Problem Set 2: Crypto

out of 51 points

due by 7:00 P.M. on Friday, 12 October 2007

Be sure that your code is thoroughly commented
to such an extent that lines' functionality is apparent from comments alone.

Goals.

The goals of this problem set are to:

- Better acquaint you with functions and libraries.
- Allow you to dabble in cryptanalysis.
- Introduce you a bit early, perhaps, to file I/O.

Recommended Reading.

Per the syllabus, no books are required for this course. If you feel that you would benefit from some supplementary reading, though, below are some recommendations.

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

Academic Honesty.

All work that you do toward fulfillment of this course's expectations must be your own unless collaboration is explicitly allowed (*e.g.*, by some problem set or the final project). Viewing or copying another individual's work (even if left by a printer, stored in an executable directory, or accidentally shared in the course's virtual terminal room) or lifting material from a book, magazine, website, or other source—even in part—and presenting it as your own constitutes academic dishonesty, as does showing or giving your work, even in part, to another student.

Similarly is dual submission academic dishonesty: you may not submit the same or similar work to this class that you have submitted or will submit to another. Moreover, submission of any work that you intend to use outside of the course (*e.g.*, for a job) must be approved by the staff.

All forms of cheating will be dealt with harshly.

You are welcome to discuss the course's material with others in order to better understand it. You may even discuss problem sets with classmates, but you may not share code. In other words, you may communicate with classmates in English, but you may not communicate in, say, C. If in doubt as to the appropriateness of some discussion, contact the staff.

Getting Started.

0. SSH to `nice.fas.harvard.edu`, create a directory called `ps2` in your `~/cs50/` directory, and then navigate your way to that directory. All of the work that you do for this problem set must ultimately reside in that directory for submission.

1. (5 points.) If you haven't seen the iPhone in action, take a peek at one or more of the videos at `http://www.apple.com/iphone/ads/`. Better yet, find someone on campus with an iPhone and ask for a demo!

   The screen on the iPhone (and iPod Touch) responds to the touch of your finger. You can touch once to click, touch and drag to scroll, and even pinch to zoom. But, in engineering terms, how does it work? In a paragraph of your own words in a file called `iphone.txt`, explain as technically as you can (*i.e.*, you should understand your own explanation) how the iPhone's "multi-touch" screen works.[1] Realize that it's a bit different from, say, your local ATM's touchscreen!

[1] For this question, you're welcome to consult *How Computers Work*, Google, Wikipedia, a friend, or anyone else, so long as your words are ultimately your own!

Passwords et cetera.

2. (45 points.) 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.[2]

   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`.[3] Below, though, are some `username:ciphertext` pairs from some outdated (fake) systems.

```
mscott:50vk7WHrTHP4U
dhelmet:50Bpa7n/23iug
homer:50CnQM/BvbIsQ
sjobs:50O9Y6r9/Q816
billg:50Pm0mjT32q02
malan:HAvc4tY/3g/HQ
```

   Crack these passwords, each of which has been encrypted with C's DES-based `crypt` function. Specifically, write, in `crack.c`, a program that accepts a single command-line argument: an encrypted password.[4] 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. Otherwise, your program must proceed to crack the given password, ideally as quickly as possible, ultimately printing to `stdout` the password in the clear. 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 six 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.

[2] If you like this stuff, consider taking Computer Science 120 or 220r.
[3] Take a look at `/etc/passwd` on `nice.fas.harvard.edu`, for instance; wherever you see `'x'` a password once was.
[4] 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.

Assume that users' passwords, as plaintext, are no longer than eight characters long. As for their ciphertexts, you'd best pull up the man page for `crypt`, 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 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 and compile your program with `-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/american-english` on `nice.fas.harvard.edu` 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.[5] 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.

Submitting Your Work.

3. (1 point.) Okay, the free point is back.

4. Ensure that your work is in `~/cs50/ps2/`. Submit your work by executing the command below.

   ```
   cs50submit ps2 hacker
   ```

   Thereafter, follow any on-screen instructions until you receive visual confirmation of your work's successful submission. You will also receive a "receipt" via email to your FAS account, which you should retain until term's end. You may re-submit as many times as you'd like; each resubmission will overwrite any previous submission. But take care not to re-submit after the problem set's deadline, as only your latest submission's timestamp is retained.

[5] In fact, do bear in mind the policies at `http://www.fas.harvard.edu/computing/rules/`.