**(ADV)Problem Set #1 Password Based Key Derivation Functions**

Best practice of password storage is to store only the hash of the function and not the plaintext password itself.

In practice, it has been shown that even very good cryptographic hash algorithms are subject to attack methods and can be compromised reasonably quickly, especially as computation speeds increase (Moore Law).

This problem set is to understand the most common attack and then the standard best practice to "harden" against that attack.

**Part A - Rainbow Table Attack**

ref Wikipedia : https://en.wikipedia.org/wiki/Rainbow\_table

For a rainbow table, we need establish the following:

* Password plaintext domain
* Hash algorithm
* Base passwords
* "Reduction" function
* Number of rounds in a rainbow chain

**Plaintext Domain**

We need to establish what types of passwords we are trying to determine. For instance, for simplicity sake, we could target only 5 letter passwords. Then we set the domain to be 5 letter passwords and build our attack based on that assumption.

**Hash Algorithm**

Naturally, we must use the same Hash Algorithm as the one that generated the target hash digest.

**Base Passwords**

We need a large number of seed passwords that fit in the plaintext domain. Fortunately, there are many password files. For instance, we can take a large password dictionary and extract just the 5-letter passwords to use (if we have a domain of 5-letter passwords)

**Reduction function**

A reduction function will take the cryptographic hash and "reduce" it back into the plaintext domain somehow. This is NOT a reverse hash function, but rather just a way to "chain" the hash back into the plaintexts so that we can continue the cycle of hashing->reducing->hashing ...

A common example would be to take the first 5 letters of the hash output and use that if our domain were 5-letter passwords.

**Number of Rounds**

The length of each chain is defined by the number of rounds. More rounds means we save space in our Rainbow Table, but also means that there is more computation involved (slower). For instance, we could use 5-round chains in our Rainbow Table.

**Building the Rainbow Table**

Take each of your base passwords, hash the password. Take the resulting hash and reduce it back into the password domain. Now hash that. Reduce that hash digest. Do this 5 times. So you should have 5 hash/reduce steps.

Store the base password and the output of the LAST reduction step. So your data row should look like two 5-digit passwords. This is a rainbow table entry that contains a 5 hash lookup.

Repeat for all your base passwords.

**Attack**

Take the hash that you are attacking. Apply the reduction function to the hash. Now you have a 5-digit "password".

Take your Rainbow Table and using the 2nd column (final reduction value) look for the "password" you generated when you reduced the hash you are attacking.

If you find it, then you have the "Rainbow Chain" to find the password. Starting with the password in column 1, you hash/reduce 4 times (1 less than 5) to find the password. You win.

If you do NOT find it, then hash/reduce the value you were looking for. Now you have another 5-digit "password". Look for this the same way as before.

NOW, if you find it, you have the "Rainbow Chain", but this time, to find the password, you take the column 1 password and then hash/reduce it 3 times.

You can continue 3 more times until you've exhausted your Rainbow Table.

**Questions**

A.1. Draw out a diagram of a Rainbow Table and explain/show how this attack works.

A.2. This is a time/storage tradeoff attack. Explain the benefit of this and why it can speed up attacks by orders of magnitude.

A.3. This is precomputation-heavy. What does that mean?

**Code(s)**

**Given** : Password hash list to attack and Common password list.

**Use** : Plaintext domain is **8** character passwords. Hash function is **SHA1**. Reduction is use **first 8 chars** of hash digest. Rounds per chain is **5**.

**TODO**: Build rainbow table. use rainbow table to compromise hashes in list.

**Part B - Salt, Key Stretching**

Salting and Key Stretching are the primary common "hardening" steps against Rainbow Table Attack.

B.1. Explain what Salt is.

B.2. Explain what Key Stretching is.

B.3. Explain what using Salt and Key Stretching does to help.

**ref:** <https://en.wikipedia.org/wiki/PBKDF2>

**Python3's hashlib provides a pbkdf2\_hmac function**

ref: https://docs.python.org/3/library/hashlib.html

**Password Based Key Derivation Function (PBKDF)**

This is another way of saying, wrap the hash, salt, and key stretching into a single algorithm. Also, the Hash Function concept is generalized into a "Hash-based Message Authentication Code". PBKDF2 is a specific implementation of the general concept by RSA Labs's PKCS series, using SHA-512 as the hashing algorithm.

for PBKDF, we need to establish the following:

* Hash Function
* HMAC
* Salt
* Number of rounds for the key stretching

RSA Labs has established the PKCS standards series which has implemented the PBKDF concept, initially as PBKDF1 and now currently as PBKDF2. PBKDF2 uses SHA-512 as the hash algorithm.

**Hash Function**

The hash function is the basic mechanism underlying the "randomness" of the result. For this we shall use SHA-512

**HMAC**

We will use the RSA Labs' HMAC

**Salt**

Each password should get a unique, randomly generated Salt that will be stored alongside the final digest. We will use a 64-bit Salt.

**Number of Rounds**

We will hash and salt repeatedly many times to increase computation effort. For this, we shall use 4096 rounds. The number of rounds should be stored alongside the uid, salt, and digest.

**Code(s)**

We will use Python3's built in pbkdf2\_hmac function in the hashlib library (ref: https://docs.python.org/3/library/hashlib.html).

Given a number of uids and passwords, write a module to generate a password digest list using pbkdf2\_hmac, random salt, num iterations (10,000), and uid.

Write an authenticator that takes in a uid, password and then determines if it is a valid password for the user.

Questions

B.4. What happens when you try to use the Rainbow Table attack on your PBKDF2 list?

B.5. What is/are the weaknesses of PBKDF2?

B.6. What is a Length Extension Attack?

**Beyond PBKDF2**

Because of significant weaknesses in the modern age, PBKDF2 is no longer considered effective and now current Best Practice is to use one of the more modern Key Derivation schemes: **scrypt, bcrypt, argon2**. Of them, as of 2019, argon2 is considered the most secure.

**Write short module to using each of scrypt, bcrypt, argon2 to store password digests and to verify password attempt**

**bcrypt**

**ref**: https://pypi.org/project/bcrypt/

**scrypt**

**ref**: https://pypi.org/project/scrypt/

**argon2**

**ref**: https://pypi.org/project/argon2-cffi/

**opt** alt: https://pypi.org/project/argon2/