Hashing – What is it

If you are an application developer , and you are working with a user login/registration system .  The most important aspect of a user account system is how user passwords are protected. It is not secure to save your password as plain text in database.

<!\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Image-sample-password\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_!>

And, Users account databases are hacked frequently, so you absolutely must do something to protect your users' passwords.

And here comes the **password hashing**.

<!\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Image-hashed-password\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_!>

Hash algorithms are one way functions (i.e the text you entered cannot be reversed ). They also have the property that if the input changes by even a tiny bit, the resulting hash is completely different (see the example above).

<!\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Image-workflow of user creates an account \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_!>

**Why we aren’t using encryption?**

* **Hashing** is simply **one-way.** You cannot get the string back, you can only check to see if a string validates against a hash. If your string validates against the hash, this does not guarantee that it's the same "password," but you can log in with it because you've found a **collision**. The "message"/password is usually limited to a small number of characters, relatively speaking.
* **Encrypting** is **two-way.** For example, you have an algorithm, a key, and a message. Using the key, you can unlock the message. Usually, the message could be of arbitrary size.

<!\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_illustrator Images\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_!>

Adding Salt

Lookup tables and rainbow tables only work because each password is hashed the exact same way. If two users have the same password, they'll have the same password hashes. We can prevent these attacks by randomizing each hash, so that when the same password is hashed twice, the hashes are not the same.

We can randomize the hashes by appending or prepending a random string, called a **salt**, to the password before hashing. As shown in the example above, this makes the same password hash into a completely different string every time. To check if a password is correct, we need the salt, so it is usually stored in the user account database along with the hash, or as part of the hash string itself.

The salt does not need to be secret. Just by randomizing the hashes, lookup tables, reverse lookup tables, and rainbow tables become ineffective. An attacker won't know in advance what the salt will be, so they can't pre-compute a lookup table or rainbow table. If each user's password is hashed with a different salt, the reverse lookup table attack won't work either.

In the next section, we'll look at how salt is commonly implemented incorrectly.