COMP30023 Project 2 – Password cracker

Worth 15% of the final mark

Due: 24 May, 2019.

1 Background

A security system is only as strong as the passwords that unlock it.

In this project, you will try to crack the passwords of a simple system that has **four- and six-character passwords**. The passwords can contain any ASCII character from 32 (space) to 126 (~), but since the passwords are chosen by people, they are not arbitrary strings. You will *use your knowledge of common human frailties to identify the passwords used* by a particular user.

This system is particularly insecure because it will accept any of the thirty previous passwords of the user. Your goal is to write code to guess as many of these as possible.

Submission will be through git, like the first project.

2 Project specification

For each student, 30 passwords have been generated and SHA256 hashes of these have been computed. Passwords 1 to 10 are **all four characters**. Use `scp` to download the file `pwd4sha256` from 172.26.37.44. This file contains 320 bytes. The first 32 bytes are the SHA256 hash of password 1, the second 32 bytes are the hash of password 2 etc.. Hashes 11 to 30 are for **six character** passwords, and can be obtained using an extension described below.

You are to write a C program to guess the passwords. That is, find the four- or six-character sequences whose SHA256 hash is one of the hashes you have downloaded.

You should create a Makefile that produces an executable named 'crack'.

- If `crack` is run with no arguments, it should generate guesses, and test them against the SHA256 hashes. When it finds one, it should print one line, consisting of the plaintext password, a space, and an integer 1-30 indicating which hash it matches. For example

  abcd 3
  Oops 5
  adam 1
  passwd 15

  This is probably the mode you will use for finding passwords. The other two modes are only for the assessment.

- If `crack` is run with one integer argument, the argument specifies how many guesses it should produce. In this mode, password guesses should be printed to `stdout`, separated by newline characters ('\n').

- If `crack` is run with two argument, it should treat the first as the filename of a list of passwords (one per line), and the second as the filename of a list of SHA256 hashes (in groups of 32 bytes, with no newline characters). It should then test each of the passwords given in the first file against each of the hashes given in the second file, and produce output as for the case of no arguments. It should print nothing else to `stdout`. If you want to produce other output, send it to `stderr`.

This option does not need to generate any guesses.

A library for SHA256 is provided at
[https://gitlab.eng.unimelb.edu.au/junhaog/comp30023-labcode-current] in directory proj-2.

Write your code so that it is easy to change the number of hash values available to you. For example, just read the whole file, and count how many 32-byte blocks it contains.

2.1 Suggestions for guesses

Part of the challenge in this project is teaching yourself about the types of weak password people choose. To help you, a list of around 10,000 common passwords is available in
[https://gitlab.eng.unimelb.edu.au/junhaog/comp30023-labcode-current] in directory proj-2.

Note: These are not generally four-letter passwords. What is the best way to get a list of candidate four-letter passwords?

You may also find the site `https://www.thefreedictionary.com/4-letter-words.htm` and the cartoon in Fig. 1 helpful.

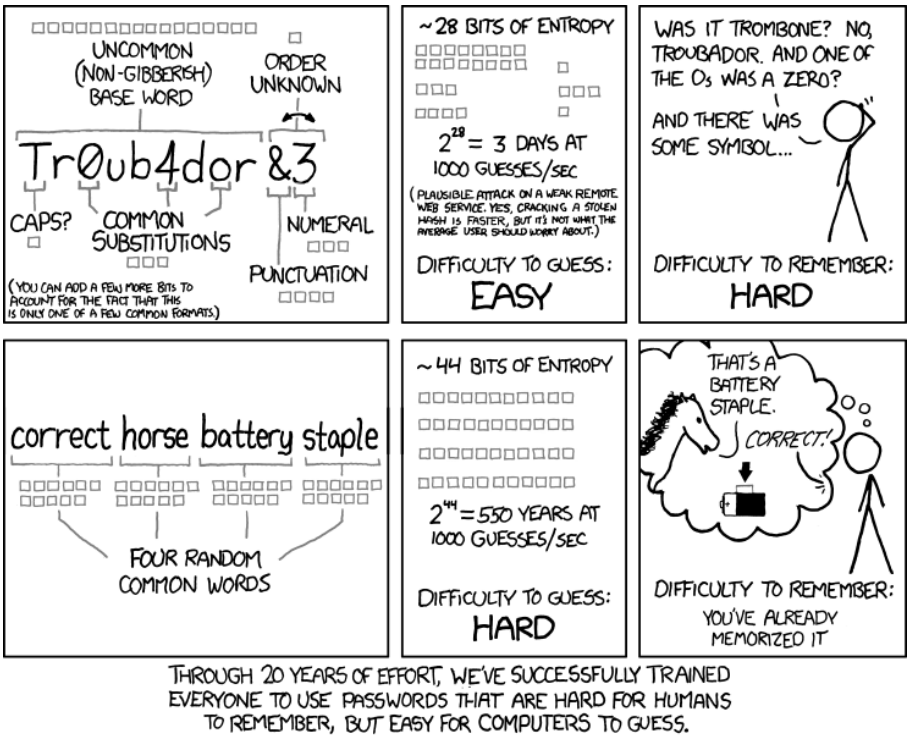

Figure 1: Anatomy of a weak password

2.2 Unlocking the additional hashes

At 0:01am on 17 May, a file `pwd6sha256` will become available for download by `scp` from the server. The first 32 bytes correspond to password 11, the second 32 bytes to password 12 etc..

You can (and are encouraged to) make these available earlier by performing a challenge-response protocol based on Diffie-Hellman key exchange with the server. Recall from lectures that the Diffie-Hellman Key Exchange protocol enables two parties to produce a shared secret in an open communication channel that is observable by an adversary. The process starts with with public parameters (g, p). Both parties choose secrets, a and b, then compute and exchanging the values $g^a \text{mod} p$ and $g^b \text{mod} p$. The computed shared secret key between the two parties is $g^{ab} \text{mod} p$.

Write a program called `DiffieHellman.c` (No. Please call it `dh.c`. I apologize for the error.) to run the key exchange protocol below. Upload this to the server, for example using

`scp` *your-local-filename.c* `172.26.37.44:dh.c`

Note: This assumes that your ssh private key is in the usual location, `~/.ssh/id_rsa`. If it is not, you will need to use

`scp -i` *path-to-your-private-key* *your-local-filename.c* `172.26.37.44:dh.c`

To verify this, create the SHA256 hash of this C source using

`openssl sha256` *dh.c*

Use the first byte of this as your Diffie-Hellman secret b. That is, convert the first two hexadecimal digits to an integer, and use that for b.

The exchange requires you to create a socket connecting to port 7800 on the server *after* uploading your source. The Diffie-Hellman component uses $g = 15$ and $p = 97$. First, send your username, followed by a new line ('\n'). Send the server $g^b (\text{mod} p)$ as text (for example, 12 would be sent as '1', '2', rather than a byte whose value is 12), followed by a new line. *Do not use a random b like with proper Diffie-Hellman; use the first byte of the SHA256 of your C source. This will let us check that your code is working correctly.* The server will send you a line in the same format. Compute what is normally the shared secret, and then send it to the server (via the socket) as text. The server will send a line of text saying whether or not you succeeded. You may want to display that the the screen. This is illustrated in Figure 2

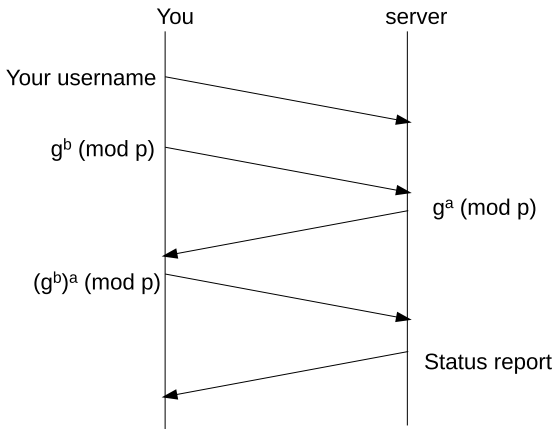

Figure 2: Protocol message sequence

This will create `pwd6sha256` for download.

Using 32-bit or 64-bit arithmetic will result in overflow. Instead, use the method hinted at in the Challenge Questions of the tutorial in Week 8.

Your C code, named `dh.c`, should be in your git repository and LMS submission. Includle any other files that may be necessary for `dh.c` to compile and run.

Note that this `dh.c` does not need to be as "polished" as the source code of `crack`. Just write something that works, and that you will still be able to understand in a month's time. It will only be marked for style in borderline cases. It will be tested for correctness.

3 Marking

The marks are broken down as follows:

Marks	Task
3	Correctly implementing the two-argument form of `crack`
2	"Good" guesses
1	Code quality
1	Build quality
3	Completing the Diffie-Hellman challenge-response
0.2 each	Discovering 4-digit passwords
0.3 each	Discovering up to 10 6-digit passwords

3.1 Code quality

Factors considered in code quality include: Choice of variable names, comments, indentation, remaining debugging code (it is OK to have a small amount if, for example, it is disabled by an #ifdef DEBUG), useful output indicating progress.

3.2 Build quality

Factors considered in build quality include: Producing an executable with the correct name, using '-Wall' but yielding no warnings

3.3 Correctly implementing the two-argument form of `crack`

When given a list of guesses and a list of hashes, the program should correctly identify the hash (if any) that matches the guess. The order of the output should equal the order of the guesses in the input file.

Your code should not crash under any circumstances, but only needs to work correctly for cases where the maximum password length is less than, say, 10,000 characters and the complete file of hashes fits in memory.

3.4 "Good" guesses

Some passwords are more likely than others. Marks will be allocated for code that starts by guessing likely passwords. This will be assessed in part by inspection of the code, and in part by the behaviour of `crack` when run with a single argument. The behaviour marks are awarded for the degree to which the first few passwords generated (say the first 100 (in practice, we will ask it to generate more than 10000, to allow a reliable statistical sample to be taken)) are statistically representative of actual passwords. For the "code inspection" marks, *please include a comment of one or two lines explaining how you generate passwords, and why it is "good".*

It is sufficient to produce only 6-character passwords. The evaluation of how good a password is will consider several factors. One is how well the frequency of characters matches that of the passwords the class has been given and how well it matches the distribution of characters in the set of passwords allocated to the class. (For the curious, we will base the mark on the minimum of the Kullback-Leibler divergence between the distribution of characters you generate and the two distributions above. That needn't bother your. Just try to make the passwords similar to the examples you have.) Another is how often the same character appears multiple times in a password.

The command line argument may be large. You have been given a list of 10,000 common passwords, and the test will consider what you do when these run out. **Your code should eventually guess every valid password, just guess likely ones first.**

3.5 Completing the Diffie-Hellman challenge-response

One mark is awarded for correctly sending $g^b \pmod p$ and the other two for correctly sending $(g^b)^a \pmod p$. These marks are awarded for doing this any time before the due date. Even if the

6-letter password hashes have been made available, completing this task will attract these marks. It is required that `dh.c` perform all of these calculations, but it may do that by calling `dc` using `pipe()` and `fork()`.

Clarification: The original spec said to write C code, upload it and run it. Some people uploaded something that was different from what they ran. In order for us to verify that you did what the original spec said, we need to run your code. Please submit `dh.c` and any files it needs to run as part of your LMS submission and Git submission. We will test your code by compiling it with `make dh` and then running

```
echo b-in-decimal | ./dh b-in-decimal
```

You do not need to use either the command line argument or stdin, but you are welcome to use either or both. **If this fails**, we will run `make run-dh` which should execute `dh` in the way you require. If you need `dh` to be run a special way then include a suitable rule in your Makefile. It should *not* create a file `run-dh`. If you do not want to learn how to write makefiles, then make your code run with the original command above.

4 Discovering passwords

Place the passwords and hash numbers in a file `found_pwds.txt` in the top level directory of your git repository, and the zip file you submit. The format should be the same as the output of `crack` namely PASSWORD space HASH_ID newline.

5 Collaboration

You can discuss this project abstractly with your classmates, such as password generation strategies, but should not share code. If possible, do not even look at anyone else's code. If you really want to help somone debug, don't look at the screen and pretend you're doing it over the phone. (Yes, seriously; it's slow but you'll both learn from it.)

5.1 Real programmers use Stack-Exchang

Code will be run through software to detect copied code. You are welcome to use code fragments from Stack Exchange, or any other source of information on the web, but be sure to quote the URL whose code you used, so that it doesn't appear that you copied a classmate.

Also note that any code you find on a site other than a question-and-answer site like Stack Exchange may be copyright. Please only use it if there is an explicit licence allowing it, such as Creative Commons.

5.2 Submission

The due date is at 11:59pm on Friday 24 May 2019. It is recommended that you submit before 5pm, in case you need assistance on the forum.

You must submit, to *both* **GitLab** and **LMS**, your source code (with a makefile) in a .zip file with a filename in the the format of <*your_username*>_comp30023_2019_project-2, e.g., llandrew_comp30023_2019_project-2. Any failure to follow the filename format will result in a **2-mark deduction**. Any project not submitted to both GitLab and LMS will receive a failing mark.

5.2.1 Special consideration

To receive special consideration, you **must** apply at least three days before the deadline. If you have a chronic condition and aren't sure if it will be a problem, tell us in advance. If you get sick a week before the deadline and aren't sure how it will affect you, tell us.

You can submit multiple times. Make sure that your first submission is at least three days before the deadline so that, even if something goes wrong, you have submitted a nearly-finished project. (Do not leave this project until last minute.)

5.3 Notes

1. To make testing easy, one password in `pwd4sha256.txt` is the first four letters of your username, and one in `pwd6sha256.txt` is the first six letters. If your username is less than four or six letters, it is padded with trailing spaces before taking the hash.

2. Brute force search (i.e., trying every combination of characters) is sufficient to find the four-character passwords, if your code is efficient and if you don't leave it to the last minute. One possible approach is to write a brute-force search first,and while that is running, develop the code to download the remaining hashes, and determine efficient ways generate candidate passwords.

3. Start early. Because searching for 6-digit passwords is slow, the sooner you complete your code, the more passwords you will discover.

4. Your code should not crash, regardless of what input it receives. If your code is given invalid input, then it is acceptable for it not to work properly, but it is not acceptable to crash or suffer buffer overflow.

6 Questions for reflection

These do no need to be submitted, but will help you understand the subject. You are welcome to discuss them.

1. Not all brute-force searches are equal. How can you make brute force find passwords earlier?

2. You can write "support programs" in other languages. For example, what is the most frequently used character in the sample passwords? Write a program in your favourite language to count them, and then use that information in your C program.

3. You have a choice between making guesses randomly or systematically (such as guessing eeeee, eeeeet, eeeeea, eeeeeo, eeeeeei). What are the strengths and weaknesses of each approach? Can you combine the strengths of each?

4. You may want to stop the program and restart it from where it left off, rather than repeating the initial sequence each time. How can you do this, without using one or two command line arguments?

5. What will happen if you create a SHA256 hash of your code and then modify it to hard-code the first byte as the value of b? How can you avoid that problem?