

Best Practices: Salting & peppering passwords?

Asked 11 years, 6 months ago Modified 3 years, 8 months ago

Viewed 71k times



187



I came across a discussion in which I learned that what I'd been doing wasn't in fact salting passwords but peppering them, and I've since begun doing both with a function like:

```
hash_function($salt.hash_function($pepper.$password)
[multiple iterations])
```

Ignoring the chosen hash algorithm (I want this to be a discussion of salts & peppers and not specific algorithms but I'm using a secure one), is this a secure option or should I be doing something different? For those unfamiliar with the terms:

- A **salt** is a randomly generated value usually stored with the string in the database designed to make it impossible to use hash tables to crack passwords. As each password has its own salt, they must all be brute-forced individually in order to crack them; however, as the salt is stored in the database with the password hash, a database compromise means losing both.

- A **pepper** is a site-wide static value stored separately from the database (usually hard-coded in the application's source code) which is intended to be secret. It is used so that a compromise of the database would not cause the entire application's password table to be brute-forceable.

Is there anything I'm missing and is salting & peppering my passwords the best option to protect my user's security? Is there any potential security flaw to doing it this way?

Note: Assume for the purpose of the discussion that the application & database are stored on separate machines, do not share passwords etc. so a breach of the database server does not automatically mean a breach of the application server.

security

hash

passwords

salt-cryptography

password-hash

Share

Improve this question

Follow

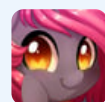
edited Jun 3, 2013 at 9:39



Charles

51.4k ● 13 ● 106 ● 143

asked Jun 3, 2013 at 7:15



Glitch Desire

15k ● 7 ● 44 ● 55

-
- 3 Not *quite* a duplicate, but *extremely* related:
stackoverflow.com/questions/16594613/... – ircmaxell Jun 3, 2013 at 11:14
-
- 2 Cross site duplicate:
security.stackexchange.com/q/3272/2113 – Jacco Jul 8, 2017 at 12:34
-

5 Answers

Sorted by:

Highest score (default)



Ok. Seeing as I need to write about this [over](#) and [over](#), I'll do one last canonical answer on pepper alone.

377



The Apparent Upside Of Peppers



It seems quite obvious that peppers should make hash functions more secure. I mean, if the attacker only gets your database, then your users passwords should be secure, right? Seems logical, right?

That's why so many people believe that peppers are a good idea. It "makes sense".

The Reality Of Peppers

In the security and cryptography realms, "make sense" isn't enough. Something has to be provable **and** make

sense in order for it to be considered secure. Additionally, it has to be implementable in a maintainable way. The most secure system that can't be maintained is considered insecure (because if any part of that security breaks down, the entire system falls apart).

And peppers fit neither the provable or the maintainable models...

Theoretical Problems With Peppers

Now that we've set the stage, let's look at what's wrong with peppers.

- **Feeding one hash into another can be dangerous.**

In your example, you do `hash_function($salt . hash_function($pepper . $password))`.

We know from past experience that "just feeding" one hash result into another hash function can decrease the overall security. The reason is that both hash functions can become a target of attack.

That's why algorithms like [PBKDF2](#) use special operations to combine them (hmac in that case).

The point is that while it's not a big deal, it is also not a trivial thing to just throw around. Crypto systems

are designed to avoid "should work" cases, and instead focus on "designed to work" cases.

While this may seem purely theoretical, it's in fact not. For example, [Bcrypt cannot accept arbitrary passwords](#). So passing `bcrypt(hash(pw), salt)` can indeed result in a far weaker hash than `bcrypt(pw, salt)` if `hash()` returns a binary string.

- **Working Against Design**

The way bcrypt (and other password hashing algorithms) were designed is to work with a salt. The concept of a pepper was never introduced. This may seem like a triviality, but it's not. The reason is that a salt is not a secret. It is just a value that can be known to an attacker. A pepper on the other hand, by very definition is a cryptographic secret.

The current password hashing algorithms (bcrypt, pbkdf2, etc) all are designed to only take in one secret value (the password). Adding in another secret into the algorithm hasn't been studied at all.

That doesn't mean it is not safe. It means we don't know if it is safe. And the general recommendation with security and cryptography is that if we don't know, it isn't.

So until algorithms are designed and vetted by cryptographers for use with secret values (peppers), current algorithms shouldn't be used with them.

- **Complexity Is The Enemy Of Security**

Believe it or not, [Complexity Is The Enemy Of Security](#). Making an algorithm that looks complex may be secure, or it may be not. But the chances are quite significant that it's not secure.

Significant Problems With Peppers

- **It's Not Maintainable**

Your implementation of peppers precludes the ability to rotate the pepper key. Since the pepper is used at the input to the one way function, you can never change the pepper for the lifetime of the value. This means that you'd need to come up with some wonky hacks to get it to support key rotation.

This is **extremely** important as it's required whenever you store cryptographic secrets. Not having a mechanism to rotate keys (periodically, and after a breach) is a huge security vulnerability.

And your current pepper approach would require every user to either have their password completely invalidated by a rotation, or wait until their next login to rotate (which may be never)...

Which basically makes your approach an immediate no-go.

- **It Requires You To Roll Your Own Crypto**

Since no current algorithm supports the concept of a pepper, it requires you to either compose algorithms or invent new ones to support a pepper. And if you can't immediately see why that's a really bad thing:

Anyone, from the most clueless amateur to the best cryptographer, can create an algorithm that he himself can't break.

- [Bruce Schneier](#)

NEVER roll your own crypto...

The Better Way

So, out of all the problems detailed above, there are two ways of handling the situation.

- **Just Use The Algorithms As They Exist**

If you use bcrypt or scrypt correctly (with a high cost), all but the weakest dictionary passwords should be statistically safe. The current record for hashing bcrypt at cost 5 is 71k hashes per second. At that rate even a 6 character random password would take years to crack. And considering my minimum recommended cost is 10, that reduces the hashes per second by a factor of 32. So we'd be talking only about 2200 hashes per second. At that rate, even some dictionary phrases or modifications may be safe.

Additionally, we should be checking for those weak classes of passwords at the door and not allowing them in. As password cracking gets more advanced, so should password quality requirements. It's still a statistical game, but with a proper storage technique, and strong passwords, everyone should be practically very safe...

- **Encrypt The Output Hash Prior To Storage**

There exists in the security realm an algorithm designed to handle everything we've said above. It's a block cipher. It's good, because it's reversible, so we can rotate keys (yay! maintainability!). It's good because it's being used as designed. It's good because it gives the user no information.

Let's look at that line again. Let's say that an attacker knows your algorithm (which is required for security, otherwise it's security through obscurity). With a traditional pepper approach, the attacker can create a sentinel password, and since he knows the salt and the output, he can brute force the pepper. Ok, that's a long shot, but it's possible. With a cipher, the attacker gets nothing. And since the salt is randomized, a sentinel password won't even help him/her. So the best they are left with is to attack the encrypted form. Which means that they first have to attack your encrypted hash to recover the encryption key, and then attack the hashes. But there's a **lot** of research into the attacking of ciphers, so we want to rely on that.

TL/DR

Don't use peppers. There are a host of problems with them, and there are two better ways: not using any server-side secret (yes, it's ok) and encrypting the output hash using a block cipher prior to storage.

Share Improve this answer

edited May 23, 2017 at 12:10

Follow



Community Bot


1 • 1

answered Jun 3, 2013 at 11:59



ircmaxell

165k • 35 • 268 • 315

-
- 7 Thanks for including that last part of encrypting the hash-value, this is an answer i can fully agree with. If the encryption would become part of your [password api](#), there would be no reason not to use it, so maybe... (i would love to write the documentation for it) – [martinstoeckli](#) Jun 3, 2013 at 13:21 
-
- 2 @martinstoeckli: I would not agree to add the encryption step to the simplified hashing API. The reason is that the storage of secrets (keys) is a lot harder than people realize, and it's quite easy to shoot yourself in the foot. For 99.9% of the users out there, raw bcrypt is more than sufficient for all but the simplest passwords... – [ircmaxell](#) Jun 4, 2013 at 14:56
-
- 8 @ircmaxell - On the other side, you would loose nothing. In the worst case, when the key becomes known, an attacker must still crack the BCrypt hash (same situation as without encryption). This is not the same as storing a key for encrypting data, this is about adding a server-side secret. Even a hardcoded key would protect those weak passwords, as long as the attacker has no control over the server/code.

This situation is not uncommon: aside from SQL-injection, also thrown away backups, discarded servers... can lead to this situation. A lot of PHP users work on hosted servers.

– [martinstoeckli](#) Jun 4, 2013 at 20:14

5 I was pointed to [OWASP Application Security Verification Standard 4.0](#) section 2.4.5 that recommends to use secret salt(aka pepper): “Verify that an additional iteration of a key derivation function is performed, using a salt value that is secret and known only to the verifier.” – [Michael Freidgeim](#) Sep 24, 2019 at 16:00 ✎

7 This answer is extremely out of date, and many claims it makes are no longer true. Modern crypto functions are designed to accept pepper as an argument, including the current best-in-class argon2id.
[stackoverflow.com/a/63222603/213191](#) – [Peter H. Boling](#) Jan 27, 2022 at 11:44



Fist we should talk about the **exact advantage of a pepper**:

30



- The pepper can protect weak passwords from a dictionary attack, in the special case, where the attacker has read-access to the database (containing the hashes) but does not have access to the source code with the pepper.



A typical scenario would be SQL-injection, thrown away backups, discarded servers... These situations are not as uncommon as it sounds, and often not under your control (server-hosting). If you use...

- A unique salt per password

- A slow hashing algorithm like BCrypt

...strong passwords are well protected. It's nearly impossible to brute force a strong password under those conditions, even when the salt is known. The problem are the weak passwords, that are part of a brute-force dictionary or are derivations of them. A dictionary attack will reveal those very fast, because you test only the most common passwords.

The second question is **how to apply the pepper ?**

An often recommended way to apply a pepper, is to combine the password and the pepper before passing it to the hash function:

```
$pepperedPassword = hash_hmac('sha512', $password, $pepper);  
$passwordHash = bcrypt($pepperedPassword);
```

There is another even better way though:

```
$passwordHash = bcrypt($password);  
$encryptedHash = encrypt($passwordHash, $serverSideKey);
```

This not only allows to add a server side secret, it also allows to exchange the \$serverSideKey, should this be necessary. This method involves a bit more work, but if the code once exists (library) there is no reason not to use it.

Follow



Mykola Semenov

802 ● 3 ● 13 ● 21

answered Jun 3, 2013 at 9:05



[martinstoeckli](#)

24k ● 6 ● 60 ● 91

So you'd say a pepper does add security over just a salt, in short? Thanks for the help on how to implement.

– [Glitch Desire](#) Jun 3, 2013 at 9:24

-
- 1 @LightningDust - Yes it does, for weak passwords, as long as the pepper stays secret. It mitigates some well defined types of threats. – [martinstoeckli](#) Jun 3, 2013 at 9:28 ✎

@martinstoeckli definitely a good way to implement this. Good to see that someone with some security experience supports this method. Would a MySQL

`AES_ENCRYPT($passwordHash, $serverSideKey)` call also be an appropriate way of implementing this?

– [The Thirsty Ape](#) Jun 27, 2013 at 19:40

-
- 1 @Foo_Chew - I don't know the implementation of the MySQL function, but it seems that they used the EBC mode, to avoid the IV-vector. Together with the known plaintext (hash-values always start with the same characters), this could be a problem. On my [homepage](#) i published an example implementation, that handles this encryption. – [martinstoeckli](#) Jun 27, 2013 at 20:14

-
- 1 Could you please suggest how would one exchange the key for the old passwords? Iterate over all of them and decrypt and re-encrypt? – [Mukul](#) Apr 4, 2018 at 13:06
-



6



I want this to be a discussion of salts & peppers and not specific algorithms but I'm using a secure one

Every secure password hashing function that I know of takes the password and the salt (and the secret/pepper if supported) as separate arguments and does all of the work itself.

Merely by the fact that you're concatenating strings and that your `hash_function` takes only one argument, I know that you aren't using one of those well tested, well analyzed standard algorithms, but are instead trying to roll your own. Don't do that.

[Argon2](#) won the Password Hashing Competition in 2015, and as far as I know it's still the best choice for new designs. It supports pepper via the K parameter (called "secret value" or "key"). I know of no reason not to use pepper. At worst, the pepper will be compromised along with the database and you are no worse off than if you hadn't used it.

If you can't use built-in pepper support, you can use one of the two suggested formulas from [this discussion](#):

```
Argon2(salt, HMAC(pepper, password))    or  
HMAC(pepper, Argon2(salt, password))
```

Important note: if you pass the output of HMAC (or any other hashing function) to Argon2 (or any other password hashing function), either make sure that the password hashing function supports embedded zero bytes or else encode the hash value (e.g. in base64) to ensure there are no zero bytes. If you're using a language whose strings support embedded zero bytes then you are probably safe, [unless that language is PHP](#), but I would check anyway.

Share Improve this answer

answered Aug 3, 2020 at 1:28

Follow



benrg

1,845 ● 17 ● 15



The point of salt and pepper is to increase the cost of a pre-computed password lookup, called a rainbow table.

5



In general trying to find a collision for a single hash is hard (assuming the hash is secure). However, with short hashes, it is possible to use computer to generate all possible hashes into a lookup onto a hard disk. This is called a Rainbow Table. If you create a rainbow table you can then go out into the world and quickly find plausible passwords for any (unsalted unpeppered) hash.



The point of a pepper is to make the rainbow table needed to hack your password list unique. Thus wasting more time on the attacker to construct the rainbow table.

The point of the salt however is to make the rainbow table for each user be unique to the user, further increasing the

complexity of the attack.

Really the point of computer security is almost never to make it (mathematically) impossible, just mathematically and physically impractical (for example in secure systems it would take all the entropy in the universe (and more) to compute a single user's password).

Share Improve this answer

answered Jun 3, 2013 at 7:32

Follow



Aron

15.8k ● 3 ● 33 ● 69

So does a salt+pepper offer any more security than just a salt? Or would I be better off dropping the pepper and running more iterations of script? – [Glitch Desire](#) Jun 3, 2013 at 7:34

2 The main thing about a rainbow table is that you don't create one for a specific attack, you download a pre-existing one. There are lengthy rainbow tables available for popular hash algorithms just a google away! – [Richard](#) Jun 3, 2013 at 7:34

1 @LightningDust I can't think of any reason myself. However Richard in the other thread came up with one. You can hide the pepper in your source code, which means another place your attacker needs to get access to, just to get a rainbow table together. – [Aron](#) Jun 3, 2013 at 7:36

@Aron - Well that's what I thought, since we have application servers separate from database servers (that is, if you get `root` on our db server, you still have no access to our application server), that hiding pepper in the source code (in our config file) would provide additional security.

– [Glitch Desire](#) Jun 3, 2013 at 7:38 



1

Can't see storing a hardcoded value in your source code as having any security relevance. It's security through obscurity.



If a hacker acquires your database, he will be able to start brute forcing your user passwords. It won't take long for that hacker to identify your pepper if he manages to crack a few passwords.

Share Improve this answer

answered Jun 3, 2013 at 7:23

Follow



Sven

1,326 ● 10 ● 12

2 It would make a precomputed rainbow table for an unsalted password table useless. In short, even if the attacker knew your pepper he would need to create a new rainbow table.

– Aron Jun 3, 2013 at 7:24

3 It strengthens the hash based on data not available in the database, which is a good thing. Things in the database can be potentially revealed in vulnerabilities - it's less likely that a value in the code will be accessed in the same way.

– Richard Jun 3, 2013 at 7:25

1 MD5 example here using `md5($pepper.$password)` (note: I know not to use MD5 on a live site, this is for example purposes), my password is `cat`, my hash is

`2d1ec2609e8265e64f2dbf6a697ebbfe`. What is my

pepper? At very best here, all you can do is try to brute force my (typically 64-character) pepper. – Glitch Desire Jun 3, 2013 at 7:31 ✎

1 @LightningDust I think you mean "Hashes are trapdoor functions". Yes it is impossible to figure out your salt and/or

pepper from a secure hash (in fact that is the definition of a secure hash). – [Aron](#) Jun 3, 2013 at 7:36

- 9 @Aron Security Through Obscurity can be a valid technique used as a layer of defense. The point is that it should **never** be relied upon as a defense, but instead used to "slow an attacker down". If that was not the case, something like a honeypot would not be used. Instead, we can effectively make use of security through obscurity to help slow attackers down, as long as we're not relying on it for the security of our application. – [ircmaxell](#) Jun 3, 2013 at 12:03
-