Password Encryption

Almost all modern web applications need, in one way or another, to encrypt their users' passwords. We could say that, from the moment that an application has users, and users sign in using a password, these passwords have to be stored in an encrypted way.

There are some intuitive reasons for this: our data stores can be compromised, and so can our communications. But the most important reason is that we have to think of our users' passwords as sensitive personal data. Their passwords are their key to their privacy, so they are personal, they are sensitive, and no one (not even us) has the right to know them. And we must honor this if we want to gain our user's trust.

What's Hashing?

Hashing is the process of generating a string, or hash, from a given message using a mathematical function known as a cryptographic hash function.

While there are several hash functions out there, those tailored to hashing passwords need to have four main properties to be secure:

It should be deterministic: the same message processed by the same hash function should always produce the same hash

It's not reversible: it's impractical to generate a message from its hash

It has high entropy: a small change to a message should produce a vastly different hash

And it resists collisions: two different messages should not produce the same hash

A hash function that has all four properties is a strong candidate for password hashing since together they dramatically increase the difficulty in reverse-engineering the password from the hash.

Also, though, password hashing functions should be slow. A fast algorithm would aid brute force attacks in which a hacker will attempt to guess a password by hashing and comparing billions (or trillions) of potential passwords per second.

Some great hash functions that meet all these criteria are PBKDF2, BCrypt, and SCrypt. But first, let's take a look at some older algorithms and why they are no longer recommended

Password Encryption Using MD-5

```
import java.util.Base64;

public class PasswordDigest {

	public static byte[] computeHash(String x) throws Exception {
		java.security.MessageDigest d = null;
		d = java.security.MessageDigest.getInstance("SHA-1");
		d.reset();
		d.update(x.getBytes());
		return d.digest();
	}

	public static void main(String[] args) throws Exception {
		String digestedPwd1 = Base64.getEncoder().encodeToString(computeHash("abc@123"));
		String digestedPwd2 = Base64.getEncoder().encodeToString(computeHash("abc@123"));
		System.out.println(digestedPwd1);
		System.out.println(digestedPwd2);
	}
}
```

Our first hash function is the MD5 message-digest algorithm, developed way back in 1992.

Java's MessageDigest makes this easy to calculate and can still be useful in other circumstances.

However, over the last several years, MD5 was discovered to fail the fourth password hashing property in that it became computationally easy to generate collisions. To top it off, MD5 is a fast algorithm and therefore useless against brute-force attacks.

Because of these, MD5 is not recommended.

Why SHA-512?

As computers increase in power, and as we find new vulnerabilities, then researchers derive new versions of SHA. Newer versions have a progressively longer length, or sometimes researchers publish a new version of the underlying algorithm.

SHA-512 represents the longest key in the third generation of the algorithm.

While there are now more secure versions of SHA, SHA-512 is the strongest that is implemented in Java.

AES

AES stands for Advanced Encryption System and its a symmetric encryption algorithm.It is a specification for the encryption of electronic data established by the U.S. National Institute of Standards and Technology (NIST) in 2001.Here is the wiki link for AES.The AES engine requires a plain-text and a secret key for encryption and same secret key is required to again decrypt it.

```
package com.example.demo;

import java.util.Base64;

import javax.crypto.Cipher;

import javax.crypto.spec.IvParameterSpec;

import javax.crypto.spec.SecretKeySpec;

public class PasswordEncryptUsingAES {

	private static final String key = "aesEncryptionKey";

	private static final String initVector = "encryptionIntVec";

	public static String encrypt(String value) {

		try {

			IvParameterSpec iv = new IvParameterSpec(initVector.getBytes("UTF-8"));

			SecretKeySpec skeySpec = new SecretKeySpec(key.getBytes("UTF-8"), "AES");


			Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5PADDING");

			cipher.init(Cipher.ENCRYPT_MODE, skeySpec, iv);

			byte[] encrypted = cipher.doFinal(value.getBytes());

			return Base64.getEncoder().encodeToString(encrypted);

		} catch (Exception ex) {

			ex.printStackTrace();
```

```
            }
            return null;
        }
        public static String decrypt(String encrypted) {
            try {
                IvParameterSpec iv = new IvParameterSpec(initVector.getBytes("UTF-8"));
                SecretKeySpec skeySpec = new SecretKeySpec(key.getBytes("UTF-8"), "AES");
                Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5PADDING");
                cipher.init(Cipher.DECRYPT_MODE, skeySpec, iv);
                byte[] original = cipher.doFinal(Base64.getDecoder().decode(encrypted));
                return new String(original);
            } catch (Exception ex) {
                ex.printStackTrace();
            }
            return null;
        }
        public static void main(String[] args) {
            String originalString = "password";
            System.out.println("Original String to encrypt - " + originalString);
            String encryptedString = encrypt(originalString);
            System.out.println("Encrypted String - " + encryptedString);
            String decryptedString = decrypt(encryptedString);
            System.out.println("After decryption - " + decryptedString);
        }
}
```

BCrypt

As per wiki, bcrypt is a password hashing function designed by Niels Provos and David Mazières, based on the Blowfish cipher. Bcrypt uses adaptive hash algorithm to store password.BCrypt internally generates a random salt while encoding passwords and hence it is obvious to get different encoded results for the same string.But one common thing is that everytime it generates a String of length 60.

```
<dependency>
      <groupId>org.mindrot</groupId>
      <artifactId>jbcrypt</artifactId>
     <version>0.3m</version>
  </dependency>
```

```
import org.mindrot.jbcrypt.BCrypt;

public class PasswordEncryptionUsingBcrypt {

	public static String encrypt(String password) {
		return BCrypt.hashpw(password, BCrypt.gensalt());
	}

	public static void main(String[] args) {
		String encryptedPwd1 = encrypt("ashok");
		System.out.println(encryptedPwd1);
		if (BCrypt.checkpw("ashok", encryptedPwd1)) {
			System.out.println("Password Matched");
		} else {
			System.out.println("Password Not Matched");
		}
	}
}
```

Secured Password with Salt Value

```
import java.security.NoSuchAlgorithmException;
import java.security.SecureRandom;
import java.security.spec.InvalidKeySpecException;
import java.util.Arrays;
import java.util.Base64;
import java.util.Random;

import javax.crypto.SecretKeyFactory;
import javax.crypto.spec.PBEKeySpec;

public class PasswordUtils {

	private static final Random RANDOM = new SecureRandom();
	private static final String ALPHABET =
"0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz";
	private static final int ITERATIONS = 10000;
	private static final int KEY_LENGTH = 256;

	public static String getSalt(int length) {
		StringBuilder returnValue = new StringBuilder(length);
		for (int i = 0; i < length; i++) {
			returnValue.append(ALPHABET.charAt(RANDOM.nextInt(ALPHABET.length())));
		}
		return new String(returnValue);
	}

	public static byte[] hash(char[] password, byte[] salt) {
		PBEKeySpec spec = new PBEKeySpec(password, salt, ITERATIONS, KEY_LENGTH);
		Arrays.fill(password, Character.MIN_VALUE);
		try {
			SecretKeyFactory skf = SecretKeyFactory.getInstance("PBKDF2WithHmacSHA1");
			return skf.generateSecret(spec).getEncoded();
		} catch (NoSuchAlgorithmException | InvalidKeySpecException e) {
			throw new AssertionError("Error while hashing a password: " +
e.getMessage(), e);
		} finally {
			spec.clearPassword();
		}
	}

	public static String generateSecurePassword(String password, String salt) {
		String returnValue = null;
		byte[] securePassword = hash(password.toCharArray(), salt.getBytes());

		returnValue = Base64.getEncoder().encodeToString(securePassword);

		return returnValue;
	}

	public static boolean verifyUserPassword(String providedPassword, String
securedPassword, String salt) {
		boolean returnValue = false;

		// Generate New secure password with the same salt
		String newSecurePassword = generateSecurePassword(providedPassword, salt);

		// Check if two passwords are equal
		returnValue = newSecurePassword.equalsIgnoreCase(securedPassword);

		return returnValue;
	}
```

```
	public static void main(String[] args) {
		String myPassword = "myPassword123";

		// Generate Salt. The generated value can be stored in DB.
		String salt = PasswordUtils.getSalt(30);

		// Protect user's password. The generated value can be stored in DB.
		String mySecurePassword1 = PasswordUtils.generateSecurePassword(myPassword, salt);

		// Print out protected password
		System.out.println("My secure password1 = " + mySecurePassword1);
		System.out.println("Salt Value : " + salt);

		// store Secured Password and Salt value in DB

		// For user login retrieve both Secured Password and Salt value then call
		// verifyUserPassword()
	}
}
```