HowToDoInJava

YOU ARE HERE: HOME > JAVA BEST PRACTICES > HOW TO GENERATE SECURE PASSWORD HASH : MD5, SHA, PBKDF2, BCRYPT EXAMPLES

How to generate secure password hash : MD5, SHA, PBKDF2, BCrypt examples

JULY 22, 2013 | LOKESH GUPTA | 57 COMMENTS

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

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 String 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.toString();
}
```

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, NoSuchProvider
    {
        String passwordToHash = "password";
        String salt = getSalt();

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

        String regeneratedPassowrdToVerify = getSecurePassword(passwordToHash, salt);
        System.out.println(regeneratedPassowrdToVerify); //Prints 83ee5baeea20b6c21635e4ea
    }

    private static String getSecurePassword(String passwordToHash, String salt)
    {
        String generatedPassword = null;
        try {
            // Create MessageDigest instance for MD5
            MessageDigest md = MessageDigest.getInstance("MD5");
            //Add password bytes to digest
            md.update(salt.getBytes());
            //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 String getSalt() throws NoSuchAlgorithmException, NoSuchProviderExcepti
    {
        //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.toString();
    }
}
```

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:

```java
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";
        String 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, String salt)
    {
        String generatedPassword = null;
        try {
            MessageDigest md = MessageDigest.getInstance("SHA-1");
            md.update(salt.getBytes());
            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, String salt)
    {
        //Use MessageDigest md = MessageDigest.getInstance("SHA-256");
    }

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

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

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

```
Output:

e4c53afeaa7a08b1f27022abd443688c37981bc4

87adfd14a7a89b201bf6d99105b417287db6581d8aee989076bb7f86154e8f32

bc5914fe3896ae8a2c43a4513f2a0d716974cc305733847e3d49e1ea52d1ca50e2a9d0ac192acd43facfb422bb

529211542985b8f7af61994670d03d25d55cc9cd1cff8d57bb799c4b586891e112b197530c76744bcd7ef135b5
```

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, InvalidKeySpec
    {
        String  originalPassword = "password";
        String generatedSecuredPasswordHash = generateStorngPasswordHash(originalPassword)
        System.out.println(generatedSecuredPasswordHash);
    }
    private static String generateStorngPasswordHash(String password) throws NoSuchAlgorit
    {
        int iterations = 1000;
        char[] chars = password.toCharArray();
        byte[] salt = getSalt().getBytes();

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

    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:f38d165fce8ce42f59d366139ef5d9e1ca1247f0e06e503ee1a611dd9ec40876bb
```

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, InvalidKeySpecExce
    {
        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 storedPassworc
    {
        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,
        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.gensa
        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,
        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.

Download sourcecode

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/

Share On:

Facebook Google+ Twitter Buffer

READ MORE:

1. **How to generate SHA or MD5 file checksum hash in java**
2. **How to get google page rank in java**
3. **How to build regex based password validator in java**
4. **How to view/generate bytecode for a java class file**
5. **Java Load/Read/Write Properties file examples**
6. **How to read file content into byte array in java**
7. **Performance comparison of different for loops in java**
8. **Usage of class sun.misc.Unsafe**

---

57 THOUGHTS ON “HOW TO GENERATE SECURE PASSWORD HASH : MD5, SHA, PBKDF2, BCRYPT EXAMPLES”

kuldeep

OCTOBER 27, 2015 AT 12:44 PM

Also , what would be size of key length below line return, what I Understand the key length we are passing is 64*8 = 512, but the password length it it return after hashing is 128.
PBEKeySpec spec = new PBEKeySpec(chars, salt, iterations, 64 * 8);

Can you please explain how it is determining key length.

kuldeep

OCTOBER 27, 2015 AT 12:05 PM

Can any one explain why we are using **fromHex** and **toHex** method.

S

OCTOBER 17, 2015 AT 12:47 PM

I'm curious about why validatePassword method, why not just hash the new pw with the same salt and see if the answer is the same? Is there a reason to do it the way you're doing it (it just seems a little more complicated)?

★ Lokesh Gupta

OCTOBER 17, 2015 AT 6:32 PM

`validatePassword()` method does exactly he same thing. You can choose to do it differently .. may be compare using database query.. but the logic remains the same : hash the new password with the same salt and compare with stored password.

Lloyd Rochester

JULY 19, 2015 AT 2:27 PM

Hello Lokesh,

Great article. My question if you store the string from generateStrongPasswordHash which contains the PBKDF2 work factor and salt in the string does this make the algorithm weak again in the sense the hacker would know both the salt and the work factor?

Thanks,
Lloyd

★ Lokesh Gupta

JULY 19, 2015 AT 3:00 PM

Work factor is for delaying the computation speed.. even if hacker know it's value.. he can't manipulate the execution speed in runtime.
Salt also will not help in guessing the password. How it can?

Mathieu

JUNE 29, 2015 AT 7:05 PM

Why there is no date for the article? I can see it's in the URL, and from the comments I could have guessed, but why is there no date to clearly show at what time this was written? Computer science is moving fast enough, that stuff that was written 2 years ago, might not be true today.

★ Lokesh

JUNE 30, 2015 AT 3:10 AM

Date is in fact part of URL. Check again.

King

JUNE 24, 2015 AT 1:46 PM

I want to encrypt zip file that i generated. inside the zip file I have files.
So I want to encrypt the zip file. Its my first time to work with encryption. I did my own research, I seem to understand but not clearly on how it works.
Can anyone help with a clear example, and good way of encrypting, please explain to me. And how do I implement it.

Example of encrypting a zip file will be a good one.

Your help will be appreciated.

Thanks in advanced

★ **Lokesh**

JUNE 24, 2015 AT 2:23 PM

I will recommend to go with a proper library to do this job e.g. zip4j. A sample program is given here : SO link.

King

JUNE 24, 2015 AT 3:07 PM

Lokesh Thanks, But I tried that one, it leave a file that is corrupt. I cant access it to see if its encrypted or not.

★ **Lokesh**

JUNE 24, 2015 AT 4:19 PM

I tried to open the file using double click, and it was format error as you said. But when I tried 7-zip software, it asked for password and accepted correct password also. Let me try few things.

★ **Lokesh**

JUNE 24, 2015 AT 4:49 PM

I tried with

```
parameters.setEncryptionMethod(Zip4jConstants.ENC_METHOD_STANDARD);
```

in place of

```
parameters.setEncryptionMethod(Zip4jConstants.ENC_METHOD_AES);
```

And it worked like charm. Have you tried it?

This one also seems good. https://code.google.com/p/winzipaes/

King

JUNE 25, 2015 AT 6:52 AM

Lokesh, its work as a charm after I replace the
parameters.setEncryptionMethod(Zip4jConstants.ENC_METHOD_AES);

with
parameters.setEncryptionMethod(Zip4jConstants.ENC_METHOD_STANDARD);

Thanks a lot, but now in my case. I generate two files in my app on android. The format of the files are .txt and .mp4, i than zip them together. Which happens successfully. Now I want to encrypt the zip file to another extension (.ecz) and put a password to it. So I will be having the zip file, which is not encrypted and the encrypted file which is the one I need. Zip file and the two files, I will delete them. I will be left with encrypted file with this extension .ecz

I will be happy if you help me accomplish this, i battle this for three days now.

★ Lokesh

JUNE 25, 2015 AT 10:33 AM

I will try to find time for it.

King

JUNE 25, 2015 AT 10:38 AM

I will be happy if you help me on this one Lokesh.

Abhishek walter

MARCH 11, 2015 AT 10:55 AM

i need a java program for generating HMAC values for sha-1 results ,can you send me?thanks

★ Lokesh

MARCH 15, 2015 AT 3:06 AM

I will work on it.

Anuj

JANUARY 13, 2015 AT 11:50 AM

Thanks for a great tutorial.
I wanna send encrypted to image to server(android).Now please can you help me out how can

i achieve this???

Francois Groulx

SEPTEMBER 29, 2014 AT 3:01 PM

Perfectly simple and clear

Manjunath

AUGUST 6, 2014 AT 7:27 PM

Hi,

In the function specified to get salt.

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

You are generating the salt by using "SHA1PRNG" but while returning you are sending the hashcode of salt. So all the efforts made to generate the secure random salt is ruined.

Instead construct new string from the salt and return.

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

himansu nayak

JUNE 25, 2014 AT 5:51 AM

Hi Lokesh,

I have gone thru one of the website which discourage use of "SHA1-PRNG" Algo. for using Random Generator.
Please have a look
http://coding.tocea.com/scertify-code/dont-use-the-sha1-prng-randomness-generator/

★ Lokesh

JUNE 25, 2014 AT 7:15 AM

Himansu, the only reason I found on quick google search is "sha1prng" is slightly slower

than native algorithms like 'Windows-PRNG' on windows and 'NativePRNG' in linux. No where I could find any bench-mark data about their performances.
So my take is that you can go with sha1prng as long as you do not hit any performance roadblock, specifically due to random generator. In all normal real life scenarios, sha1prng is more than enough.

Eric

JUNE 13, 2014 AT 3:42 AM

Great post! One question though. If I wanted to display the hashed password as a String to the user, how would I get that value if I am using JUST the hashed password from a database? I am using the advanced technique to store a password

★ Lokesh

JUNE 13, 2014 AT 3:53 AM

I have not mis-understood your question then you need to fetch the password from database, just like you fetch any other field. I really doubt what you mean by "hashed password as a String"? Isn't hashed password a string itself?

Eric

JUNE 13, 2014 AT 8:43 PM

I am sorry I should be more clear. I have an administrative user and a student user. I would like the administrator to have access to all of the student's original passwords. So when I query my database, how can I decode the hashed password back to the original, plain text password for the admin to view? After reading some other comments, I see that it is impossible to get the original password after using the advanced hash function that you provided. Can you suggest a way around this?

★ Lokesh

JUNE 14, 2014 AT 6:05 AM

No, using functions in this post, u cannot get back the original password.

artur

JUNE 4, 2014 AT 11:23 PM

How to validate entered password with securePassword SHA-512?

★ Lokesh

JUNE 5, 2014 AT 4:38 AM

Please re-read the third para in starting "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."

artur

JUNE 6, 2014 AT 12:03 AM

Thank you! I understand it! But, hash each time is the different for the same strings. So, hashes of stored and input strings constantly would be different, even strings are the same. Maybe, it is because of using salt. So this code will not work correctly.

```
String passwordToHash = "password";
String salt = getSalt();
String securePassword = get_SHA_1_SecurePassword(passwordToHash, salt);

 String passwordEntered = "password";
String salt2 = getSalt();
String securePasswordEntered = get_SHA_1_SecurePassword(passwordEntered, salt2);

 securePasswordEntered.equals(securePassword);
```

★ Lokesh

JUNE 6, 2014 AT 7:05 AM

Another similar note is written in "Making MD5 more secure using salt" section:
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.

Rocky

MAY 13, 2014 AT 11:53 AM

Thanks a lot!!
Well explained..

John

APRIL 27, 2014 AT 2:21 PM

Hi in the article about Md5 and generating salts

shouldn't you use

new String(salt, "UTF-8");

rather than

return salt.toString();

★ Lokesh

APRIL 27, 2014 AT 4:00 PM

Any particular reason?

Hoyeong Heo

MAY 22, 2014 AT 7:26 AM

It will make you use the hash code of byte array 'object' rather than the generated random byte values themselves.
More precisely, it is getClass().getName() + '@' + Integer.toHexString(hashCode()).

Hoyeong Heo

MAY 22, 2014 AT 7:30 AM

I think your getSalt() should be revised like below.

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

Malli

JUNE 3, 2014 AT 6:29 PM

It looks SecureRandom.getInstance("SHA1PRNG") returns a new instance of SecureRandom every time you call it. Just wonder, if it is a good approach to create a new SecureRandom everytime we need a salt??

Malli

JUNE 3, 2014 AT 6:36 PM

Not sure if there is any secured ThreadLocalSecuredRandom? Just puzzled about making it singleton or thread local…

Mark Dominic Libunao

APRIL 26, 2014 AT 6:28 AM

Hi, If I would be using SHA 256

will i need to change something here in the getSalt()? like change the SHA1PRNG to SHA256PRNG? and the byte size? or Is it okay to use this on SHA 256 w/o changing anything

//Add salt

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

★ Lokesh

APRIL 26, 2014 AT 7:09 AM

MessageDigest digest = MessageDigest.getInstance("SHA-256");

Above is enough. SecureRandom support only SHA1PRNG algo. No change needed here: http://docs.oracle.com/javase/7/docs/technotes/guides/security/StandardNames.html#SecureRandom

Mark Dominic Libunao

APRIL 26, 2014 AT 3:50 PM

Hi Lokesh,

Thanks for the reply, got it work but somehow I cannot make it to work with jboss login module with these options

Do you have any suggestion to make it work with it?

★ Lokesh

APRIL 26, 2014 AT 4:16 PM

On vacation for next 3 days. Let me know if still have something I have work on. May be by tuesday.

Diana

APRIL 25, 2014 AT 8:55 AM

A comment about your PBKDF2WithHmacSHA1 solution:

```
private static String generateStorngPasswordHash(String password) throws
NoSuchAlgorithmException, InvalidKeySpecException
{
int iterations = 1000;
char[] chars = password.toCharArray();
byte[] salt = getSalt().getBytes();

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 String getSalt() throws NoSuchAlgorithmException
{
SecureRandom sr = SecureRandom.getInstance(“SHA1PRNG”);
byte[] salt = new byte[16];
sr.nextBytes(salt);
return salt.toString();
}

You could just return salt without converting it to String in

return salt.toString();

and then converting it back to byte array here:

byte[] salt = getSalt().getBytes();

By just using

return salt;

in getSalt(), and then:

byte[] salt = getSalt();
in generateStorngPasswordHash().

★ **Lokesh**

APRIL 25, 2014 AT 9:26 AM

Yes, it also can be done.

Diana

APRIL 25, 2014 AT 11:02 AM

Also

PBEKeySpec spec = new PBEKeySpec(chars, salt, iterations, 64 * 8);

would be better as

KeySpec spec = new PBEKeySpec(chars, salt, iterations, 64 * 8);

Awesome article by the way! I am using it right now for an app and it’s clear, well explained and easy to follow!

★ **Lokesh**

APRIL 25, 2014 AT 11:05 AM

I am glad to hear that it is making someone’s work a bit easier.

lasantha

APRIL 2, 2014 AT 6:53 AM

Thank You

feruz

MARCH 16, 2014 AT 12:38 PM

Great post.
Got question, SCryptUtil.scrypt(originalPassword, 16, 16, 16);
what this numbers actually do?

thanks!

Marek

MARCH 4, 2014 AT 1:20 PM

I don’t like the implementation of the getSalt function. You convert the byte array to string (which does not make any sense at all, as a random variable will not have any sensible string representation), to convert the value back to byte array in the next step. More to it, you create an object with cryptographic data in memory you will not be able to override (because of immutability). This might be acceptable in this example, but it is a very bad idea in crypto. No FIPS 140-2 for you. ;)

somesh

DECEMBER 19, 2013 AT 8:55 AM

Useful

Rahmad Dawood

DECEMBER 1, 2013 AT 6:49 PM

Thank you for sharing this! Excellent post!

Cesar

OCTOBER 9, 2013 AT 8:26 AM

Excelente Post, very ilustrative
Thanks for your explication
Comment from Lima Peru.

Mostafa Ali

SEPTEMBER 22, 2013 AT 9:06 PM

WOW!! that one with the PBKDF2 algorithm is so neat :D

denger

AUGUST 5, 2013 AT 1:09 PM

Nice post!! I'll translate it to Chinese.

reza

JULY 25, 2013 AT 7:29 AM

very good
thanks!

Ummer

JULY 23, 2013 AT 10:56 AM

Hi Pradeep,
Nice article bro.

Pradeep

JULY 22, 2013 AT 5:20 PM

Excellent post….very informative.
Thanks for writing it.

Note:- In comment box, please put your code inside **[java] ... [/java]** OR **[xml] ... [/xml]** tags otherwise it may not appear as intended.