LAB 4 – SALT

Student Name: Eryk Gloginski **(L00157413)**

Course: BSc Computing

Module: Secure Programming

Lecturer: Maria Griffin

Submission Date: 22/03/2023

1. Introduction:

Salting is a well-known and often used security technique in programming that adds an extra layer of protection to user passwords. Salting involves adding a random string of characters to each password before encrypting and storing it. When a user logs in, the system retrieves the salt from the database and adds it to the user's entered password.

2. Aims:

The aims of this lab are to:

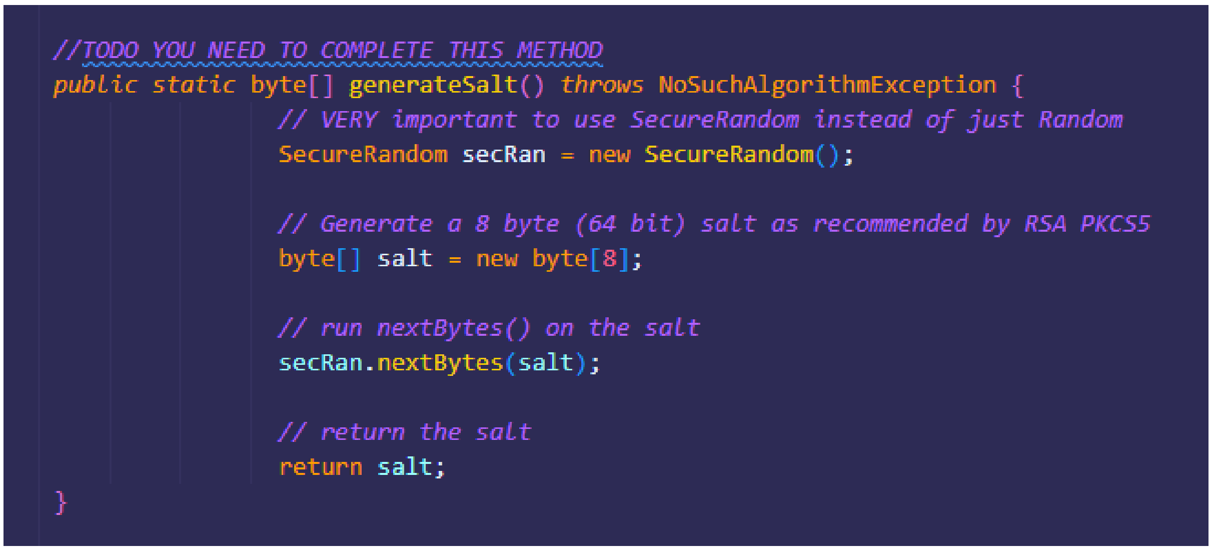
- Use the code provided for the class **PasswordEncryptionService.java**.
- Complete the code in the **generateSalt()** method to generate a salt.
- Add test code to the **main()** method to encrypt the password and check if the authentication matches the attempted password.

3. Method:

The method that we will be using for salting is called "**PBKDF2WithHmacSHA1**". "**PBKDF2**" stands for Password-Based Key Derivation Function 2 while "**WithHmacSHA1**" is a cryptographic algorithm used to hash the password with the salt. Together, it is a key derivation function that takes the password and salt as an input and produces a derived key as output.

4. Results:

We start by completing the **generateSalt()** method so it returns a **byte[] salt**. We use the **SecureRandom** class to create a **secRan** object, define a **byte[] salt** variable by giving it 8 bytes and then run the **nextBytes()** method on the **salt**. With that, we can generate and return a **salt**. The code in the method below is for generating the salt.



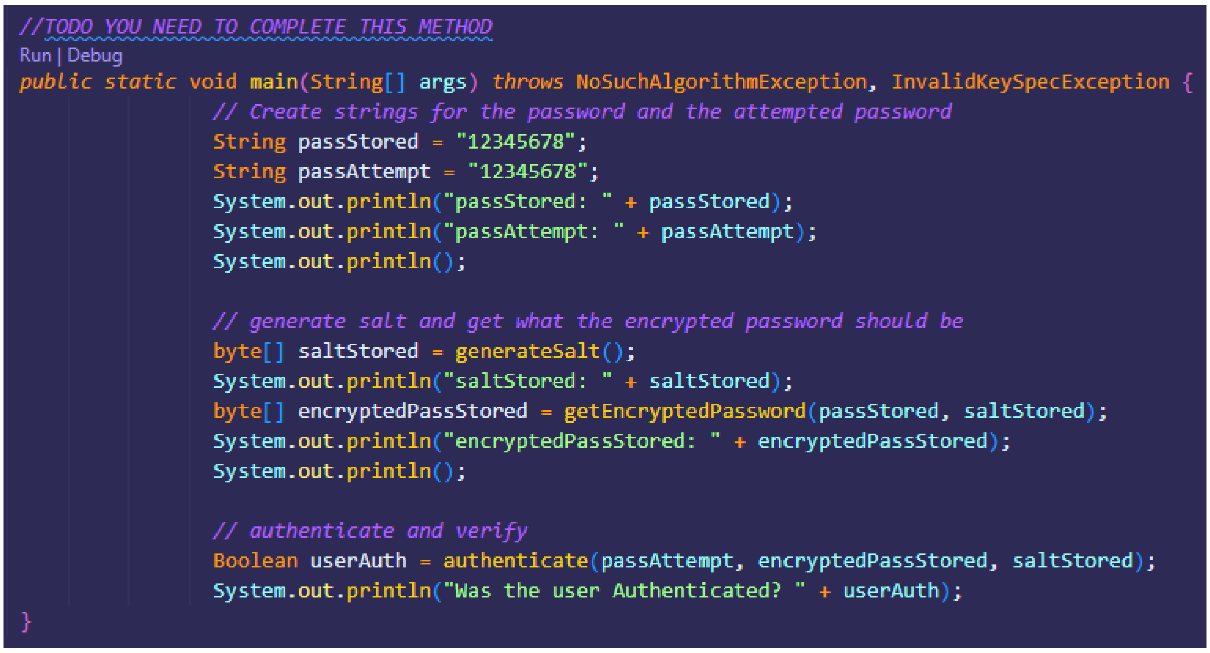
```java
//TODO YOU NEED TO COMPLETE THIS METHOD
public static byte[] generateSalt() throws NoSuchAlgorithmException {
            // VERY important to use SecureRandom instead of just Random
            SecureRandom secRan = new SecureRandom();

            // Generate a 8 byte (64 bit) salt as recommended by RSA PKCS5
            byte[] salt = new byte[8];

            // run nextBytes() on the salt
            secRan.nextBytes(salt);

            // return the salt
            return salt;
}
```

After that, we make the **test main()** method where we test out all the methods, starting with defining variables for the **stored** password, **attempted** password. We generate the salt using the **generateSalt**() method and we print out the stored salt. The code below is to check if the **passStored** and **passAttempt** store the same values.

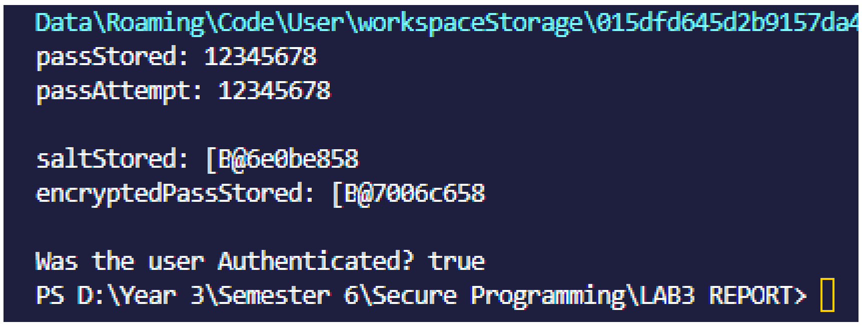


```
//TODO YOU NEED TO COMPLETE THIS METHOD
Run | Debug
public static void main(String[] args) throws NoSuchAlgorithmException, InvalidKeySpecException {
            // Create strings for the password and the attempted password
            String passStored = "12345678";
            String passAttempt = "12345678";
            System.out.println("passStored: " + passStored);
            System.out.println("passAttempt: " + passAttempt);
            System.out.println();

            // generate salt and get what the encrypted password should be
            byte[] saltStored = generateSalt();
            System.out.println("saltStored: " + saltStored);
            byte[] encryptedPassStored = getEncryptedPassword(passStored, saltStored);
            System.out.println("encryptedPassStored: " + encryptedPassStored);
            System.out.println();

            // authenticate and verify
            Boolean userAuth = authenticate(passAttempt, encryptedPassStored, saltStored);
            System.out.println("Was the user Authenticated? " + userAuth);
}
```

We can see below the output of the code in the main method. It prints out the **passStored** and **passAttempt** to verify that they are indeed the same values. After it prints out the generated salt as **saltStored** and it encrypts the password using that generated salt with the **getEncryptedPassword()** method, which is also printed out. Lastly, it verifies if the user has been authenticated using the **authenticate()** method which passes the **passAttempt**, **encryptedPassStored** and **saltStored** and returns a Boolean.

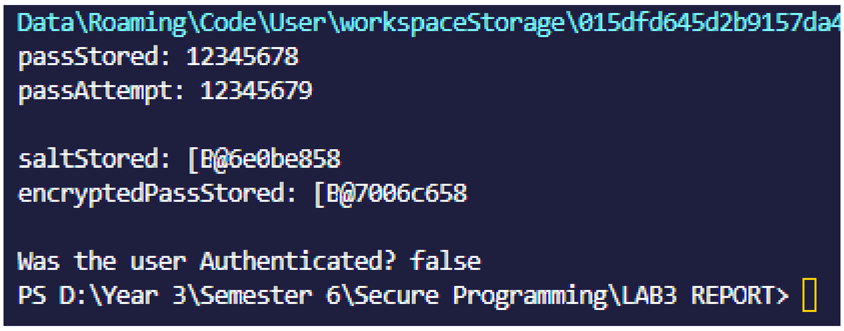


```
Data\Roaming\Code\User\workspaceStorage\015dfd645d2b9157da4
passStored: 12345678
passAttempt: 12345678

saltStored: [B@6e0be858
encryptedPassStored: [B@7006c658

Was the user Authenticated? true
PS D:\Year 3\Semester 6\Secure Programming\LAB3 REPORT>
```

However, what happens if **passStored** and **passAttempt** are not the same values? When that happens, the **authenticate()** method takes care of it and returns a false, rather than a true.

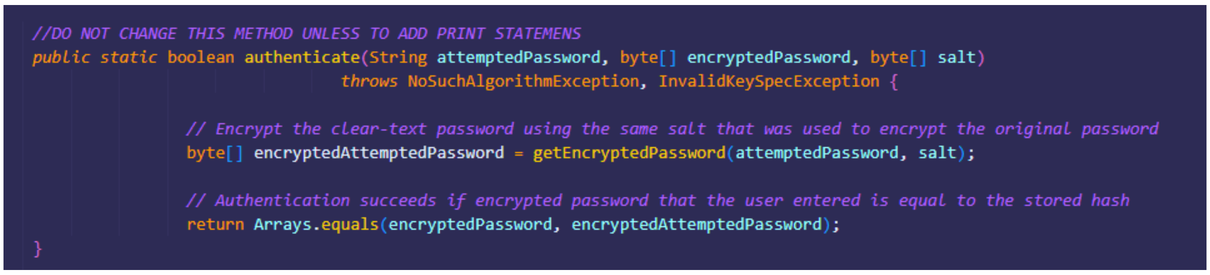


```
Data\Roaming\Code\User\workspaceStorage\015dfd645d2b9157da4
passStored: 12345678
passAttempt: 12345679

saltStored: [B@6e0be858
encryptedPassStored: [B@7006c658

Was the user Authenticated? false
PS D:\Year 3\Semester 6\Secure Programming\LAB3 REPORT>
```

The code below is a snippet of the provided **authenticate()** method. We can see here that the **attemptedPassword** and **salt** parameters are being used in the **getEncryptedPassword()** method to retrieve an **encryptedAttemptedPassword**. After that it returns a Boolean with the **Arrays.equals()** method using **attemptedPassword** and **encryptedAttemptedPassword**.

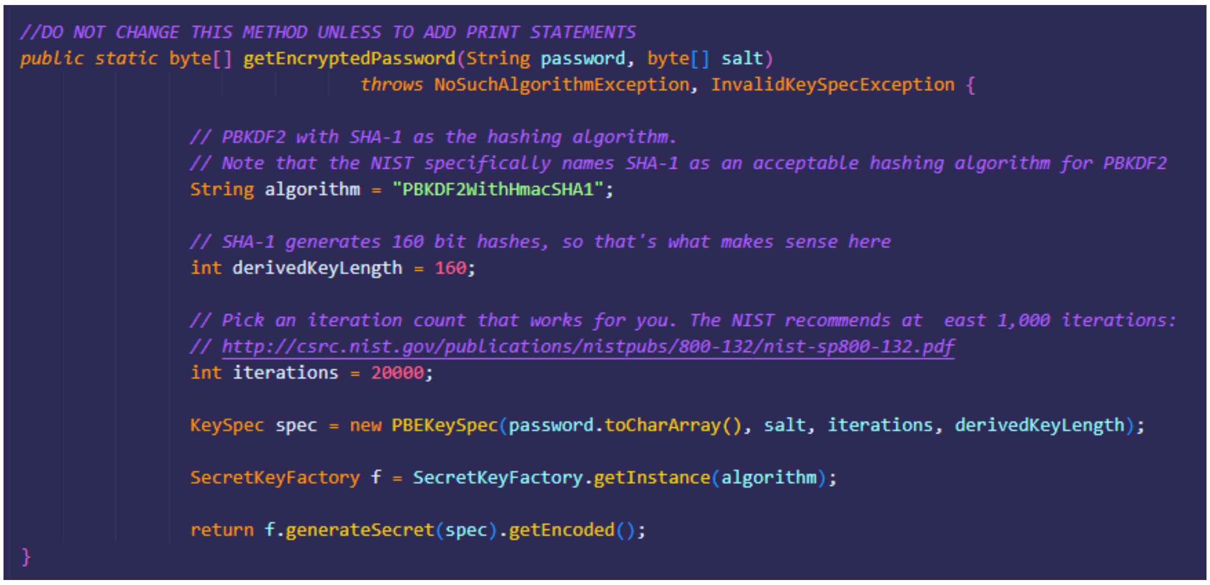


```
//DO NOT CHANGE THIS METHOD UNLESS TO ADD PRINT STATEMENS
public static boolean authenticate(String attemptedPassword, byte[] encryptedPassword, byte[] salt)
                                throws NoSuchAlgorithmException, InvalidKeySpecException {

                // Encrypt the clear-text password using the same salt that was used to encrypt the original password
                byte[] encryptedAttemptedPassword = getEncryptedPassword(attemptedPassword, salt);

                // Authentication succeeds if encrypted password that the user entered is equal to the stored hash
                return Arrays.equals(encryptedPassword, encryptedAttemptedPassword);
}
```

The code below is also a snipped of the provided **getEncryptedPassword()** method. We can see here that there are predefined **algorithm**, **derivedKeyLength** and **iterations** variables. It gets the **KeySpec spec** object by using the **password**, **salt**, **iterations** and **derivedKeyLength** variables. The **SecretKeyFactory** object is made using the **getInstance()** method and using the **algorithm** variable as a parameter. Finally, it returns the **SecretKeyFactory** object which uses the **generateSecret()** method which also uses the **getEncoded()** method.



```
//DO NOT CHANGE THIS METHOD UNLESS TO ADD PRINT STATEMENTS
public static byte[] getEncryptedPassword(String password, byte[] salt)
                                throws NoSuchAlgorithmException, InvalidKeySpecException {

                // PBKDF2 with SHA-1 as the hashing algorithm.
                // Note that the NIST specifically names SHA-1 as an acceptable hashing algorithm for PBKDF2
                String algorithm = "PBKDF2WithHmacSHA1";

                // SHA-1 generates 160 bit hashes, so that's what makes sense here
                int derivedKeyLength = 160;

                // Pick an iteration count that works for you. The NIST recommends at  east 1,000 iterations:
                // http://csrc.nist.gov/publications/nistpubs/800-132/nist-sp800-132.pdf
                int iterations = 20000;

                KeySpec spec = new PBEKeySpec(password.toCharArray(), salt, iterations, derivedKeyLength);

                SecretKeyFactory f = SecretKeyFactory.getInstance(algorithm);

                return f.generateSecret(spec).getEncoded();
}
```

5. Conclusion:

In conclusion, salting refers to the process of adding a salt value to a user's password before storing it in the database. It helps to increase the security of the stored passwords by making it harder for attackers to crack them.

The salt value is generated randomly and is unique to each user. It is added with the user password before applying a cryptographing function hashing function. The hash is then stored in the database alongside the salt value.

When a user attempts to login, the salt is retrieved from the database and added to the entered password. The retrieved hash is compared with the hash in the database. If they match, the user is granted access.

6. References:

1. https://www.baeldung.com/java-password-hashing
2. https://supertokens.com/blog/password-hashing-salting
3. https://docs.oracle.com/javase/8/docs/api/java/security/SecureRandom.html
4. https://docs.oracle.com/javase/7/docs/api/javax/crypto/SecretKeyFactory.html
5. https://docs.oracle.com/javase/7/docs/api/javax/crypto/spec/PBEKeySpec.html