

Forum:[Hack Forums](#)

Thread: [Best password hashing method?](#)

mawrek - June 11, 2018, 1:11 pm

some people use sha256 but i've heard stories that bcrypt is the best possible encryption method that exists.

bcrypt is purposely designed to be slow so it will take much longer for hackers to breach it, unlike other methods.

but does someone have a detailed technical explanation what method do you think is the best?

Anxuein - June 11, 2018, 1:13 pm

<https://security.stackexchange.com/a/6415/163582>

Read through that.

splunk - June 11, 2018, 1:14 pm

bcrypt, or you could use sha256 and do unique per-user salts (usually this is strong enough).

TheDeceptionist - June 15, 2018, 10:42 pm

The three popular choices for hashing algorithms are:
PBKDF2

Bcrypt

Script

Technically speaking Script is more secure than Bcrypt because it requires more memory than Bcrypt. Also Bcrypt is slightly better than PBKDF2 because it removes the advantages of having powerful GPU's. However when it come to implementing a hashing algorithm in a project, each algorithm presents a problem. Bcrypt, for example, offers no additional protection from FPGA and Script when implemented requires very little CPU and therefore very little RAM.

According to the recommendation of [National Institute of Standards and Technology](#), it is sufficient to implement a PBKDF2 hashing algorithm.

scrubjay - June 16, 2018, 1:16 am

<http://shop.oreilly.com/product/0636920028154.do>

K4Fr - June 18, 2018, 2:49 am

(Jun 11, 2018 01:11 PM)mawrek Wrote: [\[CITE\]](#) some people use sha256 but i've heard stories that bcrypt is the best possible encryption method that exists.

bcrypt is purposely designed to be slow so it will take much longer for hackers to breach it, unlike other methods.

but does someone have a detailed technical explanation what method do you think is the best? scrypt and argon2 can take advantage of as much memory as you are willing to spend to raise the attacker's costs. bcrypt and pbkdf2 can only take advantage of as much time as you're willing to wait.

1. Pick whichever of scrypt, argon2, or balloon hash is convenient in your application, e.g. because of an existing high-speed library for it, without worrying too much about the differences.

2. Spend most of your time on (a) writing down a design document about your application, threat model, and security properties, and (b) making sure you use the password hash to maximize utilization of your available resources.

Figure out how much RAM the user is willing to spend on every password computation -- say, 1 GB. If your user is willing to wait 1sec to compute password hashing, then measure how many iterations of scrypt/argon2/balloon requires 1sec using 1 GB RAM.

Doing this well -- maximizing utilization of the available resources subject to the constraints of the user's experience -- is the hard part of this subproblem.

(Part 2(a) -- writing down a design document -- is still more important, though.)

If you're still worrying about which one to use in ten minutes and they're all equally convenient in your application, pick argon2id.

Also consider teaching your users to let computers pick passwords for them.

That's not a substitute for taking responsibility for password-hashing, but it enables users to improve their own security far beyond what an adversary could break by trying to guess passwords-- by that I mean you should encourage users to draw passwords uniformly at random from $>2^{128}$ possibilities. However, even if you encourage users to do that, you should still hash the passwords with whichever of scrypt or argon2id is convenient.. For example, if you are building a web application, you could embed <https://mumble.net/~campbell/js/pwgen/> (or a variant with your favorite word list, like one of the ones at <https://eff.org/dice>) in the password-change dialogue as an option for users. <https://mumble.net/~campbell/js/pwgen/> is an example of a little JavaScript program (it's tiny: view source and read it!) that draws a password uniformly at random from $>2^{128}$ possibilities. I used to use `tr -cd '[:graph:]' < /dev/urandom | head -c 20` as my password generator, and memorize a handful of those for different domains of password managers, without too much trouble. But the word sequences are noticeably easier to remember.

Python is quite shoddy when it comes to cryptography though - main reason being Python has too many libraries for cryptography, and of those libraries none implement exactly the primitives you need.(i.e cryptography.io pycrypto) - however hashlib has scrypt as of python 3.6+.Are you using python 3? If so, it looks like scrypt is the obvious winner. Now go on to step (2). The plus however to the method suggested is that it guarantees >128 bits of entropy. Tack on a couple more words and you can safely reject, say, nine out of ten sequences if you don't like them.

LSDoom - June 18, 2018, 5:16 am

I would recommend 8 to 13 round bcrypt. Should be strong enough for whatever you want to use it for. Keep in mind that very common passwords will still get cracked even with bcrypt.