Safe Password Hashing ¶

This section explains the reasons behind using hashing functions to secure passwords, as well as how to do so effectively.

Why should I hash passwords supplied by users of my application?

Password hashing is one of the most basic security considerations that must be made when designing any application that accepts passwords from users. Without hashing, any passwords that are stored in your application's database can be stolen if the database is compromised, and then immediately used to compromise not only your application, but also the accounts of your users on other services, if they do not use unique passwords.

By applying a hashing algorithm to your user's passwords before storing them in your database, you make it implausible for any attacker to determine the original password, while still being able to compare the resulting hash to the original password in the future.

It is important to note, however, that hashing passwords only protects them from being compromised in your data store, but does not necessarily protect them from being intercepted by malicious code injected into your application itself.

Why are common hashing functions such as md5() and sha1() unsuitable for passwords?

Hashing algorithms such as MD5, SHA1 and SHA256 are designed to be very fast and efficient. With modern techniques and computer equipment, it has become trivial to "brute force" the output of these algorithms, in order to determine the original input.

Because of how quickly a modern computer can "reverse" these hashing algorithms, many security professionals strongly suggest against their use for password hashing.

How should I hash my passwords, if the common hash functions are not suitable?

When hashing passwords, the two most important considerations are the computational expense, and the salt. The more computationally expensive the hashing algorithm, the longer it will take to brute force its output.

PHP 5.5 provides a native password hashing API that safely handles both hashing and verifying passwords in a secure manner. There is also » a pure PHP compatibility library available for PHP 5.3.7 and later.

Another option is the crypt() function, which supports several hashing algorithms in PHP 5.3 and later. When using this function, you are guaranteed that the algorithm you select is available, as PHP contains native implementations of each supported algorithm, in case one or more are not supported by your system.

The suggested algorithm to use when hashing passwords is Blowfish, which is also the default used by the password hashing API, as it is significantly more computationally expensive than MD5 or SHA1, while still being scalable.

Note that if you are using crypt() to verify a password, you will need to take care to prevent timing attacks by using a constant time string comparison. Neither PHP's == and === operators nor strcmp() perform constant time string comparisons. As password_verify() will do this for you, you are strongly encouraged to use the native password hashing API whenever possible.

What is a salt?

A cryptographic salt is data which is applied during the hashing process in order to eliminate the possibility of the output being looked up in a list of pre-calculated pairs of hashes and their input, known as a rainbow table.

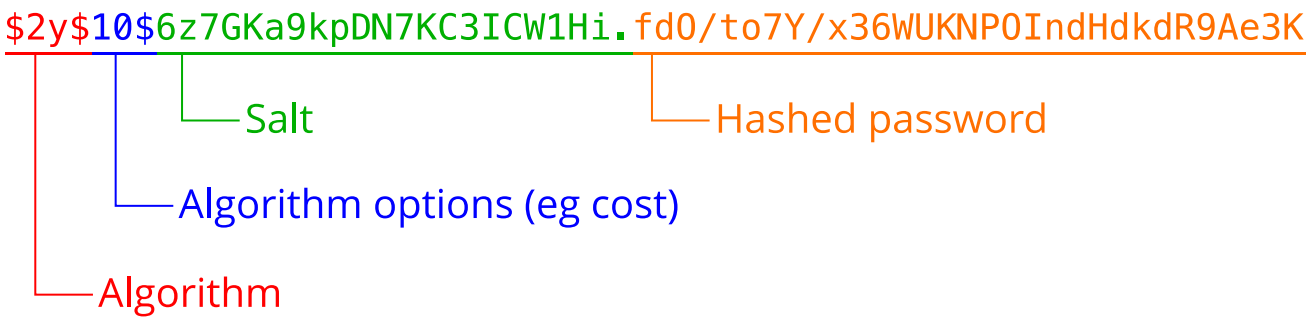
In more simple terms, a salt is a bit of additional data which makes your hashes significantly more difficult to crack. There are a number of services online which provide extensive lists of pre-computed hashes, as well as the original input for those hashes. The use of a salt makes it implausible or impossible to find the resulting hash in one of these lists.

password_hash() will create a random salt if one isn't provided, and this is generally the easiest and most secure approach.

How do I store my salts?

When using password_hash() or crypt(), the return value includes the salt as part of the generated hash. This value should be stored verbatim in your database, as it includes information about the hash function that was used and can then be given directly to password_verify() or crypt() when verifying passwords.

The following diagram shows the format of a return value from crypt() or password_hash(). As you can see, they are self-contained, with all the information on the algorithm and salt required for future password verification.



add a note

User Contributed Notes 3 notes

up
down
126
alf dot henrik at ascdevel dot com ¶
5 years ago

```
I feel like I should comment some of the clams being posted as replies here.

For starters, speed IS an issue with MD5 in particular and also SHA1. I've written my own MD5
bruteforce application just for the fun of it, and using only my CPU I can easily check a hash
against about 200mill. hash per second. The main reason for this speed is that you for most attempts
```

can bypass 19 out of 64 steps in the algorithm. For longer input (> 16 characters) it won't apply, but I'm sure there's some ways around it.

If you search online you'll see people claiming to be able to check against billions of hashes per second using GPUs. I wouldn't be surprised if it's possible to reach 100 billion per second on a single computer alone these days, and it's only going to get worse. It would require a watt monster with 4 dual high-end GPUs or something, but still possible.

Here's why 100 billion per second is an issue:
Assume most passwords contain a selection of 96 characters. A password with 8 characters would then have 96^8 = 7,21389578984e+15 combinations.
With 100 billion per second it would then take 7,21389578984e+15 / 3600 = ~20 hours to figure out what it actually says. Keep in mind that you'll need to add the numbers for 1-7 characters as well. 20 hours is not a lot if you want to target a single user.

So on essence:
There's a reason why newer hash algorithms are specifically designed not to be easily implemented on GPUs.

Oh, and I can see there's someone mentioning MD5 and rainbow tables. If you read the numbers here, I hope you realize how incredibly stupid and useless rainbow tables have become in terms of MD5. Unless the input to MD5 is really huge, you're just not going to be able to compete with GPUs here. By the time a storage media is able to produce far beyond 3TB/s, the CPUs and GPUs will have reached much higher speeds.

As for SHA1, my belief is that it's about a third slower than MD5. I can't verify this myself, but it seems to be the case judging the numbers presented for MD5 and SHA1. The issue with speeds is basically very much the same here as well.

The moral here:
Please do as told. Don't every use MD5 and SHA1 for hasing passwords ever again. We all know passwords aren't going to be that long for most people, and that's a major disadvantage. Adding long salts will help for sure, but unless you want to add some hundred bytes of salt, there's going to be fast bruteforce applications out there ready to reverse engineer your passwords or your users' passwords.

up
down
23
swardx at gmail dot com ¶
3 years ago
A great read..

https://nakedsecurity.sophos.com/2013/11/20/serious-security-how-to-store-your-users-passwords-safely/

Serious Security: How to store your users’ passwords safely

In summary, here is our minimum recommendation for safe storage of your users’ passwords:

- Use a strong random number generator to create a salt of 16 bytes or longer.
- Feed the salt and the password into the PBKDF2 algorithm.
- Use HMAC-SHA-256 as the core hash inside PBKDF2.
- Perform 20,000 iterations or more. (June 2016.)
- Take 32 bytes (256 bits) of output from PBKDF2 as the final password hash.

```
    Store the iteration count, the salt and the final hash in your password database.
    Increase your iteration count regularly to keep up with faster cracking tools.

Whatever you do, don’t try to knit your own password storage algorithm.
```
up
down
10
fluffy at beesbuzz dot biz ¶
7 years ago
```
The security issue with simple hashing (md5 et al) isn't really the speed, so much as the fact that
it's idempotent; two different people with the same password will have the same hash, and so if one
person's hash is brute-forced, the other one will as well.  This facilitates rainbow attacks.  Simply
slowing the hash down isn't a very useful tactic for improving security.  It doesn't matter how slow
and cumbersome your hash algorithm is - as soon as someone has a weak password that's in a
dictionary, EVERYONE with that weak password is vulnerable.

Also, hash algorithms such as md5 are for the purpose of generating a digest and checking if two
things are probably the same as each other; they are not intended to be impossible to generate a
collision for.  Even if an underlying password itself requires a lot of brute forcing to determine,
that doesn't mean it will be impossible to find some other bit pattern that generates the same hash
in a trivial amount of time.

As such: please, please, PLEASE only use salted hashes for password storage.  There is no reason to
implement your own salted hash mechanism, either, as crypt() already does an excellent job of this.
```
add a note