Holden Stefan
10/12/2021

Demonstrating Improper Cryptography

- "CWE-759: Use of a One-Way Hash Without a Salt"
- "CWE-327: Use of a Broken or Risky Cryptographic Algorithm"

In the case of a hash without salt, this is the concept of encrypting sensitive data, such as passwords, using a cryptographic algorithm to produce a hash value that should be computationally impractical to reverse. This practice is widely accepted and used to store sensitive data, so that in cases where a database is compromised, the stored data is still securely encrypted. It is important to understand that this strategy is not bulletproof, especially in the case of "rainbow tables" where people have compiled lists of hundreds of millions of words, phrases, and common passwords, and run this list through hashing algorithms. In the end, there are essentially hundreds of millions of compromised hash values, and as an algorithm will always produce the same hash if given the same input, this means that there are essentially hundreds of millions of compromised passwords.

There are a few mitigations for rainbow tables, the first being the practice of salting. This entails appending or prepending a string to a password before it is hashed, with the idea being that this random string will not appear in rainbow tables. It is important to follow a few best-practices when salting, including randomly generating the salt for every password that gets stored (rather than hard-coding the salt), and usually storing that salt in the same or separate database where it can be retrieved to use in logging the user in. Secondly, depending on the hashing algorithm used, it is a good idea to make the salt long enough to both make it more difficult to guess, and decrease the likelihood of reusing the same salt.

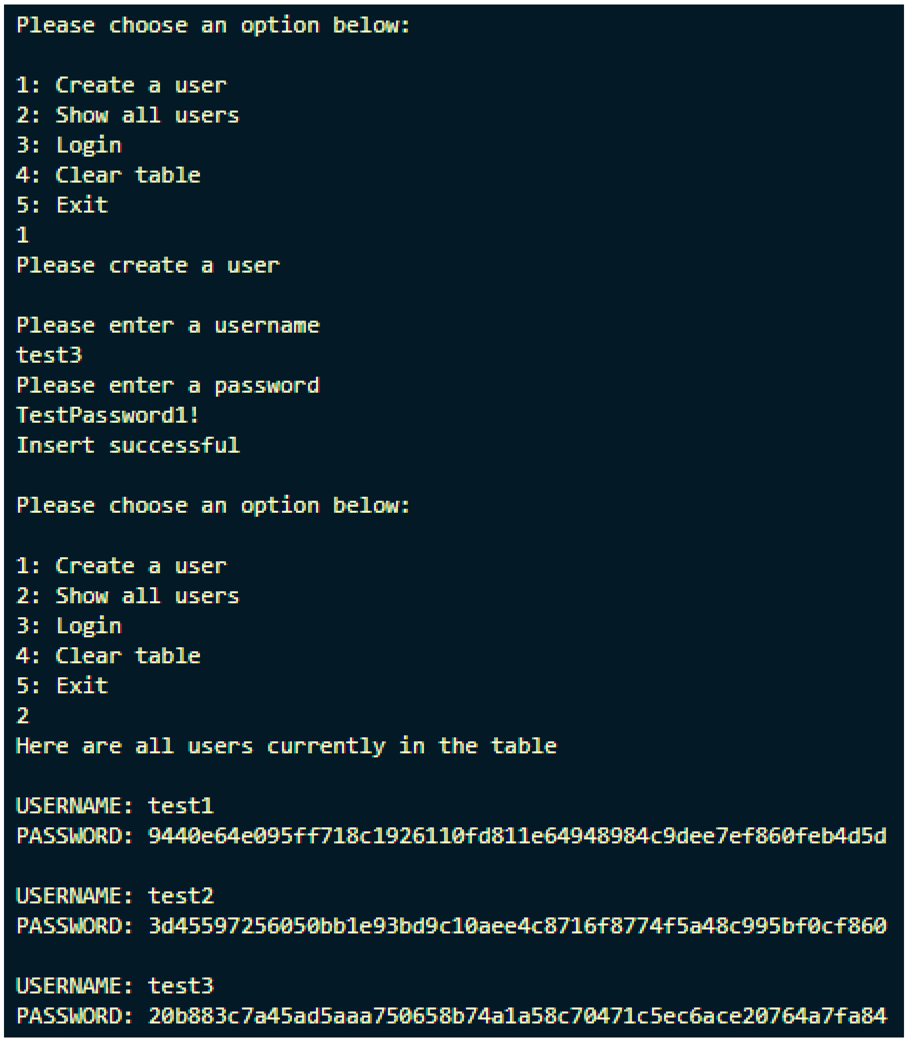
When it comes to the second weakness, using a broken cryptographic algorithm, it is a much more basic explanation, and also much easier to mitigate in most cases. Simply put, as algorithms age, and computational power becomes cheaper, older algorithms turn out to have weaknesses. In general, these weaknesses lead to what are called collisions, which is where separate inputs create the same hash digest. Theoretically, all hashing algorithms have this weakness, and it really depends on the size of the digest produced by the algorithm (MD5=2^{128}, SHA-1=2^{160}, bcrypt=2^{192}, and so on). This is known as the "pigeonhole principle", and extended in the "birthday problem". The real threat in using a broken algorithm is not necessarily that two random strings/files will happen to create the same digest, but that with enough time and cpu cores, one file can be engineered to produce the same hash as a second, and essentially mimic that file. This was used in the Flame malware, where the malware was signed with a forged CA certificate that produced the same MD5 hash value and was recognized by Windows as a legitimate Microsoft program.

In order to mitigate this weakness, developers simply have to use newer and stronger algorithms. Of the most common algorithms used in the past 2 decades, MD4, MD5, as well as Sha-1, are all considered compromised. There are plenty of newer algorithms that as of right now are safe, including Sha-2, Sha-3, bcrypt, RSA, AES, and TwoFish. The only challenge in mitigating this weakness is with legacy software, in which the encryption may be a cornerstone in the program, and thus updating to a new algorithm is not so simple. To deal with this, there is the strategy of counter-cryptanalysis, in which collisions can be detected. It is not the most secure or reliable mitigation, but is a good idea to put in place until the broken algorithm can be updated to a secure algorithm.

Example 1- [CWE-759: Use of a One-Way Hash Without a Salt]

Overview:

For this demonstration a basic authentication app was created using python, in which a menu displays options to add a user, login, show all users, and clear the database. When adding users, a username is entered along with a password which gets hashed using Sha-224, before they are both stored in a sqlite3 database. The display all users options kind of simulates a breached database, giving access to the usernames and the hashed passwords. The login option will ask for a username and password (which gets hashed using Sha-224) and compares them to the values stored in the database to determine if the login was successful or not.

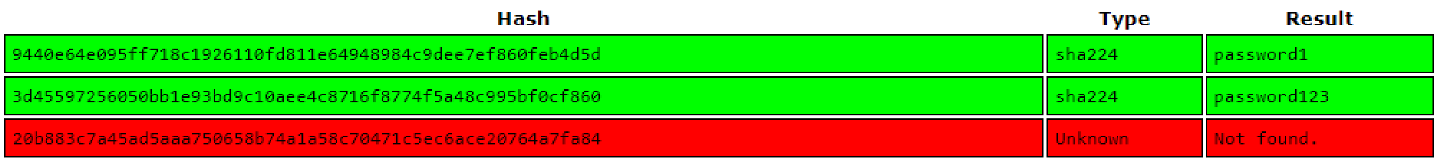


```
Please choose an option below:

1: Create a user
2: Show all users
3: Login
4: Clear table
5: Exit
1
Please create a user

Please enter a username
test3
Please enter a password
TestPassword1!
Insert successful

Please choose an option below:

1: Create a user
2: Show all users
3: Login
4: Clear table
5: Exit
2
Here are all users currently in the table

USERNAME: test1
PASSWORD: 9440e64e095ff718c1926110fd811e64948984c9dee7ef860feb4d5d

USERNAME: test2
PASSWORD: 3d45597256050bb1e93bd9c10aee4c8716f8774f5a48c995bf0cf860

USERNAME: test3
PASSWORD: 20b883c7a45ad5aaa750658b74a1a58c70471c5ec6ace20764a7fa84
```

Screenshot of adding users and showing list of all users

Analysis:

This program is vulnerable to rainbow tables or dictionary attacks, as no salt is used, so any weak password has likely already had the hash digest published somewhere. In this example a website called CrackStation was used to see if the hash values in the database have been leaked previously.

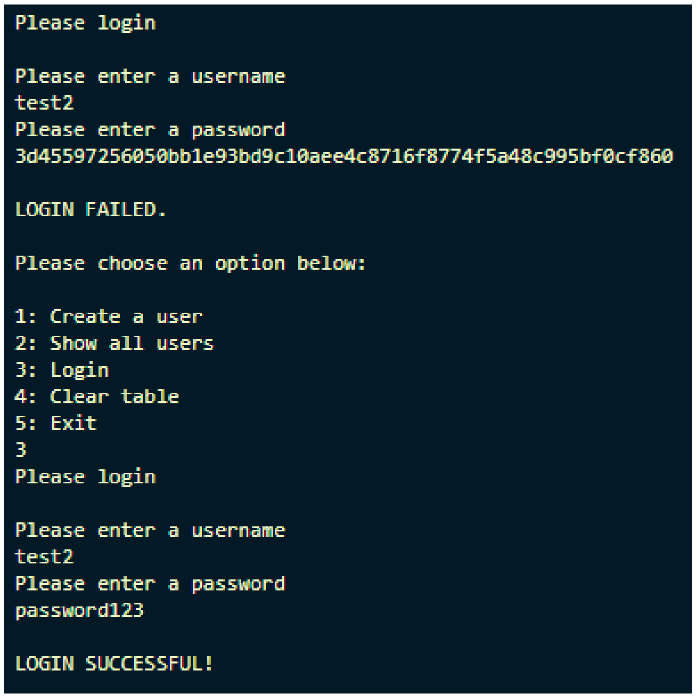


Hash	Type	Result
9440e64e095ff718c1926110fd811e64948984c9dee7ef860feb4d5d	sha224	password1
3d45597256050bb1e93bd9c10aee4c8716f8774f5a48c995bf0cf860	sha224	password123
20b883c7a45ad5aaa750658b74a1a58c70471c5ec6ace20764a7fa84	Unknown	Not found.

In the case of the three test users that were added:

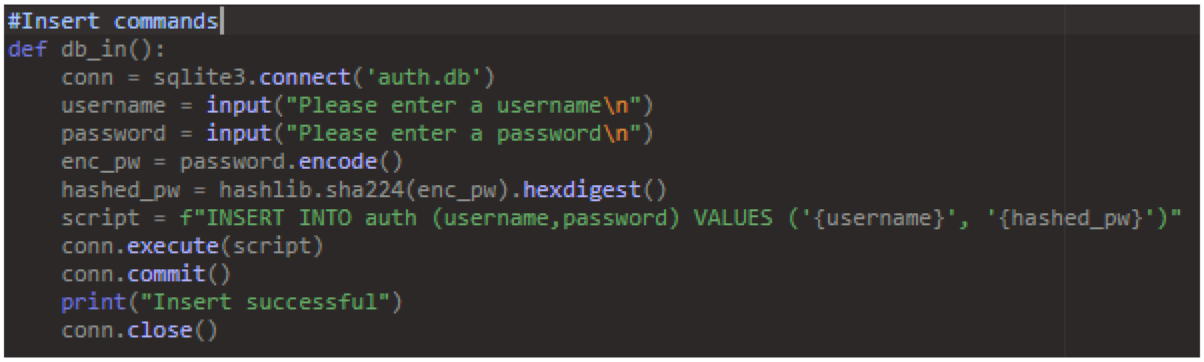
- Username: test1, Password: password1
- Username: test2, Password: password123
- Username: test3, Password: TestPassword1!

The first 2 passwords were easily found, and upon using them to login, were successful.



```
Please login

Please enter a username
test2
Please enter a password
3d45597256050bb1e93bd9c10aee4c8716f8774f5a48c995bf0cf860

LOGIN FAILED.

Please choose an option below:

1: Create a user
2: Show all users
3: Login
4: Clear table
5: Exit
3
Please login

Please enter a username
test2
Please enter a password
password123

LOGIN SUCCESSFUL!
```

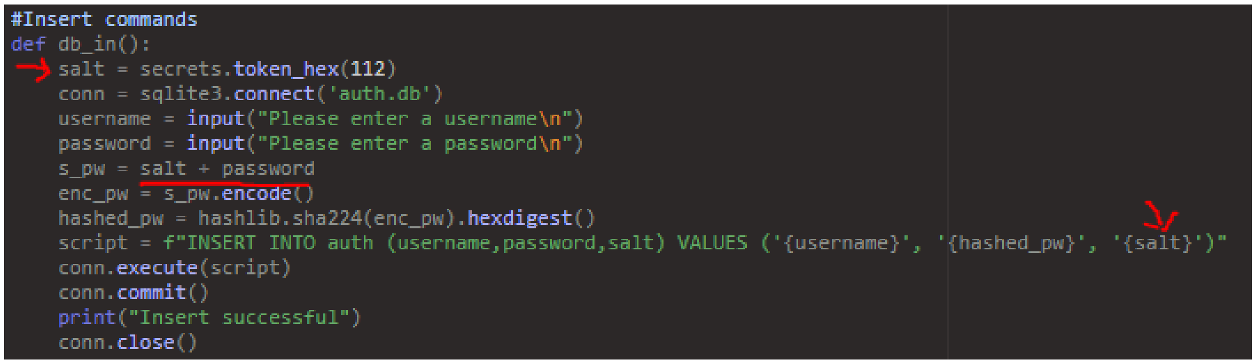
In the vulnerable function, there is not much to look at, due to the fact that the only real significance is the complete lack of salting functionality.



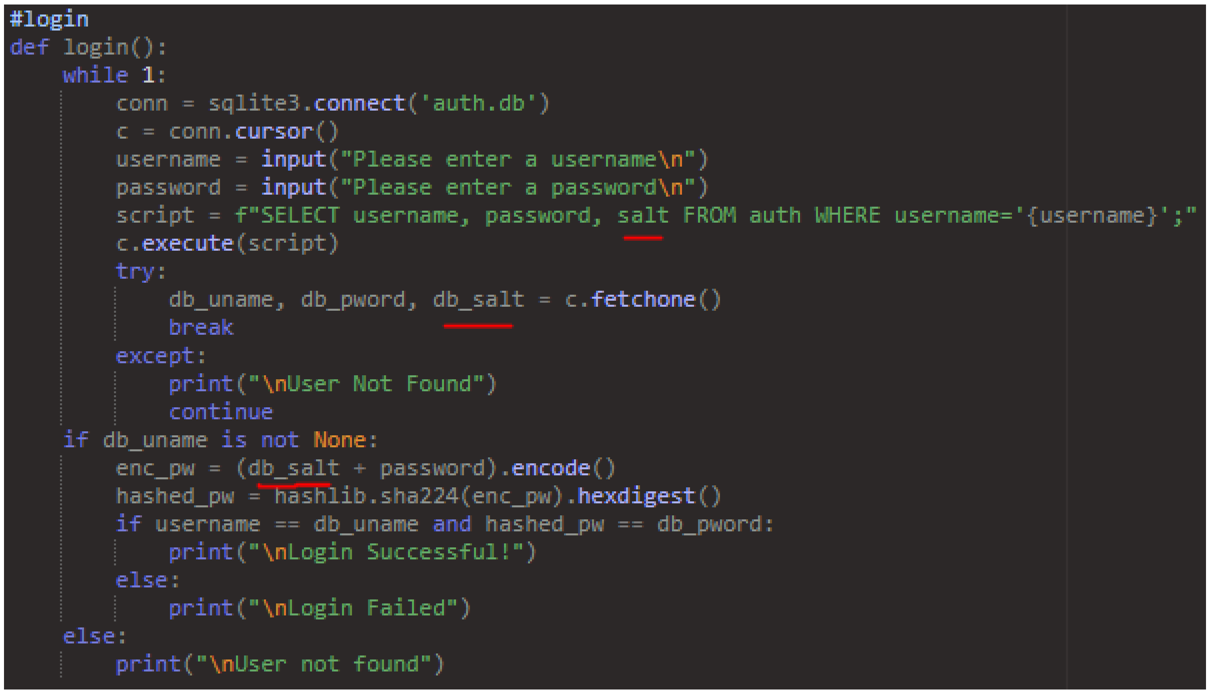
```python
#Insert commands
def db_in():
    conn = sqlite3.connect('auth.db')
    username = input("Please enter a username\n")
    password = input("Please enter a password\n")
    enc_pw = password.encode()
    hashed_pw = hashlib.sha224(enc_pw).hexdigest()
    script = f"INSERT INTO auth (username,password) VALUES ('{username}', '{hashed_pw}')"
    conn.execute(script)
    conn.commit()
    print("Insert successful")
    conn.close()
```

Mitigation:

In order to mitigate this weakness, in short, a randomly generated salt was prepended to the given password, and the salt itself was stored alongside the user information in the database. In this case, it is not considered too risky to store the salt in the same place as the password, because even with both the hash and the salt, it will be incredibly computationally expensive to solve, especially as each password gets a unique 112 character salt. This salt could be smaller, but the 2 important factors are that the salt is both long enough to make sure it is unique, and that each password gets a separately generated salt, so there is no universal salt. It made the authentication function a little bit more complicated, as the salt had to be fetched before the passwords could be compared, and this makes it important to ensure unique usernames are used, as it is possible to get a different user's information if the usernames are identical.

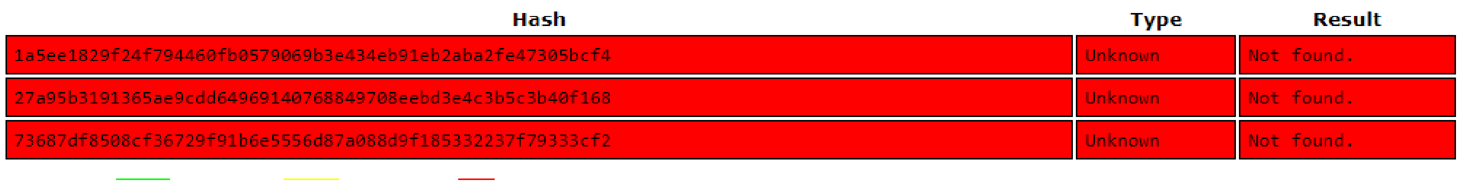


```python
#Insert commands
def db_in():
    salt = secrets.token_hex(112)
    conn = sqlite3.connect('auth.db')
    username = input("Please enter a username\n")
    password = input("Please enter a password\n")
    s_pw = salt + password
    enc_pw = s_pw.encode()
    hashed_pw = hashlib.sha224(enc_pw).hexdigest()
    script = f"INSERT INTO auth (username,password,salt) VALUES ('{username}', '{hashed_pw}', '{salt}')"
    conn.execute(script)
    conn.commit()
    print("Insert successful")
    conn.close()
```

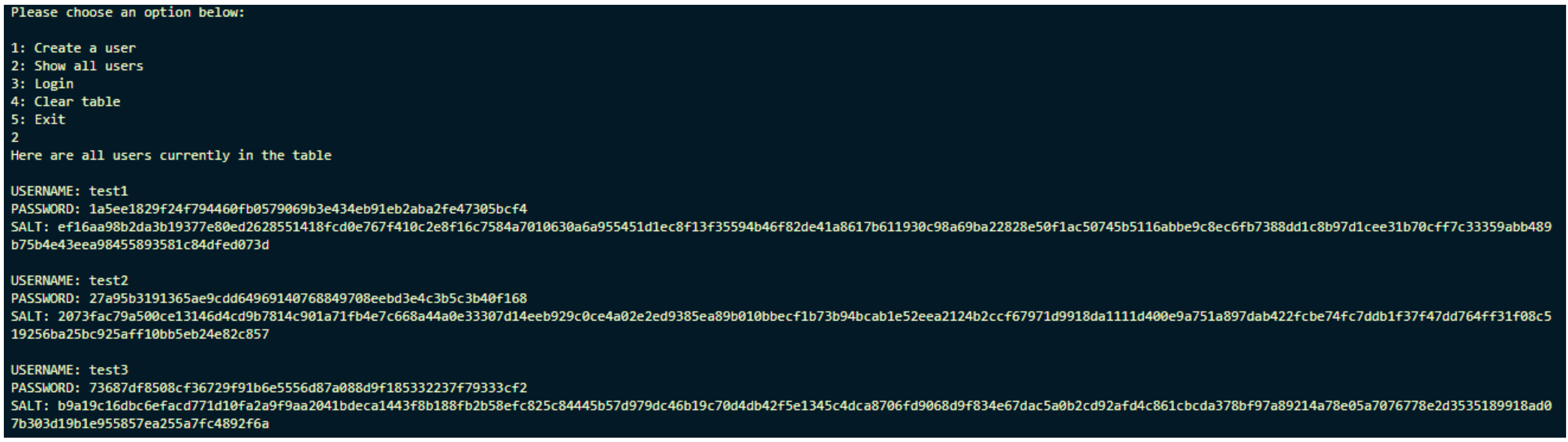


```python
#login
def login():
    while 1:
        conn = sqlite3.connect('auth.db')
        c = conn.cursor()
        username = input("Please enter a username\n")
        password = input("Please enter a password\n")
        script = f"SELECT username, password, salt FROM auth WHERE username='{username}';"
        c.execute(script)
        try:
            db_uname, db_pword, db_salt = c.fetchone()
            break
        except:
            print("\nUser Not Found")
            continue
    if db_uname is not None:
        enc_pw = (db_salt + password).encode()
        hashed_pw = hashlib.sha224(enc_pw).hexdigest()
        if username == db_uname and hashed_pw == db_pword:
            print("\nLogin Successful!")
        else:
            print("\nLogin Failed")
    else:
        print("\nUser not found")
```

When the database was shown (showing the same exact username/password combos used previously), and the hashes were searched, no matches were found, demonstrating this strategy's resistance to dictionary attacks.



Hash	Type	Result
1a5ee1829f24f794460fb0579069b3e434eb91eb2aba2fe47305bcf4	Unknown	Not found.
27a95b3191365ae9cdd64969140768849708eebd3e4c3b5c3b40f168	Unknown	Not found.
73687df8508cf36729f91b6e5556d87a088d9f185332237f79333cf2	Unknown	Not found.



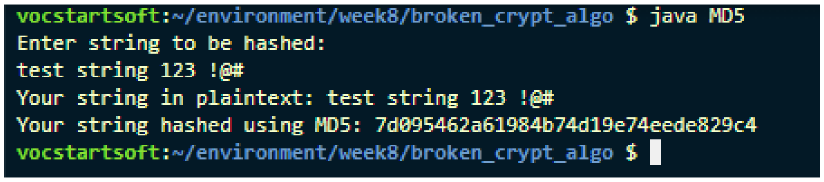
```
Please choose an option below:

1: Create a user
2: Show all users
3: Login
4: Clear table
5: Exit
2
Here are all users currently in the table

USERNAME: test1
PASSWORD: 1a5ee1829f24f794460fb0579069b3e434eb91eb2aba2fe47305bcf4
SALT: ef16aa98b2da3b19377e80ed2628551418fcd0e767f410c2e8f16c7584a7010630a6a955451d1ec8f13f35594b46f82de41a8617b611930c98a69ba22828e50f1ac50745b5116abbe9c8ec6fb7388dd1c8b97d1cee31b70cff7c33359abb489
b75b4e43eea98455893581c84dfed073d

USERNAME: test2
PASSWORD: 27a95b3191365ae9cdd6496914076884970 8eebd3e4c3b5c3b40f168
SALT: 2073fac79a500ce13146d4cd9b7814c901a71fb4e7c668a44a0e33307d14eeb929c0ce4a02e2ed9385ea89b010bbecf1b73b94bcab1e52eea2124b2ccf67971d9918da1111d400e9a751a897dab422fcbe74fc7ddb1f37f47dd764ff31f08c5
19256ba25bc925aff10bb5eb24e82c857

USERNAME: test3
PASSWORD: 73687df8508cf36729f91b6e5556d87a088d9f185332237f79333cf2
SALT: b9a19c16dbc6efacd771d10fa2a9f9aa2041bdeca1443f8b188fb2b58efc825c84445b57d979dc46b19c70d4db42f5e1345c4dca8706fd9068d9f834e67dac5a0b2cd92afd4c861cbcda378bf97a89214a78e05a7076778e2d3535189918ad0
7b303d19b1e955857ea255a7fc4892f6a
```

User list, now displays USERNAME, PASSWORD, and SALT values

Example 2-[CWE-327: Use of a Broken or Risky Cryptographic Algorithm]

Overview:

For this demonstration, I had a hard time coming up with a practical use case for encryption, so I ended up making a simple Java program that will generate a hash for a user-provided string. The broken algorithm that was used was MD5. Strictly speaking, MD5 is only technically broken in certain respects, specifically when it comes to collisions, which are not a major concern when it comes to things like passwords, but are a concern when it comes to things like checksums. Regardless of this fact, it is clear that MD5 has been thoroughly weakened, and should be avoided for this reason in addition to the fact that it is quite fast. In simpler terms, there are much newer and more suitable cryptographic algorithms available.

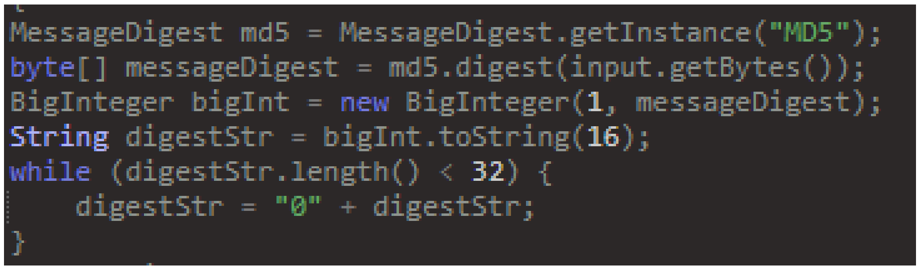


```
vocstartsoft:~/environment/week8/broken_crypt_algo $ java MD5
Enter string to be hashed:
test string 123 !@#
Your string in plaintext: test string 123 !@#
Your string hashed using MD5: 7d095462a61984b74d19e74eede829c4
vocstartsoft:~/environment/week8/broken_crypt_algo $
```

Analysis:

Initially, I had planned to use this program to generate checksums for files rather than Strings, however, I could not get the file structure sorted out in Cloud9. If I had been successful, this would have been a more interesting example, as it is relatively straightforward to generate 2 distinct files that have the same MD5 digest, using available tools.

Regardless, the vulnerable function in either case would have simply been the implementation of MD5, as it is a weakened algorithm.

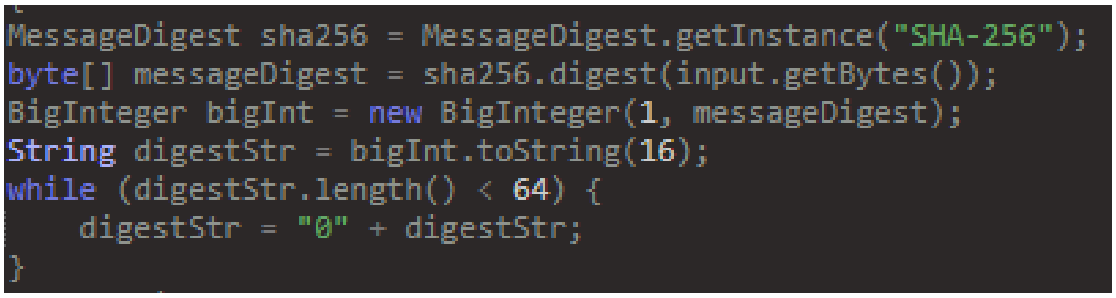


```java
MessageDigest md5 = MessageDigest.getInstance("MD5");
byte[] messageDigest = md5.digest(input.getBytes());
BigInteger bigInt = new BigInteger(1, messageDigest);
String digestStr = bigInt.toString(16);
while (digestStr.length() < 32) {
    digestStr = "0" + digestStr;
}
```

This implementation uses the Java library MessageDigest, which is also capable of generating Sha-256 hashes, using almost identical code. So especially in this case, it just does not make sense to use MD5.

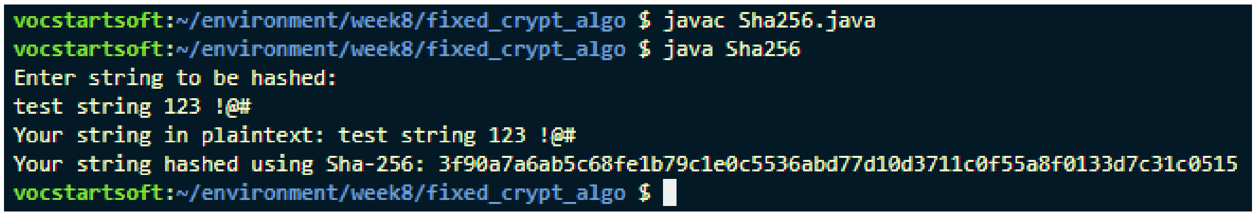
Mitigation:

To mitigate this weakness, I simply exchanged MD5 for Sha-256.



```
MessageDigest sha256 = MessageDigest.getInstance("SHA-256");
byte[] messageDigest = sha256.digest(input.getBytes());
BigInteger bigInt = new BigInteger(1, messageDigest);
String digestStr = bigInt.toString(16);
while (digestStr.length() < 64) {
    digestStr = "0" + digestStr;
}
```

I was able to do this extremely easily, only changing the MessageDigest Instance from “MD5” to “SHA-256”, and additionally doubling the length of the digest and printable string lengths, as Sha-256 creates a 256 bit digest, rather than the 128 bit digest created by MD5. It runs exactly the same as before, but is significantly more secure and reliable with the updated algorithm.



```
vocstartsoft:~/environment/week8/fixed_crypt_algo $ javac Sha256.java
vocstartsoft:~/environment/week8/fixed_crypt_algo $ java Sha256
Enter string to be hashed:
test string 123 !@#
Your string in plaintext: test string 123 !@#
Your string hashed using Sha-256: 3f90a7a6ab5c68fe1b79c1e0c5536abd77d10d3711c0f55a8f0133d7c31c0515
vocstartsoft:~/environment/week8/fixed_crypt_algo $
```

Had I been successful in generating checksums for files, at this point in time, it would be nearly impossible to generate two distinct files that produce the same hash value.

References

Arias, D. (2021, February 25). *Adding salt to hashing: A better way to store passwords*. Auth0. Retrieved October 12, 2021, from https://auth0.com/blog/adding-salt-to-hashing-a-better-way-to-store-passwords/.

Bellore, A. (2021, March 23). *Birthday attacks, collisions, and password strength*. Auth0. Retrieved October 12, 2021, from https://auth0.com/blog/birthday-attacks-collisions-and-password-strength/.

Free password hash cracker. ▼CrackStation. (2019, May 27). Retrieved October 12, 2021, from https://crackstation.net/.

MITRE. (2006, July 19). *Common weakness enumeration*. CWE. Retrieved October 12, 2021, from https://cwe.mitre.org/data/definitions/327.html.

MITRE. (2009, March 3). *Common weakness enumeration*. CWE. Retrieved October 12, 2021, from https://cwe.mitre.org/data/definitions/759.html.

Wang, X., & Yu, H. (2005). How to break MD5 and other hash functions. *Lecture Notes in Computer Science*, 19–35. https://doi.org/10.1007/11426639_2