Data security: Lab 2

Autumn 22, 02239
November 4, 2022

s174391, Sölvi Pálsson
s174434, Mads Esben Hansen
s180857, Ditte Aarøe Jepsen
s221969, Rémi Lejeune

Danmarks
Tekniske Universitet

Contents

1 Introduction

In our day to day life, people interact with multiple client/server applications each day. While some interactions are for simple and non-critical purposes, other client/server applications are highly critical. Examples of the latter could be cash withdrawal from ATMs, where it should be ensured that no other person than the account holder of the respective credit card should be able to withdraw money. In order for a server to protect a user from fraud and misuse authentication is used to *verify* that no other person is maliciously claiming another user's identity. Authentication mechanisms can make use of three means to confirm an identity depending on the critical level of security required:

- Something the user knows - *passwords, factual answers to personal questions.*
- Something the user is - *Biometrics (physical characteristics of the user).*
- Something the user has - *Identity badges, physical keys, a driver's license, etc.*

Following a confirmed client authentication in client/server applications the server will be able to determine and allow only the authorizations assigned to this specific client.
This laboratory exercise will focus on secure user authentication mechanisms through design and implementation of a password based authentication for a print server, i.e. authentication through *something the user knows*. For this purpose, the problems of password storage, password transport, and password verification are crucial aspects to consider in order to ensure a secure authentication implementation.

2 Authentication

Password based authentication is one of the most commonly used authentication mechanisms partly due to its simple structure. However, the advantages in ease of use comes with several limitations as a mean to protection from identity attacks. Some major vulnerabilities on the client side include risk of disclosure, revocation, and loss of passwords [1]. Furthermore, the strength of a password is determined by how many guesses are required to guess correctly. Many studies show that people tend to use simple heuristics and only include a few of the number of bits available when creating easy to remember passwords. This behavior makes it relatively straightforward for an attacker to obtain unauthorized access to another person's password. On the server side, the vulnerabilities within storage, transport, and verification of passwords must be carefully considered in the implementation of the system.

2.1 Password Storage

Servers must hold some information that enables matching of a client-supplied password. However, this data should by no means be stored in plain text as this constitute a major risk of disclosure. Instead, the data should be stored in obscured format regardless of the specific choice of storage method using (one-way) hashing functions as a first line of defense, to ensure that one can not obtain the plain-text password. The input to the hash function should also include a salt element, such that the risk of two identical hash values are not generated in case two clients have the same (weak) password. Going forward "password" refers to hashed values of the original password.

Using a **system file** to store passwords relies on the operating system/file system protection mechanisms to ensure the *confidentiality, integrity* and *availability* (CIA) requirements for the password data. In this context, the system file method might be a bit too cumbersome for the purpose in that it requires that the server is able to access the data and that only the admin/root has writing permission to add new users. However, this also constitutes the advantage of the system file storing method; by default, normal users will not be able to access the system file, which is only accessible for the admin/root - given that the OS protection mechanisms are adequate.

Storing password data in a **public file** would simply be implemented by mapping a user to the (hashed) password in a file record. Since it is a public file the developers of the application would in practice configure authorizations for the specific data. Thus, this method is more vulnerable with regards to all aspects of CIA. Confidentiality is to some extent ensured initially through hashing and supported further by a configuration of reading restrictions. Moreover, the availability of the data must be ensured by preventing deletion of the file, and finally strictly configured writing permissions should prevent breaches of the integrity principle. Practical risks of breaching CIA principles include mistakes in the access configuration, lack of access control maintenance (e.g. if reading rights are not terminated for a former employee), and also the fact that more users with file access yields a higher risk of unauthorized access by an attacker.

Likewise, when password data is stored in a **DBMS** the level of security also relies heavily on the access control mechanism to ensure CIA requirements, such that system adminis-

trators are not able to modify a client's password and thereby obtain access to the server and prevent the client from accessing. Though in the print server context this solution could easily be implemented using a small DBMS like SQLite and setting the necessary reading/writing permissions. Since the function of a print server is not of the highest critical function and the number of people that should be granted access would be rather limited, this would be a sufficient storage method.

Based on the considerations above it has been decided to use the public file method for storing password data in this print server application. The public file can be implemented locally relatively easily enabling writing new client data and reading the file. In this lab context, it is not necessary to consider long-term access maintenance in order to implement and test an authentication mechanism in the print server case. Though, in larger environments, this solution requires more attention on access control due to the nature of its availability as previously outlined.

2.2 Password Transport

When a client prompts a password it must be transported to the server which constitutes another risk factor in the protection of passwords. Thus, in all cases a secure communication *between* client and server should be ensured, which could be implemented through encryption of communication between the two entities by means of public/private keys. This implementation assumes that the communication is secure by other means, e.g. TLS with server-side certificates.

To limit the number of times the password authentication process is required a system can make use of session authentication, this works when a user logs in: they will receive a session ID that they can they use to authenticate. When using session authentication all subsequent messages exchanged are implicitly authenticated using the session ID which is terminated and invalid upon sign-out, more details in 3.2.4.

2.3 Password Verification

The password verification mechanism is no matter choice of storage method based on some way of matching the client provided user name and password to the corresponding credentials in the system. To authenticate a client the server compares the hash derived from the user's password against the stored hash values.

3 Design and Implementation

The goal of the software is to be able to run a print server with everything that it entails. The server should be callable from a client that is able to interact with the said server in a reasonable manner. Besides this, few strict requirements are given, and as a result, this can be implemented in many ways.

3.1 Design

We are asked to design and implement a system, where the print server function as one would expect in a small company, meaning it should be callable from possibly multiple clients and should manage possibly multiple printers. We believe the best, and most natural approach is to have a main server manage the print server. This main server should not hold any of the print functionalities but merely facilitate communications between clients and the print server. The print server should then be contactable from any client that has access to the local network of the main server.
In order to achieve this, we define a main server, `ApplicationServer`, that has ownership of the print server, `HelloServant`. Any client, `Client`, that has access to the local network should now be able to contact the print server. All authentication and/or print functionalities are handled between the print server and client directly, the only job of the main server is to facilitate communication pathways. Figure 1 shows an overview of the architecture design and the objects involved.

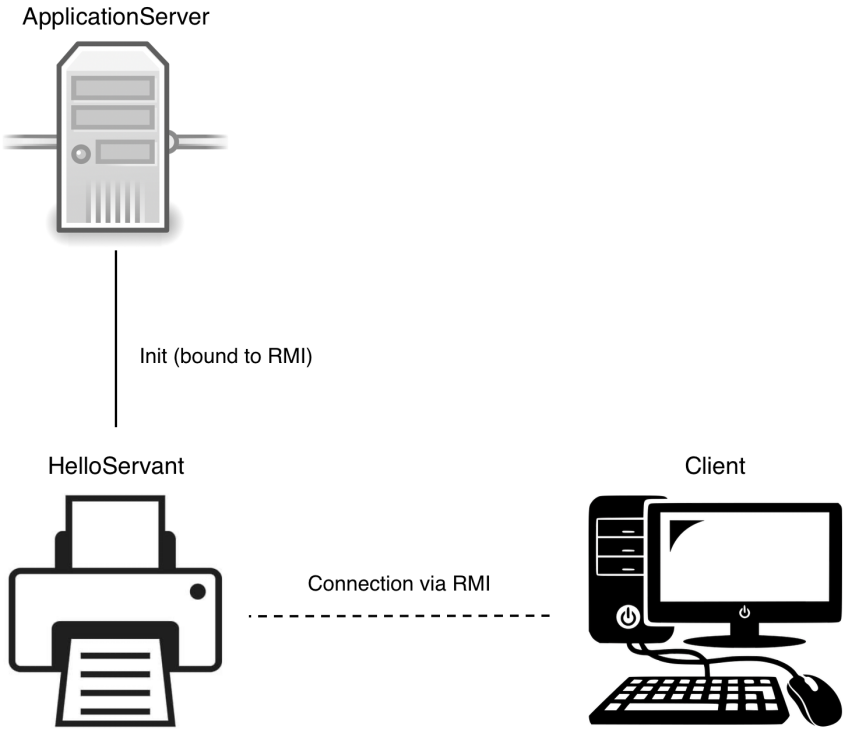

Figure 1 – Overview of design architecture.

3.2 Implementation

In our implementation, we follow the hints given in the problem description and use RMI[1] to set up a server that is callable from a client. Specifically, we define `ApplicationServer` that initializes a `HelloServant`, i.e., the print server. The `ApplicationServer` binds the *HelloServant* to an RMI registry that is connectable locally. The `Client` is now able to connect to the print server via the local reference using the RMI framework. Notice the connection between `HelloServant` and `Client` is via RMI and there is no ownership. Figure 2 shows a class diagram of the entire server/client framework, including the main server.

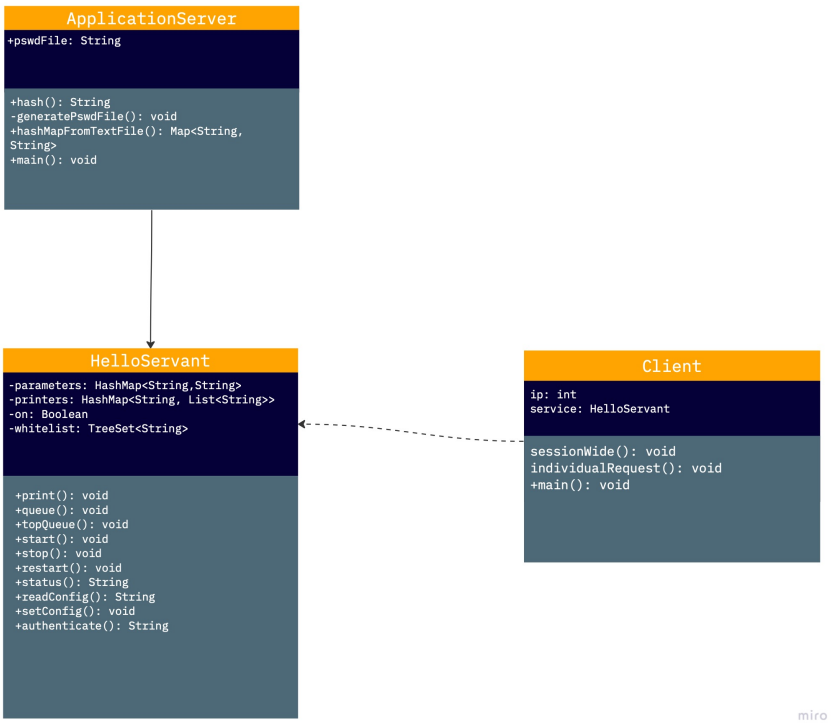

Figure 2 – Class diagram of server/client print setup.

3.2.1 ApplicationServer

The functionality of `ApplicationServer` is two-sided: Firstly, the `ApplicationServer` generates a password file with relevant users as described in section 2 that can be used for authentication by the print server. The `ApplicationServer` object holds the necessary methods to read the password file that is public and statically callable, which enables the print server to read the correct password file. Secondly, `ApplicationServer` initializes an instance of `HelloServant` that will function as a print server. This instance is bound to the RMI registry.

3.2.2 HelloServant

In our implementation `HelloServant` is the object that works as print server. As mentioned the object is initialized by `ApplicationServer`. The print server holds some pri-

[1]Remote Method Invocation, https://www.javatpoint.com/RMI

vate variables. For functionality, the print server holds a `HashMap<string, list<String»`, `printer`, that maps from the print name to the queue of said printer. Notice the queue is implemented as a simple list (specifically a LinkedList). Additionally, the print server holds a private variable `HashMap<String, String>`, `parameters`, which holds all parameters for the printer. In our implementation there are no restrictions on what parameters are definable, nor are any parameters defined by default. Finally, there are two auxiliary variables, namely `Boolean`, `on`, and `Set<Integer>`, `whitelist`. `on` defines whether the print server is started or not (defined by the `start()` method). `whitelist` is a set of whitelisted session IDs used as part of the session wide authentication protocol. Our implementation of session ID is a secure random number that is then encoded as base64 and added to the whitelist,

3.2.3 Client

The client object, `Client`, works by establishing a connection to the print server via RMI. Then it works by invoking methods in the print server. In this way, the client and server work by running two simultaneous applications. When the print server is initiated (via `ApplicationServer`) it opens a continuously running application that hosts the print server, which is called from the client.

3.2.4 Authentication

In order to call any of the methods on the print server from the client, authentication must be done. We have implemented a publicly readable password file. This file contains user ids and the hash value of the corresponding passwords. In such a setup it is paramount that the passwords are not de-hashable. Therefore, a cryptographic hash function has been used. Specifically, *SHA3-256* is used, this is simply because it is the most used cryptographic hash function that, to the best of our knowledge, is de-hashable.

System-wide

To do a session wide authentication, the client must invoke `authenticate()` on the print server. This function takes two inputs: User name and password. If the user name and password are in the public file, the user will receive a session ID (added to the `whitelist` set). This session ID is securely generated using the *SecureRandom* class from *java.security* and then encode as base64. All session wide methods take a session ID as the last input, they all check if the session ID is whitelisted, and are invoked if it is.

Single-request

If you do not want to do a session wide authentication, you can call all methods on a single request basis. All methods are overloaded such that they either take a session ID (as described above) OR a user name and password as input. If user name and password are given, a check with the password file is performed and if present the methods are invoked. For these types of calls, no session ID is whitelisted.

For both session wide and single request, the user name and password are essentially sent in plain text from client to server, hence encryption of commutation is obligatory. In practice, this can be obtained using e.g. asymmetric encryption with the public key of the print server.

4 Evaluation

This section aims to evaluate the requirements for the password storage, password transportation, and password verification as defined in section Authentication. To help with that, we have designed and implemented test cases and logged every request to the server to a log file to better see what happens each time a simulation is ran.

4.1 Password Storage

As previously stated, a public file was used to store the password data for authentication of users for the purpose of this assignment. In context of the CIA principles, the confidentiality of the password storage is protected by cryptographic means, as the passwords in the public file used in the assignment are hashed with the *SHA3-256* algorithm including salt element. The password is still protected even if an unauthorized user can read the file since it is a one way encryption, meaning that it is not possible to go back to the password from the hashed value. Adding the salt element to the hash algorithm also adds more safety, for example, attacks using rainbow tables. As stated in section 2.1, the developers for an authentication mechanism for a system using a public file should especially consider the configuration of read/write permissions to the file. Configuration of reading permission would further protect the confidentiality of the storage since it would limit who can access it. Configuration of writing permissions to the public file would further ensure the integrity of the storage since only trusted entities could modify the file. Furthermore, the availability of the file could be further ensured by making sure that it can not be deleted.

4.2 Password Transportation

As previously mentioned in Section 2.2, this implementation assumes that the communication is secure by the means of TLS with server side certificate. To minimize the risk factor of transporting the password every time a method is invoked on the server, a session ID is whitelisted on the server side after authentication of the user the first time around. The session ID can then be used for the remainder of the session for verification purposes. This increases the confidentiality of the password since the number of times it must be transported is decreased.

4.3 Password Verification

When a user tries to log in the first time around for each session, a username and a password must be transported to the server in order to verify the identity of the user. In the implementation, the server hashes the password with the same algorithm, *SHA3-256* and salt element from when the user was added to the public file. The server then reads the public file and tries to match the hashed values where the username is the same. If the server finds a match, the user is authenticated and a session ID for the user is added to a whitelist. The implemented unit tests verify that the authentication mechanism works as it should.

4.4 Testing

To be sure that the user authentications worked, we used testing. We tested every method on the server side to be sure that you needed to be authenticated for it to work. We checked

in case the user was authenticated using his password and also when he was using a session ID.

To do this we created a test class, this class tests all methods for the following case:

- If the user authenticates using a correct password
- If the user authenticates using a wrong password
- If the user authenticates using a session ID on the whitelist
- If the user authenticates using a session ID not on the whitelist

Furthermore, in the Client class, two functions are available that test individual requests and session based requests to the server, *individualRequest()* and *sessionWide()*

4.5 Logging

The logging on the server side was implemented by having a method that writes to the *.log file*. Every request to the server is recorded to a log file with information:

- Access: Granted, Denied, Server Offline
- User: UserId, SessionId
- Method: All methods defined in interface
- Timestamp: Timestamp
- Result of the request

An example of a recorded request in the log file can be seen here:

AccessGranted;UserId=usr0;Method=print();TimeStamp=2022-11-04T00:28:59.132368900
To Print file: file1 on printer: printer0

An example of a log file can be seen in appendix A

5 Conclusion

To make our application secure, we first choose to identify users using: user/password authentication. We assume that each user has a unique hard-to-guess password that is only known by them, this increases the hardness of having an intruder.

We store the users with their passwords in a file, each password is hashed using the SHA3-256 algorithm including the salt element, this means that if an intruder manages to access the file, it would be almost impossible for them to decode the passwords.

For password transportation, we assumed that the connection is secure by the means of TLS with a server-side certificate. We can also send server requests using a user unique secure random session ID, this eliminates the necessity to send our password for every request, and thus it increases the confidentiality of the password.

Password verification is done by comparing the hashed password of the user from our user/password file to the given password that we hash with the same algorithm, if it matches the user is authenticated and receives a session ID. Since hash functions have collisions it could be possible that a user authenticates with the wrong password but this probability is of $4.3 * 10^{-60}$.

Testing is done for each method that needs authentication, furthermore, each method is tested both for user/password and session ID authentication, and tested for both cases of authentication success or failure. We also have two more advanced tests (*testSessionWide*, *testIndividualRequestSuccess*), those tests check if there is no issue when multiple requests are done in a row. All of our tests combine make sure that if you are not authenticated you will not be able to access authentication required requests.

For logging, we have a file *.log* where each request is written to, the request is written even if it is denied. In the *.log* file we have one line per request, each line contains: if the access was granted, user ID/session ID, method name, time, and a sentence that explains what the user tried to do.

Appendix

A LogFile

```
------IndividualRequest()
Access Granted;UserId=usr0;Method=print();TimeStamp=2022-11-04T20:36:17.080616400 To Print file: file0 on printer: printer0
Access Granted;UserId=usr0;Method=print();TimeStamp=2022-11-04T20:36:17.083617900 To Print file: file1 on printer: printer0
Access Granted;UserId=usr0;Method=print();TimeStamp=2022-11-04T20:36:17.086617300 To Print file: file2 on printer: printer0
Access Granted;UserId=usr0;Method=print();TimeStamp=2022-11-04T20:36:17.088618500 To Print file: file3 on printer: printer0
Access Granted;UserId=usr0;Method=print();TimeStamp=2022-11-04T20:36:17.091621600 To Print file: file4 on printer: printer0
Access Granted;UserId=usr0;Method=print();TimeStamp=2022-11-04T20:36:17.099616900 To Print file: file0 on printer: printer1
Access Granted;UserId=usr0;Method=print();TimeStamp=2022-11-04T20:36:17.102617100 To Print file: file1 on printer: printer1
Access Granted;UserId=usr0;Method=print();TimeStamp=2022-11-04T20:36:17.105617900 To Print file: file2 on printer: printer1
Access Granted;UserId=usr0;Method=print();TimeStamp=2022-11-04T20:36:17.107619400 To Print file: file3 on printer: printer1
Access Granted;UserId=usr0;Method=print();TimeStamp=2022-11-04T20:36:17.112618500 To Print file: file4 on printer: printer1
Access Granted;UserId=usr0;Method=queue();TimeStamp=2022-11-04T20:36:17.116622200 To Queue For Printer : printer0
Access Granted;UserId=usr0;Method=queue();TimeStamp=2022-11-04T20:36:17.119618900 To Queue For Printer : printer1
Access Granted;UserId=usr0;Method=topQueue();TimeStamp=2022-11-04T20:36:17.124616900 To Move file: file4 to the top of the queue of printer: printer0
Access Granted;UserId=usr0;Method=topQueue();TimeStamp=2022-11-04T20:36:17.127618100 To Move file: file2 to the top of the queue of printer: printer1
Access Granted;UserId=usr0;Method=queue();TimeStamp=2022-11-04T20:36:17.129621100 To Queue For Printer : printer0
Access Granted;UserId=usr0;Method=queue();TimeStamp=2022-11-04T20:36:17.135617200 To Queue For Printer : printer1
Access Granted;UserId=usr0;Method=stop();TimeStamp=2022-11-04T20:36:17.140616900 To Stop the server
Access Granted;UserId=usr0;Method=start();TimeStamp=2022-11-04T20:36:17.143618700 To Start the server
Access Denied;UserId=usr123;Method=authenticate();TimeStamp=2022-11-04T20:36:24.316732900 To Authenticate User usr123
Access Denied;UserId=usr0;Method=authenticate();TimeStamp=2022-11-04T20:36:24.322695400 To Authenticate User usr0

------SessionWide()
Access Granted;UserId=usr0;Method=authenticate();TimeStamp=2022-11-04T20:36:24.324693 To Authenticate User usr0
Access Granted;SessionId=1086530614;Method=print();TimeStamp=2022-11-04T20:36:24.326692900 To Print file: file0 on printer: printer0
Access Granted;SessionId=1086530614;Method=print();TimeStamp=2022-11-04T20:36:24.328694700 To Print file: file1 on printer: printer0
Access Granted;SessionId=1086530614;Method=print();TimeStamp=2022-11-04T20:36:24.330694300 To Print file: file2 on printer: printer0
Access Granted;SessionId=1086530614;Method=print();TimeStamp=2022-11-04T20:36:24.332692800 To Print file: file3 on printer: printer0
Access Granted;SessionId=1086530614;Method=print();TimeStamp=2022-11-04T20:36:24.334694400 To Print file: file4 on printer: printer0
Access Granted;SessionId=1086530614;Method=print();TimeStamp=2022-11-04T20:36:24.336694300 To Print file: file0 on printer: printer1
Access Granted;SessionId=1086530614;Method=print();TimeStamp=2022-11-04T20:36:24.338730300 To Print file: file1 on printer: printer1
Access Granted;SessionId=1086530614;Method=print();TimeStamp=2022-11-04T20:36:24.340696400 To Print file: file2 on printer: printer1
Access Granted;SessionId=1086530614;Method=print();TimeStamp=2022-11-04T20:36:24.342693300 To Print file: file3 on printer: printer1
Access Granted;SessionId=1086530614;Method=print();TimeStamp=2022-11-04T20:36:24.344697100 To Print file: file4 on printer: printer1
Access Granted;SessionId=1086530614;Method=queue();TimeStamp=2022-11-04T20:36:24.346695700 To Queue For Printer : printer0
Access Granted;SessionId=1086530614;Method=queue();TimeStamp=2022-11-04T20:36:24.350703 To Queue For Printer : printer1
Access Granted;SessionId=1086530614;Method=topQueue();TimeStamp=2022-11-04T20:36:24.355698500 To Move file: file3 to the top of the queue of printer: pr
Access Granted;SessionId=1086530614;Method=topQueue();TimeStamp=2022-11-04T20:36:24.358695200 To Move file: file2 to the top of the queue of printer: pr
Access Granted;SessionId=1086530614;Method=queue();TimeStamp=2022-11-04T20:36:24.360693500 To Queue For Printer : printer0
Access Granted;SessionId=1086530614;Method=queue();TimeStamp=2022-11-04T20:36:24.364694600 To Queue For Printer : printer1
Access Granted;SessionId=1086530614;Method=stop();TimeStamp=2022-11-04T20:36:24.368696700 To Stop the server
```

Figure 3 – Example of a log file with individual requests and session requests

References

[1] C. P. Pfleeger, S. L. Pfleeger, and J. Margulies, *Security in Computing*. Prentice Hall, 2015.