420-LCW-MS Programming Techniques and Applications - Lab Exercise 7

May 11, 2022

Goals for this lab:

- See a few basic hashing functions.
- Understand password hashing and salting.
- Create a hashable Python class that can be used as a key in a dictionary.

This lab assignment is just an exercise - it will not be graded. I do **not** expect you to finish all of the exercises, but do think about them and see if you can imagine how a solution might work.

You do not need to hand in this lab, it is just for your benefit. Material from labs may appear on later exams or assignments.

Introduction

Hash functions come in a number of varieties to serve the different application areas in which they are used.

The file `hashfunc.py` contains a few standard "hashing" functions. Most are relatively simple and fast to compute, and so are intended for use with the symbol table application of hashing, although the `crc32()` algorithm is intended for data integrity checking. None of these are what are considered "cryptographic" hashing functions, which are typically much more complicated and take much longer to compute.

Because cryptographic hashing functions are so large and complex, we won't study the details of their implementation. Thankfully, Python includes support for several of the most useful of these functions. One commonly used cryptographic hash is SHA256, "secure hashing algorithm, 256 bits". SHA256 reads an input of any length and produces a 256-bit hash that is extremely hard to reverse engineer. SHA256 is used for various data security tasks, such as digitally signing documents.

Your tasks

1. If a system needs to implement password security, it is tempting to implement this by just saving the user's passwords as a plain text file somewhere on the system, and just make it hard for malicious users to read that file. This is how passwords were implemented in the UNIX operating system back in the dark ages.

   The problem with this approach is that if a wily and malicious hacker gains access to the password file, they immediately learn the plain-text passwords of every user. Since this is so obviously bad, systems were developed that hashed the passwords, so that even if someone read the password file, they would not be able to learn the passwords of the system's users.

   Even this became problematic as computers sped up and it became practical to discover hashed passwords by simply computing all of the hashes for common English words and names, and then simply searching for the hashes in the password file. Versions of this approach are sometimes called *rainbow table* attacks.

   The solution to this issue was *salting* a password. By adding a few bits of random data to the password before hashing, it becomes much more costly to check the passwords against a table of known hashes.

   In this exercise, you'll see how this works.

   The file `passwords.txt` contains 20 SHA256 hash values for 20 passwords. Each of these passwords is an uncommon word taken from the word list file `dictionary-yawl.txt`. The first ten passwords are stored "unsalted", whereas the second ten are salted. For the second 10, the salt values are give after the hash value.

   Python 3 includes an implementation of SHA256. You will use it to decrypt the passwords. See the file `decrypt.py` for the key details, but the relevant code is the class `sha256` located in the module `hashlib`. The `sha256` class supports two relevant methods:

- `update(bytes)` - adds the `bytes` object to the SHA256 computation.
- `hexdigest()` - returns a string that represents the value of the digest based on the data passed to `update()` so far.

Using the code in `decrypt.py` as a model, see if you can find the words used to generate the 20 passwords. Notice how much more computation is required to find the plaintext passwords for the salted passwords.

In practice, this is *not* a good method for storing passwords. Other, well-tested methods are available - search for PBKDF2 and/or Argon2 if you are curious.

2. Python dictionaries rely on hashing to allow rapid retrieval of entries in the dictionary by key. To act as a key, a class should:

   (a) Be immutable.

   (b) Implement a method `__hash__(self)` that returns a hash value for this object.

   (c) Implement a method `__eq__(self, other)` that returns True if the two objects are equal.

   Note `obj1 == obj2` implies that `hash(obj1) == hash(obj2)` (is the reverse necessarily true?).

   Create a class that represents an immutable *rectangle* consisting of a top y, left x, bottom y and right x coordinates, and implement the methods needed to allow the class to be used as a key in a Python dictionary. Use the hash functions in `hashfunc.py` for inspiration.

   One note about performance - because your class is immutable, consider computing the hash value once when the object is created. Then your `__hash__(self)` method will just return the precomputed value.

3. Once you've written your class, write some code to test out this class. Create a dictionary and add a bunch of items using your class as a key. Verify that it works correctly.

4. Evaluate your hash calculation. Generate a bunch of objects at random, and see how many different values yield the same hash value.