Securing Passwords with Salt, Pepper and Rainbows

You've heard again and again that storing passwords in plain-text is a bad idea so now you store your passwords as MD5 or SHA1 hashes... if someone steals your password database your users passwords are safe, right?

Actually no, they're not. They're never totally safe. You can however make the amount of effort required to break into an individual account too large for all but the most dedicated (and longevous) attacker.

Unfortunately most of the web applications that I get a chance to examine don't take the chance to make their password storage more secure, which is a shame because it's really not difficult.

For completeness, let's start with the simplest possible way of storing passwords.

Plain Text Passwords

Anathema to account security, if your password database is compromised then none of your accounts have any protection. Congratulations, you have just given away the details of your entire userbase.

That's not the worst of it either, anyone listening to the traffic between your password database and your application could pluck the passwords out of the air - very few people secure their database connections using TLS or SSH. If you're on an unswitched network spying on something like MySQL traffic becomes as easy as running one command:

```
# tcpdump -l -i eth0 -w - src or dst port 3306 | strings
```

Any query such as `SELECT users.id FROM users where password = 'foo'` or results from a query like `SELECT users.* FROM users` will show up in plain text and you won't even know that your passwords have been stolen.

In a switched environment it's possible to trick the switch into sending you traffic (although this can be detectable). Depending on the switching hardware it's also possible that in the event of a failure the switch will fail open and start acting like you're on an unswitched network.

Simple hashed passwords

Hashed passwords are generally looked at as the solution to this problem. No passwords are stored in plain-text, and it's hard to guess a password that'll match a certain hash so even if the password database is compromised or snooped you don't need to worry too much.

That may have been true, but hashes for many common words, passwords and passphrases have already been calculated. Translating from those hashes back to a password that'll match is trivial. Remember, since the original password isn't stored, all you need to do is match the hash - the actual input doesn't matter.

How easy is it to break into an account that's protected by an MD5 hashed password? Let's say we had attacked a site and found the following table of usernames and passwords:

```
Username : Hashed Password
Alice    : a34bc26f864ed5f404eac5b7a20cd9aa
Bob      : 7a75a532aaab234ad4bd33ed67e67242
Malory   : 39579c8d4a536eb092f959b4a3d14aa8
Zebedee  : 57208d910b63e879d2bae3b3a5f8366d
```

All we'd need to do is take the hashed password and look it up in a Rainbow Table for the hash that we think we're using. Given the length of these hashes (32 hexadecimal characters) it's pretty likely that they're MD5 hashed passwords so we could use something like GData to look for these hashes in an MD5 hash table.

```
Username : Password
Alice    : alphabets
Bob      : ch1cken
Malory   : blue41
Zebedee  : ?????
```

Only Zebedee is safe, and that's only for two reasons: (1) He's a freaky little spring creature that can do magic things who has a magnificent moustache, and (2) no one has added his password or some value that matches the hash of it to the rainbow table... yet.

Rainbow tables exist for several hashing algorithms including the shown MD5 and SHA1. If the hash wasn't on the rainbow table then causing a collision for a specific account would cost only USD$2,000 and take about a day for MD5. I'm not sure how much this would cost using SHA1.

Multiply hashed passwords

Rainbow tables take quite some time to populate and that time can be made longer by requiring that the hashing function is applied to the data several times before the value is taken.

```
MD5(alphabets)                          = a34bc26f864ed5f404eac5b7a20cd9aa
MD5(a34bc26f864ed5f404eac5b7a20cd9aa) = dd3f1bf5a36529705d08fe50b966d41a
MD5(...)                                = ...
MD5(...)                                = b5fdbbd055fcbfd3958a28f15661aea0
```

Each step takes a certain amount of CPU time to complete so the cost of generating a rainbow table that matches these hashes is higher, but CPU time is cheap these days. The advantage here is that the attacker doesn't know how many times you've applied the hash function unless they also have your code. Unfortunately that's reasonably easy to brute-force by starting at the hash and working backwards through the generated rainbow tables until you get to a value that allows the account to log into your site. After that magic number has been established you have just a few days before the rest of your accounts are compromised.

Peppered hashes

Rainbow tables can be calculated reasonably fast and while they're not hugely cheap, they're no longer exactly prohibitively expensive. How can we make it so that this is not a viable attack vector?

The rainbow tables don't enforce any rules on their input - they're maps of hashes to the input that generates them. If we require that each password contain a little data - a piece of spice (let's call it pepper) - that we specify then we can make limit the usefulness of the rainbow tables... A new rainbow table has to be generated where each input to the hash function has this pepper.

The pepper is stored in the authentication code in your application and never gets to the database except as part of a hash. In this way it behaves much like the magic number in the multiply hashed passwords above. Similarly, this one has the disadvantage that once the pepper has been discovered it can be used for other values.

Someone could - and if they're intent on it, will - calculate a rainbow table with a decent hit rate given time, but this does mean that there's no off-the-shelf rainbow table if you pick a strong, unique enough pepper.

Spicy hashes

By combining the pepper and the multiple runs of the hash function we could make the attacker have to guess two elements - the number of times the hash function is applied and the pepper.

```
pepper = ...aliesc3ifCTAasd4$af...
MD5(pepper + password)    = ...b5f34...
MD5(pepper + ...b5f34...) = ...ea28c...
MD5(pepper + ...ea28c...) = ...

SELECT users.id FROM users WHERE hashed_password = ...
```

I'm not sure that this actually buys anything over applying the hash function lots of times, but it sure looks pretty and I really wanted to make some sort of pun about using lots of pepper to make hashes spicy. Sorry.

Salted hashes

The pepper or the multiple-hash approach use only one value for the entire database - either the number of times the hash function is run on the input or the value that's added to the password. What if there were several values? What if there was one for each account? A small value that's unique to each account that's added in the same way as the pepper above - a salt for the hash.

In this case the rainbow table that needed to be used for an individual account would be of no use when it came to guessing the password for the next account.

Sounds awesome but where do we store the salt? I quite like storing it in the first few characters of the hashed password field, though you may prefer to store it in a separate column in the password database.

That's right, store it in the password database. Sounds like it'd make it easier to crack, right? Well, all this salt tells the attacker is that the password is somehow combined with this salt to generate the hash. The how is still hidden in your application code, and a valid password is still several iterations of rainbow table generation away... and that's for each individual account in your password database.

Safe now?

Hell no. With a decent combination of the above using strong salts, peppers, and a decent number of hashing calls you've made it unlikely that someone who has stolen your password database can access your users' accounts using it. That doesn't mean that your users will pick sane or safe passwords, that your system is free of bugs, or that there aren't other ways into your system or of finding your users passwords.

-- Written by **Craig** on **2009-08-03**

Disagree? Found a typo? Got a question?
If you'd like to have a conversation about this post, email craig@barkingiguana.com. I don't bite.

You can verify that I've written this post by following the verification instructions:

```
curl -LO http://barkingiguana.com/2009/08/03/securing-passwords-with-salt-pepper-and-rainbows.html.orig
curl -LO http://barkingiguana.com/2009/08/03/securing-passwords-with-salt-pepper-and-rainbows.html.orig.asc
gpg --verify securing-passwords-with-salt-pepper-and-rainbows.html.orig{.asc,}
```

Running Starling under DaemonTools | Home | First Steps with Rabbit MQ in Ruby 1.8.6