Salt Generation and open source software

Asked 15 years, 1 month ago Modified 5 months ago Viewed 29k times

79

As I understand it, the best practice for generating salts is to use some cryptic formula (or even magic constant) stored in your source code.

I'm working on a project that we plan on releasing as open source, but the problem is that with the source comes the secret formula for generating salts, and therefore the ability to run rainbow table attacks on our site.

I figure that lots of people have contemplated this problem before me, and I'm wondering what the best practice is. It seems to me that there is no point having a salt at all if the code is open source, because salts can be easily reverse-engineered.

Thoughts?

security cryptography open-source salt-cryptography rainbowtable

Share

Improve this question

edited Jul 7 at 19:22

Dharman ♦

Follow

33.2k ● 27 ● 99 ● 146

asked Oct 29, 2009 at 17:00

user199085

927 ● 2 ● 8 ● 8

6 Answers

Sorted by: Highest score (default)

247

Since questions about salting hashes come along on a quite regular basis and there seems to be quite some confusion about the subject, I extended this answer.

What is a salt?

A salt is a *random* set of bytes of a fixed length that is added to the input of a hash algorithm.

Why is salting (or seeding) a hash useful?

Adding a random salt to a hash ensures that the same password will produce many different hashes. The salt is usually stored in the database, together with the result of the hash function. Salting a hash is good for a number of reasons:

1. Salting greatly increases the difficulty/cost of precomputated attacks (including rainbow tables)

2. Salting makes sure that the same password does not result in the same hash. This makes sure you cannot determine if two users have the same password. And, *even more important*, you cannot determine if the same person uses the same password across different systems.
3. ~~Salting increases the complexity of passwords, thereby greatly decreasing the effectiveness of both~~ Dictionary~~ and~~ Birthday attacks~~.~~ (This is only true if the salt *is* stored separate from the hash).
4. Proper salting *greatly* increases the storage need for precomputation attacks, up to the point where they are no longer practical. (8 character case-sensitive alpha-numeric passwords with 16 bit salt, hashed to a 128 bit value, would take up just under 200 exabytes without rainbow reduction).

There is no need for the salt to be secret.

A salt is not a secret key, instead a salt 'works' by making the hash function specific to each instance. With salted hash, there is not *one* hash function, but one for every possible salt value. This prevent the attacker from attacking *N* hashed passwords for less than *N* times the cost of attacking one password. This is the point of the salt.

A "secret salt" is not a salt, it is called a "key", and it means that you are no longer computing a hash, but a

Message Authentication Code (MAC). Computing MAC is tricky business (much trickier than simply slapping together a key and a value into a hash function) and it is a very different subject altogether.

The salt **must be random** for every instance in which it is used. This ensures that an attacker has to attack every salted hash separately.
If you rely on your salt (or salting algorithm) being secret, you enter the realms of Security Through Obscurity (won't work). Most probably, you do not get additional security from the salt secrecy; you just get the warm fuzzy feeling of security. So instead of making your system more secure, it just distracts you from reality.

So, why does the salt have to be random?

Technically, the salt should be *unique*. The point of the salt is to be distinct for each hashed password. This is meant *worldwide*. Since there is no central organization which distributes unique salts on demand, we have to rely on the next best thing, which is random selection with an unpredictable random generator, preferably within a salt space large enough to make collisions improbable (two instances using the same salt value).

It is tempting to try to derive a salt from some data which is "presumably unique", such as the user ID, but such schemes often fail due to some nasty details:

1. If you use **for example the user ID**, some bad guys, attacking distinct systems, may just pool their resources and create precomputed tables for user IDs 1 to 50. A user ID is unique *system-wide* but not *worldwide*.
2. The same applies to the **username**: there is one "root" per Unix system, but there are many roots in the world. A rainbow table for "root" would be worth the effort, since it could be applied to millions of systems. Worse yet, there are also many "bob" out there, and many do not have sysadmin training: their passwords could be quite weak.
3. Uniqueness is also temporal. Sometimes, users change their password. For each **new password**, a **new salt** must be selected. Otherwise, an attacker obtained the hash of the old password and the hash of the new could try to attack both simultaneously.

Using a random salt obtained from a cryptographically secure, unpredictable PRNG may be some kind of overkill, but at least it *provably* protects you against all those hazards. It's not about preventing the attacker from knowing what an *individual* salt is, it's about not giving them the big, fat target that will be used on a substantial number of potential targets. Random selection makes the targets as thin as is practical.

In conclusion:

Use a random, evenly distributed, high entropy salt. Use a new salt whenever you create a new password or change a password. Store the salt along with the hashed password. Favor big salts (at least 10 bytes, preferably 16 or more).

A salt does not turn a bad password into a good password. It just makes sure that the attacker will at least pay the dictionary attack price for *each* bad password he breaks.

Usefull sources:
stackoverflow.com: *Non-random salt for password hashes*
Bruce Schneier: *Practical Cryptography* (book)
Matasano Security: *Enough with the Rainbow Tables*
usenix.org: *Unix crypt used salt since 1976*
owasp.org: *Why add salt*
openwall.com: *Salts*

Disclaimer:
I'm not a security expert. (Although this answer was reviewed by Thomas Pornin)
If any of the security professionals out there find something wrong, please do comment or edit this wiki answer.

edited May 23, 2017 at 12:32

community wiki
23 revs, 3 users 82%
Jacco

4 it needs to be different for each row across all machines the system is used, the salt also needs to be unpredictable. This is achieved by creating a random salt. See also: stackoverflow.com/questions/536584/… – Jacco Oct 29, 2009 at 17:49

1 @Jacco, this is fantastic. And I agree with your argument for randomness. This is the most effective way to prevent attackers from 'guessing' the salt across all systems. (Though a crypto RNG is definitely overkill). – Jeremy Powell Nov 2, 2009 at 17:35

3 If you know the salt, you can run a brute force attack on this particular hash. There is no advantage in creating a rainbow table if you want to break a single hash. Also, creating a million hashes for a good hashing scheme will take longer than you seem to assume. – Jacco Jan 12, 2010 at 0:10

1 Thanks for the quick reply. Do you think weak passwords aren't common, or that adaptive cost hashing algorithms aren't a good idea? I appreciate the benefits of not straying from the focus of this question, but I see so much bad practice and advice out there about password hashing that I think at least linking to another good answer on the algorithm front would be very helpful. Do you know of any on stackoverflow? On the new it security stackexchange, security.stackexchange.com/questions/211/password-hashing is pretty good, so linking to it from here be the right way to help folks. – nealmcb Mar 29, 2011 at 3:45

2 One other point that is more relevant here. You say "Technically, the salt should be unique", but that isn't strong enough. It does also need lots of entropy as you say elsewhere. If it was just unique, handed out by some "central organization", they might just assign sequential salts, and

knowledge of how many salts had been assigned to date, or which salts had gone to your target, would reduce the attack space for a precomputed attack. – nealmcb May 10, 2011 at 21:34

23

Really salts just need to be unique for each entry. Even if the attacker can calculate what the salt is, it makes the rainbow table extremely difficult to create. This is because the salt is added to the password before it is hashed, so it effectively adds to the total number of entries the rainbow table must contain to have a list of all possible values for a password field.

Share Improve this answer

Follow

answered Oct 29, 2009 at 17:03

kemiller2002

115k • 28 • 199 • 253

8

Since Unix became popular, the right way to store a password has been to append a random value (the salt) and hash it. Save the salt away where you can get to it later, but where you hope the bad guys won't get it.

This has some good effects. First, the bad guys can't just make a list of expected passwords like "Password1", hash them into a rainbow table, and go through your password file looking for matches. If you've got a good two-byte salt, they have to generate 65,536 values for each expected password, and that makes the rainbow table a lot less practical. Second, if you can keep the salt from the bad guys who are looking at your password file,

you've made it much harder to calculate possible values. Third, you've made it impossible for the bad guys to determine if a given person uses the same password on different sites.

In order to do this, you generate a random salt. This should generate every number in the desired range with uniform probability. This isn't difficult; a simple linear congruential random number generator will do nicely.

If you've got complicated calculations to make the salt, you're doing it wrong. If you calculate it based on the password, you're doing it WAY wrong. In that case, all you're doing is complicating the hash, and not functionally adding any salt.

Nobody good at security would rely on concealing an algorithm. Modern cryptography is based on algorithms that have been extensively tested, and in order to be extensively tested they have to be well known. Generally, it's been found to be safer to use standard algorithms rather than rolling one's own and hoping it's good. It doesn't matter if the code is open source or not, it's still often possible for the bad guys to analyze what a program does.

answered Oct 30, 2009 at 15:34

David Thornley

57k ● 9 ● 95 ● 158

1 +1 for "impossible to determine if a given person uses the same password on different sites." – Jacco Oct 30, 2009 at

17:14

1

You can just generate a random salt for each record at runtime. For example, say you're storing hashed user passwords in a database. You can generate an 8-character random string of lower- and uppercase alphanumeric characters at runtime, prepend that to the password, hash *that* string, and store it in the database. Since there are 62^8 possible salts, generating rainbow tables (for every possible salt) will be prohibitively expensive; and since you're using a unique salt for each password record, even if an attacker has generated a couple matching rainbow tables, he still won't be able to crack *every* password.

You can change the parameters of your salt generation based on your security needs; for example, you could use a longer salt, or you could generate a random string that also contains punctuation marks, to increase the number of possible salts.

Share Improve this answer

Follow

answered Oct 29, 2009 at 17:10

mipadi

410k ● 90 ● 531 ● 487

You then have to store the salts in the database along with the hashed passwords, correct? – user199085 Oct 29, 2009 at 17:20

2 You will gain additional protection if you store the salts in a separate database, but even if they're stored alongside the

hashed password, just using the salt method will significantly increase the complexity of a successful attack. The key here is that by including a random element in the hash generation process, you've made cracking ALL the passwords considerably more difficult. See this Wikipedia entry for more info: en.wikipedia.org/wiki/Salt_%28cryptography%29
– Dave R. Oct 29, 2009 at 17:45

0

Use a random function generator to generate the salt, and store it in the database, make salt one per row, and store it in the database.

I like how salt is generated in django-registration.
Reference: http://bitbucket.org/ubernostrum/django-registration/src/tip/registration/models.py#cl-85

```
salt =
sha_constructor(str(random.random())).hexdigest()
[:5]
activation_key =
sha_constructor(salt+user.username).hexdigest()
return self.create(user=user,
           activation_key=activation_key)
```

He uses a combination of sha generated by a random number and the username to generate a hash.

`Sha` itself is well known for being strong and unbreakable. Add multiple dimensions to generate the salt itself, with random number, sha and the user specific component, **you have unbreakable security!**

Share Improve this answer edited Oct 29, 2009 at 17:20

Follow

answered Oct 29, 2009 at 17:13

lprsd

87k ● 48 ● 139 ● 169

7 Unbreakable security? I think that is *a bit* too optimistic. – Jacco Oct 29, 2009 at 21:54

6 SHA-1 is broken: schneier.com/blog/archives/2005/02/sha1_broken.html, so use SHA-256. – Jacco Oct 29, 2009 at 21:57

2 SHA-256 also become outdated now. currently suggested methods are BCrypt and PBKDF2. – Gnanz May 9, 2014 at 8:59

0

In the case of a desktop application that encrypts data and send it on a remote server, how do you consider using a different salt each time?

Using PKCS#5 with the user's password, it needs a salt to generate an encryption key, to encrypt the data. I know that keep the salt hardcoded (obfuscated) in the desktop application is not a good idea.

If the remote server must NEVER know the user's password, is it possible to user different salt each time? If the user use the desktop application on another computer, how will it be able to decrypt the data on the remote server if he does not have the key (it is not hardcoded in the software) ?

Share Improve this answer

Follow

answered May 26, 2011 at 15:48

Normand Bedard
2,685 2 20 24