CS334: Operating Systems
Spring 2022. Assignment 2
Due: Sunday, February 13th 2022 NLT 11:59 PM
Individual, 100 pts possible

Educational Objectives:

- Illustrate the performance differences between using different library calls.
- Practice using the kernel level library calls
- Practice using the library level calls.
- Illustrate how hashing and security works.

Logistics:
The is your first individual programming assignment. It has to compile and run on UP's Linux VDI.

Description:

Every modern, multi-user system uses some form of user authentication using a user name and a password. A user types in a password which is translated to a hash value. The remote machine stores the user names and a password for each user. The password stored on the remote machine is also stored as a hash of the original plain text password that was done once, when the user account was created (or password updates). When a user attempts to gain access to the system, the authentication attempt password is translated to a hash, the hash value is transmitted to the remote system where it is compared against the hash values stored on the remote system for the given user. Modern operating systems came a long way and the authentication schema have evolved from storing plain text passwords, to hashed passwords, to encrypted passwords, to sophisticated protocols such as Karbeross.

We will use one, not so strong, version of generating, storing and retrieving hashed passwords. When user creates a password as plain text, the plain text (a string of characters) is processed through a special hashing function that is stored on a host system. This translation is easy, fast, and unique when translating a plain text to a hash value. If you try to translate the hash back to the plain text, this is extremely hard. This is why security works, in other words, if you had the hash value, reversing it back to the user's plain text is computationally impractical. There are several hashing algorithms – SHA, SHA1, SHA3, MD5… (https://en.wikipedia.org/wiki/SHA-1) and they all have different advantages and vulnerabilities.

As SHA1 is not particularly strong, security firmes published a list of reverse engineered ("pwned") passwords from hacked systems, where if you have a hash it is very simple to get the original password as plain text. The cracked hashes from all compromised servers are published annually as one very large plain text file. If you were a system adminitrator who uses SHA1 hashing, this file is super useful, because when uses changes a password, the candidate password's hash can be checked if it exists in the cracked file and you (the sys admin) will know that the user wants to use a password that is weak and can be cracked easily. The only problem is that the file of hashes for cracked passwords is 25GB big and if you are running for example an e-commerce server farm managing thousands of users you have to have very fast way of checking this hash file.

The assignment:

In this assignment we will test two different ways of reading a file to see which way is faster. Your job is to write a C program that will (1) take a plain text password as a parameter, (2) generates a SHA1 hash of the password, and (3) preforms lookup for the hash key in the file of cracked hashed. You make two version of this program. One using the user library primitives you know how to use: fopen, fscanf, fprintf, strcmp etc, and the other one with the kernel level libraries with parallel functionality: open, fread, read, write, memcmp etc.

Remember, hashing a password is trivial and fast so that will be the same, but the fast look-up is not and that's where two versions will differ. Functional requirements:

- Version 1: Create file text2shaLookup_v1.c that will use user level file operations primitives including: fopen, fscanf, fprintf, strlen, EOF, strcmp, strncmp... to read, lookup, and compare the hashed string passed in from CLI against the password hashes read from stored entries in the file. Read one line from the opened file and compare the two hash strings, and repeat until you check all file entries or you find a match
- Version 2: Create file ext2shaLookup_v2.c will be identical, but the read, lookup, compare will use byte based operations and memory compares. This time, you will read 41bytes (which is 40bytes hash + 1 character that is '\n'. The fscanf in the version 1 of your solution will consume the \n and that's how fscanf knows that it hit the end of the line. **fread(2)** does not do that as it reads raw un-buffered input and it will read exactly the number of bytes you want it to read. Be careful and troubleshoot your program behavior on the lookup of a known hash in a file (Hint: make sure when you are comparing the hashes that you only compare the hash itself and no 'additional' characters.
- Compare the run-time of these two versions to see if there is an advantage of using one set of calls over the other.
- Both programs will print if the password's hash was found: "Found" and "Not Found". If the hash was found, print the line number where it is in the pwned file (see sample output below)
- Sanity check: use https://haveibeenpwned.com/Passwords to see if a password exists in a file and http://www.sha1-online.com/ to check if your sha1 hashing works correctly. This will help you to check manually if a password has been hashed before and how many times this password has been seen. For example, password: "password" has been hashed 9,545,824 times and its SHA1 hash is "5baa61e4c9b93f3f0682250b6cf8331b7ee68fd8**" (hmmm, this is 300 time increase since LAST YEAR, this means there is 6milion more servers out there that were compromised because someone is still using password 'password')**

Implementation notes:

- Download the hash file from: https://upedu-my.sharepoint.com/:u:/g/personal/cenek_up_edu/ER7g5argi75KuyjMestK_scBNs90swSRrysFgncLq62T8A?e=JvOL9V
- The downloaded file (after you unpack it) is a ASCII written file that with the following structure: <40bytes ASCII hash>\n
- Be careful as SHA algorithm will return hex hash in all lower case, but the file has all entries in the upper case. I suggest converting the password's hash key into all upper case before you start reading hashes from file and running the comparison between the password's hash and hash from the file. Hints: use toupper function for this string conversion, sprintf might be useful as well
- Rather than using the entire 20GB file, I suggest extracting the first 200 lines or so to build and error check the program on before you do run-time bench-marking on the full file.
- Don't overthink the assignment, my solution is cca 40 lines each (and 30 of them are the same).
- Do NOT re-implement the SHA1 algorithm. SHA1 comes packaged with your Ubuntu operating system. All you need to do in your C code is to include openssl/sha.h. You have to include the compiled binary at the compilation time (read below).
    - There is an issue setting a SHA1 result back and actually using the result as a text.
    - SHA1 returns a SHA_DIGEST_LENGTH array of unsigned chars, you will have to convert the returned array of chars into a two times longer array of characters (SHA_DIGEST_LENGTH*2) as each unsigned char in the original result encodes two hex values (recall, a hex value is 4 bites and since unsigned char is 8bits, which accounts for 2 hex values).

- To convert/sprintf the unsigned char to two hex characters, use sprintf with '%02x' formatted string.

- When compiling with external C libraries, use the following CLI:
  - gcc -o <application name> <source code file name> -l<library name>
  - In our case: gcc -o <application name> <source code file name> -lcrypto
- List the content of openssl/sha.h to find out the parameters of for SHA1 function.
- To time your execution, this time from inside of your code:
  - the first lines of your program should declare two variables of type clock_t to record start and end time.
  - Before you go into the while loop that reads the file, store clock() time into the start variable
  - After the loop concludes (before printing output), store clock() time into the end variable
  - Report the total elapsed time as the difference between end time and the start time. If you divide the resulting tick count by the CLOCKS_PER_SEC constant, you will get the runtime in seconds(see the sample output below)
- If you don't have a Linux installed on your home computer, I suggest using the lab SH304 systems that you just installed, as I don't think your UP account disk quota allows you to host a 25G file - I might be wrong. You still can develop your code base on VM, just use the first couple of pages from the pwned file to test and troubleshoot your program.
- Benchmark your code (for the write-up) on the entire 25GB file
- The command line arguments for your programs will take the file name and the plain text password (no spaces allowed).
  Example: ./ext2shaLookup_v1<input hash file> <password string>

- **Sample output:**

```
$./ext2shaLookup_v2 pwned_hashes.txt default
Searching for password "default":
Searching in file "pwned_hashes.txt"
sha hash: 7505D64A54E061B7ACD54CCD58B49DC43500B635
Found, line: 7844
Time searching: 0.000960s
```

Program embellishment (additional implementation, not extra credit):

Create a version 3 that will alter your solution that uses the kernel level calls (v2). Instead of reading 40 bytes at a time, let's see if we can save some additional time on the I/O operations. Instead, read one page (4K bytes) into a buffer and your hash checked will go through the 4k buffer to check each 40 bytes. This way instead of doing filesize/40bytes number of file I/O, we will only do filesize/4K number of file I/O operations and the rest is all done in memory which is faster. Please include v3 and let us know that you completed this additional requirement (question 1 in write up).

What to turn in

An archive (.zip) with:

1. The source code files
2. makefile to compile the code and produce two targets: text2shaLookup_v1 and text2shaLookup_v2
3. The write-up
4. A readme file if there are any specific instructions I need to know when compiling and running the file. I will not read the file readme, unless the code does not compile or run.

5. DO NOT INCLUDE THE FILE CONTAINING THE HASHED PASSWORDS or the binaries!

Write up

1. Did you complete all required section or functionality of this assignment. If not state what is successfully implemented and what is not.
2. Write a simple pseudo-code that explains the design of your solution.
3. Profile the your solution and fill out the following table. Profile your code execution on the entire 25GB fine, not the small tester file with 10k hashes I supplied with the HW. Please use the physical machines in SH304, so the computational resources do not vary based on the load:

Password	Found? If Y: line number in file	Time ext2shaLookup_v1 (seconds)	Time ext2shaLookup_v2 (seconds)
"password"			
"administrator"			
"chubaka"			
Try your current password			
"PassW0sD"			
Non-pwned password			

4. Describe at least one additional design consideration you would use to decrease the search speed even more (excluding suggestion stated in the program embellishment section)?

5. What differentiates fscanf from read, strcmp from memcmp and what makes some commands faster than others?

After completing this assignment, you should be able to answer the following questions:

1. Why is storing hashed passwords on a host machine so much better than storing plain text?
2. In what situations would you use low or high level calls to read information from files?
3. Can you use read to read file input that is not fixed length? If you did what would be the additional considerations?
4. Why is using a password with a known hash undesirable?
5. Why does SHA1 always produce a hash of the same length regardless what the plain text input is?
6. How to use clock() to profile the program's execution times