

Danmarks
Tekniske
Universitet



Authentication Lab

AUTHORS

Joao Mena - s223186
Tomas Estacio - s223187
Aidana Nursultanova - s212994
Renjue Sun - s181294

November 4, 2022

Contents

1	Introduction	1
2	Authentication	2
2.1	Requirements	2
2.2	Password Storage	2
2.2.1	System File	2
2.2.2	Public File	3
2.2.3	DBMS	3
2.3	Password Transport	4
2.4	Password Verification	4
3	Design and Implementation	5
4	Evaluation	8
5	Conclusion	10
	Nomenclature	11

1 Introduction

This report describes the development and results of the second laboratory exercise for the Data Security course. In this activity we are expected to write a simple client-server application on Java using the RMI [12] functions in order to create a print server, for which we will design and implement a password based authentication mechanism, taking into account relevant problems such as the password storage, transportation and verification methods. The purpose of this activity is to put into practise some concepts covered in the lectures and provide the students with a hands on experience in the development of a user authentication mechanism.

We start by going through the importance of authentication in a secure system and reflect over relevant topics for the system that we put together, namely the the password storage, password transport and password verification. For the password storage we consider 3 methods of implementation: public file, system file and DBMS, and later on the design and implementation section we put the pros and cons of each option against each other, and decide on which one to implement. We also establish a few security requirements for our system based on the core expectations of this acitivity, and come up with some requirements of our own that we believe can be an upgrade for our system.

For the design and implementation, we explore our options, explain our design choices, such as the use of a DBMS for password storing and PBKDF2 for hashing, and expand on how we implemented them into the system. After that, on the evaluation section, we compare our expectation of the system to the final product.

Finally, in the conclusion, we go over which requirements were met and which weren't, and reflect on what can be added or improved in future work.

2 Authentication

Authentication is a very common concept in security, used when a party verifies the identity of another party. In this activity it is expected that our server authenticates a user, given a username and a secret password, in order to get access to the services available. This process is broken down in 3 different phases: the password storage, the password transport and the password verification.

2.1 Requirements

For this system we require that in order for a user to access the printer services, it must be authenticated using a username and a password that are verified by the server. For added security, the user session should also be terminated if the user is inactive for a certain amount of time, and when logging in, if the authentication fails a certain amount of times, the user should be unable to try to authenticate again for a certain amount of time. These measures are used to prevent that, for example, a user leaves their device with a session open and someone else takes advantage of that, and make it more difficult for attacks like brute force or dictionary attacks to be attempted, as it would take a lot of failed iterations which would lead to several cool down periods in between, adding computational time to the attacks.

2.2 Password Storage

For the server to be able to authenticate a user, it must already have the knowledge of that user's existence and credentials. This means that the user must be enrolled and its password must be stored. To prevent possible attackers from having direct access to the users passwords in the event of an attack where they can access the file or platform where they are stored, the passwords are salted, hashed and encrypted before being stored, and only the final encoded message is used for the password verification. More on this can be read further ahead on the report, in section 3. For this activity, we assumed that the user had already enrolled in our application and considered the following methods of password storage.[7]

2.2.1 System File

The first option considered for our client/server application is storing passwords in a "system" file, like a shadow password file, also known as */etc/shadow* on modern Linux distributions [4], or a SAM file, used on Windows desktop platform [6]. The shadow password file stores the hashed passphrase format for the user with additional properties related to the password and is only accessible to the root user, restricting the access. However, there are some set-user-ID programs that drop all of the program's user privileges and are able to access */etc/shadow* in a secure way. All fields of the file are separated by the `:` symbol and in the following order: username; passphrase, in hash format, containing the reference to the algorithm used for hashing, the salt used (a unique, randomly generated string to each

password) and the output of the hashing function used with salt and password as inputs; the time since the password was changed; the time left before the user is allowed to change his password again; the time in which the password is still valid; the time until user is warned his password will expire; the time that user is disabled after his password is expired and the time until the date of expiration of the account. [2]

Assuming now that our user is using the Windows operating system, the file that contains usernames and passwords is the SAM file. This is the method used by Windows to authenticate users during login attempts. The file is stored locally on the hard disk, its access is restricted to users that have *HKLM/SAM* and *SYSTEM* privileges and, like the Linux shadow file, all the data it contains is encrypted.

In this project, to implement this type of password storage solution, we would create a file, in which only *root* username (the Operating System administrator) has permissions to access it and we would store all those parameters the same way as discussed previously, assuring also that the password selected by the user has strong features, securing against password snooping and dictionary attacks.

2.2.2 Public File

Traditionally, on Unix Systems the */etc/passwd* file would store the value generated by a "one-way function" called *crypt()*, using the password selected by the user and an unique and randomly generated salt, to encrypt a block of zero bits. However, the file was public for all the accounts registered in the system, having permissions to read it, but no permissions to write on it, making the administrator of the Operating System the only one capable to change information. So, the security of the storage method was not in the access of the file, but only on the strength of the encryption used on the password, making it hard for hackers to try to break the algorithm, such that the *crypt()* function is still highly resistant. Inside the file, one would find user's: username, encrypted password, user identification number (UID), group identification number (GID), full name, home directory and shell. [14]

To implement this method in our project, we would create a file that can be read by all users that have registered in our system, but only the *root* user with all permissions could change the information inside it, which contains the parameters discussed in the previous paragraph.

2.2.3 DBMS

Another option considered for password storage was DBMS. Databases play an important role in the storage of information for most modern technology infrastructures, and are a common concern in security contexts due to the need to prevent against ill intended individuals from stealing, tampering or deleting the stored data.[8] Using a DBMS for the password storage, to ensure the confidentiality of the passwords we need to store, these would first need be salted, to add randomness to the key, and secondly hashed using a secure algorithm such as PBKDF2, Bcrypt or Scrypt, which use common hash functions and iteratively add complexity to make the key computationally hard to decode, and consequentially make the password inaccessible to attackers. Hashing the password by itself, without the salt, would

not be enough for it to be protected, as it would still be susceptible to brute force attacks, dictionary attacks, rainbow tables, lookup tables and reverse lookup tables.[3]

2.3 Password Transport

The client/server communication happening in our project is secured by the use of Transport Layer Security. This method combines symmetric and asymmetric cryptography and provides end-to-end security for the data sent, which means that the client and the server have their own code used to verify the integrity of each message sent, avoiding possible eavesdropping and tampering with the information shared. [13]

The communication used also takes advantage of server-side certificates, which provide a extra layer of security to the connection between client and server, by using the TLS/SSL handshake. This handshake is initiated by the client, followed with the response by the server of his TLS/SSL certificate, that is a key pair made of a public key and private key, to establish a session between them, with a unique valid session key. Then, the client uses a Certificate Authority to confirm and trust the TLS/SSL certificate received and sends an encrypted symmetric session key that the server is able to decrypt by using its own private key. Finally, both of the parties involved have a session key and have an available communication session that maintains confidentiality of the messages sent. This way, the communication is protected from Man-In-The-Middle attacks, guaranteeing that the certificate sent by the server is not expired. [1]

2.4 Password Verification

In every password storage method discussed previously, we store it inside the file or database at least three important data: the username, the salt used and the output of the hash function that takes as inputs the password and the salt. These are all used to verify the client's identity by doing the following steps:

- The client sends his username and password though the connection, secured by the TLS/SSL protocol, to the server.
- Assuming that the username is already in the database, the server receives the username and password, and searches for an equal username stored on the file/database where the users' information reside.
- After finding the entry in the file/database with the same username, we check for the salt value associated and perform the function used to encrypt the password with the password given by the client and the salt associated with the username.
- Then, we verify if the result obtained is equal to the one stored in the file/database. If it is, then the client is verified and can access to all the server's features. If not, then the server sends an error message and the client is not able to access the server.

3 Design and Implementation

Our system is composed of two main parts which are *Server* and *Client*. On the Client side, a user enters a username and password, which will be sent to the server through a secure communication channel using TLS with server side certificates. This secure communication channel was not actually implemented, we just assumed it's presence. The steps of the protocol needed for it's implementation if we were to do it manually are explained in section 2.3. The following figure illustrates the architecture of our system:

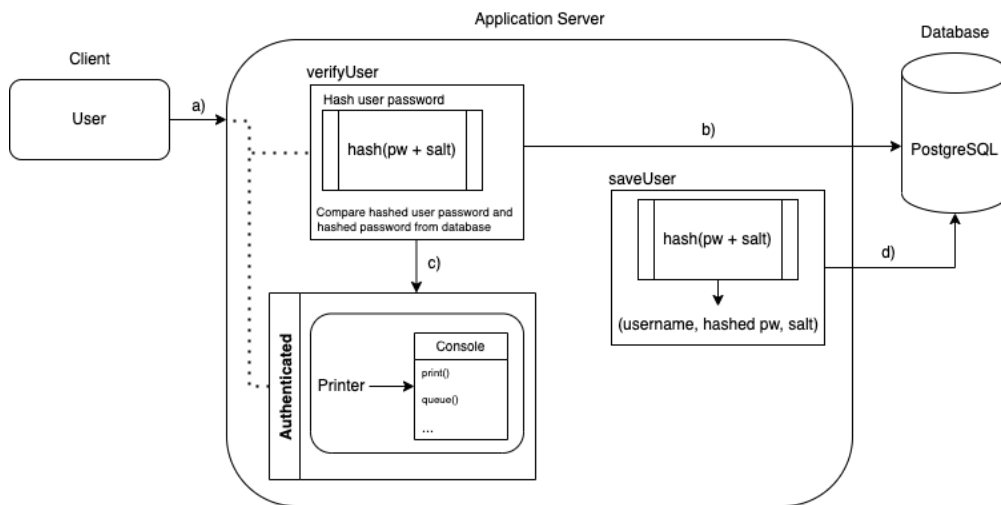


Figure 1: Architecture of our Printer application system

As it can be seen, we chose to use a DBMS for our password storage. For this purpose we used PostgreSQL [5], an open-source relational database management system. Before taking this decision, we considered the methods described in section 2.2. All three methods ensure the confidentiality aspect of the passwords stored. The public file method lacks, however, when it comes to the integrity of the information. Since any user can access the file, it can be more easily tampered with. The DBMS and system file methods are similarly adequate for our system, security wise, but storing the passwords in a single File system would be limiting for a larger system involving many users across various devices and where scalability could be needed. For this reasons, we considered the DBMS method to be the most balanced option for our needs.

We started the implementation of our system by performing the connection of the client and the server, using a Java RMI application. Firstly, we created the server program with remote and accessible objects through references. Then, we created the client that obtains the remote references to the objects on the server side, and invokes methods on them. Finally, RMI provides the form of communication between client and server, so that they can pass information back and forth.

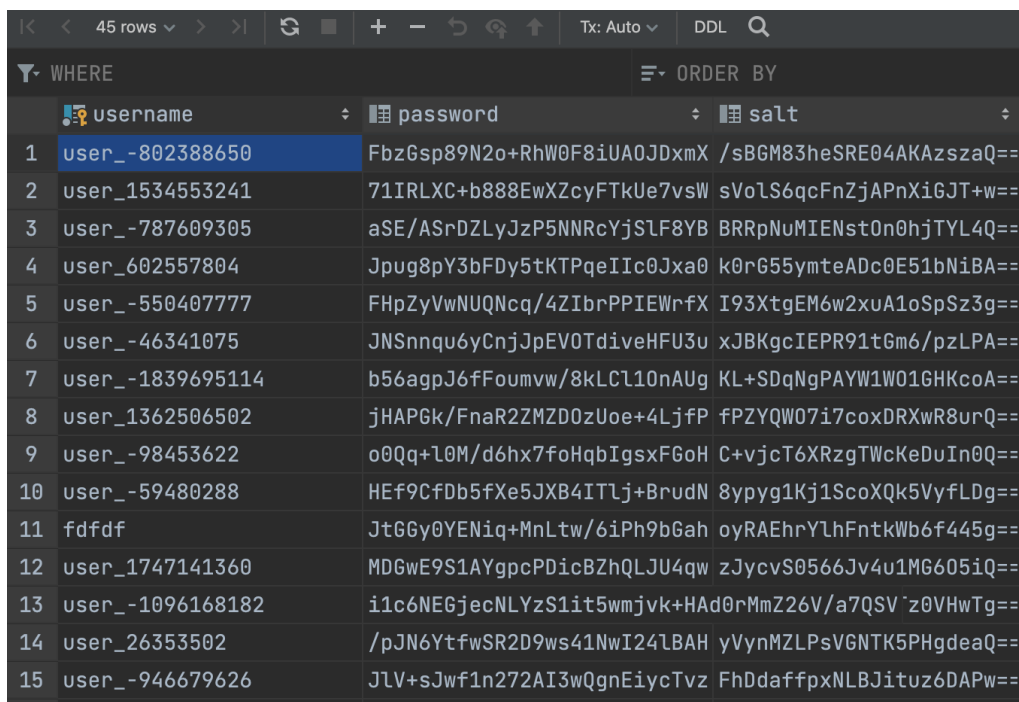
Next, we start the authentication process of the client by the server, to make sure that only authenticated users can access all the features associated with the server. The user is prompted to type his username and password and that data is sent to the server and

processed. For added security, we agreed to add a feature that generates a penalty cooldown if the user fails to successfully insert valid credentials three times in a row, making him wait 60 seconds to try to login again, in order to prevent against brute-force attacks in our system. Unfortunately, due to time constraints, we were not able to finish a working implementation of this feature.

In the server side, the data is processed as follows: The given username is used to retrieve the *salt* and the hashed password that are stored in the database associated to that user. After this, the password inserted by the user and the retrieved *salt* are hashed using the PBKDF2 algorithm with SHA-1 as a base encryption function, an iteration count of 10000 and a derived key length of 256 bits. PBKDF2 is a KDF that applies a HMAC, in this case SHA-1, to the password and salt passed as inputs, and repeats it for as many iterations as we desire, adding computational complexity to the key.[3] To implement this we used Java's Security [9] and [10] libraries.

This generated key is then compared to the one retrieved from the database and, if they match, the user is authenticated. At this point, a user session is started. For added security, if the session duration exceeds 5 minutes, it is automatically ended and the user must authenticate again to access the printer services. Ideally, the session would only be ended after a period of inactivity, and the session would not have a fixed time limit, but we were also unable to correctly implement that in time.

To populate the database, we implemented a function that takes the username and password as inputs, generates a random 16 byte *salt* and hashes the password and salt pair, with the same PBKDF2 algorithm as mentioned before. It then stores the username, hashed password and salt in the database, using SQL, resulting in the following table:



	username	password	salt
1	user_-802388650	FbzGsp89N2o+RhW0F8iUA0JDxmX	/sBGM83heSRE04AKAzsaZa==
2	user_1534553241	71IRLXC+b888EwZcyFTkUe7vsW	sVoLS6qcFnZjAPnXi6JT+w==
3	user_-787609305	aSE/ASrDZLjYzP5NNRcYjS1F8YB	BRRpNuMIENst0n0hjTYL4Q==
4	user_602557804	Jpug8pY3bFDy5tKTPqeIic0Jxa0	k0rG55ymteADc0E51bNiBA==
5	user_-550407777	FHpZyVwNUQNcq/4ZiBrPPIEWrfX	I93XtgEM6w2xuA1oSpSz3g==
6	user_-46341075	JNSnnqu6yCnjJpEV0TdivHFU3u	xJBKgcIEPR91tGm6/pzLPA==
7	user_-1839695114	b56agpJ6fFoumvw/8kLC110nAUg	KL+SDqNgPAYW1W01GHKcoA==
8	user_1362506502	jHAPGk/FnaR2ZMZD0zUoe+4LjfP	fPZYQW07i7coxDRXwR8urQ==
9	user_-98453622	o0Qq+l0M/d6hx7foHqbIgsxF6oH	C+vjcT6XRzgTWcKeDuIn0Q==
10	user_-59480288	HEf9CfDb5fXe5JXB4ITLj+BrudN	8ypyg1Kj1ScoXQk5VyfLDg==
11	fdfdf	JtG6y0YENiq+MnLtw/6iPh9bGah	oyRAEhrYlhFntkWB6f445g==
12	user_1747141360	MDGwE9S1AYgpcPDicBZhQLJU4qw	zJycvS0566Jv4u1MG6051Q==
13	user_-1096168182	i1c6NEGjecNLYzS1it5wmjvk+HAd0rMmZ26V/a7QSV`z0VHWtg==	
14	user_26353502	/pJN6YtfwSR2D9ws41NwI24LBAH	yVynMZLPsV6NTK5PHgdeaQ==
15	user_-946679626	JLV+sJwf1n272AI3wQgnEiycTzv	FhDdaFFpXNLBJituz6DAPw==

Figure 2: DBMS table structure

Throughout our code, we store the passwords sent by the user to the client, to be saved in the database, in the variable type `char[]` instead of `String`, because the first is preferred for storing sensitive information for several reasons: `String` variables are immutable, which means there is no form to change or overwrite the content of the variable, `char[]` variables are less vulnerable of being accidentally printed to some insecure place and they are able to be "sanitized", meaning that after usage one can override a clear password with junk. [11]

After authenticating the client, we authorize his access to the server features, like performing commands on the printer server. By default, we have created 3 different printers (`printer1`, `printer2` and `printer3`) that are ready to be used. The commands are sent by the user, received by the printer server and then performed accordingly. We have implemented the following commands:

- `print(Stringfilename, Stringprinter)`, the client specifies the *filename* that needs to be printed and which *printer* is used and the server prints the file and puts the printer job in the queue.
- `queue(Stringprinter)`, prints the queue list of jobs performed by the specific *printer*.
- `topQueue(Stringprinter, intjob)`, changes the order of the queue list, putting the specific *job* of the *printer* in the first position of the queue.
- `start()`, command sent by the client when he wants to initiate the printer server, only after being authenticated.
- `stop()`, command sent by the client when he wants to stop the printer server, finishing his connection to the printer server.
- `restart()`, command sent by the client when he wants to stop the printer server, clears the print queue created in the session and starts the printer server again, only after the user authenticates again with his username and password.
- `status(Stringprinter)`, shows the status (active or not) of the specific *printer* on the client's display.
- `readConfig(Stringparameter)`, prints the value of the *parameter* on the client's display.
- `setConfig(Stringparameter, Stringvalue)`, sets the *parameter* to *value*.

The final feature that was implemented to achieve better security was a connection timeout that closes the connection between the client and the printer server if the client remains inactive for 60 seconds. After that, the user has to authenticate again to regain access to the printer server.

4 Evaluation

After the design and implementation of our system, all the discussed features and requirements were evaluated by a test program done in the *Client.java*. Firstly, we tested the RMI connection between client and server on the local host and on the specified port number 5099 and successfully were able to open the communication between the two agents.

The transportation of the information from client to server and vice-versa was ensured by the TLS with server side certificates protocol, however that feature was not implemented in the system, just assumed to be part of the system.

The authentication part of the system that was described previously was fully implemented, using DBMS as the password storage mechanism, which was populated by giving random usernames and passwords to the server, and when tested, the results were as expected.

Finally, to test the sending and receiving of commands and the execution of the actual commands, we used a random username and password who was authenticated in the server to send commands from the client side and verified that the results was as predicted and all features discussed and planned were actually working, except the "penalty cooldown" feature that was mentioned.

Listing 1 shows the output of the test program created, on the client's side, as we attempted to perform the following tasks:

- authenticate one user who was already in the database, but insert the wrong password, so he did not receive access to the server.
- authenticate the same user, using now the correct password, so he receives access to the server.
- send the command *status(Stringprinter)* to the printer server, so it prints the status of the printer in question.
- send the command *setConfig(Stringparameter,Stringvalue)* to the printer server, setting the parameter to have that specific value, printing in the terminal that the configuration is being set.
- send the command *readConfig(Stringparameter)* that prints the new value for the parameter.
- send the command *print(Stringfilename,Stringprinter)* that prints the files added in the specific printer.
- send the command *queue(Stringprinter)* that prints the queue of printing jobs of the specific printer.
- send the command *topQueue(Stringprinter,intjob)* that puts the specific job in the first place of the printer's queue and prints that the job is being moved on the client's side and the queue of the printer after the change has been made.

- send the command *restart()* that stops the printing server, clears the printing queue, prints it to see that it is empty, and starts the printer server again, if the user is authenticated.

Listing 1: Output of code execution

```
1 Trying to verify user with incorrect password..
2 Authentication of user with username: user_1269463851 failed. User cannot
  use the printer
3
4 Trying to verify user with correct password..
5 User with username 'user_1269463851'successfully verified!
6
7 Starting the server...
8
9 -----STATUS-----
10 Printer server is started and can be used
11
12 setting config...
13
14 -----READ_CONFIG-----
15 The value of the parameter 'param1' : value1
16
17 -----PRINT-----
18 File 'file1' successfully added to the print queue with job number 1
19 File 'file2' successfully added to the print queue with job number 2
20 File 'file3' successfully added to the print queue with job number 3
21
22
23 -----QUEUE-----
24 job number: 1, file name: file1
25 job number: 2, file name: file2
26 job number: 3, file name: file3
27
28 moving job 2 to the top...
29
30 -----QUEUE_AFTER_MOVING_TO_TOP-----
31 job number: 1, file name: file2
32 job number: 2, file name: file1
33 job number: 3, file name: file3
34
35 restarting the server...
36
37 -----QUEUE_AFTER_RESTART-----
```

5 Conclusion

In conclusion, most requirements were met, or partially met. Our system requires a user to authenticate itself using a username and a password to get access to the printer service, takes commands from authenticated users and executes them. The system has a secure password storage implementation using DBMS, a secure password transport protocol is assumed to be in place, and a password verification mechanism is implemented and working. Our system also has a session timeout functionality. The requirement about the cooldown period after a certain amount of failed authentication attempts was not met, but could be implemented in future work, if we are to build upon this system. A role-based access control to certain functionalities of the system could also be something worth considering in future work.

Nomenclature

DBMS Database Management System

HMAC Hash-based Message Authentication Code

KDF Key Derivation Function

RMI Remote Method Invocation

SAM Security Accounts Manager

SSL Secure Sockets Layer

TLS Transport Layer Security

References

- [1] *A lock, two keys, strong identity, and Gold Standard Encryption*. URL: <https://www.digicert.com/how-tls-ssl-certificates-work>.
- [2] Rahul Awati. *What is a shadow password file?* Sept. 2021. URL: <https://www.techtarget.com/searchsecurity/definition/shadow-password-file>.
- [3] Levent Ertaul, Manpreet Kaur, and Venkata Arun Kumar R Gudise. "Implementation and performance analysis of pbkdf2, bcrypt, scrypt algorithms". In: *Proceedings of the International Conference on Wireless Networks (ICWN)*. The Steering Committee of The World Congress in Computer Science, Computer &S. 2016, p. 66.
- [4] Vivek Gite. *Understanding /etc/shadow file format on Linux - nixCraft*. Apr. 2022. URL: <https://www.cyberciti.biz/faq/understanding-etcshadow-file/>.
- [5] PostgreSQL Global Development Group. Oct. 2022. URL: <https://www.postgresql.org/>.
- [6] Katie Terrell Hanna. *What is the Windows Security Accounts manager (Sam)?* Feb. 2022. URL: <https://www.techtarget.com/searchenterprisedesktop/definition/Security-Accounts-Manager>.
- [7] Neeraj Kushwaha. *Best practices for storing passwords securely in a database*. July 2022. URL: <https://www.learncsdesign.com/best-practices-for-storing-passwords-securely-in-a-database/>.
- [8] Afonso Araujo Neto and Marco Vieira. "Towards assessing the security of DBMS configurations". In: *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*. 2008, pp. 90–95. DOI: 10.1109/DSN.2008.4630074.
- [9] Oracle. *Package: java.security*. June 2020. URL: <https://docs.oracle.com/javase/7/docs/api/java/security/package-summary.html>.
- [10] Oracle. *Package: javax.crypto*. June 2020. URL: <https://docs.oracle.com/javase/7/docs/api/javax/crypto/package-summary.html>.
- [11] Raja. *Why char[] array is more secure (store sensitive data) than string in Java?* Feb. 2020. URL: <https://www.tutorialspoint.com/why-char-array-is-more-secure-store-sensitive-data-than-string-in-java>.
- [12] *Trail: RMI*. 1995. URL: <https://docs.oracle.com/javase/tutorial/rmi/index.html>.
- [13] *What is TLS amp; How Does It Work?: ISOC internet society*. Aug. 2022. URL: <https://www.internetsociety.org/deploy360/tls/basics/>.
- [14] Alan Williamson. *What is clear text passwords and why you shouldn't store them*. Aug. 2019. URL: <https://medium.com/macclaurin-group/what-is-clear-text-passwords-and-why-you-shouldnt-store-them-e61c604b1fb7>.