Adding Salt to Hashing

CS 166 Section 4
Professor Tarng

By
Madhuri Manohar, Christian Oh, Cameron Lau, Gurteg Singh

December 7th, 2021

Section 1 : Project Overview

Password security has been a major problem since the start of the internet age and it is a problem that has only gotten worse as time goes on. Many hackers continue to improve with newer tech and many people increasingly rely on simple passwords for their many applications. As we put more and more of our personal information onto the internet for safekeeping, the risk of this information being leaked and used against us in some way greatly increases significantly. With weak passwords being the only thing between a hacker and one's most important information online, it's no wonder that the importance of password security has been greatly advertised and stronger password education methods have risen in the past couple of years.

Many methods have been discussed on how to improve password security. Among them are things like two factor authentication and password generators or using encryption and hashing for storing. One method for storing passwords is "adding salt" to the hashed password in order to make common attacks used against hashing ineffective. Salting essentially includes using a randomly generated value called the 'salt value' and adding that value to the end of the plaintext user password, which is then hashed using any 'SHA-2' algorithm. This is then stored into the database along with the salt value. Salting makes hashing more secure because without salt, the hashed passwords would end up being identical and in the case of commonly used passwords like 'password123', the hashed value stored would be the same for a lot of users in the database thereby compromising the security of user information. The LinkedIn hack of 2012 led to the release of almost 6.5 million user's password information(later revealed to actually be over 100 million emails/passwords), but the worst part about the leak was that the data was unsalted and secured using a SHA-1 algorithm. While the use of SHA-1 in the modern age by a big tech company is a whole different case, a massive and terrible one at that, the fact that LinkedIn was

not salting their stored passwords meant that many users had to face the very real chance of their information being bought and sold online. The lack of effort by LinkedIn led to the panic of millions of people worldwide and was a problem that could have easily been avoided had the passwords been salted along with their hashes.

Section 2 : Project Objectives and Scope

The objective of this project is to showcase the security of salt||password hashes. Salt||password is how we denote adding salt to a password and using a hash function on both the salt and password. We plan on learning about the adding salt process and how it helps increase its security compared to the traditional or past methods used. Furthermore, we'll discuss the positives and negatives of the salted passwords.

In the scope of this project, we will study and familiarize ourselves with different hashing algorithms like bCrypt and SHA-256. This includes learning about the various forms of attacks that are effective against hashing algorithms like rainbow tables and dictionary/brute force attacks. We will also study the differences and history between the hashing algorithms and how to effectively utilize them.

Section 3 : Project Architecture

SHA stands for secure hash algorithm. SHA-256 came after SHA-1 and this algorithm and it's 256 bit key has not been compromised yet.

Section 3.1 : Algorithm for SHA-256

SHA-256 is a block cipher. It works on block size of 512 so we need to edit our input string to be a multiple of 512. First, SHA-256 appends a bit 1. Then it appends zeros until the length is 64 bits short of being a multiple of 512. Finally a big-endian representation of the length of the original string is appended.

SHA-256 uses several functions and constants. SHA-256 uses 8 hash values stored in state registers labeled A - H or h_t. These are hard coded constants that every SHA-256 algorithm starts with no matter the input. These inputs are the fractional part of the square root of the first 8 prime numbers.

```
h0 := 0x6a09e667
h1 := 0xbb67ae85
h2 := 0x3c6ef372
h3 := 0xa54ff53a
h4 := 0x510e527f
h5 := 0x9b05688c
h6 := 0x1f83d9ab
h7 := 0x5be0cd19
```

```
-----------
compression: (H0) (initial hash value)
-----------
a = 01101010000010011110011001100111
b = 10111011011001111010111010000101
c = 00111100011011101111001101110010
d = 10100101010011111111101010011010
e = 01010001000011100101001001111111
f = 10011011000001010110100010001100
g = 00011111110000011111011001101011
h = 01011011111000001100110100011001
```

There are 64 other constants taken from the fractional part of the cube root of the first 64 prime numbers. It is denoted with K[0]-K[63] or K_t

$$\sigma_0(x) = ROTR^7(x) \oplus ROTR^{18}(x) \oplus SHR^3(x)$$

$$\sigma_1(x) = ROTR^{17}(x) \oplus ROTR^{19}(x) \oplus SHR^{10}(x)$$

$$Ch(e,f,g) = (e \wedge f) \oplus (\neg e \wedge g)$$

$$Maj(a,b,c) = (a \wedge b) \oplus (a \wedge c) \oplus (b \wedge c)$$

$$\sum_1^{\{256\}}(e) = ROTR^6(e) \oplus ROTR^{11}(e) \oplus ROTR^{25}(e)$$

$$\sum_1^{\{256\}}(a) = ROTR^2(a) \oplus ROTR^{13}(a) \oplus ROTR^{22}(a)$$

There are also 6 functions used in SHA256 besides basic binary addition and XOR. There is **σ**0, **σ**1, **Σ**0, **Σ**1, choice, and majority. The sigma functions are already defined for us and are a combination of rotate right, shift right, and xor operations. The Ch(e,f,g) function refers to choice. The first input determines if it should take the bit from the second input or the third input. Majority function, Maj(a,b,c), just takes the bit that occurs the most out of the three.

Now we can work on the blocks. We need to create a message schedule. We need to create a message schedule of 64 words. We can use the first 256 bits of the block and split it into 16 words to create the first 16 inputs of the message schedule. We generate the rest of the schedule using the previous inputs. The formula is sigma1(t-2) + (t-7) + sigma0(t-15) + (t-16). T corresponds to the index of the array and starts at 16. T gets incremented until it reaches 63.

$$T_1 = \Sigma_1(e) + Ch(e, f, g) + h + K_t + W_t$$

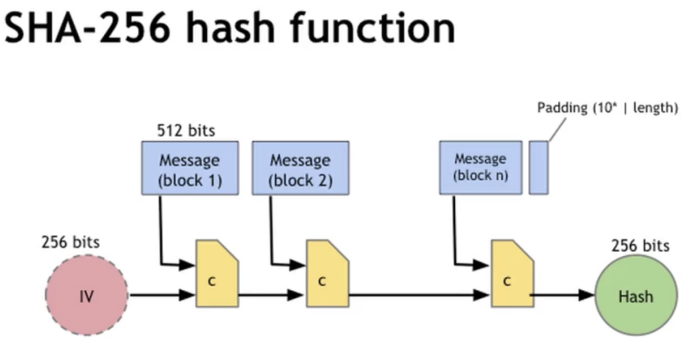
$$T_2 = \Sigma_0(a) + Maj(a, b, c)$$

```
-----------
compression: (H0) (initial hash value)
-----------
a = 01101010000010011110011001100111
b = 10111011011001111010111010000101
c = 00111100011011101111001101110010
d = 10100101010011111111101010011010
e = 01010001000011100101001001111111
f = 10011011000001010110100010001100
g = 00011111110000011111011001101011
h = 01011011111100000110011010011001
```

The next step is the compression. First we get the first input in the message schedule or W0. You also need the first constant or K0. With the state registers and these two numbers you get the first temporary word, T1. T1 = Uppercase Sigma 1(e) + Ch(e,f,g) + h + K0 + W0. Then you do another algorithm for a second temporary word, T2. T2 = Uppercase Sigma 0(a) + Maj(a,b,c). You shift every state register to its next register which leaves the first state register empty. In register A, you add T1 and T2 and store it in the first register. You also add T1 to

register E. You repeat this step until you get through every word in the message schedule and constant array.

The final step is to add the initial registers block to the final register block. Then put all the final bits into one line by concatenating them.



Section 3.2 : Salt Generation

Salt generation is fairly simple. Randomly generated salt can be used for whatever length. The only thing necessary is that the pseudo random number generator is cryptographically secure or CSPRNG for short. CSPRNG stands for cryptographically secure pseudo random number generator. Most languages already have some sort of package or library that provides a CSPRNG. Python has the secrets package, java has java.security.SecureRandom, and C/C++ has CryptGenRandom. We coded a simple python file to demonstrate these functions.

Section 4 : Results and Analysis

You can find our code for this project at this link :

https://github.com/madhurimanohar/PasswordSalting

The output below is just an example of the secrets package in python. The secrets package has a CSPRNG meaning it is a random number generator that is safer to use than normal random generators.

```
Secrets Package - Random Bits
Binary:  0b11010101001110101
Decimal:  109173

Binary:  0b10110010111011110100
Decimal:  732916


EXTRA

Random Package
Random Number from 1 - 50
18
Secrets Package - Random Range
500
Secrets Package - Random Int from 1 - 100
25
Secrets Package - Random Choice from non empty list
[1, 2, 3, 5, 7, 11, 13, 17]
2
Secrets Package - Random Sample from list
[11, 2, 5]
```

```
Secrets Package - Random Bits
Binary:  0b10101111111101110100
Decimal:  720756

Binary:  0b100011110000101000
Decimal:  146472


EXTRA

Random Package
Random Number from 1 - 50
36
Secrets Package - Random Range
300
Secrets Package - Random Int from 1 - 100
17
Secrets Package - Random Choice from non empty list
[1, 2, 3, 5, 7, 11, 13, 17]
5
Secrets Package - Random Sample from list
[2, 3, 17]
```

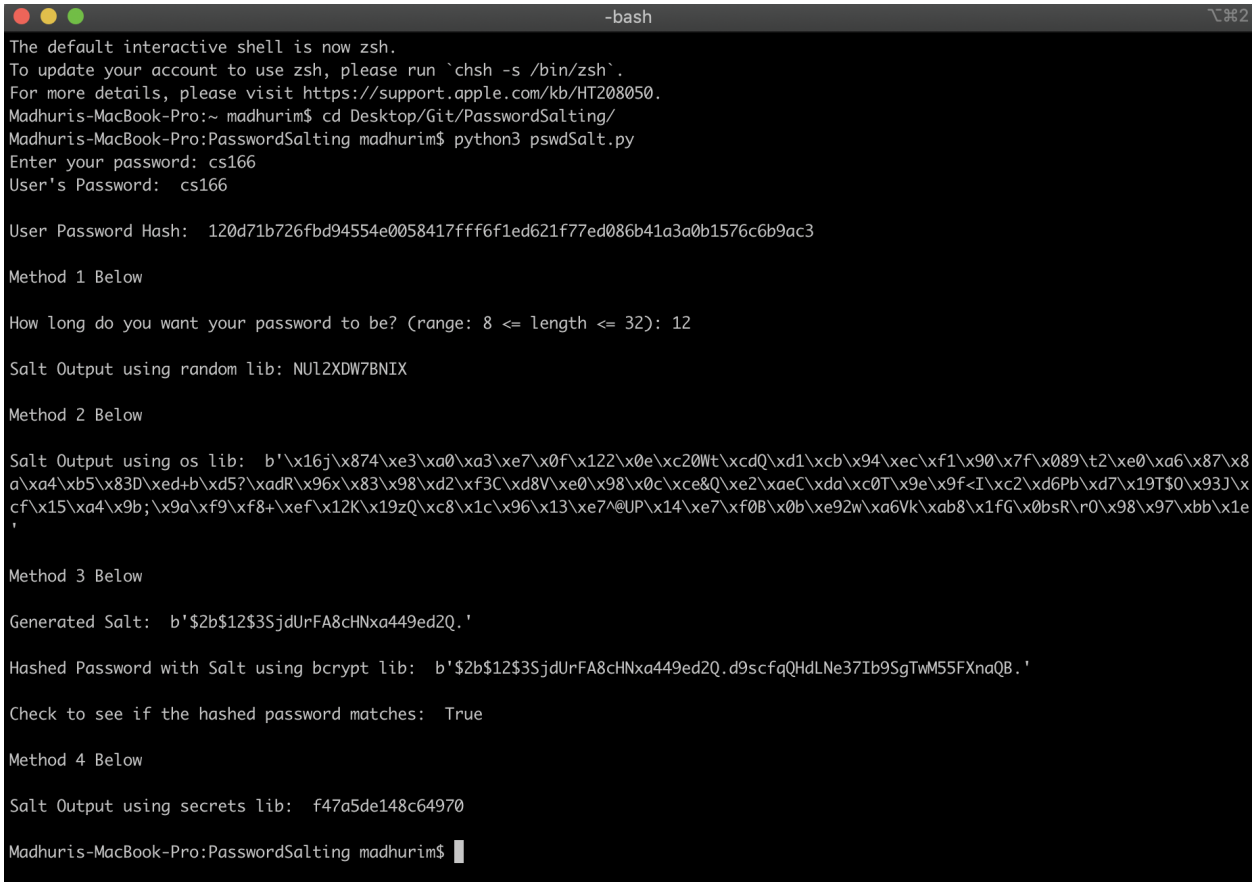
For our code, we have 4 different methods of creating a password. For the first method, the user inputs the desired length of their password and we generate a password for them out of 62 character options. This method isn't ideal because it's easy to crack this kind of a password.

Our second method requires the user to input a password of their choice, which we then hash using the hash function SHA-256. We imported Python's hashlib library to use SHA-256. After encoding the password, we add salt to it to make all the passwords unique. We used Python's os library to generate a random array of characters for the salt. Using os.random() is better than using random(). In python, random() is a pseudorandom number generator (PRNG). The pattern is more repeatable and is also called a deterministic PRNG. On the other hand,

os.random() does not showcase a repeatable pattern and draws it's entropy from many unpredictable sources that helps it to generate a more random array of characters. In terms of cryptography, os.random() is more desirable over random().

Our third method takes the user's inputted password and then generates a salt that is added when hashing the password. It uses a different algorithm than the previous method. We imported Python's bcrypt library to use it's function of salting and hashing. Bcrypt can sometimes be more desirable than using SHA-256 because it forces you to follow security practices since it requires salt while hashing. When combining salt with hashing, it helps protect the user's password from rainbow attacks.

The final method uses a Python library called 'secrets' that generates cryptographically strong random numbers that are then used to generate the password. It uses a function called secrets.token_hex(). The function returns a randomly generated hex string based on the bytes specified by the user.



Section 5 : Future Work

After learning about the different kinds of ways to make passwords more secure, we were also curious about the opposite, how easy or difficult is it for an attacker to retrieve a plaintext password? To start, we would want to create a database that would store the different stages of the password hashing of the plaintext, hashed password, then hashed plus salted password. In doing this, we can evaluate the differences between different hashing algorithms. In addition to creating a database for the different types of hashed passwords, we would want to implement a program that has the ability to brute force attack hashed passwords in order to obtain the plaintext. Not only will this allow us to determine the security reliability of different hashing algorithms, but also illustrate how fast a password can be cracked when comparing a hashed

password versus a salted hashed password. Lastly, in putting all these ideas together, we want to create a web application that will display the different kinds of passwords stored in our database. Not only will this show the user the complexity in the different kinds of password encryptions, we would want to showcase the security of the passwords with the program to brute force attack the passwords. Overall, this application will visually illustrate to the user how salt can make their password more secure against attackers.

Section 6: Conclusion

As discussed throughout this paper, salted passwords are far more secure and less prone to getting hacked as it is incredibly difficult to crack a salted hashed password with traditional hacking techniques like brute force attacks. It would be far easier for them to try and take a crack at some less protected passwords that would return a greater profit to them. We showcased in this paper the pros and cons of salting a hash along with why the SHA-256 algorithm hash function is best used along with salting for best password protection. We hope that our project was educational enough to inform the readers why they should make sure they use a stronger plaintext password and do some research on how the apps they are using are storing their passwords because if its SHA-1 and unsalted, it's just asking for a repeat of the 2012 LinkedIn incident.

Section 7: References

1. Gauravaram, Praveen. "Security Analysis of Salt||Password Hashes." *IEEE Xplore*, https://ieeexplore.ieee.org/abstract/document/6516321.

2. Manjarres, Sam. “2021 World Password Day: How Many Will Be Stolen This Year? - Secplicity - Security Simplified.” *Secplicity*, Secplicity, 4 May 2021, https://www.secplicity.org/2021/05/04/2021-world-password-day-how-many-will-be-stolen-this-year/.

3. OBE, Prof Bill Buchanan. “To Salt or Not to Salt? - Salting Is Not the Only Answer to Securing Passwords.” *Medium*, Coinmonks, 10 Aug. 2018, https://medium.com/coinmonks/to-salt-or-not-to-salt-salting-is-not-the-only-answer-to-securing-passwords-cdab26bd20ad.

4. Shanika, Shanika. “LinkedIn: 2012 Data Breach Much Worse than We Thought.” *CBS News*, CBS Interactive, 19 May 2016, https://www.cbsnews.com/news/linkedin-2012-data-breach-hack-much-worse-than-we-thought-passwords-emails/.

5. *SHA-256 animation*. Sha256 Animation. (n.d.). Retrieved December 3, 2021, from https://awesomeopensource.com/project/in3rsha/sha256-animation.

6. Wagner, Lane. “How Sha-256 Works Step-by-Step.” *Qvault*, 20 Oct. 2021, https://qvault.io/cryptography/how-sha-2-works-step-by-step-sha-256/.

7. Vollebregt, Brent. “How to Hash Passwords in Python.” *Brand Icon*, https://nitratine.net/blog/post/how-to-hash-passwords-in-python/.

8. “Generating Random Passwords.” *Stack Overflow*, 1 Oct. 1956, https://stackoverflow.com/questions/54991/generating-random-passwords.

9. Link to our code : https://github.com/madhurimanohar/PasswordSalting

Section 8: Contributions

Madhuri - Coding, Presentation Slides, Final Report, Research

Cameron - Presentation Slides, Final Report, Research,

Gurteg - Presentation Slides, Final Report, Research, Planning

Christian - Research, Coding, Final Report, Presentation Slides