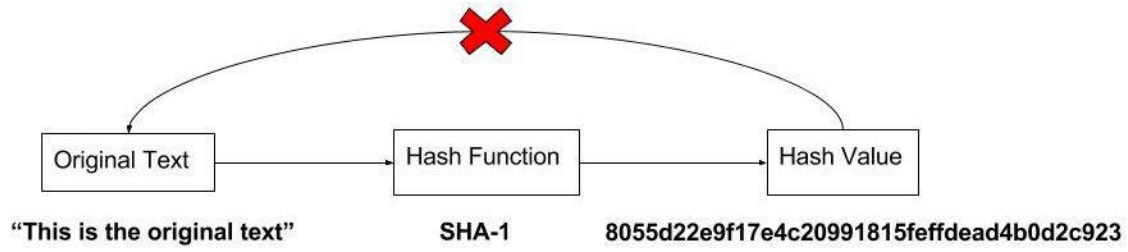
Week 5 FIT 5003 Software Security

Hashing, Salted Hashing and Permissions

1. Introduction

Hashing is a technique used to transform a variable length input into an irreversible and fixed-sized output which is known as a message digest or hash value. Storing passwords in software systems, ensuring the integrity of messages during communication and creating indexes in databases are some examples where hashing is used. Hashing constitutes an interesting example for a simple security application in Java.

The following diagram simply explains hashing. We have some input (text) value and using a hash function (e.g. MD5, SHA-1, SHA-256) we can transform this input into a fixed length output which we call as the hash value. Using this hash value, it is not possible to obtain the original input. That is why we call hashing is irreversible.

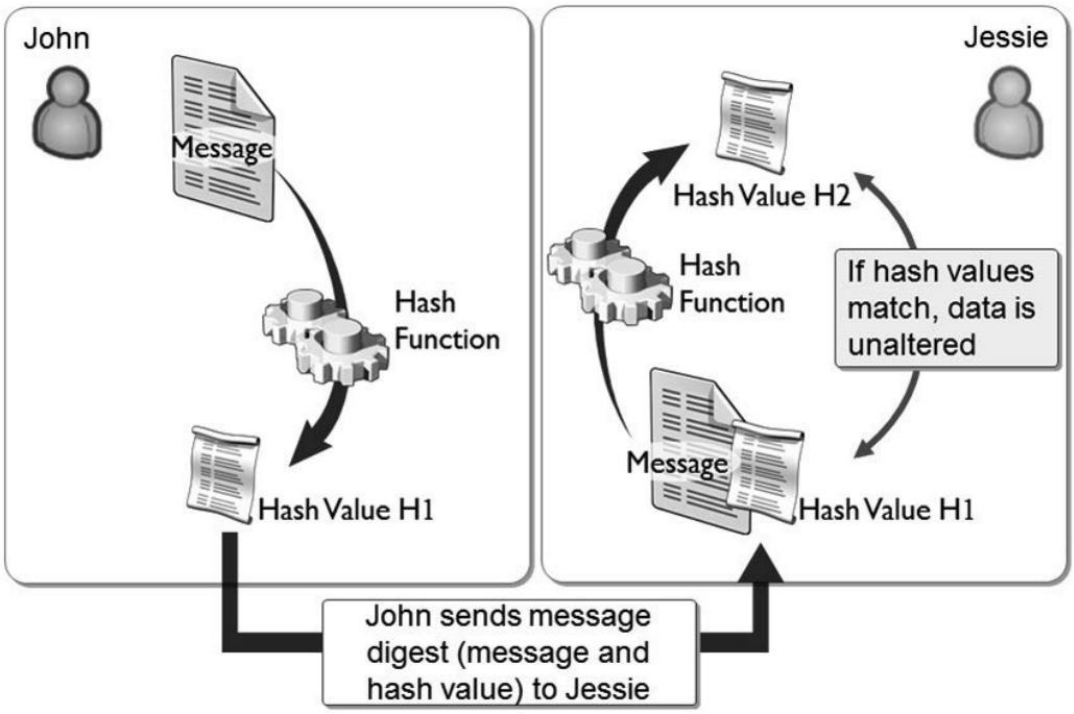


If we need to preserve the data in original format for later use, we cannot use hashing. In such cases, we can encrypt the data and keep where we can later decrypt it and obtain the original value.

Some of the examples discussed in this post are taken from the **Official (ISC)2 Guide to the CSSLP** (https://www.crcpress.com/Official-ISC2-Guide-to-the-CSSLP/Paul/9781439826058) book.

2. Hashing for Integrity Verification

Let's say John wants to send a message to Jessie. Once Jessie receives the message, how can she verify that it is the original message sent by John and not altered by some other party during the communication ?



Data Integrity using Hash Functions

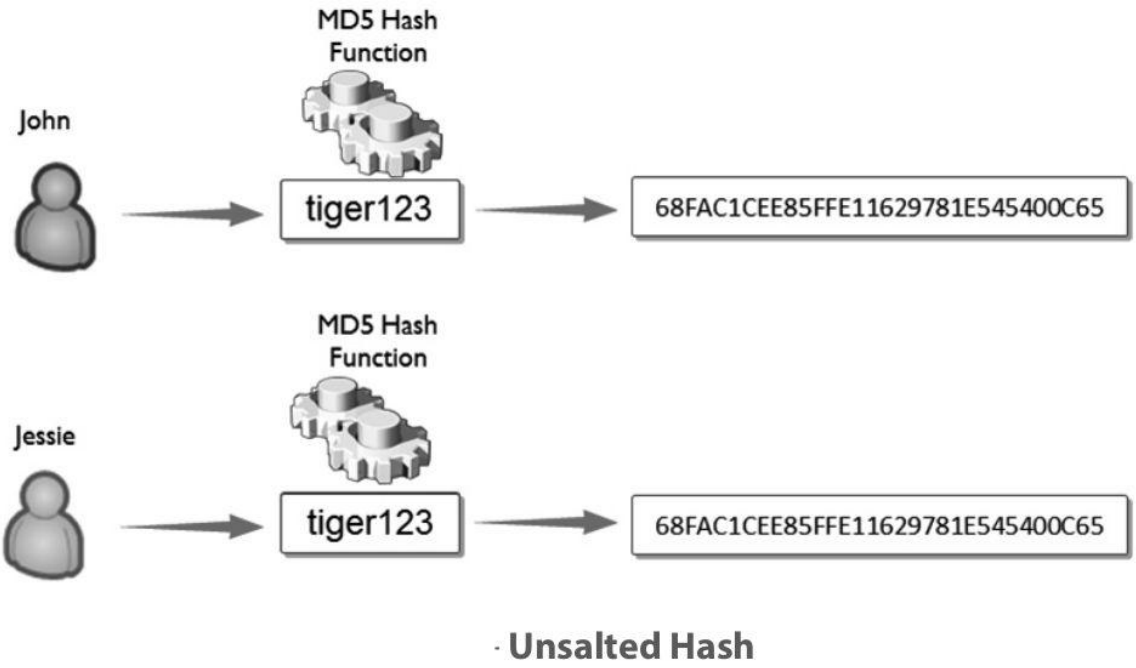
John can write the message and calculate the hash value of the message (H1) using a hash function (e.g. SHA-1). Then he can send the message to Jessie along with the hash value (H1) which we call as the message digest. When Jessie receives the message and the hash value (H1), she can calculate the hash value of the message again using the same hash function used by John. Now if the calculated hash value (H2) matches with the hash value (H1) sent by John, it ensures that the original message sent by John is not altered. If it was altered, then the new hash value (H2) would not match with the previous hash value (H1).

3. Hashing for Password Storage and Authentication

If you develop a software application in which you need to manage user accounts, how are you going to store the passwords of the users ? If you store the passwords in plain text, anyone that has access to the user store (database, LDAP or Active Directory) where the user accounts are stored, will have be able to view the passwords and that person can even login to the system using the credentials of any user which should not be done at any cost. A simplified way to overcome this risk is through hashing

What we can do is, instead of storing plain text password as it is, we can hash the password using a hashing function and store the hashed value. When a user tries to

login to the system by providing his plain text password, the system must hash the password using the same hashing function and compare the hashed value with the value stored in the user store. If they match, then the user is successfully authenticated.



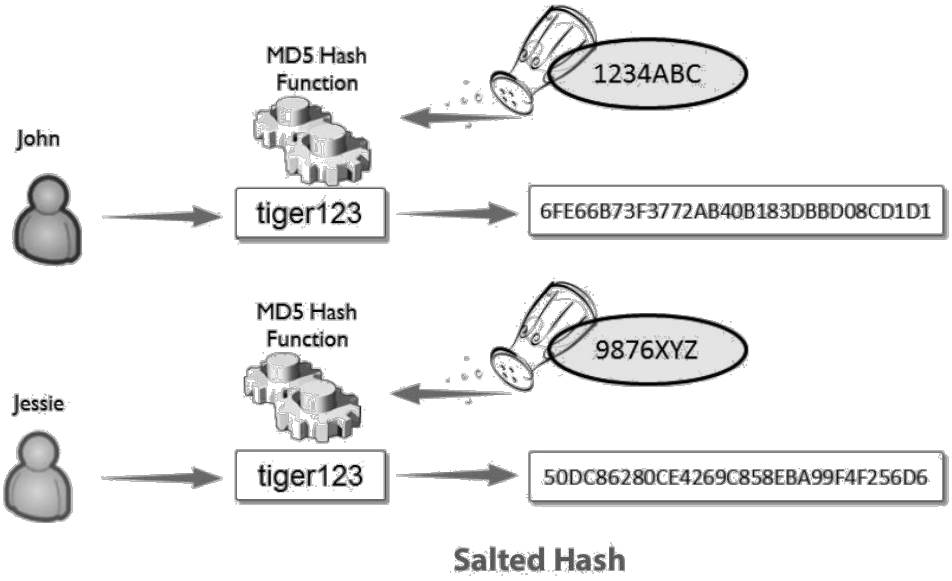
· **Unsalted Hash**

In above diagram, both John and Jessie are having the same password ‘tiger123’. When we hash this password using a particular hashing algorithm, we get the same hashed value. Even with this, the security of the passwords cannot be fully guaranteed. Let’s say John uses a word in the Dictionary as his password. An attacker can prepare a list of hash values for all the words in the dictionary. Then if he has access to the user store where the hashed passwords are stored, then he can compare the hashed dictionary words with the hashed user passwords and find if any matching record is found. That way, the attacker can get to know the password of John. In order to avoid dictionary attacks for hashed passwords, we can use the technique called Salting.

In Salting, the system can generate a random set of characters which we call as the salt. Then the user’s plaintext password is appended with this salt value. After that, this combined password is sent to the hashing function to obtain the hash value. This value is called as the salted hash. The system must remember the salt value generated for each user. When a user tries to login to the system, the system can retrieve the salt value of the user, append it to the plaintext password, hash it and compare the salted hash value with the value stored in the system. If they match, then the user is successfully authenticated.

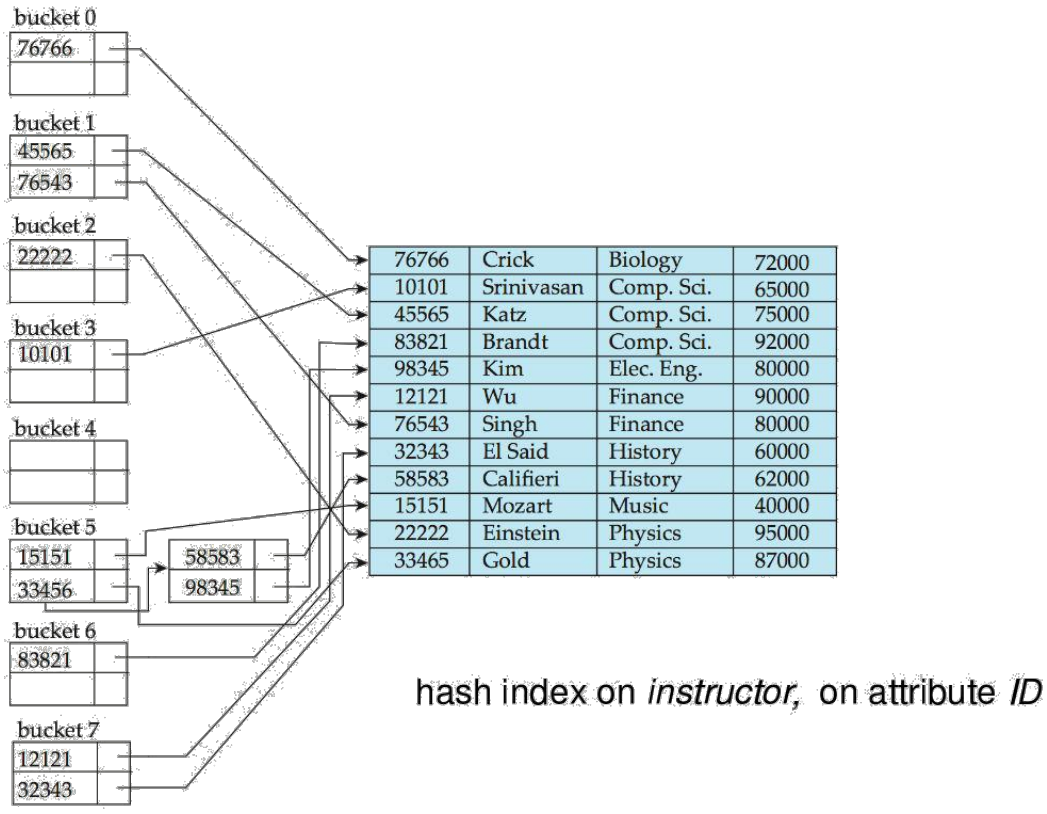
In following example, both John and Jessie have the same password ‘tiger123’. The system generates the salt ‘1234ABC’ for John and ‘9876XYZ’ for Jessie.

After adding the salt to John’s password, we get “tiger123 hashed with **1234ABC**” and for Jessie, we get “tiger hashed with **9876XYZ**”. When these values are sent to the hashing function, John and Jessie get different hash values. Therefore, with this approach we can avoid dictionary attacks.



4. **Hashing for Indexing in Databases**

When using hashing in database indexing, we can divide a set of records to different groups which we call as a bucket, based on a key. This key is a hash value. When we need to perform a search, the input is sent to a hashing function which returns a hash value that points to a particular bucket. Then we can get to know the particular records that are associated with the search.



5. Hashing Functions

Some of the most common hash functions are the MD2, MD4, and MD5, which were all designed by Ronald Rivest; the Secure Hash Algorithms family (SHA-0, SHA-1, SHA- and SHA-2) designed by NSA and published by NIST to complement digital signatures, and HAVAL. The Ronald Rivest MD series of algorithms generate a fixed, 128-bit size output and has been proven to be not completely collision free. The SHA-0 and SHA-1 family of hash functions generated a fixed, 160-bit sized output. The SHA-2 family of hash functions includes SHA-224 and SHA-256, which generate a 256-bit sized output and SHA-384 and SHA-512 which generate a 512-bit sized output. HAVAL is distinct in being a hash function that can produce hashes in variable lengths (128 bits - 256 bits). HAVAL is also flexible to let users indicate the number of rounds (3-5) to be used to generate the hash for increased security. As a general rule of thumb, the greater the bit length of the hash value that is supported, the greater the protection that is provided, making cryptanalysis work factor significantly greater. So when designing the software, it is important to consider the bit length of the hash value that is supported.

Hash Function	Hash Value Length (in bits)
MD2, MD4, MD5	128
SHA	160
HAVAL	Variable lengths (128, 160, 192, 224, 256)

Hash functions and supported hash value lengths

When selecting a hashing function for your systems, it is important to check the strength of the function. For example, MD5 is proven to have collisions. A collision is when two different input values result in getting the same hash value.

Questions

1. What is the purpose of using salt? Should the value of the salt be kept secret? Why or why not?
2. What is the purpose of the iteration count?
3. What are the possible uses of password-based cryptography?

6. Programming Task for Hashing

Following java class MessageDigestTest is written for converting plaintext inputs into their hashes using SHA-1 hashing function.

Task 1

Modify the given java program given in Listing 1. Change the hash function from SHA1 to SHA256 and SHA512 (you need to change other variables in the program accordingly otherwise you may encounter errors).

Listing 1:

```
import java.math.BigInteger;
import java.nio.charset.StandardCharsets;
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;

// Java program to calculate SHA-1 hash value

class MessageDigestTest {
   public static byte[] getSHA(String input) throws NoSuchAlgorithmException
   {
      // Static getInstance method is called with hashing SHA-1
      MessageDigest md = MessageDigest.getInstance("SHA-1");


      // digest() method called
      // to calculate message digest of an input
      // and return array of byte
      return md.digest(input.getBytes(StandardCharsets.UTF_8));
   }

   public static String toHexString(byte[] hash) {
      // Convert byte array into signum representation
      BigInteger number = new BigInteger(1, hash);

      // Convert message digest into hex value
      StringBuilder hexString = new StringBuilder(number.toString(16));

      // Pad with leading zeros
      while (hexString.length() < 32) {
         hexString.insert(0, '0');
      }
      return hexString.toString();
   }
   // Driver code
   public static void main(String args[]) {
      try
      {
         System.out.println("HashCode Generated by SHA-1 for:");

         String s1 = "FIT5003";
         System.out.println("\n" + s1 + " : " + toHexString(getSHA(s1)));

         String s2 = "hello world";
         System.out.println("\n" + s2 + " : " + toHexString(getSHA(s2)));
      }
      // For specifying wrong message digest algorithms
      catch (NoSuchAlgorithmException e) {
```

```
            System.out.println("Exception thrown for incorrect algorithm: " +
e);
        }
    }
}
```

Task 2

Each time a user logs in to the application, the program has to regenerate the password hash again and match with the hash stored in the database. So, if the user forgets his/her password, you will have to send him a temporary password and ask him to change it with his new password. So, modify the java program with the reset password function.

Following java class SecureUtils is an example for using a salt in password hashing.

Listing 2:

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

public class SecureUtils {

   public static String getSecurePassword(String password, byte[] salt) {

      String generatedPassword = null;
      try {
         MessageDigest md = MessageDigest.getInstance("SHA-256");
         md.update(salt);
         byte[] bytes = md.digest(password.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 byte[] getSalt() throws NoSuchAlgorithmException {
      SecureRandom random = new SecureRandom();
      byte[] salt = new byte[16];
      random.nextBytes(salt);
      return salt;
   }
```

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

      // same salt should be passed
      byte[] salt = getSalt();
      String password1 = getSecurePassword("Password", salt);
      String password2 = getSecurePassword("Password", salt);
      System.out.println(" Password 1 -> " + password1);
      System.out.println(" Password 2 -> " + password2);
      if (password1.equals(password2)) {
         System.out.println("passwords are equal");
      }
   }
}
```

7. Adding Permissions and Creating Policies in Java

Java has a mechanism for assigning specific permissions to programs using the Java security Manager to enforce specific security policies.

Info: The security manager is a class that allows applications to implement a security policy. It allows an application to determine, before performing a possibly unsafe or sensitive operation, what the operation is and whether it is being attempted in a security context that allows the operation to be performed. The application can allow or disallow the operation.

The security manager can be activated either through the command prompt when we execute a java program, but also can be created within the Java application by creating a security manager object.

Furthermore, we can also create custom permissions that fit better our Java application and add appropriate security policy rules to grad such permissions in the Java program or the JVM in general.

Java has a default policy file that is used whenever we activate the security manager. This default security policy file `java.policy`—found in the `/path/to/java.home/lib/security` directory on UNIX-like systems and its equivalent on Microsoft Windows systems—grants a few permissions (reading system properties, binding to unprivileged ports, and so forth).

A user-specific policy file can also be created and should be located in the user's home directory. The union of these policy files specifies the permissions granted to a program. Also, the `java.security` file can specify which policy files are used. If either of the systemwide `java.policy` or `java.security` files is deleted, no permissions are granted to the executing Java program.

A permission inside a policy file should have the following structure (in simplified syntax):

```
grant signedBy "signer_names", codeBase "URL",
        {
      permission permission_class_name "target_name", "action",
```

```
        signedBy "signer_names";
    ...
};
```

- A `signedBy` value indicates the alias for a certificate stored in the keystore
- A `codeBase` value indicates the code source location; you grant the permission(s) to code from that location. An empty `codeBase` entry signifies "any code"; it doesn't matter where the code originates from.
- A **permission entry** must begin with the word `permission`. The word *`permission_class_name`* in the template above would actually be a specific permission type, such as `java.io.FilePermission` or `java.lang.RuntimePermission`.
- The "*`action`*" is required for many permission types, such as `java.io.FilePermission` (where it specifies what type of file access is permitted). It is not required for categories such as `java.lang.RuntimePermission` where it is not necessary--you either have the permission specified by the "*`target_name`*" value following the *`permission_class_name`* or you don't.
- The `signedBy` name/value pair for a permission entry is optional. If present, it indicates a signed permission.

Example:

Let’s assume that we want to create our own permission and be able to give or restrict access to our program based on this permission.

We can create a `CustomPermission` java file with the following class that extends the `java.security.BasicPermission` abstract class.

```
import java.security.BasicPermission;
public class CustomPermission extends BasicPermission {
    public CustomPermission(String name) {
        super(name);
    }

    public CustomPermission(String name, String actions) {
        super(name, actions);
    }
}
```

Then let’s assume that we want to create a program and assign to it an object of this permission and have the security manager check if the policy dictates that the program should be executed if this permission is granted.

```
import java.security.BasicPermission;
public class program {

 public static final String OPERATION = "my-operation";
```

```
 public static void main(String[] args) {
 try {
      CustomPermission cpermit=new CustomPermission(OPERATION);

      SecurityManager securityManager = new SecurityManager();
      System.setSecurityManager(securityManager);

       if (securityManager != null) {
       securityManager.checkPermission(cpermit);
      // program code goes here
      System.out.println("Operation is performed");

        }
      } catch (SecurityException se) {
           System.out.println(se.getMessage());
       }
 }
}
```

In the above code, we create a new `CustomPermission` object that has the target name my-operation. Then we create the at security manager object and assign it as the defauly security manager. Finally, we check that the created permission has been granted in the java security policy using the `securityManager.checkPermission` method. If it is not granted, a `SecurityException` exception is thrown and handled using the try- catch statements inside the code.

For the above code to be executed there must be a security policy permission entry in the Java policy file similar to the following:

```
grant codeBase "file:<our-code-source>" {
    permission CustomPermission "my-operation";
};
```

Info:
We can execute the java program with a custom security policy by providing the following lines when starting the program:

```
java -Djava.security.policy==test.policy -
Djava.security.debug=access -cp . program
```

Note also that I've activated the `java.security debug` mechanisms so as to get printed in the system out the access that have beengiven or denied when the program is been executed.
In the above example, a custom security policy file has been created (test.policy) and we assume that the Program has an active security manager object.

Task 3

Modify the Listing 2 program so that it can have appropriate permission that will allow it to be executed only when it is stored in a specific folder in a computer's hard drive.

You can create a security manager object, appropriate permission and policy entries to achieve this goal.