Generate Secure Password Hash : MD5, SHA, PBKDF2, BCrypt Examples

July 22, 2013 by Lokesh Gupta

A password hash is an encrypted sequence of characters obtained after applying certain algorithms and manipulations on user provided password, which are generally very weak and easy to guess.

There are many such hashing algorithms in java also, which can prove really effective for password security. In this post, I will discuss some of them.

Note: Please remember that once this password hash is generated and stored in database, **you can not convert it back to original password**. Each time user login into application, you have to regenerate password hash again, and match with hash stored in database. So, if user forgot his/her password, you will have to send him a temporary password and ask him to change it with his new password. It's common now-a-days, right?

Sections in this post:

Simple password security using MD5 algorithm
Making MD5 more secure using salt
Medium password security using SHA algorithms
Advanced password security using PBKDF2WithHmacSHA1 algorithm
More Secure password hash using BCrypt and SCrypt algorithms
Final notes

Simple password security using MD5 algorithm

The **MD5 Message-Digest Algorithm** is a widely used **cryptographic hash function** that produces a 128-bit (16-byte) hash value. It's very simple and straight forward; the **basic idea is to map data sets of variable length to data sets of a fixed length**. In order to do this, the input message is split into chunks of 512-bit blocks. A padding is added to the

end so that it's length can be divided by 512. Now these blocks are processed by the MD5 algorithm, which operates in a 128-bit state, and the result will be a 128-bit hash value. After applying MD5, **generated hash is typically a 32-digit hexadecimal number**.

Here, the password to be encoded is often called the "**message**" and the generated hash value is called the message digest or simply "**digest**".

```
public class SimpleMD5Example
{
    public static void main(String[] args)
    {
        String passwordToHash = "password";
        String generatedPassword = null;
        try {
            // Create MessageDigest instance for MD5
            MessageDigest md = MessageDigest.getInstance("MD5");
            //Add password bytes to digest
            md.update(passwordToHash.getBytes());
            //Get the hash's bytes
            byte[] bytes = md.digest();
            //This bytes[] has bytes in decimal format;
            //Convert it to hexadecimal format
            StringBuilder sb = new StringBuilder();
            for(int i=0; i< bytes.length ;i++)
            {
                sb.append(Integer.toString((bytes[i] & 0xff) + 0x100, 16).substring(1));
            }
            //Get complete hashed password in hex format
            generatedPassword = sb.toString();
        }
        catch (NoSuchAlgorithmException e)
        {
            e.printStackTrace();
        }
        System.out.println(generatedPassword);
    }
}

Console output:

5f4dcc3b5aa765d61d8327deb882cf99
```

Although MD5 is a widely spread hashing algorithm, is far from being secure, MD5 generates fairly weak hashes. It's main advantages are that it is fast, and easy to implement. But it also means that it is **susceptible to brute-force** and **dictionary attacks**. **Rainbow tables** with words and hashes generated allows searching very quickly for a known hash and getting the original word.

Also, It is **not collision resistant**: this means that different passwords can eventually result in the same hash.

Still, if you are using MD5 hash then consider adding some salt to your security.

Making MD5 more secure using salt

Keep in mind, adding salt is not MD5 specific. You can add it to other algorithms also. So, please focus on how it is applied rather than its relation with MD5.

Wikipedia defines salt as **random data that are used as an additional input to a one-way function that hashes a password or pass-phrase**. In more simple words, salt is **some randomly generated text, which is appended to password before obtaining hash.**

The original intent of salting was primarily to defeat pre-computed rainbow table attacks that could otherwise be used to greatly improve the efficiency of cracking the hashed password database. A greater benefit now is to slow down parallel operations that compare the hash of a password guess against many password hashes at once.

Important: We always need to use a **SecureRandom** to create good Salts, and in Java, the SecureRandom class supports the "**SHA1PRNG**" pseudo random number generator algorithm, and we can take advantage of it.

Let's see how this salt should be generated.

```
private static byte[] getSalt() throws NoSuchAlgorithmException
{
    //Always use a SecureRandom generator
    SecureRandom sr = SecureRandom.getInstance("SHA1PRNG");
    //Create array for salt
    byte[] salt = new byte[16];
    //Get a random salt
    sr.nextBytes(salt);
    //return salt
    return salt;
}
```

SHA1PRNG algorithm is used as cryptographically strong pseudo-random number generator based on the SHA-1 message digest algorithm. Note that if a seed is not provided, it will generate a seed from a true random number generator (**TRNG**).

Now, lets look at the modified MD5 hashing example:

```
public class SaltedMD5Example
{
    public static void main(String[] args) throws NoSuchAlgorithmException, NoSuchProviderException
    {
        String passwordToHash = "password";
        byte[] salt = getSalt();

        String securePassword = getSecurePassword(passwordToHash, salt);
        System.out.println(securePassword); //Prints 83ee5baeea20b6c21635e4ea67847f66

        String regeneratedPassowrdToVerify = getSecurePassword(passwordToHash, salt);
        System.out.println(regeneratedPassowrdToVerify); //Prints 83ee5baeea20b6c21635e4ea67847f66
```

```
    }

    private static String getSecurePassword(String passwordToHash, byte[] salt)
    {
        String generatedPassword = null;
        try {
            // Create MessageDigest instance for MD5
            MessageDigest md = MessageDigest.getInstance("MD5");
            //Add password bytes to digest
            md.update(salt);
            //Get the hash's bytes
            byte[] bytes = md.digest(passwordToHash.getBytes());
            //This bytes[] has bytes in decimal format;
            //Convert it to hexadecimal format
            StringBuilder sb = new StringBuilder();
            for(int i=0; i< bytes.length ;i++)
            {
                sb.append(Integer.toString((bytes[i] & 0xff) + 0x100, 16).substring(1));
            }
            //Get complete hashed password in hex format
            generatedPassword = sb.toString();
        }
        catch (NoSuchAlgorithmException e) {
            e.printStackTrace();
        }
        return generatedPassword;
    }

    //Add salt
    private static byte[] getSalt() throws NoSuchAlgorithmException, NoSuchProviderException
    {
        //Always use a SecureRandom generator
        SecureRandom sr = SecureRandom.getInstance("SHA1PRNG", "SUN");
        //Create array for salt
        byte[] salt = new byte[16];
        //Get a random salt
        sr.nextBytes(salt);
        //return salt
        return salt;
    }
}
```

Important: Please note that now you have to **store this salt value for every password you hash**. Because when user login back in system, you must use only originally generated salt to again create the hash to match with stored hash. **If a different salt is used (we are generating random salt), then generated hash will be different.**

Also, you might heard of term **crazy hashing and salting**. It generally refer to creating custom combinations like:

```
salt+password+salt => hash
```

Do not practice these things. They do not help in making hashes further secure anyhow. If you want more security, choose a better algorithm.

Medium password security using SHA algorithms

The **SHA (Secure Hash Algorithm)** is a family of cryptographic hash functions. It is very similar to MD5 except it **generates more strong hashes**. However these hashes are not always unique, and it means that for two different inputs we could have equal hashes. When this happens it's called a "collision". Chances of collision in SHA is less than MD5. But, do not worry about these collisions because they are really very rare.

Java has 4 implementations of SHA algorithm. They generate following length hashes in comparison to MD5 (128 bit hash):

- SHA-1 (Simplest one – 160 bits Hash)
- SHA-256 (Stronger than SHA-1 – 256 bits Hash)
- SHA-384 (Stronger than SHA-256 – 384 bits Hash)
- SHA-512 (Stronger than SHA-384 – 512 bits Hash)

A longer hash is more difficult to break. That's core idea.

To get any implementation of algorithm, pass it as parameter to MessageDigest. e.g.

```
MessageDigest md = MessageDigest.getInstance("SHA-1");
//OR
MessageDigest md = MessageDigest.getInstance("SHA-256");
```

Lets create a test program so demonstrate its usage:

```
package com.howtodoinjava.hashing.password.demo.sha;

import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;
import java.security.SecureRandom;

public class SHAExample {

    public static void main(String[] args) throws NoSuchAlgorithmException {
        String passwordToHash = "password";
        byte[] salt = getSalt();

        String securePassword = get_SHA_1_SecurePassword(passwordToHash, salt);
        System.out.println(securePassword);

        securePassword = get_SHA_256_SecurePassword(passwordToHash, salt);
        System.out.println(securePassword);

        securePassword = get_SHA_384_SecurePassword(passwordToHash, salt);
        System.out.println(securePassword);

        securePassword = get_SHA_512_SecurePassword(passwordToHash, salt);
```

```
        System.out.println(securePassword);
    }

    private static String get_SHA_1_SecurePassword(String passwordToHash, byte[] salt)
    {
        String generatedPassword = null;
        try {
            MessageDigest md = MessageDigest.getInstance("SHA-1");
            md.update(salt);
            byte[] bytes = md.digest(passwordToHash.getBytes());
            StringBuilder sb = new StringBuilder();
            for(int i=0; i< bytes.length ;i++)
            {
                sb.append(Integer.toString((bytes[i] & 0xff) + 0x100, 16).substring(1));
            }
            generatedPassword = sb.toString();
        }
        catch (NoSuchAlgorithmException e)
        {
            e.printStackTrace();
        }
        return generatedPassword;
    }

    private static String get_SHA_256_SecurePassword(String passwordToHash, byte[] salt)
    {
        //Use MessageDigest md = MessageDigest.getInstance("SHA-256");
    }

    private static String get_SHA_384_SecurePassword(String passwordToHash, byte[] salt)
    {
        //Use MessageDigest md = MessageDigest.getInstance("SHA-384");
    }

    private static String get_SHA_512_SecurePassword(String passwordToHash, byte[] salt)
    {
        //Use MessageDigest md = MessageDigest.getInstance("SHA-512");
    }

    //Add salt
    private static byte[] getSalt() throws NoSuchAlgorithmException
    {
        SecureRandom sr = SecureRandom.getInstance("SHA1PRNG");
        byte[] salt = new byte[16];
        sr.nextBytes(salt);
        return salt;
    }
}

Output:

e4c53afeaa7a08b1f27022abd443688c37981bc4

87adfd14a7a89b201bf6d99105b417287db6581d8aee989076bb7f86154e8f32

bc5914fe3896ae8a2c43a4513f2a0d716974cc305733847e3d49e1ea52d1ca50e2a9d0ac192acd43facfb422bb5ace88

529211542985b8f7af61994670d03d25d55cc9cd1cff8d57bb799c4b586891e112b197530c76744bcd7ef135b58d47d65a0bec221e
```

Very easily we can say that SHA-512 generates strongest Hash.

Advanced password security using PBKDF2WithHmacSHA1 algorithm

So far we learned about creating secure hashes for password, and using salt to make it even more secure. But the problem today is that hardwares have become so much fast that any brute force attack using dictionary and rainbow tables, any password can be cracked in some less or more time.

To solve this problem, general **idea is to make this brute force attack slower so that damage can be minimized**. Our next algorithm, works on this very concept. The goal is to make the hash function slow enough to impede attacks, but still fast enough to not cause a noticeable delay for the user.

This feature is essentially implemented using some CPU intensive algorithms such as **PBKDF2, Bcrypt** or **Scrypt**. These algorithms take a work factor (also known as security factor) or iteration count as an argument. This value determines how slow the hash function will be. When computers become faster next year we can increase the work factor to balance it out.

Java has implementation of "**PBKDF2**" algorithm as "**PBKDF2WithHmacSHA1**". Let's look at the example how to use it.

```
public static void main(String[] args) throws NoSuchAlgorithmException, InvalidKeySpecException
{
    String  originalPassword = "password";
    String generatedSecuredPasswordHash = generateStorngPasswordHash(originalPassword);
    System.out.println(generatedSecuredPasswordHash);
}
private static String generateStorngPasswordHash(String password) throws NoSuchAlgorithmException, Inv
{
    int iterations = 1000;
    char[] chars = password.toCharArray();
    byte[] salt = getSalt();

    PBEKeySpec spec = new PBEKeySpec(chars, salt, iterations, 64 * 8);
    SecretKeyFactory skf = SecretKeyFactory.getInstance("PBKDF2WithHmacSHA1");
    byte[] hash = skf.generateSecret(spec).getEncoded();
    return iterations + ":" + toHex(salt) + ":" + toHex(hash);
}

private static byte[] getSalt() throws NoSuchAlgorithmException
{
    SecureRandom sr = SecureRandom.getInstance("SHA1PRNG");
    byte[] salt = new byte[16];
    sr.nextBytes(salt);
    return salt;
}

private static String toHex(byte[] array) throws NoSuchAlgorithmException
{
    BigInteger bi = new BigInteger(1, array);
```

```
        String hex = bi.toString(16);
        int paddingLength = (array.length * 2) - hex.length();
        if(paddingLength > 0)
        {
            return String.format("%0"  +paddingLength + "d", 0) + hex;
        }else{
            return hex;
        }
    }
```

Output:

```
1000:5b4240333032306164:f38d165fce8ce42f59d366139ef5d9e1ca1247f0e06e503ee1a611dd9ec40876bb5edb8409f5abe550
```

Next step is to have a function which can be used to validate the password again when user comes back and login.

```
public static void main(String[] args) throws NoSuchAlgorithmException, InvalidKeySpecException
    {
        String  originalPassword = "password";
        String generatedSecuredPasswordHash = generateStorngPasswordHash(originalPassword);
        System.out.println(generatedSecuredPasswordHash);

        boolean matched = validatePassword("password", generatedSecuredPasswordHash);
        System.out.println(matched);

        matched = validatePassword("password1", generatedSecuredPasswordHash);
        System.out.println(matched);
    }

    private static boolean validatePassword(String originalPassword, String storedPassword) throws NoSuchA
    {
        String[] parts = storedPassword.split(":");
        int iterations = Integer.parseInt(parts[0]);
        byte[] salt = fromHex(parts[1]);
        byte[] hash = fromHex(parts[2]);

        PBEKeySpec spec = new PBEKeySpec(originalPassword.toCharArray(), salt, iterations, hash.length * 8
        SecretKeyFactory skf = SecretKeyFactory.getInstance("PBKDF2WithHmacSHA1");
        byte[] testHash = skf.generateSecret(spec).getEncoded();

        int diff = hash.length ^ testHash.length;
        for(int i = 0; i < hash.length && i < testHash.length; i++)
        {
            diff |= hash[i] ^ testHash[i];
        }
        return diff == 0;
    }
    private static byte[] fromHex(String hex) throws NoSuchAlgorithmException
    {
        byte[] bytes = new byte[hex.length() / 2];
        for(int i = 0; i<bytes.length ;i++)
        {
            bytes[i] = (byte)Integer.parseInt(hex.substring(2 * i, 2 * i + 2), 16);
        }
        return bytes;
    }
```

Please care to refer functions from above code samples. If found any difficult then download the sourcecode attached at end of tutorial.

More Secure password hash using bcrypt and scrypt algorithms

The concepts behind bcrypt is similar to previous concept as in PBKDF2. It just happened to be that java does not have any inbuilt support for bcrypt algorithm to make the attack slower but still you can find one such implementation in source code download.

Let's look at the sample usage code(BCrypt.java is available in sourcecode):

```
public class BcryptHashingExample
{
    public static void main(String[] args) throws NoSuchAlgorithmException
    {
        String  originalPassword = "password";
        String generatedSecuredPasswordHash = BCrypt.hashpw(originalPassword, BCrypt.gensalt(12));
        System.out.println(generatedSecuredPasswordHash);

        boolean matched = BCrypt.checkpw(originalPassword, generatedSecuredPasswordHash);
        System.out.println(matched);
    }
}

Output:

$2a$12$WXItscQ/FDbLKU4mO58jxu3Tx/mueaS8En3M6QOVZIZLaGdWrS.pK
true
```

Similar to bcrypt, i have downloaded scrypt from **github** and added the source code of scrypt algorithm in sourcecode to download in last section. Lets see how to use the implementation:

```
public class ScryptPasswordHashingDemo
{
    public static void main(String[] args) {
        String originalPassword = "password";
        String generatedSecuredPasswordHash = SCryptUtil.scrypt(originalPassword, 16, 16, 16);
        System.out.println(generatedSecuredPasswordHash);

        boolean matched = SCryptUtil.check("password", generatedSecuredPasswordHash);
        System.out.println(matched);

        matched = SCryptUtil.check("passwordno", generatedSecuredPasswordHash);
        System.out.println(matched);
    }
}

Output:
```

```
$s0$41010$Gxbn9LQ4I+fZ/kt0glnZgQ==$X+dRy9oLJz1JaNm1xscUl7EmUFHIILT1ktYB5DQ3fZs=
true
false
```

Final Notes

1. Storing the text password with hashing is most dangerous thing for application security today.
2. MD5 provides basic hashing for generating secure password hash. Adding salt make it further stronger.
3. MD5 generates 128 bit hash. To make ti more secure, use SHA algorithm which generate hashes from 160-bit to 512-bit long. 512-bit is strongest.
4. Even SHA hashed secure passwords are able to be cracked with today's fast hardwares. To beat that, you will need algorithms which can make the brute force attacks slower and minimize the impact. Such algorithms are PBKDF2, BCrypt and SCrypt.
5. Please take a well considered thought before applying appropriate security algorithm.

To download the sourcecode of above algorithm examples, please follow the below link.

Happy Learning !!

References:

- https://en.wikipedia.org/wiki/MD5
- https://en.wikipedia.org/wiki/Secure_Hash_Algorithm
- http://en.wikipedia.org/wiki/Bcrypt
- http://en.wikipedia.org/wiki/Scrypt
- http://en.wikipedia.org/wiki/PBKDF2
- https://github.com/wg/scrypt
- http://www.mindrot.org/projects/jBCrypt/