**Lab 0 - Hashing**

**Right Click anywhere on the purple background an “Open Terminal”**

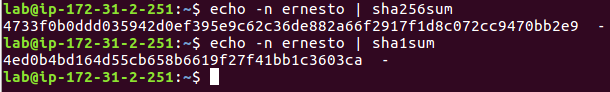
**Type: bash**

Say your password is "ernesto". (make your password in this exercise anything you want) I could simply store it raw, but anyone who gets my database gets the password. So instead I do an SHA1 or SHA256 hash on it, and get this:

**$ echo -n ernesto | sha1sum**

****

**Or**

****

Theoretically it's impossible to reverse a SHA1 hash. But go do [a google search on that exact string](https://www.google.com/search?q=a2c901c8c6dea98958c219f6f2d038c44dc5d362), and you will have no trouble recovering the original password.

Plus, if two users in the database have the same password, then they'll have the same SHA1 hash. And if one of them has a [password hint](http://www.rockpapershotgun.com/2013/11/25/one-down-your-dumb-password-adobe-crossword/) that says try "ernesto" -- well now I know what *both* users' passwords are.

**So before we hash it, we prepend a unique string. Not a *secret*, just something unique. How about WquZ012C. So now we're hashing the string WquZ012Cernesto. That hashes to this:**

**c5e635ec235a51e89f6ed7d4857afe58663d54f5**



Googling that string turns up nothing (except perhaps *this* page), so now we're on to something. And if person2 also uses "ernesto" as his password, we use a different salt and get a different hash.

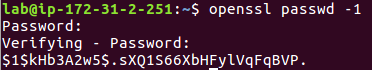
Of course, in order to test out your password, you have to know what the salt is. So we have to store that somewhere. Most implementations just tack it right on there with the hash, usually with some delimiter. Try this if you have openssl installed:

[labl ~]$ openssl passwd -1

Password: ernesto

Verifying - Password: ernesto

$1$oaagVya9$NMvf1IyubxEYvrZTRSLgk0



This gives us a hash using the standard crypt library. So our hash is $1$oaagVya9$NMvf1IyubxEYvrZTRSLgk0: it's actually 3 sections separated by $. I'll replace the delimiter with a space to make it more visually clear:

$1$oaagVya9$NMvf1IyubxEYvrZTRSLgk0

1 oaagVya9 NMvf1IyubxEYvrZTRSLgk0

* **1** means "algorithm number 1" which [is a little complicated](https://passlib.readthedocs.io/en/stable/lib/passlib.hash.md5_crypt.html#algorithm), but uses MD5. There are [plenty others which are much better](http://pythonhosted.org/passlib/modular_crypt_format.html#mcf-identifiers), but this is our example.
* **oaagVya9** is our salt. Plunked down right there in with our hash.
* **NMvf1IyubxEYvrZTRSLgk0** is the actual Hash algorithm MD5 sum, base64-encoded.

If I run the process again, I get a completely different hash with a different salt. In this example, there are about 1014 ways to store this one password. All of these are for the password "ernesto":

$1$9XsNo9.P$kTPuyvrHqsJJuCci3zLwL.

$1$nLEOCtx6$uSnz6PF8q3YuUhB3rLTC3/

$1$/jZJXTF3$OqDuk8T/cEIGpeKWfsamf.

$1$2lC.Cb/U$KR0jkhpeb1sz.UIqvfYOR.

But, if I deliberately specify the salt I want to check, I'll get back my expected result:

[labl ~]$ openssl passwd -1 -salt oaagVya9

Password: ernesto

Verifying - Password: ernesto

$1$oaagVya9$NMvf1IyubxEYvrZTRSLgk0



And that's techon to check to see if the password is correct. Find the stored hash for the user, find the saved salt, re-run that same hash using saved salt, check to see if the result matches the original hash.

Implementing This Yourself

To be clear, this post is not an implementation guide. **Don't simply salt your MD5 and call it good. That's not enough in today's risk climate**. ***You'll instead want to run an iterative process which runs the hash function thousands of times. This is the COST variable.*** This has been [explained elsewhere](https://security.stackexchange.com/a/52065/2264) many times over, so I won't go over the *"why"* here.

There are several well-established and trusted options for doing this:

* [**crypt**](http://en.wikipedia.org/wiki/Crypt_%28C%29): The function I used above is an older variation on the unix crypt password hashing mechanism built-in to all Unix/Linux operating systems. The original (DES-based) version is horribly insecure; don't even consider it. The one I showed (MD5-based) is better, but still shouldn't be used today. Later variations, including the SHA-256 and SHA-512 variations should be reasonable. All recent variants implement multiple rounds of hashes.
* [**bcrypt**](http://en.wikipedia.org/wiki/Bcrypt): The blowfish version of the crypt functional call mentioned above. Capitalizes on the fact that blowfish has a very expensive key setup process and takes a "cost" parameter which increases the key setup time accordingly.
* [**PBKDF2**](http://en.wikipedia.org/wiki/PBKDF2): ("Password-based Key Derivation Function version 2") Created to produce strong cryptographic keys from simple passwords, this is the only function listed here that [actually has an RFC](http://tools.ietf.org/html/rfc2898). Runs a configurable number of rounds, with each round it hashes the password plus the previous round's result. The first round uses a salt. It's worth noting that its original intended purpose is *creating strong keys*, not *storing passwords*, but the overlap in goals makes this a well-trusted solution here as well. If you had no libraries available and were forced to implement something from scratch, this is the easiest and best-documented option. Though, obviously, using a well-vetted library is always best.
* [**scrypt**](http://en.wikipedia.org/wiki/Scrypt): A recently-introduced system designed specifically to be difficult to implement on dedicated hardware. In addition to requiring multiple rounds of a hashing function, *scrypt* also has a very large working memory state, so as to increase the RAM requirement for implementations. While very new and mostly unproven, it looks at least as secure as the others, and possibly the most secure of them all.
* argon2: This is presently one of the strongest hashes. The number of iterations (or t variable) is what provides a cost to systems attempting to brute force hack the algorithm.

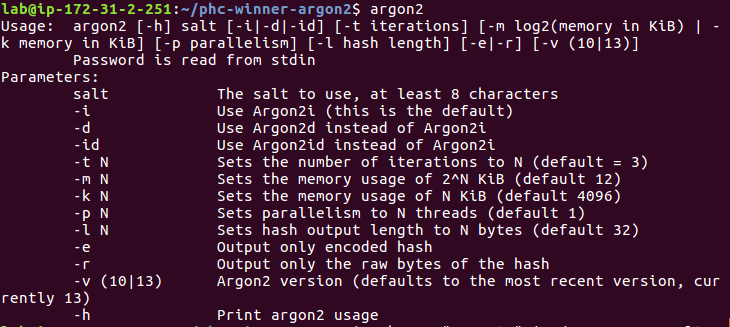
**Execute the following:**

**git clone https://github.com/P-H-C/phc-winner-argon2.git**

**cd ./phc-winner-argon2**

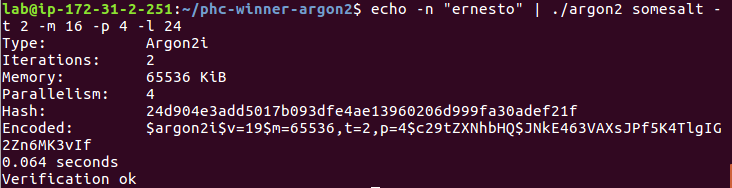
**sudo make install**

**./argon2**



**Notice that the “t” is cost.**

**Execute the command below:**



Change the t to a value of 4 and observe the results.

Change the value of t to 999 and observe the results. Is it taking longer to process?