Homework

潘亦晟, 515021910384 04/16/2018

1 Background

Any computer system that requires password authentication must contain a database of passwords, either hashed or in plaintext, and various methods of password storage exist. Most databases, therefore, store a cryptographic hash of a user's password in the database.

After gathering a password hash, using the said hash as a password would fail since the authentication system would hash it a second time. To learn a user's password, a password that produces the same hashed value must be found.

Brute-force attacks and dictionary attacks are the most straightforward methods available. However, these are not adequate for systems that use long passwords because of the difficulty of storing all the options available and searching through such an extensive database to perform a reverse lookup of a hash.

Rainbow tables are one tool that has been developed to derive a password by looking only at a hashed value.

2 Precomputed hash chains

Hash chains are a technique for decreasing this space requirement. The idea is to define a reduction function R that maps hash values back into values in P. Note, however, that the reduction function is not actually an inverse of the hash function. The cipher text is longer that the key, hence the reduction. By alternating the hash function with the reduction function, chains of alternating passwords and hash values are formed.

$$\textbf{aaaa} \underset{H}{\longrightarrow} 281DAF40 \underset{R}{\longrightarrow} sgfnyd \underset{H}{\longrightarrow} 920ECF10 \underset{R}{\longrightarrow} \textbf{kiebgt}$$

The ciphertexts are organised in chains whereby only the first and the last element of a chain is stored in memory. Storing only the first and last element of a chain is the operation that yields the trade-off (saving memory at the cost of cryptanalysis time).

In the example chain above, **"aaaaaa"** would be the starting point and **"kiebgt"** would be the endpoint, and none of the other passwords (or the hash values) would be stored.

Given a hash value h that we want to invert (find the corresponding password for), compute a chain starting with h by applying R, then H, then R, and so on. If at any point we observe a value matching one of the endpoints in the table, we get the corresponding starting point and use it to recreate the chain. There's a good chance that this chain will contain the value h, and if so, the immediately preceding value in the chain is the password p that we seek.

For example, if we're given the hash **920ECF10**, we would compute its chain by first applying R:

$$920ECF10 \underset{R}{\longrightarrow} \textbf{kiebgt}$$

Since **"kiebgt"** is one of the endpoints in our table, we then take the corresponding starting password **"aaaaaa"** and follow its chain until **920ECF10** is reached:

$$\textbf{aaaa} \underset{H}{\longrightarrow} 281DAF40 \underset{R}{\longrightarrow} sgfnyd \underset{H}{\longrightarrow} 920ECF10$$

Thus, the password is **"sgfnyd"** (or a different password that has the same hash value).

collide and merge

Most serious if at any point two chains collide (produce the same value), they will merge and consequently the table will not cover as many passwords despite having paid the same computational cost to generate. Because previous chains are not stored in their entirety, this is impossible to detect efficiently.

For example, if the third value in chain 3 matches the second value in chain 7, the two chains will cover almost the same sequence of values, but their final values will not be the same.

The chance of finding a key by using a table of m rows of t keys is

$$P_{table} \geq \frac{1}{N} \sum_{i=1}^{m} \sum_{j=0}^{t-1} (1 - \frac{it}{N})^{j+1}$$

False alarm

When searching for a key in a table, the key may be part of a chain which has the same endpoint but is not in the table. In that case generating the chain from the saved starting point does not yield the key, which is referred to as a false alarm.

False alarms also occur when a key is in a chain that is part of the table but which merges with other chains of the table. In that case several starting points correspond to the same endpoint and several chains may have to be generated until the key is finally found.

3 Rainbow tables

Rainbow tables effectively solve the problem of collisions with ordinary hash chains by replacing the single reduction function R with a sequence of related reduction functions R_1 through R_k. In this way, for two chains to collide and merge they must hit the same value on the same iteration.

Since the hash value of interest may be found at any location in the chain, it's necessary to generate k different chains. To lookup a key in a rainbow table we proceed in the following manner:

First we apply R_{n-1} to the ciphertext and look up the result in the endpoints of the table. If we find the endpoint we know how to rebuild the chain using the corresponding starting point. If we don' t find the endpoint, we try if we find it by applying R_{n-2}, f_{n-1} to see if the key was in the second last column of the table. Then we try to apply R_{n-3}, f_{n-2}, f_{n-1}, and so forth. The total number of calculations we have to make is thus $\frac{t(t-1)}{2}$

4 Defense against rainbow tables

A rainbow table is ineffective against one-way hashes that **include large salts**

saltedhash(password) = hash(password||salt)

saltedhash(password) = hash(hash(password)||salt)

The salt value is not secret and may be generated at random and stored with the password hash. A large salt value prevents precomputation attacks, including rainbow tables, by ensuring that each user's password is hashed uniquely. This means that two users with the same password will have different password hashes (assuming different salts are used). In order to succeed, an attacker needs to precompute tables for each possible salt value. The salt must be large enough, otherwise an attacker can make a table for each salt value.

5 Problem Two

Problem. 设计一个重设密码的口令 hashing 流程

1. user request to reset password(user should enter old password and new password)
2. server retrieve the corresponding salt value and hash from database
3. server prepend the salt to the old password and hash it using the same hash function
4. server compare the hash of old password with the hash from the database to validate a password

5. if validation is sucessful, server generate a new long random salt
6. server prepend the salt to the new password and hash it with a standard passwrd hashing function.
7. server save both the salt and the hash in the user's database record