Authentication

02239 Data Security

RAGHAV PATNE *s161227*

Consulted s103652, Mikkel Bilgrav Andersen
for choice of the java library for password hashing

15-11-2017

Contents

1 Introduction

Using passwords for authentication is the most widely used method for authentication. It is simple for the user to use. Passwords can take many forms, from strings of combinations of letters and special characters to meaningful phrases. But it is not a full proof secure way for authentication, as it is prone to many different kinds of attacks. If a password, falls into the hands of malicious entities, then a valid user is compromised, and it can bring a lot of harm to him/her.

Apart from this, a hacker could also carry out various kinds of attacks to either assume the identity of their victims, or take full control of the User's credentials. Some of the most common types of attacks are :-

1. **Dictionary Attacks** - In a dictionary attack the intruder prey's upon users having very trivial combination of characters as their password. For example; a '0000','1234' or a 'qwerty'. Sometimes, intruders can even get past the encryption, by observing the hash values of such simple combinations.
2. **Inferring likely Passwords for a user** - in this type of an attack the intruder will try to guess the password for a user based on some prior information about the user gained through unfair means.
3. **Brute Force Method** - In a brute force attack, the intruder tries all possible combinations of the domain space of the password to guess the password of her victim. But this type of attack requires a lot of time, specially when the password is a complex combination of letters, numbers and special characters.

Biometric methods offer a very lower error rate, when it comes to authentication. But, there are ways to assume the identity of another person on an biometric system also. Attaining a full-proof secure system thus may seem an impossible task, but we certainly can reduce the risk of intruders by using combinations of various methods.

Prevention by **Fencing** could be one of the methods. For example, in operating systems, the critical areas of the memory are fenced off ,i.e a certain section of the memory is not mapped by the O.S for use by the end user. Sometimes the partitioning may be at the hardware level itself, and that provides an even better cover.

Making **Access layers** or **Access control models** can also reduce intruder risk. Certain critical resources of the system could be only given access to by detailed verification and authentication, while other non-critical ones could be semi-compromised.

But these things are very expensive to implement, and thus, cryptography offers a cheaper, and faster solution.

2 Authentication

In a Client-Server architecture, the three areas of concern for the authentication mechanisms is **Password Storage, Password Transport** and **Password Verification**.

There are 3 possible solutions to the problem of password storage:-

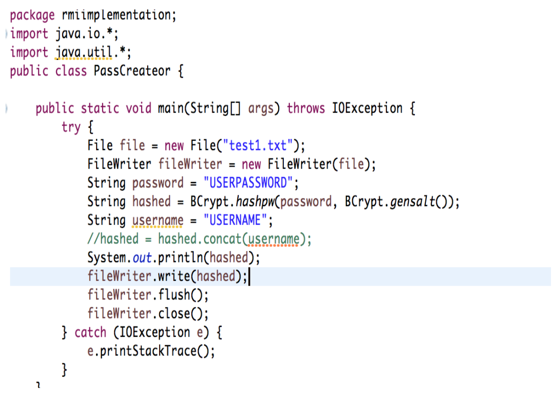
1. Using the **operating system security mechanisms** for storing passwords in a system file.
   - The O.S could provide some security features, like hiding the file from the users,i.e making it unreadable or unwritable.
   - Paging Algorithms are implemented when mapping files to memory locations. This masks the memory location of the password file from the users.
   - Otherwise, the O.S can store the password file in the section of the memory which is not mapped for the end-user.
   - Access-level models are generated by the O.S, which delegate, what are the rights of every user for a file in the memory.
2. Storing passwords in a **database management System** is a more secure option than the earlier one.
   - This is because, the DBMS provides all the features discussed above for the O.S.
   - On top of that, we can use cryptography to store the passwords also.
   - Access levels can be managed on the DBMS side of the architecture, thereby reducing the load on the Server side.
3. **Storing passwords on a "public" file** where cryptography is used is also a viable option.
   - Hash functions are a good option for this case. This is because it is not easy for trace back a password from its Hash Value.
   - Hash Functions also provide collision resistance, meaning that, if for a password m the hash value is H(m) then it is very hard to find a different password m' such that H(m)=H(m')
   - At the same time, the Hash value of the password m remains the same, every time it is implemented.
   - The hash can be computed by adding a certain salt value to the password, when passing it to the hash function. This makes it even harder to guess the password. So, basically, for a password string m,H is computed by the function H(m+salt). Salt value may be fixed and generated on the server side.

For my implementation, I used the third option, for the problem of Password Storage.

Password Transport and Verification : This is also an important area for the design of the security infrastructure. A password can be compromised if it is not encrypted on the communication channels. Packet Sniffers can pick up the data from the communication line. If the information being sent is encrypted, then the eves-dropper cannot cipher the password until the encryption is broken. Thus, an HTTP with SSL certification secured connection, which is a industry standard, should be used for password transport. The verification mechanism for the password can be implemented on the server side. Only the hash values of the passwords can be stored, and be compared to the hash values of the incoming requests. This can be done, since the output of the hash function is unique for every string m.

3 Design And Implementation

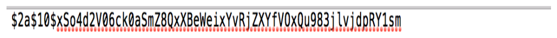
For my implementation, I first implemented a simple **RMI**[1] Client and Server. As discussed earlier, I am storing my passwords in a public file, but using cryptography. After consultation with a colleague, i decided to use the **jBcrypt**[2] package to get the hash values of the passwords. I used the method *BCrypt.hashpw(password, BCrypt.gensalt());* for doing that. It takes the password string and a salt value as arguments. This package uses the **OpenBSD Blowfish Hashing Algorithm** [3]. The package also has the method *BCrypt.checkpw(candidate, hashed)* to check if the *candidate* password matches with the hashed password. For simulation purposes I initialized only 1 password and stored it in the file *"test1.txt"*. The initialization code is as follows and also the output of the file.



```
package rmiimplementation;
import java.io.*;
import java.util.*;
public class PassCreateor {

    public static void main(String[] args) throws IOException {
        try {
            File file = new File("test1.txt");
            FileWriter fileWriter = new FileWriter(file);
            String password = "USERPASSWORD";
            String hashed = BCrypt.hashpw(password, BCrypt.gensalt());
            String username = "USERNAME";
            //hashed = hashed.concat(username);
            System.out.println(hashed);
            fileWriter.write(hashed);
            fileWriter.flush();
            fileWriter.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
```

(a) Code for initializing the test file containing the passwords



```
$2a$10$xSo4d2V06ck0aSmZ8QxXBeWeixYvRjZXYfVOxQu983jlvjdpRY1sm
```

(b) Hashed Password stored in test1.txt

Figure 1

Further, I implemented the verification logic on the server side implementation of the interface. The client side has only the implementation of the simple menu. The client side uses the **Naming.lookup service** for accessing the data objects implemented in the Server-side Implementation of the Interface. The Client Stub and the Server Skeleton are implemented by the JDK on the fly and thus need not be implemented seperately. The architecture of the system is as shown[4].

[1]https://docs.oracle.com/javase/tutorial/rmi/
[2]https://github.com/jeremyh/jBCrypt
[3]https://man.openbsd.org/crypt.3
[4]obtained from —> https://way2java.com/rmi/java-rmi-architecture/

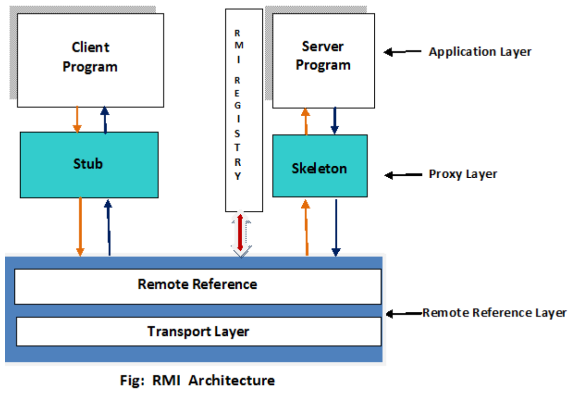


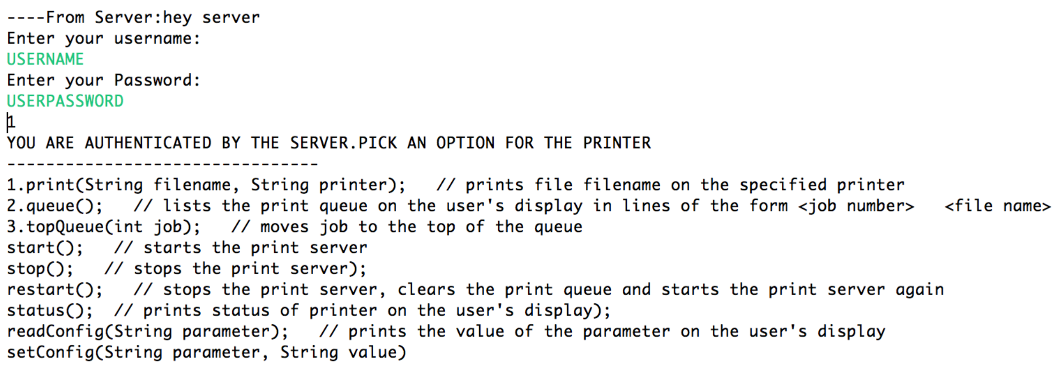
Figure 2: RMI ARCHITECTURE

4 Implementation

The server side contains the logic for the hash function, and also the verification. This insures that the client side does not learn about the hash function at all. This in turn ensures that an intruder cannot carry out a **Dictionary attack** from the client side.

In my implementation, the Servant class has a method called authenticate, which basically reads the test1.txt file, and compares it with the password provided by the user, using the *Bcrypt.checkpw* method. The result of this verification is a value returned by the function. The client side then decides based on this value, on whether to show the further menu to the user or not. Since,This is implemented in the class which implements the interface, it can't be changed about by the client. Thus, in the implementation i have successfully authenticated 1 user, whose password was stored in the test1.txt file. Although i should ideally have shown more users, but due to a few time constraints and personal reasons i couldn't show it. But, I feel, I can do that as well, because that is just a matter of coding trivial logic in the implementation.

The output obtained on the console was as follows.



```
----From Server:hey server
Enter your username:
USERNAME
Enter your Password:
USERPASSWORD
1
YOU ARE AUTHENTICATED BY THE SERVER.PICK AN OPTION FOR THE PRINTER
--------------------------------
1.print(String filename, String printer);   // prints file filename on the specified printer
2.queue();   // lists the print queue on the user's display in lines of the form <job number>   <file name>
3.topQueue(int job);   // moves job to the top of the queue
start();   // starts the print server
stop();   // stops the print server);
restart();   // stops the print server, clears the print queue and starts the print server again
status();  // prints status of printer on the user's display);
readConfig(String parameter);   // prints the value of the parameter on the user's display
setConfig(String parameter, String value)
```

Figure 3: Output of the implementation

5 Conclusion

Thus, all in all I was able to meet most of the requirements required of the assignment. A successful RMI implementation was made, with a certain level of authentication for the users. Although the creation of the log file, and a simple implementation of the printer functions could have been done, but I feel, given more time, i would be able to do it as well. I am sure that I have understood the objectives of this part of the course and can move on the next part of the course which entails access control methods.