Assignment 4:
Cryptographic Hashing

CSCI 360

April 3, 2017

In this assignment we will build a program that simulates the way password storage works on a Unix system. Please read the article by Morris and Thompson on password security included in this assignment folder.

0.1 Storing Passwords

Some say that an axiom of computer security should be

Assume that you are already hacked and try to mitigate the damage.

How could we store peoples' passwords so that, even if our database is hacked, we are most able to mitigate damage? Applied to password storage, one implication of the axiom is that you don't want to store peoples' passwords in "plaintext." This means that you don't want to have a big file with all the user names listed next to the corresponding passwords. The reason is that someone could steal this file and compromise every user on the system.

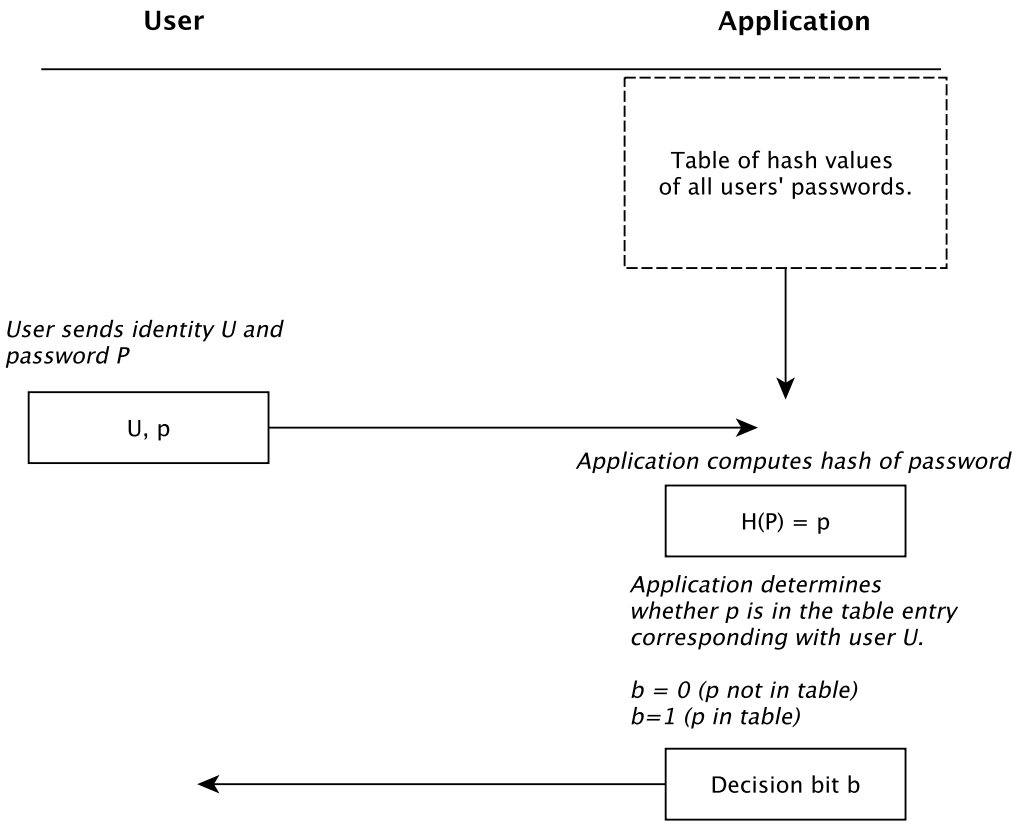
One solution is to encrypt the passwords before you store them. The problem with this is that if someone ever gets the encryption key then all passwords are compromised. It would be better if passwords could only be compromised "one at a time".

A better idea is to *hash* the passwords before storing them, so that if someone steals the password file they only get a bunch of gibberish. When the user enters his or her password, it can be hashed and compared to the entry in the file. There is no reason to store the password itself.

Remember that a *hash function* (in cryptography) is a function which is very difficult or impossible to invert. The input is frequently a string (or a file) and the output is usually a long number (say 256 bits). The two most important features of a cryptographic hash function h are

1. Given $h(M)$ for a message M, it is practically impossible to find some M' (not necessarily the same M) such that $h(M') = h(M)$.
2. (Corollary to the above) From $h(M)$ it is practically impossible to recover M. [1]

[1] The Birthday Paradox says that pairs M and M' such that $h(M) = h(M')$ can be found more easily than one might think, for combinatorial reasons that have nothing to do with the strength or weakness of h.



One example of a cryptographic hash function seen in class is `SHA-512`. The existence of cryptographic hashes allows for a new approach to storing passwords: Store the hashes instead. Thus the password file would be a long list of users next to big hex numbers $h(P)$ which are the hashes for the true passwords. Verifying a user password is then a matter of hashing what the user entered, and making sure it matches the value in the file.

There are still problems with this idea, mostly revolving around an attacker precomputing a bunch of hashes. An attacker can put together a "dictionary" which is a long list of plausible passwords

```
challa_back
D0nk3yK0nge
sickAshtray911
blink182
...
```

Assuming the attacker then gets access to the password file, she can then hash all the words in the dictionary, match these hashes against the hashes in the password file, and find out tons of passwords by noting the matches. This is called a **dictionary attack**.

A widely adopted solution to this problem is to use **salted passwords**. In a salted password scheme, when a user first selects his or her password a random sequence of bits (called the salt) is selected at the same time. The system then prepends the salt to the password, hashes both, and stores the hash (as well as the salt) in a file. On most Linux systems this file is `/etc/shadow` which you can look at if you have `sudo` permissions (don't modify it though).

Salting doesn't protect passwords from being attacked individually. The salt is public knowledge, so the attacker can just prepend it to her dictionary words and then carry out a dictionary attack as usual. However it does force the attacker to deal with each password individually instead of trying to crack them all in bulk. This is a big difference on a system with thousands of users.

In this lab we'll be working with `SHA512`, which you can use either from the command line or from a library (C++) or the `hashlib` module in Python.

1 Challenge 1 (password setter)

In this exercise you should write some code that does the following:

1. Generate some random salt. (Advice on how to do this below.)
2. Prompt the user for a password and read in the response.
3. Trim the newline character (if there is one) from what the user entered.
4. Store the salt and `SHA-512` applied to the salt and password in a file.

Here is a sample run of a possible solution to this challenge:

```
your salt is 4cffffff
Please enter a password
swordfish
your salt and password is 4cffffffswordfish
your SHA-512 digest of salt and password is

9719a6439375c9115e01dceda86e210e5f2d78a6cf3f4872997746832c4c0f58c5
ae0923fabe5acfb923dfc94a117a7d444e453622912dfa193fc6636581f159

Writing output to file: password.txt
```

The file `password.txt` should contain the user's salt and her `SHA-512` password – **not** the actual password!

Code for Python and C implementations for generating random salt are included in the project folder.

Remember that you will need to write `import hashlib` at the beginning of your file in order to import the hash library (python).

2 Challenge 2 (password verifier)

In this exercise we will use the output from the previous exercise to verify a user's password. You should write a program which does the following:

1. Prompt the user for a password and store the response.
2. Retrieve the salt and the digest from the file you created in the previous exercise.
3. Prepend the salt to the entered password and make a new MD5 digest.
4. Compare the result to the digest in the file and make sure there is a match. If so, accept the password. If not, reject it.

A sample exchange for the resulting program is like this:

```
Please enter a password
pikachu
your retrieved salt is 4cffffff
your retrieved hash is:

f22024007725185ea59c539ae8f99717f0a86e6d0d18ef1bc6bb483392099f1aa7
99bc0d7fd157bc00f21f15c2bca67423582d0f60aed640d5ecce2ee3c7893b

your salt and password is 4cffffffpikachu
your md5 digest of salt and password is

9719a6439375c9115e01dceda86e210e5f2d78a6cf3f4872997746832c4c0f58c5
ae0923fabe5acfb923dfc94a117a7d444e453622912dfa193fc6636581f159

ACCESS DENIED
```

Another run:

```
Please enter a password
swordfish
your retrieved salt is 4cffffff
your retrieved hash is:
9719a6439375c9115e01dceda86e210e5f2d78a6cf3f4872997746832c4c0f58c5
ae0923fabe5acfb923dfc94a117a7d444e453622912dfa193fc6636581f159

your salt and password is 4cffffffswordfish
your md5 digest of salt and password is

9719a6439375c9115e01dceda86e210e5f2d78a6cf3f4872997746832c4c0f58c5
ae0923fabe5acfb923dfc94a117a7d444e453622912dfa193fc6636581f159

ACCESS GRANTED
```