watch?v=HHmiHx7GGLE
Be sure you're reasonably comfortable answering the below when it comes time to submit this problem set's form!

- How does a while loop differ from a do-while loop? When is the latter particularly useful?
- What does `undeclared identifier` usually indicate if outputted by `make` (or, really, `clang`)?
- Why is Caesar's cipher not very secure?
- What's a function?
- Why bother writing functions when you can just copy and paste code as needed?

Next, take a self-paced tour through Week 2's examples, the source code for which can be found at http://cdn.cs50.net/2014/fall/lectures/2/m/src2m/ and http://cdn.cs50.net/2014/fall/lectures/2/w/src2w/, fast-forwarding through any programs with which you're already comfortable:

http://www.youtube.com/watch?v=1VbHJz2L6dM&list=PLhQjrBD2T380bVx_CQ2EXtlqZh0frwOsn
Finally, take a closer look at command-line arguments by way of these additional examples from Week 3, the source code for which can be found at http://cdn.cs50.net/2014/fall/lectures/3/m/src3m/:

http://www.youtube.com/watch?v=1VbHJz2L6dM

Getting Started

Alright, here we go again!

Open a terminal window if not open already (as via **Menu > Accessories > gedit** or via **Menu > Accessories > Terminal**). Then execute

```
update50
```

to make sure your appliance is up-to-date. In general, if you run into errors like "No such file or directory" with the staff's solutions, best to re-run `update50`, just to make sure you have the latest updates.

Next, execute

```
mkdir ~/Dropbox/hacker2
```

at your prompt in order to make a directory called `hacker2` in your `Dropbox` directory. Take care not to overlook the space between `mkdir` and `~/Dropbox/hacker2` or any other character for that matter! Keep in mind that `~` denotes your home directory, `~/Dropbox` denotes a directory called `Dropbox` therein, and `~/Dropbox/hacker2` denotes a directory called `hacker2` within `~/Dropbox`.

Now execute

```
cd ~/Dropbox/hacker2
```

to move yourself into (i.e., open) that directory. Your prompt should now resemble the below.

```
jharvard@appliance (~/Dropbox/hacker2):
```

If not, retrace your steps and see if you can determine where you went wrong. You can actually execute

```
history
```

at the prompt to see your last several commands in chronological order if you'd like to do some sleuthing. You can also scroll through the same one line at a time by hitting your keyboard's up and down arrows; hit Enter to re-execute any command that you'd like.

All of the work that you do for this problem set must ultimately reside in your `hacker2` directory for submission.

Initializing

Alright, let's get more comfortable with `string` (aka, `char *`, as we'll eventually see).

Write, in a file called `initials.c`, a program that prompts a user for their name (using `GetString` to obtain their name as a `string`) and then outputs their initials in uppercase with no spaces or periods, followed by a newline (`\n`) and nothing more. You may assume that the user's input will contain only letters (uppercase and/or lowercase) plus spaces. Folks like `Joseph Gordon-Levitt`, `Conan O'Brien`, and `David J. Malan` won't be using your program. But the user's input might be sloppy, in which case there might be one or more spaces at the start and/or end of the user's input or even multiple spaces in a row.

So that we can automate some tests of your code, your program must behave per the examples below. Assumed that the underlined text is what some user has typed.

```
jharvard@appliance (~/Dropbox/hacker2): ./initials
Zamyla Chan
ZC
jharvard@appliance (~/Dropbox/hacker2): ./initials
   robert   thomas bowden
RTB
```

If you'd like to check the correctness of your program with `check50`, you may execute the below.

```
check50 2014.fall.hacker2.initials initials.c
```

And if you'd like to play with the staff's own implementation of `initials` in the appliance, you may execute the below.

```
~cs50/hacker2/initials
```

Passwords *Et Cetera*

On most, if not all, systems running Linux or UNIX is a file called `/etc/passwd`. By design, this file is meant to contain usernames and passwords, along with other account-related details (e.g., paths to users' home directories and shells). Also by (poor) design, this file is typically world-readable. Thankfully, 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/passwd`. If the two ciphertexts match, the user is allowed in. If you've ever forgotten some password, you may have been told that "I can't look up your password, but I can change it for you." It could be that person doesn't know how. But, odds are they just can't if a one-way hash function's involved.

Even though passwords in `/etc/passwd` are encrypted, the crypto involved is not terribly strong. Quite often are adversaries, upon obtaining files like this one, able to guess (and check) users' passwords or crack them using brute force (i.e., trying all possible passwords). Only in recent years have (most) system administrators stopped storing passwords in `/etc/passwd`, instead using `/etc/shadow`, which is (supposed to be) readable only by `root`. (Take a look at `/etc/passwd` in the appliance, for instance; wherever you see `x` a password once was.) Below, though, are some `username:ciphertext` pairs[1] from an outdated (fake) system.

```
belindazeng:50q/EEJnOmtxc
caesar:50zPJlUFIYY0o
jharvard:50yoN9fp966dU
malan:50q.zrL5e0Sak
rob:HAYRs6vZAb4wo
zamyla:50NwUtF.OmQNY
```

[1] http://cdn.cs50.net/2013/fall/psets/2/hacker2/passwd

Crack these passwords, each of which has been encrypted with C's DES-based (not MD5-based) crypt function. Specifically, write, in `crack.c`, a program that accepts a single command-line argument: an encrypted password. (In case you test your code with other ciphertexts, know that command-line arguments with certain characters (e.g., `?`) must be enclosed in single or double quotes; those quotation marks will not end up in `argv` itself.) If your program is executed without any command-line arguments or with more than one command-line argument, your program should complain and exit immediately, with `main` returning any non-zero `int` (thereby signifying an error that our own tests can detect). Otherwise, your program must proceed to crack the given password, ideally as quickly as possible, ultimately printing to standard output the password in the clear followed by `\n`, nothing more, nothing less, with `main` returning `0`. The underlying design of this program is entirely up to you, but you must explain each and every one of your design decisions, including any implications for performance and accuracy, with profuse comments throughout your source code. Your program must be designed in such a way that it could crack all of the passwords above, even if said cracking might take quite a while. That is to say, it's okay if your code might take several minutes or days or longer to run. What we demand of you is correctness, not necessarily optimal performance. Your program should certainly work on inputs other than these as well; hard-coding into your program the solutions to the above is not acceptable.

Your program must behave per the below; underlined is some sample input.

```
jharvard@appliance (~/Dropbox/hacker2): ./crack 50yoN9fp966dU
crimson
```

Assume that users' passwords, as plaintext, are composed of printable ASCII characters[2] and are no longer than eight characters long. As for their ciphertexts, you'd best pull up the "man page" (i.e., manual) for `crypt` by executing

```
man crypt
```

in a terminal window so that you know how the function works. In particular, make sure you understand its use of a "salt." (According to the man page, a salt "is used to perturb

[2] http://en.wikipedia.org/wiki/ASCII#ASCII_printable_characters

the algorithm in one of 4096 different ways," but why might that be useful?) As implied by that man page, you'll likely want to put

```
#define _XOPEN_SOURCE
#include <unistd.h>
```

at the top of your file. Moreover, you'll want to link with `-lcrypt`, as by compiling not with `make` but with:

```
clang -o crack crack.c -lcrypt
```

You might also want to read up on C's support for file I/O, as there's quite a number of English words in `/usr/share/dict/words` in the appliance that might (or might not) save your program some time.

By design, `/etc/passwd` entrusts the security of passwords to an assumption: that adversaries lack the computational resources with which to crack those passwords. Once upon a time, that may have been true. Perhaps some still do. But when it comes to security, assumptions are dangerous. May that this problem set make that claim all the more real.

We should note that this problem set is no invitation to seek out other passwords to crack. Do not conflate these Hacker Editions with "black hat" editions. We hope, though, that by understanding better the design of today's systems, you might one day build better systems yourself. Besides acquainting you further with C, this problem set urges you to start questioning designs, as vulnerabilities (if not regrets) often result from poor ones.

If you'd like to play with the staff's own implementation of `crack`, well, sorry! :-) Where'd be the fun in that?