Graded Assignment: Homework 5

Hiren Shah

4/23/21

University of Maryland Global Campus

SDEV 325 – Detecting Software Vulnerabilities

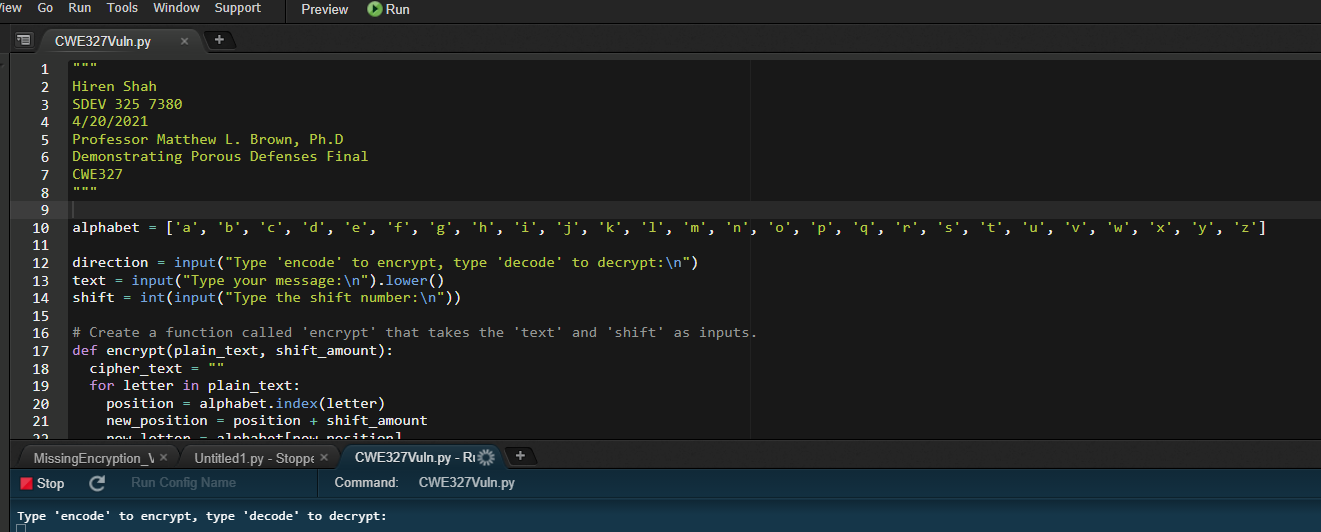
# Executive Summary

For this assignment on Demonstrating Porous Defenses, I chose CWE-327 : Use of Broken or Risky Cryptographic Algorithms and CWE-759 : Use of a One-Way Hash without SALT. I was successful on fixing both the vulnerabilties.

# Example 1 – CWE-327 : Use of Broken or Risky Cryptographic Algorithms

## Overview

This is a python program that uses an encrypt method and asks user to give a clear text, then the program encrypts it by using the Ceaser Cipher algorithm. The program generates the encrypted cipher text.



## Analysis of the Vulnerability

In my example, if an attacker figures out the code above, they can find out the length of the cipher text is the same length of the plain text. Therefore, this may give them an idication that the algorithm used for the encryption maybe a the famous Ceaser Cipher algorithm. By comparing the plain text and the cipher text, an attacker can figure out how many times the new word has shifted, therefore this algorithm is not secure.

**Vulnerable Code:**

*def encrypt(plain\_text, shift\_amount):*

*cipher\_text = ""*

*for letter in plain\_text:*

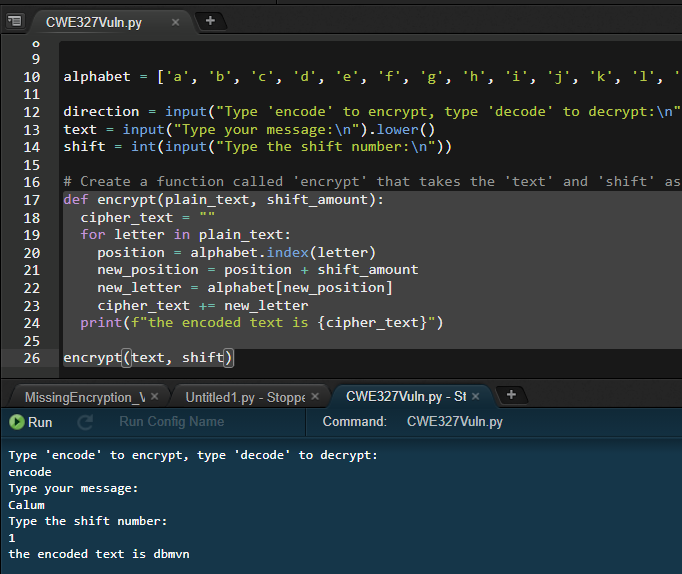
*position = alphabet.index(letter)*

*new\_position = position + shift\_amount*

*new\_letter = alphabet[new\_position]*

*cipher\_text += new\_letter*

*print(f"the encoded text is {cipher\_text}")*



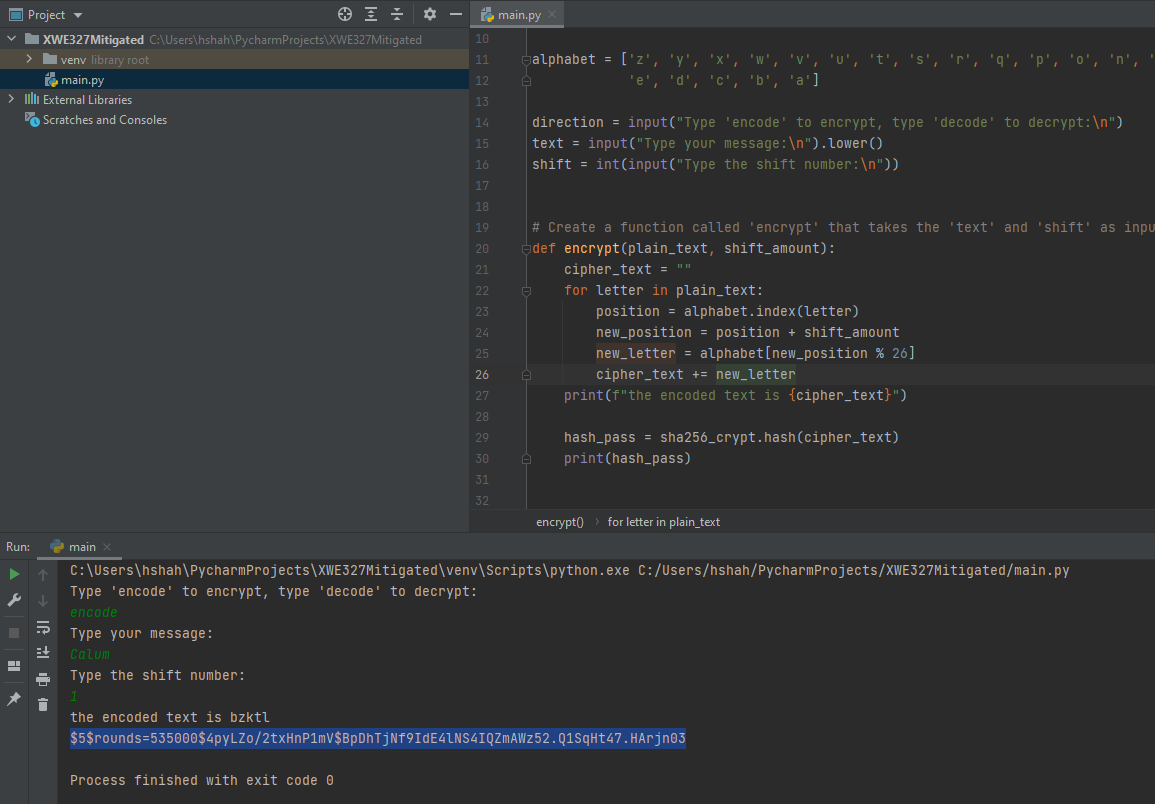
## Mitigation

In the mitigated example of CWE-327, I changed the alphabet from ascending to descending order as to make it more difficult to break the code. In addition, I added sha256 signature method to add a hash function to generate a signature. This way the hash function acts like a signature for the cipher\_text, so an attacker will have no way to get back the original plain text from the hashed output.

alphabet = ['z', 'y', 'x', 'w', 'v', 'u', 't', 's', 'r', 'q', 'p', 'o', 'n', 'm', 'l', 'k', 'j', 'i', 'h', 'g', 'f',  
 'e', 'd', 'c', 'b', 'a']

def encrypt(plain\_text, shift\_amount):  
 cipher\_text = ""  
 for letter in plain\_text:  
 position = alphabet.index(letter)  
 new\_position = position + shift\_amount  
 new\_letter = alphabet[new\_position % 26]  
 cipher\_text += new\_letter  
 print(f"the encoded text is {cipher\_text}")

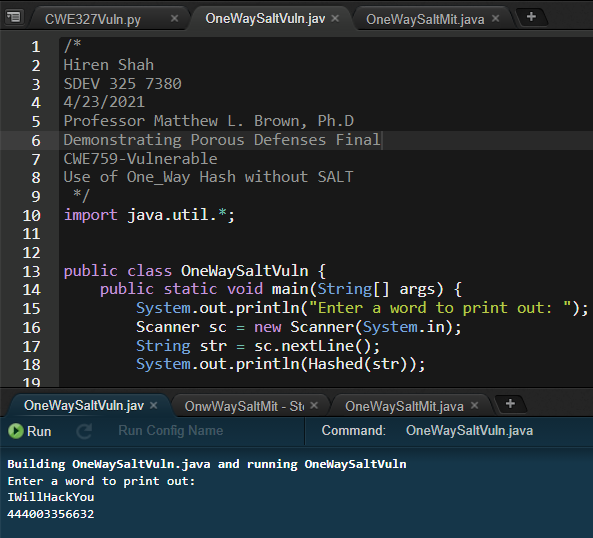
hash\_pass = sha256\_crypt.hash(cipher\_text)  
print(hash\_pass)



# Example 2 – CWE-759 : Use of a One-Way Hash without SALT

## Overview

In my example that is demonstrating a one way hash without SALT, the user is entering a word and the hash method is generating a hash value for that word. The reason that this is vulnerable is because the hashvalue is the same so the attacker can run against a big dictionary of plain text to generate a hash code and try to find a match. The application is written in java.



## Analysis of the Vulnerability

As stated above, this program is vulnerable because the hash value generated is going to be the same, hence an attacker can find a way to get senstivie information. As you can see in the screen shot above the hashed integers for “IwillHackYou” will be the same everytime. So now the attacker can use a dictionary of words to compare the generated hash value to try to find a match.

Vulnerable Code:

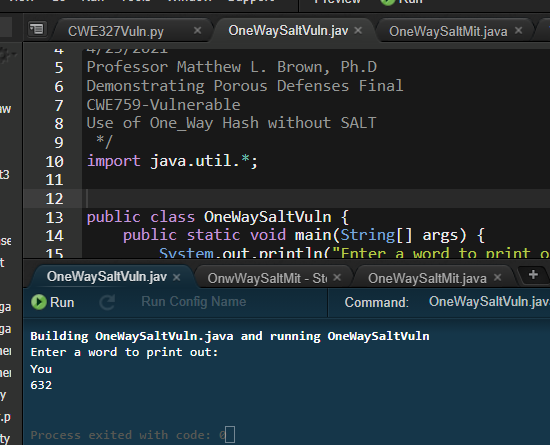
*private static String Hashed(String str) {*

*String result = "";*

*for(int i=0; i<str.length(); i++) {*

*result += Character.getNumericValue(str.charAt(i)) % 7;*

*}*

Showing “You” will always equal ‘632’

## Mitigation

In the mitigated version we are using the secure random generator in our code to generate a random binary byte array as a SALT, then we use that SALT combined with the user entered text to generate the hash.

private static String Hashed(String str) {

String result = "";

try {

str = getSalt() + str + getSalt();

}

catch(Exception e) {

e.printStackTrace();

}

String newStr = "";

for (int i=0; i<str.length(); i++) {

if (i % 2 == 1) {

newStr += str.charAt(str.length()-i);

}

else {

newStr = str.charAt(i) + newStr;

}

}

for(int i=0; i<newStr.length(); i++) {

result += Character.getNumericValue(newStr.charAt(i)) % 7;

}

return result;

}

private static String getSalt() throws NoSuchAlgorithmException {

try {

SecureRandom secureRand = SecureRandom.getInstance("SHA1PRNG");

//create array for the salt (16 bytes length)

byte[] salt = new byte[16];

secureRand.nextBytes(salt);

return new String(salt);

}

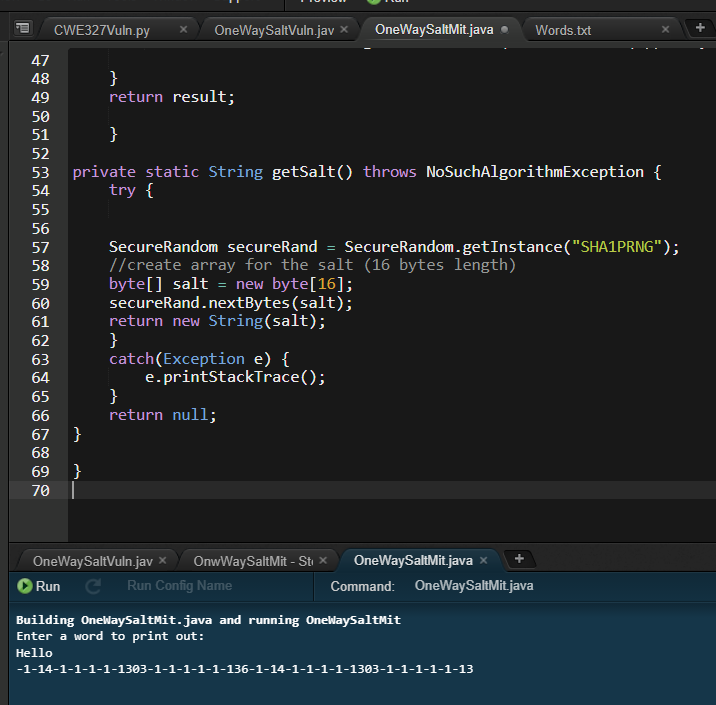
catch(Exception e) {

e.printStackTrace();

}

return null;

}



# References