Password Hashing

This article is covering password hashing, a subject which is often poorly understood by newer developers.
In this article I'm going to cover password hashing, a subject which is often poorly understood by newer developers. Recently I've been asked to look at several web applications which all had the same security issue - user profiles stored in a database with plain text passwords. Password hashing is a way of encrypting a password before it's stored so that if your database gets into the wrong hands, the damage is limited. Hashing is nothing new - it's been in use in Unix system password files since long before my time, and quite probably in other systems long before that. In this article I'll explain what a hash is, why you want to use them instead of storing real passwords in your applications, and give you some examples of how to implement password hashing in PHP and MySQL.

Foreword

As you read on you'll see that I advocate the use of a hashing algorithm called Secure Hashing Algorithm 1 (or SHA-1). Since I wrote this article, a team of researchers - Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu - have shown SHA-1 to be weaker than was previously thought. This means that for certain purposes such as digital signatures, stronger algorithms like SHA-256 and SHA-512 are now being recommended. For generating password hashes, SHA-1 still provides a more than adequate level of security for most applications today. You should be aware of this issue however and begin to think about using stronger algorithms in your code as they become more readily available.

For more information please see Bruce Schneier's analysis of the issue at
http://www.schneier.com/blog/archives/2005/02/cryptanalysis_o.html

What Is A Hash?

A hash (also called a hash code, digest, or message digest) can be thought of as the digital fingerprint of a piece of data. You can easily generate a fixed length hash for any text string using a one-way mathematical process. It is next to impossible to (efficiently) recover the original text from a hash alone. It is also vastly unlikely that any different text string will give you an identical hash - a 'hash collision'. These properties make hashes ideally suited for storing your application's passwords. Why? Because although an attacker may compromise a part of your system and reveal your list of password hashes, they can't determine from the hashes alone what the real passwords are.

So How Do I Authenticate Users?

We've established that it's incredibly difficult to recover the original password from a hash, so how will your application know if a user has entered the correct password or not? Quite simply - by generating a hash of the user-supplied password and comparing this 'fingerprint' with the hash stored in your user profile, you'll know whether or not the passwords match. Let's look at an example:

User Registration And Password Verification

During the registration process our new user will provide their desired password (preferably with verification and through a secure session). Using code similar to the following, we store their username and password hash in our database:

Figure 1. Our user enters their preferred access details

The next time our user logs in, we check their access credentials using similar code as follows:

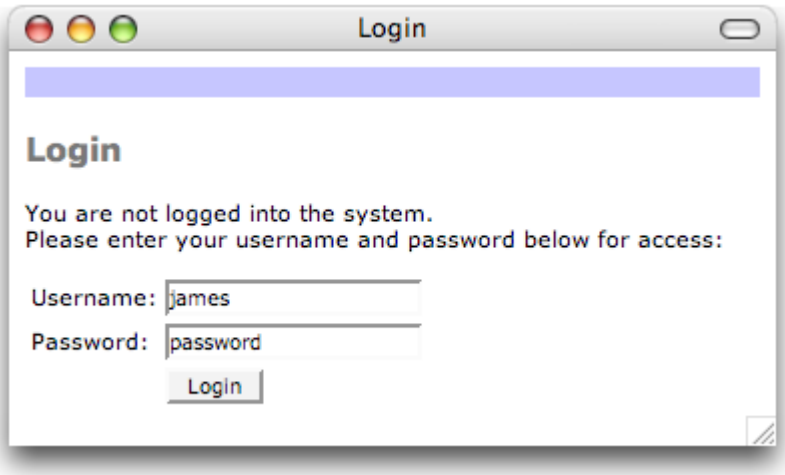

Figure 2. Logging back in

Types Of Hashes

There are a number of strong hashing algorithms in use, the most common of which are MD5 and SHA-1. Older systems - including many Linux variants - used Data Encryption Standard (DES) hashes. With only 56 bits this is no longer considered an acceptably strong hashing algorithm and should be avoided.

Examples

In PHP you can generate hashes using the md5() and sha1 functions. md5() returns a 128-bit hash (32 hexadecimal characters), whereas sha1() returns a 160-bit hash (40 hexadecimal characters). For example:

This code will output the following:

Original string: PHP & Information Security
MD5 hash: 88dd8f282721af2c704e238e7f338c41
SHA-1 hash: b47210605096b9aa0129f88695e229ce309dd362

In MySQL you can generate hashes internally using the password(), md5(), or sha1 functions. password() is the function used for MySQL's own user authentication system. It returns a 16-byte string for MySQL versions prior to 4.1, and a 41-byte string (based on a double SHA-1 hash) for versions 4.1 and up. md5() is available from MySQL

version 3.23.2 and sha1() was added later in 4.0.2.

```
mysql> select PASSWORD( 'PHP & Information Security' );
+------------------------------------------+
| PASSWORD( 'PHP & Information Security' ) |
+------------------------------------------+
| 379693e271cd3bd6 |
+------------------------------------------+
1 row in set (0.00 sec)

mysql> select MD5( 'PHP & Information Security' );
+-----------------------------------+
| MD5( 'PHP & Information Security' ) |
+-----------------------------------+
| 88dd8f282721af2c704e238e7f338c41 |
+-----------------------------------+
1 row in set (0.01 sec)
```

Note: Using MySQL's password() function in your own applications isn't recommended - the algorithm used has changed over time and prior to 4.1 was particularly weak.

You may decide to use MySQL to calculate your hash rather than PHP. The example of storing our user's registration details from the previous section then becomes:

Weaknesses

As a security measure, storing only hashes of passwords in your database will ensure that an attacker's job is made that much more difficult. Let's look at the steps they'll now take in an effort to compromise your system. Assuming that they've managed to access your user database and list of hashes, there's no way that they can then recover the original passwords to your system. Or is there?

The attacker will be able to look at your hashes and immediately know that any accounts with the same password hash must therefore also have the same password. Not such a problem if neither of the account passwords is known - or is it? A common technique employed to recover the original plain text from a hash is cracking, otherwise known as 'brute forcing'. Using this methodology an attacker will generate hashes for numerous potential passwords (either generated randomly or from a source of potential words, for example a dictionary attack). The hashes generated are compared with those in your user database and any matches will reveal the password for the user in question.

Modern computer hardware can generate MD5 and SHA-1 hashes very quickly - in some cases at rates of thousands per second. Hashes can be generated for every word in an entire dictionary (possibly including alpha-numeric variants) well in advance of an attack. Whilst strong passwords and longer pass phrases provide a reasonable level of protection against such attacks, you cannot always guarantee that your users will be well informed about such practices. It's also less than ideal that the same password used on multiple accounts (or multiple systems for that matter) will reveal itself with an identical hash.

Making It Better

Both of these weaknesses in the hashing strategy can be overcome by making a small addition to our hashing algorithm. Before generating the hash we create a random string of characters of a predetermined length, and prepend this string to our plain text password. Provided the string (called a "salt") is of sufficient length - and of course sufficiently random - the resulting hash will almost certainly be different each time we execute the function. Of course we must also store the salt we've used in the database along with our hash but this is generally no more of an issue than extending the width of the field by a few characters.

When we validate a user's login credentials we follow the same process, only this time we use the salt from our database instead of generating a new random one. We add the user supplied password to it, run our hashing algorithm, then compare the result with the hash stored in that user's profile.

Note: The function above is limited in that the maximum salt length is 32 characters. You may wish to write your own salt generator to overcome this limit and increase the entropy of the string.

Calling generateHash() with a single argument (the plain text password) will cause a random string to be generated and used for the salt. The resulting string consists of the salt followed by the SHA-1 hash - this is to be stored away in your database. When you're checking a user's login, the situation is slightly different in that you already know the salt you'd like to use. The string stored in your database can be passed to generateHash() as the second argument when generating the hash of a user-supplied password for comparison.

Using a salt overcomes the issue of multiple accounts with the same password revealing themselves with identical hashes in your database. Although two passwords may be the same the salts will almost certainly be different, so the hashes will look nothing alike.

Dictionary attacks with pre-generated lists of hashes will be useless for the same reason - the attacker will now have to recalculate their entire dictionary for every individual account they're attempting to crack.

Summary

We've seen now what hashes are and why you should store them instead of the plain text passwords they represent in your database. The examples above are a starting point and will get you on the right track with using hashes in your PHP applications. A little bit of work now may well mean much less of a headache further down the track!

About The Author

James McGlinn is a developer and project manager for Servers.co.nz where he provides application design, development and auditing services for a range of clients in New Zealand and abroad. PHP has been his language of choice since 1999. He is a Zend Certified Engineer and founded and facilitates the NZ PHP Users Group.

For more information he can be reached through his home page at http://james.mcglinn.org/.
