Method for Storing User Password Securely

Gunjan Jha*
Navneet Popli**

Abstract

Computer users are asked to generate, secret passwords for uses host accounts, email, e-commerce sites, and various online services. In this paper, I'll explain the theory for how to store user passwords securely; we propose technique that uses hashing, salting and Bcrypt to compute secure passwords for many accounts while needed user to memorize only a single short password. The combination of security and convenience will, we believe, entice users to adopt our method, we discuss various methods, compare its strengths and weaknesses to those of related approaches.

Keywords: Password Security, website user authentication, hashing, salting, Bcrypt.

Introduction

Logging in with usernames and passwords has become one of the most ubiquitous and most reviled rituals of the Internet age. On the web, passwords are used by publications, blogs, and webmail providers.

We have multiple methods to store password securely in database. In this paper we will discuss about different methods in detail. We have methods like hashing, salting, and Bcrypt. We compare all the methods and analyze that which one is best in which scenario.

Related Work

Bad Solution : plain text password

It is not secure to store each users "plain text" password in database:

user account	plain text password
gunjan@hotmail.com	password
jassy@gmail.com	password123
...	...

This is insecure if a hacker gains access to database, they'll be able to use that password to login as that user on your database. This is even worse, if that user uses the same password for all other sites on the internet, the hacker can login there as well. Users will be very unhappy.

Gunjan Jha*
GGSIP University (Meri College)
Navneet Popli**
GGSIP University (Meri College)

Bad Solution: sha1(password)

defis_password_correct(user, password_attempt):
return sha1(password_attempt) == user ["sha1_password"]

A better solution is to store a "one-way hash" of the password, typically using a function likemd5 () or sha1 ():

user account	sha1(password)
gunjan@hot mail.com	5baa61e4c9b93f3f0682250 b6cf8331b7ee68fd8
jassy@gmail.com	cbfdac6008f9cab4083784c bd1874f76618d2a97
...	...

The server does not store the plain text password it can still authenticate user:

This solution is secure than storing the plain text , because in theory it should be impossible to "undo" a one-way hash function and find an input string that output the same hash value. Unfortunately, hackers found ways around this.

One problem is that many hash functions (including md5 () and sha1 ()) are not so "one-way" after all, and security expert suggest that these functions not be used anymore for security application. (Instead, you should use better hash functions like sha256 () which do not have any known vulnerabilities so far.)

But there's a bigger problem: hackers don't need to "undo" the hash function at all; they can just keep guessing input passwords until they find a match, It

is similar to trying all the combinations of a combination lock, Here what the code would look like:-

```
database_table = {
"5baa61e4c9b93f3f0682250b6cf83
31b7ee68fd8":"gunjan@hotmail.com",
"cbfdac6008f9cab4083784cbd187
4f76618d2a97":"jassy@gmail.com",
...}
for password
inLIST_OF_COMMONPASSWORDS:
if sha1(password) in databasetable:
print "Yepieee I win! guessed a password!"
```

You might think that there are too many possible passwords for this technique to be possible. But there are far fewer common passwords than you'd think. People use passwords that are based on dictionary words (possibly with a few extra numbers or letters). And most hash functions like sha1 () can be executed very quickly -- one computer can literally try billions of combinations per second. It means **most passwords can be figured out in less than one cpu-hour.**

Aside: years ago, computers were not this fast, so the hacker community created tables that have pre-computed a large set of these hashes ahead of time, Today nobody uses rainbow tables anymore because computers are fast enough without them.

So the bad news is any user with a simple password like "password" or "password12345"or any of the billion most-likely passwords will have their password guessed, if you have an extremely complicated password (over 16 random numbers & letters) you were probably safe.

Also notice that the code above is effectively **attacking all of the passwords at the same time.** It doesn't matter if there are ten users in your database or ten million, it doesn't take the hacker any longer to guess a matching password, All matters is that how fast the hacker can iterate through potential password, (And in fact, having lots of user actually **help** the hackers. because it is more likely that someone in the system was using the password "password12345".)

sha1(password) which LinkedIn used to store its password, And in 2012 a large set of password hashes were leaked, Over time hacker were able to figure out the plain text password to **most** of these hashes.

Summary: storing a simple hash (with no salt) is not secure - if a hacker gain access to your database, they will be able to figure out the majority of the passwords of the users.

Bad Solution : sha1(FIXED_SALT + password)

One attempt to make things more secure is to "salt" the password before hashing it:

user account	sha1("salt123456789" + password)
gunjan@hotmail.com	b467b644150eb350bbc1 c8b44b21b08af99268aa
jassy@gmail.com	31aa70fd38fee6f1f8b31 42942ba9613920dfea0
...	...

The salt is suppose to be a long random string of bytes, If the hacker gains access to these new password hashes (not the salt), will make it much more difficult for the hacker to guess the passwords because they would also require to know the salt, However if the hacker has broken into server, probably also have access to your source code as well so they'll learn the salt too, That is why security designers just assume the worst, & don't rely on the salt being secret.

But even if the salt is not a secret it still makes it harder to use those old-school **rainbow tablesI** mentioned before Those rainbow tables are built assuming there is no salt so salted hashes stop them. However since no one uses rainbow tables anymore adding a fixed salt does not help much, The hacker can still execute the same basic for-loop from above:

```
for password
inLIST_OF_COMMONPASSWORDS:
if sha1(SALT + password) in databasetable:
print "Yepieee I win! guessed a password!",
password
```

Summary: adding a fixed salt still is not secure enough.

Bad Solution : sha1(PER_USER_SALT + password)

The next step in security is to create a new column in database and store a different salt for each user, Salt is

randomly created when the user account is first created. (or when user changes their password).

user account	salt	sha1(salt + password)
gunjan@hotmail.com	2dc 7fcc	1a74404cb136dd600 41dbf694e5c2ec0e 7d15b42
jassy@gmail.com	afad b2f	e33ab75f29a9cf3f70d3 fd14a7f47cd752e 9c550
...	...	...

Authenticating the user is not much harder than before:

```
defis_password_correct(user, password_attempt):
return sha1(user["salt"] + password_attempt) == user["password_hash"]
```

By having a per-user-salt we get one huge benefit, the hacker cannot attack all of your user's passwords at the same time Instead his attack code has to try each user one by one:

```
for user in users:
PER_USER_SALT = user["salt"]
for password
inLIST_OF_COMMONPASSWORDS:
if sha1(PER_USER_SALT + password) in databasetable:
print "yepieee I win! guessed a password!", password
```

So basically if you have 1 million users having a per-user-salt makes it 1 million times harder to figure out the passwords of *all* your users. But still is not impossible for a hacker to do this. Instead of 1 cpu-hour now they need 1 million cpu-hours which can easily be rented from Amazon for about forty thousand dollar.

The real problem with all the systems we have discussed so far is that hash functions like sha1 () (even sha256 ()) can be executed on passwords at a rate of hundred M+/sec (or even faster by using GPU) Even though hash functions were designed with security in mind they were also designed so they would be fast when executed on longer inputs like entire files. These hash functions were not designed to be used for password storage.

Good Solution: bcrypt (password)

Instead there are a set of hash functions that were specifically designed for passwords. In addition to being secure "one-way" hash functions they were also **designed to be slow.**

One example is Bcrypt, bcrypt() takes about hundred ms to compute which is about 10,000x slower than sha1(). Hundred ms is fast enough that the user won't notice when they login but slow enough that it becomes less feasible to execute against a long list of likely passwords, instance if hackers want to compute bcrypt() against a list of a billion likely passwords it will take about 30,000 cpu-hours about $1200 and that is for a single password, not impossible but way more work than most hackers are willing to do.

Basically the trick is, it executes an internal encryption or hash function many times in a loop, there are other alternative to Bcrypt such as PBKDF2 uses the same trick.

Also Bcrypt is configurable with log_rounds parameters that tells it how many times to execute that internal hash function, If all of a sudden Intel comes out with a new computer that is thosand times faster than the state of the art today, you can reconfigure your system to use a log_rounds that is ten more than before (log_rounds is logarithmic) which will cancel out the 1000x faster computer.

Because bcrypt() is too slow it makes the idea of rainbow tables attractive again so a per-user-salt is built into the Bcrypt system, In fact libraries like py-bcrypt store the salt in the same string as the password hash so you won't even have to create a separate database column for the salt.

Let us see the code in action, First let's install it:

```
wget "http//py-bcrypt.googlecode.com/files/py-bcrypt-0.2.tar.gz"
tar -xzf py-bcrypt-0.2.tar.gz
cd py-bcrypt-0.2
python setup.py build
sudo python setup.py install
cd ..
python -c "import bcrypt"# did it work?
```

Now that it is installed, here is the Python code you'd run when creating a new user account (or reset their password):

```
from bcrypt import hashpw, gensalt
hashed = hashpw(plaintext__password, gensalt())
print hashed    # save this value to the database for the user
```

```
'$2a$12$8vxYfAWCXe0Hm4gNX
8nzwuqWNukOkcMJ1a9G2tD71ipotEZ9f80Vu'
```

Let us dissect that output string a little:

```
$2a$12$8vxYfAWCXe0Hm4gNX8nzwuqWNukOkcMJ1a9G2tD71ipotEZ9f80Vu

$bcrypt_id$log_rounds$128-bit-salt184-bit-hash
```

As you can see it stores both the salt, & the hashed output in the string, It also stores the log_rounds parameter that was used to generate the password which controls how much work that is how slow it is in computation, If you want the hash to be slower you pass a larger value to gensalt():

```
hashed = hashpw(plaintext_password,
gensalt(log_rounds=14))
print hashed
'$2a$13$ZyprE5MRw2Q3WpNOGZW
GbeG7ADUre1Q8QO.uUUtcbqloU0yvzavOm'
```

Notice that there is now a 14 where there was a 13 before, In any case you store this string in to the database, & when that same user attempts to log in you retrieve that same hashed value and do this:

```
if hashpw(password_attempt, hashed) == hashed:
print "matches"
else:
print "does not match"
```

You might be wondering why you pass in hashed as the salt argument to hashpw() The reason this works is that the hashpw() function is smart and can extract the salt from that$2a$13$.... string This is great because it means you never have to store parse or handle any salt values yourself , the only value you need to deal with is that single hashed string which contains everything you need.

Conclusion

Final Thoughts for choosing a good password If your user has the password "password" then no amount of hashing,salting,bcrypt etc is going to protect that user The hacker will always try simpler passwords first so if your password is toward the top of the list of likely passwords the hacker will probably guess it.

The best way to prevent password from being guessed is to create a password that is as far down the list of likely passwords as possible, Any password based on a dictionary word even if it has simple mutations like a letter/number at the end is going to be on the list of the first few million password guesses.

Unfortunately difficult-to-guess passwords are also difficult-to-remember, If that was not an issue I would suggest picking a password that is a 16 character random sequence of numbers and letters ,people have suggested using passphrases instead, like "shally is a police officer", your system allows long passwords with spaces then this is definitely better than a password like "shally123". (But I actually suspect the entropy of most user's pass phrases will end up being about the same as a password of eight random alphanumeric characters.)

Acknowledgment

This research paper is made possible through the help and support from everyone, including teachers, family and friends.

References

1. OpenSSL: The open source toolkit for SSL/TLS.http://www.openssl.org.
2. Mart´?n Abadi, T. Mark A. Lomas, and RogerNeedham. Strengthening passwords. Technical Report1997 - 033, 1997.
3. Mihir Bellare, David Pointcheval, and Phillip Rogaway. Authenticated key exchange secure against dictionary attacks. In EUROCRYPT, pages 139-155,2000.

4. E. Felten, D. Balfanz, D. Dean, and D. Wallach. Web spoofing: An Internet con game. Proc. 20th National Information Systems Security Conference, 1997.

5. Eran Gabber, Phillip B. Gibbons, Yossi Matias, and Alain J. Mayer. How to make personalized web browsing simple, secure, and anonymous. In Financial Cryptography, pages 17-32, 1997.

6. Rosario Gennaro and Yehuda Lindell. A framework for password-based authenticated key exchange. In EUROCRYPT, pages 524-543, 2003.

7. J. Jeff, Y. Alan, B. Ross, and A. Alasdair. The memorability and security of passwords - some empirical results, 2000.

8. Ian Jermyn, Alain Mayer, Fabian Monrose, Michael K. Reiter, and Aviel D. Rubin. The design and analysis of graphical passwords. 1999.

9. Jonathan Katz, Rafail Ostrovsky, and Moti Yung. Efficient password-authenticated key exchange using human-memorable passwords. In EUROCRYPT '01: Proceedings of the International Conference on the Theory and Application of Cryptographic Techniques, pages 475-494. Springer-Verlag, 2001.

10. J. Kelsey, B. Schneier, C. Hall, and D. Wagner. Secure applications of low-entropy keys. Lecture Notes in Computer Science, 1396:121-134, 1998.

11. David P. Kormann and Aviel D. Rubin. Risks of the Passport single signon protocol. In Proc. 9th international World Wide Web conference on computer networks, pages 51-58. North-Holland Publishing Co., 2000.

12. U. Manber. A simple scheme to make passwords based on one-way functions much harder to crack, 1996.

13. Robert Morris and Ken Thompson. Password security: A case history. CACM, 22(11):594-597, 1979.

14. Blake Ross, Collin Jackson, Nicholas Miyake, Dan Boneh, and John C. Mitchell. A browser plug-in solution to the unique password problem, 2005.Technical report, Stanford-SecLab-TR-2005-1.

15. Bruce Schneier et al. Password Safe application.http://www.schneier.com/passsafe.html.

16. Joe Smith. Password Safe cracker utility.