# CA3 -Lab 3 – SALT

*“Password salting is a form of password encryption that involves appending a password to a given username and then hashing the new string of characters. This is usually done via an MD5 hashing algorithm. Password-salting is most commonly found within Linux operating systems, and it is generally considered a more secure password encryption model than any of the models used within the various Microsoft distributions****.”***

* Techopedia.com. (2019)

## Introduction

Salting a user’s password add extra protection and make it harder for a hacker to discover the password. It is a technique that can be applied in any programming language. This exercise will show you how it is implemented in Java code. If you are used to working in another language you should research the best way to implement salting for that language.

Salting is simply the addition of a unique, random string of characters known only to the site to each password before it is hashed, typically this “salt” is placed in front of each password. The salt value needs to be stored by the site, which means sometimes sites use the same salt for every password. This makes it less effective than if individual salts are used. (Gibbs, 2016)

## Lab Instructions:

### Complete the following, using the code in Appendix 1. Submit your work in a lab report.

1. Complete the code in the generateSalt()method (see notes/video) to generate a salt
   * public static byte[] generateSalt() throws NoSuchAlgorithmException {...
2. Add test code to main ()method to encrypt the password and authentic to check if they match.
   * public static void main(String [] args) { ... ...

## Submission and Grading Details

This lab will form part of your complete Lab book, which must be submitted at the end of the module. You must reference your work. You should follow the layout of a typical lab book, and add extra headings as necessary.

|  |  |
| --- | --- |
| Introduction | 5 |
| Aims & Objectives | 5 |
| Method | 10 |
| Results:  -Salt Code  -Understanding/Explanation | -  30  25 |
| Conclusions | 25 |
| **Total** | **100** |

## References

Gibbs, S. (2016). *Passwords and hacking: the jargon of hashing, salting and SHA-2 explained*. [online] the Guardian. Available at: https://www.theguardian.com/technology/2016/dec/15/passwords-hacking-hashing-salting-sha-2 [Accessed 20 Jan. 2019].

Techopedia.com. (2019). What is Password Salting? - Definition from Techopedia. [online] Available at: https://www.techopedia.com/definition/28546/password-salting [Accessed 21 Jan. 2019].

## Appendix 1 – Copy and paste this code to get started

import java.security.NoSuchAlgorithmException;

import java.security.SecureRandom;

import java.security.spec.InvalidKeySpecException;

import java.security.spec.KeySpec;

import java.util.Arrays;

import javax.crypto.SecretKeyFactory;

import javax.crypto.spec.PBEKeySpec;

public class PasswordEncryptionService {

**//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);

}

**//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();

}

//TODO YOU NEED TO COMPLETE THIS METHOD

public static byte[] generateSalt() throws NoSuchAlgorithmException {

// VERY important to use SecureRandom instead of just Random

// Generate a 8 byte (64 bit) salt as recommended by RSA PKCS5

// ...

//see notes

return salt;

}

//TODO YOU NEED TO COMPLETE THIS METHOD

public static void main(String[] args) throws NoSuchAlgorithmException, InvalidKeySpecException {

//Create strings for the password and the attempted password

//Print out the password, the attempted password, the salt and whether they match or not.

// ...

// ...

}

}