Passwords, Hashes and Rainbow Tables

By Steven Gordon on Thu, 14/02/2013 - 8:48am

Many computer systems, including online systems like web sites, use passwords to authenticate human users. Before using the system, the user is registered, where they normally select a *username* and *password* (or it is allocated to them). This information is then stored on the computer system. When the user later wants to access the computer system they submit their username and password, and the system checks the submitted values against the stored values: if they match the user is granted access.

There are many problems with using passwords for authentication, including being easy to guess, hard to remember, and possible to intercept across a network. In this article I focus on just one problem: the storage of the registered password on the system must be performed in a manner so that someone with access cannot discover other users' passwords.

1. Storing Actual Passwords

Consider a web site with user login as an example. Users of the website first register, and then once registered may login to gain personalized web content. Upon registration each user selects a unique username and their own password. Assume that the system stores these two values, *username* and *password*, in a database. So a website with 1000's users will have a database table such as:

username	*password*
john	mysecret
sandy	ld9a%23f
daniel	mysecret
...	...
steve	h31p_m3?

The obvious problem with this approach is that anyone who gains access to this database can see other users' passwords. Although such database will not be publicly accessible, within the organisation maintaining the website there may be multiple people who require read access to the database. It is therefore very easy for these people to view the actual passwords of many other people. Although this is a potential security issue for storing actual passwords, in many cases you will trust the organisation providing the database/website. Even if they couldn't read the database, since you are sending them your password it may be possible for people within that organisation to see your password.

A worse scenario is if the database becomes available to people outside the organisation. For example, the security of the organisations computer system has flaws such that a malicious user can gain unintended read access to the database. That malicious user has then discovered all passwords of the 1000's of users. They can use this information to masquerade as those users on the website, and since many people re-use passwords across different systems, the malicious user can also can gain unintended access to other systems.

Its this last scenario, of an external malicious user being able to read all passwords, that we want to prevent. From now on we will assume it is possible for a malicious user to gain read access to the database, hence storing actual passwords is not a secure option.

2. Storing Hashed Passwords

Rather than storing the actual password in the database, a hash of the password can be stored. Recall that good hash functions [3] have several useful practical properties:

1. Take a variable sized input and produce a fixed length, small output, i.e. the hash value
2. Hash of two different inputs produces two different output hash values (i.e. no collisions)
3. Given the output hash value, its practically impossible to find the corresponding input (i.e. a one-way function)

Further discussion of hash functions can be found in my lecture notes [4] or screencast [5] on the topic.

So for example with MD5 as a hash function, `john`'s password of `mysecret` would not be stored, but instead `MD5(mysecret)` is stored, i.e. `06c219e5bc8378f3a8a3f83b4b7e4649`. Note that MD5 produces a 128-bit hash value - here it is stored in hexadecimal. The database stored is now:

username	*H(password)*
john	06c219e5bc8378f3a8a3f83b4b7e4649
sandy	5fc2bb44573c7736badc8382b43fbeae
daniel	06c219e5bc8378f3a8a3f83b4b7e4649
...	...
steve	75127c78fd791c3f92a086c59c71ece0

When user `john` logs in to the web site he submits his username and password `mysecret`. The website calculates the MD5 hash of the submitted password and gets `06c219e5bc8378f3a8a3f83b4b7e4649`. Now the website compares the hash of the submitted password with the hash value stored in the database. As secure hash functions do not produce collisions, if the two hash values are the same then it implies the submitted password is the same as the original registered password. If they don't match, then the login attempt is unsuccessful.

Now assume a malicious user gains access to the database. They can see the hash values, but because of the one-way property of secure hash functions they cannot easily determine what the original password was. So by storing the hash of the password, instead of the actual password, the system offers significantly increased security.

3. Brute Force Attacks on Hashed Passwords

Above I said with a hash function it is practically impossible to find the input (password) given only the output hash value. What does "practically impossible" mean? Using the best known algorithms, with current (and near future) computing capabilities, it takes too long or will be too expensive to find the input password. I will not attempt to explain, and in fact some details I don't understand myself, but the amount of effort to find the input given an n-bit hash value is approximately equivalent to the effort of guessing a n-bit random number. That is, requires on order of 2^n attempts. MD5 uses a 128-bit hash, so it will take about 2^{128} or 3×10^{38} attempts to find the password. At a rate of 10^9 attempts per second, that is around 10^{21} years.

But the above is generally only true with large inputs (at least larger than the hash value). This is NOT the case with passwords. Most users choose short passwords (e.g 4 to 8 characters) so that they are easy to remember and input when logging in. Consider the case when users choose passwords that are always 8 characters long. Lets look at how many possible passwords there are and then see what an malicious user needs to do to find a password given only the hash value.

Lets assume a password is chosen from the set of characters that can be entered on an English keyboard. There are 52 letters (uppercase and lowercase), 10 digits, and another 32 punctuation characters (!, @, #, ...). So with a set of 94 characters to choose from, the number of 8 character-long passwords is 94^8 or about 6×10^{15}.

Now lets assume the malicious user has the database of users and hashed passwords. They are looking for John's password, i.e. they know the hash value *06c219e5bc8378f3a8a3f83b4b7e4649*. They then calculate the hashes of all possible passwords. When they find a resulting hash value that matches John's hash value, then they've found John's password. The m attempts the malicious user makes are summarised below:

Stored hash: 06c219e5bc8378f3a8a3f83b4b7e4649
Attempt 1: $password_1$ = 00000000; $hash_1$ = dd4b21e9ef71e1291183a46b913ae6f2
Attempt 2: $password_2$ = 00000001; $hash_2$ = ced165163e51e06e01dc44c35fea3eaf
Attempt 3: $password_3$ = 00000002; $hash_3$ = cc540920e91f05e4f6e4beb72dd441ac
...
Attempt m-1: $password_{m-1}$ = mysecres; $hash_{m-1}$ = 38a83897d7f7a8a2889bf6472e534567

```
Attempt m: password_m = mysecret; hash_m = 06c219e5bc8378f3a8a3f83b4b7e4649 <== matches stored hash
```

The worst case for the malicious user, assuming users choose random passwords, 94^8 hashes would need to be calculated to find a user's password. Can this be done within a reasonable time and cost? To have an idea we need to be able to estimate how long it takes to perform a hash (since the hash operation will be the most time consuming by far). This of course depends on the hardware performing the operation (and to a lesser extent the software). Consider for example oclHashcat [6], software for performing hashes on GPUs (GPUs are generally much faster than CPUs because they are designed to support many parallel operations at once). The performance benchmarks using an AMD HD6990 GPU indicate about 7×10^9 hashes per second can be calculated. Another site, by Ivan Golubev [7], estimates hash calculations on the same GPU at a rate of upto 10×10^9 hashes per second. The HD6990 is about 2 years old (costing about 20,000 Baht when released). For simplicty lets assume, we can calculate 10^{10} hashes per second, for a cost of about 10,000 Baht of hardware.

With 94^8 hashes to attempt at a rate of 10^{10} hashes per second, the malicious user would take about 7 days to try all possible passwords. This is definitely possible, although whether its worth the time and money of the malicious user depends on the value of the information that can be gained by discovering the password.

4. Pre-calculated Hashes and Rainbow Tables

The above simple example showed it would take about a week for a malicious user to find a password running with a recent GPU. Is it possible to make it even faster (without increasing the hardware capabilities)? Yes, it is. Just get someone else to calculate the hash values for you!

Assume someone has already calculated all 94^8 hash values. And they conveniently stored the hash value and corresponding password in a database. Then if you have that database, then its just a matter of performing a lookup with the users stored hash value against the set of pre-calculated hash values. Once a match is found, the password is found. The advantage of this approach is that performing a lookup (i.e. comparing one value against another value) is much, much faster than calculating a hash. So although one person took 7 days to calculate all the hash values, other malicious users can then re-use these values, and quickly check a known hash value against the set of pre-calculated values. This can reduce time to password discovery down to 10's of minutes or hours.

A potential problem with such pre-calculated hash values is the storage requirements. Considered how much raw data needs to be stored if no compression is used. There are 94^8 entries in a table stored. Each entry consists of a 8 character password (for simplicity, assume each character is 1 Byte) and a 128-bit MD5 hash value. That is at least 146,000 TB. This is not practical.

Of course compression can be used to store the data, but most general purpose compression techniques still would not reduce to a manageable size (a factor of 1000 size reduction would still result in 146 TB). However, using special purpose data structures to store the data is possible. Rainbow tables [8] are one such data structure. I will not attempt to explain how they work (because I don't know), but in brief rainbow tables are a data structure designing specifically for storing the hash and password. The result is a significant reduction in the total storage space needed. Consider Project RaindowCrack [9], an effort to pre-calculate the hashes of many possible passwords and distribute them (at a price) to whoever is interested. The have a list of password sets [10] already hashed and stored in rainbow tables, including the md5_ascii-32-95#1-8 set.

The md5_ascii-32-95#1-8 rainbow table contains the MD5 hashes of all combinations of 95 printable ASCII characters, ranging in length from 1 character to 8 characters long. The total number of passwords in this set is: $95^1 + 95^2 + 95^3 + 95^4 + 95^5 + 95^6 + 95^7 + 95^8 \approx 6\times10^{15}$. That is, about the same number as the example above using a set of 94 characters and 8 character passwords only. The raw data set is at least 146,000 TB. But using rainbow tables, the information is stored in 576 GB - thats a reduction in size by a factor of about 250,000. 576GB is a manageable size. In fact they sell this data set for $US1250, delivered in a 3TB hard disk.

So by using rainbow tables, the challenge of storing and distributing the set of passwords and hashes make it much easier/cheaper for a malicious user to quickly find a password, given only a hash. Some example tests

by Project RainbowCrack show that if given a hash of a random password, using the above rainbow table it takes between 5 and 30 minutes to find the password.

5. Salting a Password

Can we make it harder for malicious users that have discovered the hashed password database to use rainbow tables to quickly find passwords? Yes, there are several approaches including:

1. Require the users to use longer passwords. A 9 character random password requires almost 100 times more space and time to generate the rainbow table, again becoming much harder for the malicious user to manage. But requiring long, random passwords is inconvenient for users - they most likely not be able to remember such passwords - leading to other security problems.
2. Use different hash algorithms/implementations to slow down the calculation rate. If for example hashes could only be calculated at a rate of 10^8 hashes per second (instead of 10^{10}), then the time to generate the rainbow table would grow from 1 week to almost 2 years. But this approach doesn't help for exisiting algorithms (such as the popular MD5).
3. Use a salt before hashing the password, as explained below.

Requiring the user to increase their password length makes it harder for malicious users discovering passwords, but is inconvenient for users. An alternative is for the system to effectively increase the users password length by adding random characters to their chosen password. These extra characters are called a salt [11]. When a user account is created, the system chooses a random salt, concatenates it with the password and then hashes the resulting value. So a hash of the password with salt is stored. In addition, the salt is also stored in the password database. For example, with a 5 character salt, our example password database will be:

username	*salt*	*H(password || salt)*
john	a4H*1	ba586dcb7fe85064d7da80ea6361ddb6
sandy	U9(-f	816a425628d5dee17839fffeafb67144
daniel	5<as4	11842ced4203d4067ed6a6667f3f18d9
...	...	...
steve	LqM4^	184b7f9c6126c568ee50cd3364257973

Note that the salt is often measured in bits: our 5 character salt is approximately equivalent to a 32 bit value.

What can a malicious user do? Well they can attempt a brute force attack, trying all possible combinations of passwords. As the salt is stored in the password database, it is known to the malicious user, so it provides no additional security: the malicious user in the worst case still needs to 94^8 different passwords. Its just that for each password they try they must also concatenate with the salt for the appropriate user. It will still take about 7 days to find the password.

But what if the malicious user wants to use pre-calculated hashes, i.e. rainbow tables? This will no longer work because a rainbow table contains the hashes of passwords *without a salt*. The malicious user would need to use a rainbow table that contains the correct salt. For example, if trying to find John's password, a rainbow table must have been pre-calculated using the salt `a4H*1`. But if trying to find Sandy's password a rainbow table must have been pre-calculated using a different salt, `U9(-f`. In general, a separate rainbow would be needed for each possible salt. With a 32-bit salt, then about 4×10^9 rainbow tables are needed. The amount of space and time needed to generate the rainbow tables (previously 576GB and 7 days, respectively) have now both been increased by a factor of 4 billion. This is obviously unachievable for the malicious user.

In summary, an advantage of including a random salt before hashing the password is that it makes the use of pre-calculated tables of hashes and passwords (e.g. rainbow tables) ineffective. But note in most cases it does little to prevent a brute force attack, i.e. hasing each password plus salt and comparing with the stored hash value.

6. Summary and Other Issues

The main conclusion:

> When storing user login information, always store a hash of a salted password. Never store the actual password and avoid storing unsalted password hashes.

That is, select a long random salt, concatenate with the users password, calculate the hash of the result using a strong hash function, and store both the salt and hash value.

The above discussion made various assumptions and did not address other important issues about passwords, such as selecting passwords, dictionary attacks, selecting hash algorithms and speed of different hardware. There are many websites and textbooks that discuss this issues further and are worth reading.

Content: Articles [12]
Topic: Security [13]

Source URL: http://sandilands.info/sgordon/passwords-hashes-and-rainbow-tables

Links:
[1] http://sandilands.info/sgordon/passwords-hashes-and-rainbow-tables
[2] http://sandilands.info/sgordon/user/2
[3] http://en.wikipedia.org/wiki/Cryptographic_hash_function
[4] http://ict.siit.tu.ac.th/~sgordon/css322y12s2/lectures.html#hash
[5] http://www.youtube.com/watch?v=_kE2M5gSros
[6] http://hashcat.net/oclhashcat-plus/
[7] http://golubev.com/gpuest.htm
[8] http://en.wikipedia.org/wiki/Rainbow_table
[9] http://www.project-rainbowcrack.com/
[10] http://www.project-rainbowcrack.com/table.htm
[11] http://en.wikipedia.org/wiki/Salt_%28cryptography%29
[12] http://sandilands.info/sgordon/taxonomy/term/144
[13] http://sandilands.info/sgordon/taxonomy/term/116