Dustin Boswell - Brain Dumps about Computers, Programming, and Everything Else

dustwell.com
About Me
Past Academic Work
Photo Album
Links I Use

Articles by Me
Subscribe: RSS

Oil ETFs leak money
Yes/No proposition bets
div, span, and CSS "display:"
Pair programming + screen
Dear VirginAmerica.com
SSH keys in 2 easy steps
"An hour" vs. "A hour"
How to hash passwords
Snapshotting with Rsync
MacBook Pro Sharp Edge
Fixing Your Flaky Internet
How X-over-SSH works
Drinking Distilled Water
Politician != Decider
Understanding iTunes files
My Audi A4 Gas Mileage
djb-dns installation
Vim Cheat Sheet

Storing User Passwords Securely: hashing, salting, and Bcrypt

June 18, 2012

In this article, I'll explain the theory for how to store user passwords securely, as well as some example code in Python using a Bcrypt library.

Bad Solution #1: plain text password

It would be very insecure to store each user's "plain text" password in your database:

user account	plain text password
john@hotmail.com	password
betty@gmail.com	password123
...	...

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

(Oh, and if you think no one would ever store passwords this way, Sony did just this in 2011.)

Bad Solution #2: sha1(password)

A better solution is to store a "one-way hash" of the password, typically using a function like md5() or sha1():

user account	sha1(password)
john@hotmail.com	5baa61e4c9b93f3f0682250b6cf8331b7ee68fd8
betty@gmail.com	cbfdac6008f9cab4083784cbd1874f76618d2a97
...	...

Even though the server doesn't store the plain text password anywhere, it can still authenticate the user:

```
def is_password_correct(user, password_attempt):
    return sha1(password_attempt) == user["sha1_password"]
```

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

One problem is that many hash functions (including md5() and sha1()) aren't so "one-way" afterall, and security experts suggest that these functions not be used anymore for security applications. (Instead, you should use better hash functions like sha256() which don't 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. This is similar to trying all the combinations of a combination lock. Here's what the code would look like:

```
database_table = {
  "5baa61e4c9b93f3f0682250b6cf8331b7ee68fd8": "john@hotmail.com",
  "cbfdac6008f9cab4083784cbd1874f76618d2a97": "betty@gmail.com",
  ...}

for password in LIST_OF_COMMON_PASSWORDS:
    if sha1(password) in database_table:
        print "Hacker wins! I guessed a password!"
```

You might think that there are too many possible passwords for this technique to be feasible. But there are far fewer common passwords than you'd think. Most people use passwords that are based on dictionary words (possibly with a few extra numbers or letters thrown in). And most hash functions like sha1() can be executed very quickly -- one computer can literally try billions of combinations each second. That means **most passwords can be figured out in under 1 cpu-hour.** Programs like John The Ripper are able to do just this.

Aside: years ago, computers weren't this fast, so the hacker community created rainbow 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 that any user with a simple password like `"password"` or `"password123"` or any of the billion most-likely passwords will have their password guessed. If you have an extremely complicated password (over 16 random numbers and 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 10 users in your database, or 10 million, it doesn't take the hacker any longer to guess a matching password. All that matters is how fast the hacker can iterate through potential passwords. (And in fact, having lots of users actually **helps** the hacker, because it's more likely that **someone** in the system was using the password `"password123"`.)

sha1(password) is what LinkedIn used to store its passwords. And in 2012, a large set of those password hashes were leaked. Over time, hackers 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 gains access to your database, they'll be able to figure out the majority of the passwords of the users.

Bad Solution #3: sha1(FIXED_SALT + password)

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

user account	sha1("salt123456789" + password)
john@hotmail.com	b467b644150eb350bbc1c8b44b21b08af99268aa
betty@gmail.com	31aa70fd38fee6f1f8b3142942ba9613920dfea0
...	...

The salt is supposed to be a long random string of bytes. If the hacker gains access to these new password hashes (but not the salt), it will make it much more difficult for the hacker to guess the passwords because they would also need to know the salt. However, if the hacker has broken into your server, they probably also have access to your source code as well, so they'll learn the salt too. That's why security designers just assume the worst, and 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 tables** I 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 doesn't help much. The hacker can still execute the same basic for-loop from above:

```
for password in LIST_OF_COMMON_PASSWORDS:
    if sha1(SALT + password) in database_table:
        print "Hacker wins! I guessed a password!", password
```

Summary: adding a fixed salt still isn't secure enough.

Bad Solution #4: sha1(PER_USER_SALT + password)

The next step up in security is to create a new column in the database and store a different salt for each user. The salt is randomly created when the user account is first created (or when the user changes their password).

user account	salt	sha1(salt + password)
john@hotmail.com	2dc7fcc...	1a74404cb136dd60041dbf694e5c2ec0e7d15b42
betty@gmail.com	afadb2f...	e33ab75f29a9cf3f70d3fd14a7f47cd752e9c550
...	...	...

Authenticating the user isn't much harder than before:

```
def is_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 can't 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 in LIST_OF_COMMON_PASSWORDS:
        if sha1(PER_USER_SALT + password) in database_table:
            print "Hacker wins! I 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 this still isn't impossible for a hacker to do. Instead of 1 cpu-hour, now they need 1 million cpu-hours, which can easily be rented from Amazon for about $40,000.

The real problem with all the systems we've discussed so far is that hash functions like sha1() (or even sha256()) can be executed on passwords at a rate of 100M+/second (or even faster, by using the GPU). Even though these 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. **Bottom line: 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 100ms to compute, which is about 10,000x slower than sha1(). 100ms is fast enough that the user won't notice when they log in, but slow enough that it becomes less feasible to execute against a long list of likely passwords. For instance, if a hacker wants to compute bcrypt() against a list of a billion likely passwords, it will take about 30,000 cpu-hours (about $1200) -- and that's for a single password. Certainly not impossible, but way more work than most hackers are willing to do.

If you're wondering how Bcrypt works, here's the paper. Basically the "trick" is that it executes an internal encryption/hash function many times in a loop. (There are other alternatives to Bcrypt, such as PBKDF2 that use the same trick.)

Also, Bcrypt is configurable, with a `log_rounds` parameter 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 1000 times faster than the state of the art today, you can reconfigure your system to use a log_rounds that is 10 more than before (log_rounds is logarithmic), which will cancel out the 1000x faster computer.

Because bcrypt() is so 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's 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's installed, here's the Python code you'd run when creating a new user account (or resetting their password):

```
from bcrypt import hashpw, gensalt
hashed = hashpw(plaintext_password, gensalt())
print hashed    # save this value to the database for this user
'$2a$12$8vxYfAWCXe0Hm4gNX8nzwuqWNukOkcMJ1a9G2tD71ipotEZ9f80Vu'
```

Let's 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, and 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 (i.e. how slow) it is to compute. If you want the hash to be slower, you pass a larger value to `gensalt()`:

```
    hashed = hashpw(plaintext_password, gensalt(log_rounds=13))
    print hashed
    '$2a$13$ZyprE5MRw2Q3WpNOGZWGbeG7ADUre1Q8QO.uUUtcbqloU0yvzavOm'
```

Notice that there is now a `13` where there was a `12` before. In any case, you store this string in the database, and when that same user attempts to log in, you retrieve that same `hashed` value and do this:

```
    if hashpw(password_attempt, hashed) == hashed:
        print "It matches"
    else:
        print "It 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$12$...` 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.

Final Thoughts: 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 your 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 wasn't an issue, I would suggest picking a password that is a 16-character random sequence of numbers and letters. Other people have suggested using passphrases instead, like `"billy was a turtle for halloween"`. If your system allows long passwords with spaces, then this is definitely better than a password like `"billy123"`. (But I actually suspect the entropy of most user's pass phrases will end up being about the same as a password of 8 random alphanumeric characters.)

24 Comments Dustin Boswell's Blog Login

Sort by Oldest Recommend Share

Join the discussion…

Greg Lumley · 3 years ago
Nice article!
3 · Reply · Share ›

Sanjay · 3 years ago
Superb explanation, great expression
1 · Reply · Share ›

soundofjw · 3 years ago
This is a great explanation of current password tech - thanks for sharing your insights!
1 · Reply · Share ›

Sundar Machi · 3 years ago
Thanks itz useful ..

• Reply • Share ›

Rezade2004 · 2 years ago
Thanks a lot
1 • Reply • Share ›

guest · 2 years ago
Great article, thanks!
One question: assuming that "the hacker has broken into your server". If the bcrypt hash&salt are stored in the same location, couldn't they been reverse-engineered just like in #3?
1 • Reply • Share ›

dustwell Mod guest · 2 years ago
Yes, in all the solutions I discussed, if a hacker breaks in and learns both the hash and the salt, they can write code to try all combinations of passwords until they guess the right one. Solution #4 is better than Solution #3 because it prevents the hacker from attacking all the passwords at the same time. Solution #5 (bcrypt) is better than #4 because the hash function is a million times slower to compute. Other than that, bcrypt is logically equivalent to Solution #4.
1 • Reply • Share ›

Jlewczyk dustwell · 2 years ago
Just a thought.. Wouldn't it be more secure to store the salt for the user in a different place than the hash? That way just stealing the user table doesn't get you enough information to brute force the stolen user table, no matter what Hash you utilize.
1 • Reply • Share ›

dustwell Mod Jlewczyk · 2 years ago
Yes, in theory it would be more secure. However, it makes things a little more complicated to implement: now when someone logs in, the backend server has to talk to 2 separate databases; and you have to deal with backing up and keeping both of them in sync.

Also, whatever reason caused your first database to be broken into (rogue employee, weak root password, insecure server, physical access to the server) would probably make it so your second database (the one with the salts) to be broken into as well.

If the user accounts are that important, you might want to consider other measures as well, such as Multi-Factor-Authentication
• Reply • Share ›

jooyoung · 2 years ago
I salute you for writing this. Thank you!!!
• Reply • Share ›

Kdkrun · 2 years ago
Thank you for valuable
 lessons
• Reply • Share ›

Harry Sufehmi · 2 years ago
Thanks for the insighful post.
• Reply • Share ›

Girish · 2 years ago
Nice article! I liked how you explained all the bad methods and the rationale.
• Reply • Share ›

PTap · 2 years ago
Nice work! Great info, explanation, and reasoning.
• Reply • Share ›

Matt Willsher · 2 years ago
This is not a binary "You must use bcrypt" decision.

If an attacker already has access to the password field in your DB you already screwed up big time. How do you know the entire DB hasn't gone? That being the case it really is a moot point as to how you're storing the passwords as, of course, your clued up users don't reuse passwords, yeah? So what's the real consequence of those passwords being lost? You did of course know the password fields of your super-web site had been compromised?

But lets assume that while the website in question failed to protect the username & password columns but managed to do the rest of it right and the whole DB isn't gone. Every time someone uses a username & password on your site you have >100ms of 100% CPU usage of one or more core. It can quickly becomes an expensive exercise for the site operator too, as well as using CPU time that could arguable have been better spent serving users. Sure, small sites aren't going to sweat it, but you have a 200 logins occur you suddenly have 20s of CPU time doing just bcrypt rounds.

As with so many things, the approach needs to be proportional. salt+SHA-256 is more than adequate most cases. If it's not, perhaps passwords are a bad choice too and the site in question should be looking at client SSL certs and OTP.

That said, it's a good explanation of the various methods of hashing (or not :)) passwords. It's just bcrypt isn't the only sensible option.

• Reply • Share ›

punund → Matt Willsher • 2 years ago

You can just lower the "logRounds" value if the performance is your concern.

• Reply • Share ›

Steve Ivy → Matt Willsher • 2 years ago

Thanks for the reminder that these things are tradeoffs, mattwillsh. Make a choice but know what it's going to mean for your app.

• Reply • Share ›

Scott R. Godin • 2 years ago

the App::bmkpasswd module from CPAN provides similar functionality to the py-bcrypt package, for those of you that require a perl solution:

```
>$ perl -MApp::bmkpasswd=mkpasswd,passwdcmp -M5.14.0 -e '\
my $hashed = mkpasswd(q[killer227bees@#$], "bcrypt", 12);\
say $hashed; say "matched" if passwdcmp(q[killer227bees@#$], $hashed);'

$2a$12$2ieEMAJSZW7qWUi6wlqfjO2ZQzpaqXMuwxSGsB/up0Rre8Fms2hEq
matched
```

• Reply • Share ›

thePeoplesFriend • a year ago

Excellent! Thanks! I've read many explanations on this topic, but this is clear, concise and current.

• Reply • Share ›

esoriano • a year ago

Warning: the hash length is wrong. The hash is truncated in the bcrypt python implementation and the reference implementation. It's not 192 bits, one byte is lost. For more info, see:

http://blog.rongarret.info/201...

Thanks for the article, it's very useful. Please, update it.

• Reply • Share ›

dustwell Mod → esoriano • a year ago

Thanks for the correction. 192 changed to 184.

• Reply • Share ›

anony • 3 months ago

Awesome article!

• Reply • Share ›

big dick bro • 22 days ago

a per_user_salt in no way makes it harder to guess passwords.

one hacker attempt involves one hacker attempt. It's not like having a per user salt stops them from being able to do a million attempts simultaneously.

if they're able to do simultaneous attempts.. then they would do it either way.

i think the author of this article got so hell bent on pushing bcrypt that his brain literally turned off for the per_user_salt segment of this article. it makes no logical sense whatsoever.

· Reply · Share ›

Joris · 8 days ago

Great round-up, I'm sharing this with my students!

· Reply · Share ›

Subscribe Add Disqus to your site Privacy