02239 - Data Security

Authentication Lab

s171783 - Onat Buyukakkus
s181314 - Altug Tosun

01.11.2018

Contents

1 Introduction

In every application of a client/server architecture, authentication is a problem that needs to be addressed. The client needs to authenticate the server, and the server needs to authenticate the client in order to prevent attacks. In the context of this assignment, within an enterprise business there are often lots of tools and accounts being used day to day by people within the company, such as printing services. So the problem is how to manage all of these identities and ensure that you can trust that a hacker is not intercepting an employee's email or printing service for malicious purposes. In order to securely sign a user in with a password it is a good idea to add an encryption layer on top of the password authentication.

A possible approach for the enrycption layer can be adding Secure Sockets Layer (SSL) since RMI uses SSL for client authentication. SSL is a security protocol that allows sensitive information to be transmitted securely. Also, it allows you to secure the data exchange session and allows transparent client identification. Normally, data sent between client and servers is sent in plain text—leaving you vulnerable to eavesdropping. If an attacker is able to intercept all data being sent between a client and a server, they can see and use that information.

When the SSL session is established, the client and the server exchange certificates. The server can check the certificate's validity and find out if the user who presents the certificate is allowed to access the server. An ideal exchange would include the client sharing its public key certificate with the server. By this way the server would be able to authenticate client.

An authenticated connection between the client and the server is not the only concern of this solution. Another important aspect that needs to be handled securely is password security i.e. password storage, transportation and verification.

Our solution for password storage will be using SHA-512 hashing algorithm with 64-bit salts in 1000 hashing rounds and storing hashed passwords and salts in a database. We assume that the database connection will be secure. We also assume that all users are already registered in the system so they exist in the database.

2 Authentication

2.1 Password Storage

For storing passwords on the server side, we will consider three options.

2.1.1 System File

This approach considers storing passwords within a file in the operating system. With this approach one can use built-in permission and access systems of operating systems to ensure confidentiality and integrity of the stored data.

One of disadvantages of this approach is that there are not enough ready-to-use libraries and functionality of programming languages for efficient I/O utilization. This utilization should include inserting and searching which may be a problem with huge amount of registered users. So, one would need to implement an efficient and secure I/O utilization.

Another disadvantage of this approach is the fact that system files are open for physical theft. Even though operating system does not allow unauthorized users to access this system file, an attacker may get access to the user credentials if he can get hold of the physical hard drive.

2.1.2 Public File

This approach considers storing passwords within a public encrypted file in the operating system.

This approach eliminates the possibility of the physical theft discussed with system file approach. Even if an attacker can get hold of the physical hard drive, he will not be able to get access to the user credentials since they are encrypted.

However, as discussed with system file approach, public file approach also requires implementing a sophisticated (efficient and secure) I/O utilization.

2.1.3 DBMS

This approach considers storing passwords in a database. By this way, confidentiality and integrity of the data relies on the DBMS. One can either store passwords unencrypted or encrypted.

This approach eliminates all disadvantages discussed for system file and public file approaches. We will not be need to implement an I/O utilization system since Database Management Systems provide such tools. And also the encryption on the data stored in the database resolves the issue of physical or digital theft.

Since this is the most viable and feasible option out of three, in our implementation we will use DBMS.

2.2 Password Transport

In a system where clients are sending passwords to the server, transportation of passwords plays a major role. It is crucial to utilize a secure protocol system between the client and the server in order to avert man-in-the-middle attacks.

As mentioned in Section 1, it is a a good idea to add an encryption layer on top of the password authentication so that password transportation happens through a channel of confidentiality and integrity.

SSL(Secure Sockets Layer) is the standard security technology for establishing an encrypted link between a server and a client. Since we do not assume that there is a secure channel between client and server, SSL is a key factor to provide secure password transportation for our work.

2.3 Password Verification

Password verification is a very important part of the authentication process and it changes for different approaches of password storage. Since we have decided to go with DBMS approach for password storage, we will store hashed passwords and generated salt values in the database with usernames. We will retrieve corresponding hashed password and salt to given username. Then we will generate a hash value with given password and retrieved salt. To verify password, we will check generated the hash value to the hashed password retrieved from the database.

3 Design and Implementation

3.1 Design

Since our implementation of RMI will be over SSL, server-client communication security is reached. As for authentication of invocation, we will authenticate users for every remote method invocation, so we will pass user credentials with methods through SSL sockets.

As covered in Section 2.1.3, we will use a DBMS for storing passwords. Reasoning behind this approach is the fact it is easy to implement since there are already a lot of tools available and connection between server and the database is secure. We will store encrypted password in the database so that we will get in front of physical or digital theft. Storing passwords in a database also allows portability of the private credentials. We assume that users are already created in the database.

Passwords will be hashed with SHA-512 hashing algorithm with 64-bit random salts in 1000 re-hashing rounds. Only hashed password and generated salts will be stored in the database with the corresponding username.

Figure 1 consists of the sequence diagram for an example remote method invocation (`print`).

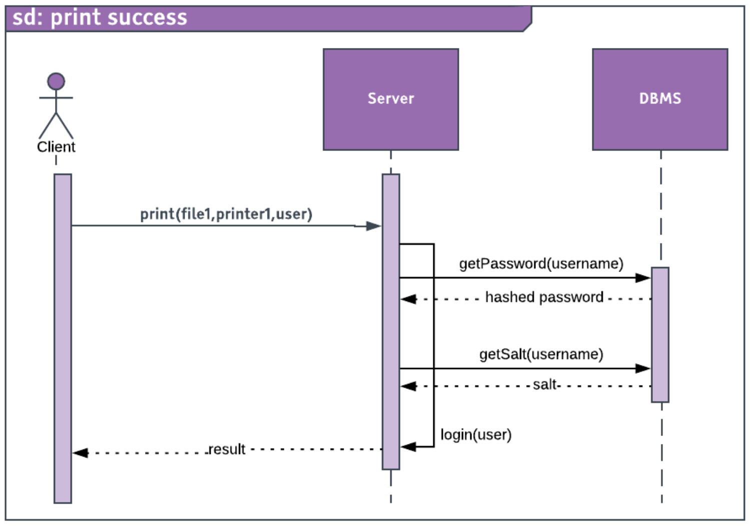

Figure 1: `print` method success sequence diagram

3.2 Implementation

3.2.1 Hashing

As mentioned in the Section 3.1, we decided to use SHA-512 hashing algorithm with 64-bit random salts in 1000 hashing rounds.

We have created a singleton class for authentication. We have used `java.security.MessageDigest` library to hash passwords. Figure 2 is the code snippet for hashing passwords.

```java
public String hashWithGivenSalt(String password, String salt) throws
    ↪ NoSuchAlgorithmException, IOException {
    byte[] byteSalt = base64ToByte(salt);
    MessageDigest digest = MessageDigest.getInstance("SHA-512");
    digest.reset();
    digest.update(byteSalt);
    byte[] digestedPassword = digest.digest(password.getBytes("UTF-8"));

    for (int i = 0; i < 1000; i++) {
        digest.reset();
        digestedPassword = digest.digest(digestedPassword);
    }
    return byteToBase64(digestedPassword);
}
```

Figure 2: Java code for hashing passwords

3.2.2 Database

We have decided to use SQLite since it is easy to use and implement. We have created a table for users with fields `username`, `password` and `salt`. Field `password` is for hashed passwords.

We have created a utility class for database connection. We have used `java.sql` library and Figure 3 is the code snippet for retrieving hashed password of the given user from the database.

```java
public static String getPassword(String username) {
    Connection c = null;
    Statement stmt = null;
    String password = null;

    try {
        Class.forName("org.sqlite.JDBC");
        c = DriverManager.getConnection("jdbc:sqlite:printauthentication.db");
        c.setAutoCommit(false);

        stmt = c.createStatement();
        String sql = "SELECT␣password␣FROM␣users␣" +
                "WHERE␣username='" + username + "';";
        ResultSet rs = stmt.executeQuery(sql);

        if (rs.next()) {
            password = rs.getString("password");
        }

        rs.close();
        stmt.close();
        c.close();
    } catch (Exception e) {
        System.err.println(e.getClass().getName() + ":␣" + e.getMessage());
        System.exit(0);
    }
    return password;
}
```

Figure 3: Java code for retrieving password of the given user

3.2.3 Transport

To be able to transport password through a channel of confidentiallity and integrity, we have used `RMIServerSocketFactory` and `RMIClientSocketFactory`. Code snippet for `ApplicationServer` class is shown in Figure 4.

```java
public class ApplicationServer implements RMIServerSocketFactory, Serializable {

public ApplicationServer() throws IOException {
    RegistryImpl impl = new RegistryImpl(5099);
    Naming.rebind("rmi://localhost:5099/printauthentication", new ServiceImpl());
}

public ServerSocket createServerSocket(int port) throws IOException {
    ServerSocketFactory factory = SSLServerSocketFactory.getDefault();
    ServerSocket socket = factory.createServerSocket(port);
    return socket;
}

public static void main(String[] args) throws IOException {
    new ApplicationServer();
}
}
```

Figure 4: Java code for RMI over SSL server

3.2.4 Authentication

To be able to authenticate invocations, we have decided to change the interface so that each remote method will also pass a serializable `User` object. Each method will try to authenticate user first then if authentication succeeds, it will execute its functionality. We have considered each request as its own and we have not adopted a session concept.

A request from the client for remote method invocation is shown in Figure 5.

```java
Service printService = (Service) Naming.lookup("rmi://localhost:5099/printauthentication
    ↪ ");
    User onat = new User("onat", "123456");
    System.out.println("---" + printService.print("file1", "printer1", onat));
```

Figure 5: Java code for RMI over SSL client

After receiving above request, Print Service will try to login the given user by

1. Accessing database to retrieve hashed password and salt.
2. Hashing given password and retrieved salt.
3. Comparing newly created hash value to the retrieved.

Figure 6 shows the code snippet of above process for `print` method.

```java
    public String print(String filename, String printer, User user) throws IOException,
        ↪ NoSuchAlgorithmException {
        if(!login(user)) return "Authentication␣failed.";
        PrintJob printJob = new PrintJob(filename, printer);
        this.printQueue.add(new Pair<Integer, PrintJob>(this.jobId++,printJob));
        return "File␣added␣to␣queue.";
    }

    public Boolean login(User user) throws IOException, NoSuchAlgorithmException {
        String storedPassword = SQLiteJDBC.getPassword(user.getUsername());
        String storedSalt = SQLiteJDBC.getSalt(user.getUsername());
        String hashed = Authentication.getInstance().hashWithGivenSalt(user.getPassword(),
            ↪ storedSalt);
        return(hashed.equals(storedPassword));
    }
```

Figure 6: Java code for `print` and `login`

4 Evaluation

If we create two users with same username and different passwords and try to invoke methods with two different credentials, only the user with the right password will be able to invoke the method. As an example we have created two users in Figure 7. First created user has the right password that is hashed in the database.

```java
    Service printService = (Service) Naming.lookup("rmi://localhost:5099/printauthentication
        ↪ ");
        User onat = new User("onat", "123456");
        User intruder = new User("onat", "password");

        printService.start(onat);
        printService.start(intruder);
        printService.print("file1", "printer1", onat);
        printService.print("file11", "printer1", intruder);
```

Figure 7: Java code for RMI test

We created log files on the server side for each remote method. Whenever a remote method is invoked, log file is appended with either authentication success or fail. Figure 8 shows the log file for code snippet in Figure 7.

```
    Nov 10, 2018 5:51:17 PM server.ServiceImpl <init>
    INFO: Server started.
    Nov 10, 2018 5:52:11 PM server.ServiceImpl start
    INFO: Authentication successful for method: start user: onat
    Nov 10, 2018 5:52:11 PM server.ServiceImpl start
    INFO: Authentication failed for method: start user: onat
    Nov 10, 2018 5:52:11 PM server.ServiceImpl print
    INFO: Authentication successful for method: print user: onat
    Nov 10, 2018 5:52:11 PM server.ServiceImpl print
    INFO: Authentication failed for method: print user: onat
```

Figure 8: Log file for example test

As it can be seen in Section 3.2 and above test, we were able to satisfy following requirements:

- *DBMS:* Passwords are hashed using SHA-512 and 64-bit salt with 1000 re-hashing and stored in the database. If needed for verification, hashed password and salt are retrieved from the database.
- *Password transport:* Each remote method invocation is authenticated before executing by passing user credentials. Information for each invocation is logged into a log file (Figure 8). As mentioned in Section 2.2, we have used SSL over RMI which means that we established communication between the client and the server from SSL sockets. This allowed us to establish confidentially, integrity and availability in our channel between the server. By this way we were able to securely transport usernames and passwords between the client and the server.
- *Password verification:* Transported password is hashed with the corresponding salt value in the database to username. For verification, hashed password is checked with stored one in the database.

5 Conclusion

As shown in Section 4, our implementation successfully authenticates valid users for remote method invocation and denies access for invalid users. This is done by storing passwords in a strong hashed form in database. By this way, in case of the database being compromised, passwords are still safe. So, we were able to satisfy authentication requirement with password storage and verification.

Also, we have used SSL over RMI as mentioned in Section 4, which means that requirement of transporting passwords through a channel of confidentiality and integrity is met.