BENSCOMPUTERNOIP.ORG .Devel

Storage and Manipulation of Passwords: A Developers Guide

Ben Tasker
IT Manager & Linux Specialist

Contents

Introduction

For many users, creating and entering passwords is an everyday occurrence. On today's internet, very few services will allow access without some form of credential. Whether it's internet banking or social networking, the user is required to enter a username and a password.

Although passwords have a number of weaknesses when compared to alternative methods (such as One Time Tokens), they continue to be the most common form of authorisation. As a developer, it is highly likely that you will need to process and store passwords at some point.

The aim of this whitepaper is to look at the strengths and weaknesses of the various methods available. We will also look into the available methods of processing supplied credentials to establish whether to permit the user access to the system.

This paper is not intended to focus on any particular type of system, and the main body of information provided here should apply to any system, whether a web application or a local application. For convenience, we will assume that your application data is stored in a CSV based database. In reality the data can be stored using your preferred method

Storage

Although it would obviously be more secure *not* to store passwords, it is an unfortunate necessity. Without storing some form of the password, how will your application verify the password provided by a user upon login?

This section will examine the various methods of password storage available to you.

Storage

Plaintext

This is the simplest form of password storage, your application simply writes the entered password to the database. In order to verify a user's credentials upon login you simply retrieve the stored password from the database and compare the two strings.

Clearly, it is very simple to write some code to both store and retrieve the password. A simple CSV file could be as follows;

"Username","Password",
"Ben","MyPassword1",

However, this method should be avoided at all costs. User's have a habit of using the same password for multiple applications/services. If an attacker can gain access to your database, the passwords are available to him with no extra work. The attacker could achieve access to this data as a result of a flaw in your application (such as SQL Injection) or the database could

potentially be stolen by someone in your organisation.

There have been numerous news stories [1] about users' credentials being compromised as a result of being stored in plaintext.

It is easily possible to store an 'encrypted' version of passwords, so developers will need to have a *very* good reason to justify storing passwords in plaintext form! Developers around the world are realising that storing passwords in plaintext offers no tangible benefit, and is simply a symptom of laziness.

Hashed

The simplest form of encryption is to generate a 'hash' or a checksum. This is a cryptographic signature which is theoretically [2] unique to the original string. Which hashing algorithm you use depends on how accurate you need the hash to be (i.e. to reduce the number of possible collisions [2]).

Many people opt to use the MD5 algorithm, so using our previous example our CSV file would look like this;

"Username","Password",
"Ben","97274d5652cb9522ec0ed285b845b55f",

As you can see the password is no longer so easy for an attacker to retrieve. When the user attempts to log in, your application simply needs to generate a MD5 checksum of the password provided and then compare it to the sum stored by your application.

This method, whilst far better than plaintext, is however still quite easy for an attacker to circumvent. A dedicated attacker will generate what's known as rainbow tables [3].

These tables contain hashes of dictionary words and commonly used passwords. The attacker then utilises a program designed to compare the hash(es) in your database to those stored in his rainbow tables.

So in the above example, if our attacker's rainbow tables contain the hash *97274d5652cb9522ec0ed285b845b55f,* he/she will be able to deduce that our password is MyPassword1.

Note: *It's important to note that the rainbow tables only need to contain a matching checksum. If there is a 'hash collision' the password does not need to match the user's password. For example if the string "123456" generated the same hash as "MyPassword1" (it doesn't), the attacker would be able to log in by supplying "123456" as a password. It is for this reason that some opt to use a more advanced algorithm such as SHA256* [4]

Using hashed passwords requires the attacker to invest more time and energy into trying to retrieve the users' passwords, without overtly complicating your application. A dedicated attacker, however, will be able to compromise all stored passwords by using his 'rainbow tables'.

Hash Salting

Salted

Salted hashes are one step up from hashed passwords. The method of generating and verifying hashes differs in one slight way – we add a character (a salt) to the password.

So although our user enters “MyPassword1” to identify himself, we add one or more characters to the password before generating our hashes. So for example, if we used “SALT” as our salt, we could transform the password into either “SALTMyPassword1” or “MyPassword1SALT”. So utilising the former, our CSV file would look like this;

“Username”,”Password”,
“Ben”,”24a4fafc673bda0acad134362e72f4cc”,

As you can see, although the password remains the same, the generated hash is very different to that in our previous example. The salt used is generally used throughout the application, so we would salt every user’s password by adding “SALT” to the beginning of their password.

By salting our hashes, we have stopped an attacker from using generic rainbow tables, he must now invest time in order to generate Rainbow tables specific to our salt (Rainbow tables take a very long time to generate).

The use of salted hashes makes it impossible for attackers to generate and use one set of Rainbow tables to compromise multiple applications (assuming the applications use different salts). Adding a salt adds very little overhead in comparison to using plain hashes.

A dedicated attacker will take the time to generate Rainbow tables specific to your application, but use of salts will deter opportunistic attackers.

Random Salted

This method is very similar to that described above, except that instead of using a generic salt throughout the application, we generate a user specific salt.

An example of an application utilising this method is our very own BUGGER[5].

In order to use this method, we need to add an additional field to our table (BUGGER in fact uses three!), so our CSV file would be as follows;

“Username”,”Password”,”Salt”,
“Ben”,”341d1b981be66881618a16dcea1858f8”,”BenSALT”,

Each user would be allocated a different salt, so if we add a second user the CSV would look like this;

“Username”,”Password”,”Salt”,
“Ben”,”341d1b981be66881618a16dcea1858f8”,”BenSALT”,
“Bill”,”d55d02ce8ebe68be86d2febc0c2b114c “,”BillSALT”,

Hash Salting

Although Bill has (very coincidentally) also opted to use “MyPassword1” as his password, the stored hashes differ. This is because we used a different string to salt each one.

Although an attacker could still use rainbow tables to try and compromise our users’ credentials, he would need to generate a set of rainbow tables for each user account. It also carries an additional benefit, had we been using a per application salt, our CSV file would have looked like this;

“Username”,”Password”,”Salt”,
“Ben”,”341d1b981be66881618a16dcea1858f8”,
“Bill”,”341d1b981be66881618a16dcea1858f8”,

If either Bill or Ben were able to gain access to the database, they would be able to tell that the other user’s password was identical to their own. If Ben is an administrator and Bill is not, this could allow Bill to gain additional privileges.

Using this method does carry a slightly higher processing overhead than the methods we discussed earlier;

- When setting a password, your application must generate a unique salt.
- Your application needs to retrieve the relevant salt from your database every time the user is required to enter his password.

However, the additional work required to compromise users’ credentials will deter all but the most dedicated of attackers.

Notes on Storage

Although it is possible to increase the difficulty of compromising *all* the credentials stored by your application, an attacker only needs to compromise a single user account to cause harm.

Utilising Random Salts will, however, help prevent the embarrassment of having all stored credentials compromised. Some developers go so far as to store ‘dummy’ credentials in the hope that an attacker may waste their time decrypting a useless hash. Whether you feel this is worthwhile depends largely on personal taste and the performance required from the application.

Credential Verification

So, you've decided how you're going to store your users' details. But how do you process the supplied credentials to check they match those you have stored? This section is most relevant to web-based applications.

When talking about processing the credentials, we simply mean generating a (salted) hash of the supplied password. **At no point** should your application send the stored hash to the client!

Any authentication data passing between the server and the client (whether a salt, plaintext password or a hash) **must** be encrypted. The easiest method is to use a SSL connection, however, you should be aware that submitted data could still be compromised using a Man In The Middle (MITM) attack [6].

Server Side

The most convenient method, for you as a developer, is to use a server side script to process the supplied credentials. This does, however, pose a security risk in that it requires the client to send the entered password (as opposed to a hash) to your server.

Processing credentials Server-side allows you to write this aspect of the application in your preferred language, whether it be PHP, Perl or even C(++/#). It also means that you don't need to reveal your salt to the client (and thus a potential attacker).

Because the user's password is sent in plaintext form (albeit within a SSL session), a successful MITM attack would mean our attacker knows the password itself. He could then utilise this password to attack other applications/services that our unfortunate user uses.

So whilst your application may be a simple social networking platform, an attacker could take advantage of your complacency to gain a user's password, and then proceed to steal that user's identity on other sites. This could include accessing the user's e-mail and requesting a password reminder for that user's Internet Banking account!

Processing credentials server-side does carry an overhead, and could easily reduce the number of simultaneous connections that your server can support.

Client Side

If you decide to process the credentials client side, you will probably use Javascript.

You will need to send the relevant salt to the client for use in generating a hash of the password submitted as part of logging in. The disadvantage of this is that you have made it easily possible for an attacker to identify the salt that you use (they simply need to attempt a log in!).

However, by generating the hash client side, a MITM attack will only allow the attacker to view the submitted hash. If they do identify the hash, they will be able to log into your application (by writing a simple app that sends the hash in the same manner as your login form).

However, the compromised hash will only allow them to log into **your** application. Unless the attacker identifies the salt and generates dedicated Rainbow tables, the hash will not permit them to log into other applications/services that the user may use.

Processing credentials client-side will also lessen the load on your server, allowing it to support a greater number of simultaneous connections.

The Final Stage

Whether you've opted to process credentials client-side or server-side, comparison of the generated hash against the known hash should **always** take place server side.

In order to compare the two hashes, you obviously need to know the stored hash. If you send this information to the client, it becomes very easy for an attacker to compromise the hash (as simple as 'Right Click --> View Source').

You simply need to create a function that checks whether the hashes are identical, and then responds accordingly;

```
if [  "$SUPPLIED_HASH" == "$STORED_HASH" ]
then

# Hashes Matched
Access_Permitted

else

# Hashes don't match
Access_Denied

fi
```

Once the user has been authorised, you need not generate the hash for each request, use some form of token to identify that the session has been authorised. This could be a cookie or even a variable in the request URI, but ensure that the authorisation will automatically be revoked after a reasonable period of time.

Conclusion

Conclusion

As a result of our huge reliance on passwords, stolen credentials have become a regularly traded commodity. Most users do not understand just how severe the consequences of a breach could be, and find an easy complacency because “what would they want with my e-mail anyway?”.

Many users still utilise the same password for multiple applications/services and will probably always do so. As developers, it is our duty to ensure that our applications protect these credentials. Many, many breaches are caused by a developer’s incompetence and/or laziness.

There are numerous methods available to try and discourage attackers from attempting to compromise the details of **your** users. Which of these are most suited will obviously vary between applications, but it remains the responsibility of the developer.

Over time, these development practices will become second nature to you. The end result will be better applications and safer users.

References

[1] http://www.google.co.uk/search?q=Passwords+exposed+%2BPlaintext
[2] http://en.wikipedia.org/wiki/Collision_(computer_science)
[3] http://en.wikipedia.org/wiki/Rainbow_table
[4] http://en.wikipedia.org/wiki/Secure_Hash_Algorithm
[5] http://benscomputer.no-ip.org/BUGGER/Project_summary.shtml?USERNAME=&PROJ=4
[6] http://en.wikipedia.org/wiki/Man-in-the-middle_attack

Resources and Further Reading

- Basic Password Handling
- Create Effective Passwords
- Designing an Authentication System: a Dialogue in Four Scenes

For more free white papers visit

http://benscomputer.no-ip.org/Whitepapers/