Technical University of Denmark

02239

Data Security

Authentication Lab

Authors:	*Student Nr.:*
Jeff Gyldenbrand	s202790
Marcus Pagh	s151714

December 20, 2021

Contents

1 Introduction

When a service is running, it is often a requirement that the use of said service is limited to a finite set of clients. In order to enforce such a limitation, it is necessary to identify which clients belong in the allowed set and which do not. The process of authentication is used to allow a client to identify him or herself to the server running the service. Techniques to authenticate a client includes (But is not limited to) passwords (Something the client should know), various biometrics (Something the client is) and physical access cards (Something the client possesses). In the case of this lab, a password based authentication mechanism is implemented, with each client having their own unique identifier (username) which is not secret and a corresponding secret key (password) known only by the client.

In order to securely authenticate without an intruder gaining illicit access by snooping the password, a secure connection is necessary. One might argue it is enough to hash the secret password on the client side before sending it to the server, but the fact is that whatever is sent will serve as the key, which the server will authenticate the client with. Therefore sending a password as plain text is just as secure as sending a hashed version of said password. For the sake of the implementation in this lab, assumptions about a secure connection is made rather than explicitly implemented.

Storing and verifying clients passwords are also addressed throughout the lab. It would be a breach of confidentiality if a password became known to anyone but the client as well as a breach of integrity if verification was possible without the correct key/password.

Furthermore the implementation of this lab tries to mitigate human error on the client side by limiting the amount of time any client can be authenticated at one time. This is useful to prevent an intruder accessing the service from previously authenticated device, like a stolen laptop.

Lastly an attempt at preventing brute force attacks are made. While completely removing the possibility of a successful brute force attack is impossible while also maintaining availability, the implementation of this lab tries to find a balance that severely limits the risk of such an attack being successful while maintaining an acceptable availability.

2 Authentication

In this lab any communication to the printer service at all must be passed through an authenticated channel. This means it will not be possible to send or receive any data between the printer service and the outside without first proving ones identity to the server running the service. If and only if the server accepts this identity as belonging to a certified client, will the client be able to append print jobs, modify parameters et cetera.

In order to limit the possibility of a third party gaining access by utilizing a previously authenticated connection, time restricted sessions has been implemented. This ensures that any authentication will only be accepted within a specified time frame and thus the client must authenticate anew should more communication be required.

Furthermore to prevent attacks relying on brute force, a time delay has been added after a certain amount of unsuccessful attempts. While this does limit availability in the case of a forgetful client, it is but a short delay. However while this is such a short delay for the client, it severely limits the amount of guesses an attacker can do within any reasonable amount of time, rending a brute force attack unfeasible.

2.1 Password Storage

To authenticate a client with a secret key, one can be persuaded to think that the server must also know said key. Luckily that is not the case, at it would not just be a great security risk if the server was ever compromised, but also vastly inappropriate as the server admin (Or anyone with access from the inside) would be able to know the secrets of every client. Secrets that might not even be limited to this one specific service.

Instead a unique hash of the password is stored on the server. And to prevent dictionary or even rainbow table attacks, the password is concatenated with a unique salt before being hashed.

There is a number of ways to store such hashes. Below is described three such options, arguing pros and cons of each solution.

System File — Storing passwords in a system file relies on the operating system/file system protection mechanisms and are thus arguably protected against both illicit reading and writing. However there is two main caveats; It is necessary to implement some mechanism or service to provide controlled access to the data stored in the file in order to actually utilize the information stored within. Secondly any person or otherwise with root access would be able to read or even tamper with the stored credentials.

Public File — Storing passwords in a public file is the traditional way to store passwords on Unix systems. The public file can be read by anyone, but mechanisms are put in place to ensure only authorized users and/or services can modify it. Confidentiality of the passwords is normally protected by cryptographic means, not unlike described earlier. The two main caveats are similar to the ones of the system file; While it is not necessary to implement a means of modifying the public file as with the system file, it is necessary to implement a means of restricting modification of the file. Secondly as before, any user or service with root access will be able to read or even tamper with the stored credentials. While the added cryptography would ensure confidentiality even from a snooping root user, integrity could easily be broken by swapping the passwords between clients or similar illicit actions.

DBMS Passwords stored in databases are historically often not encrypted or at least not encrypted *well* and therefore relies on architecture and access control to maintain confidentiality and integrity, much like the system file. As the previous two, an ill-intended superuser will be able to breach confidentiality and/or integrity depending on the level of encryption.

It seems with every choice exists the risk of a rogue system administrator breaching integrity. While this is true, a non-compromised system administrator will have little to no motive to do so, as he or she is in place only to maintain the service provided for the clients. Worse is the breach of confidentiality, which seems easily mitigated by cryptographic means similar to those previously stated as being implemented in this lab.

When deciding a means to store credentials already secured by cryptography, it really comes down to a choice between using a database versus a file stored locally on the server.

While one could argue a database superuser could be different than the root user of the system on which the database is running, thus adding extra security by means of separation, this had little influence on the decision. While cryptographically secured files has the same advantages as a cryptographically secured database at first glance, there is a few practical advantages to using a database rather than a file:

For passwords to offer maximum security, it must be possible for a client to change them. A client might use the same passwords for multiple services and if one service is ever breached, very little security is left where ever that same key might be in use. In an environment with multiple simultaneous clients, a flat file might cause locking issues thus preventing availability of the service for the clients. Furthermore using a locally stored flat file would cause headaches if it was ever necessary or desirable to scale up to multiple servers. While there are certainly ways of mitigating those issues, solutions have already been built into most database systems.

2.2 Password Transport

Transporting sensitive information like passwords between a server and a client requires a secure connection. There is a number of ways such a connection can be established, including Kerberos as explained in the previous lab and Public key authentication with client certificates (TLS). In this lab we assume such a connection is already made and instead focus on how to identify the client. As such the password entered by the client is transported in plain text to the server. As explained earlier, cryptographically encrypting the password on the client side would not provide any security on a non-secure connection, as whatever is sent, be it plain text or otherwise, will be the key to authenticating the client making the encrypted key virtually become the actual key[1].

Once the server receives the username and password, it will query the database for the salt and hash-value associated with the specified username, add the salt to the received password, hash it and compare the value to the hash-value gotten from the database. If the server reaches a state where the two values are identical, the client identified by the provided username is authenticated.

In general there are two categories of such authentication: There is the single use where the key is bundled with every operation the client wants to do and the second option where the user is somehow remembered to be logged in.

The first option would be beneficial for quick operations, but in case of this lab, one might think the user would like to print multiple documents and maybe even changing paper sizes etc. in between and it would thus be cumbersome to have the client enter their password many times in a short while.

That leaves the second category which is lasting authentication. In this case after the initial authentication, the server will provide the client with some kind of knowledge he or she can include in every message going forth. The main reason this is superior to providing the password in every message is the possibility to time-restrict or even close down a connection server-wise. This greatly reduces the risk of a replay attack, as any sniffed message would only be useful within the lifetime of the knowledge. The two main types of knowledge are tokens and session ids. The main difference is that tokens are often files that can be stored and even shared between devices for often longer periods of times, providing means of password-less authentication, Where a session id is most often a freshly generated unique id that corresponds to a unique session on the server coupled with the username. Session ids often last much shorter than tokens and provides the ability to differentiate between sessions, thus being able to know if the user is logged in at multiple locations at once.

For the sake of simplicity, our implementation does not explicitly utilize neither tokens nor session ids, but does emulate the use of sessions. Just like we assume the connection is secure, we assume such a session id is generated and sent to the client at the initial authentication and that this session id is then concatenated with every message the client sends while navigating around the presented UI. We emulate this by saving a time stamp of the initial authentication and unauthorize the user automatically after a preset amount of time has passed.

[1]This is a known issue on windows: https://en.m.wikipedia.org/wiki/Pass_the_hash

2.3 Password Verification

In order to authenticate a client with the server by means of a password, the server must have some way of verifying that the entered password in indeed coupled with the specified user. Since there is no third party involved in authentication, it is not necessary for the server to store the password in plain text and thus only a hashed value and a used salt is saved. When the server received the plain text password and username, it will look up the corresponding hash and salt of that username in the database, hash the received password with the retrieved salt and compare the two hashes.

3 Design and Implementation

This project uses a client / server architecture based on the *RMI*[3] API. This project offers a printer service which allows already registered users to connect to it via the application server's ip address and port number. Users must authorize themselves by entering their username and associated password in the terminal. If a user enters his password incorrectly three times, he will be barred from trying for the next 10 seconds. Once a user is logged in, he can select (1) run an automatic test program or (2) manually use the printer service features. If the user is inactive for 10 seconds, he is automatically logged off. The features users can use includes:

1. **print(filename, printer):** sends file to a particular printer.
2. **queue(printer):** returns print queue of a particular printer to user.
3. **topQueue(printer, job):** sends job to top of queue.

4-6. **start()** / **stop()** / **restart():** starts/stops/restarts server.

7. **status(printer):** returns status of a particular printer to user.

8-9. **readConfig(param)**/**setConfig(param)**: read/set server config

10. **authenticateUser(uid, password):** authenticates the users

Figure 3.1: Operations

1

3.1 Architecture

This section will describe the methods, processes and cryptographic means used by our system. Figure[3.2] illustrates an overview of the java project. Titles in bold letters represents package-names. Colored boxes are classes and arrows are processes between methods in these classes, also between classes and the database, logfiles and printers. These processes are denoted with letters (a) to (m).

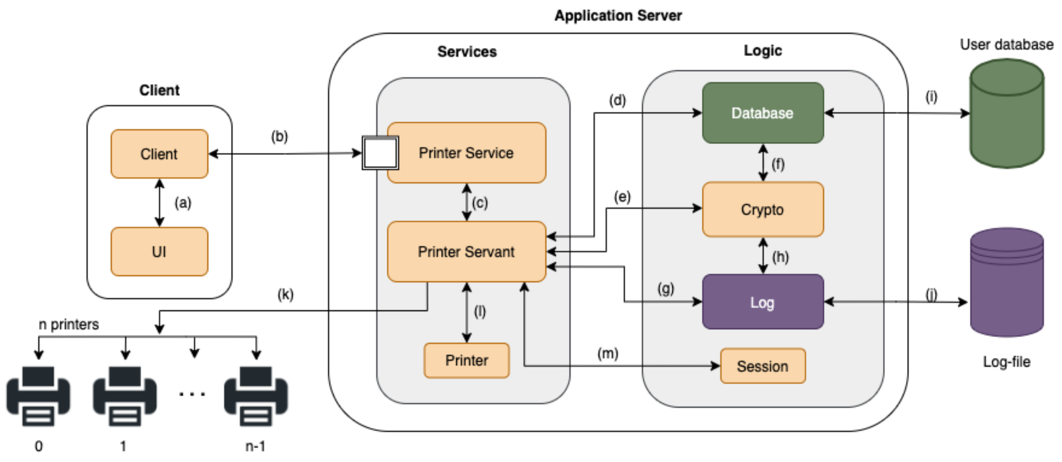


Figure 3.2: Auth Lab Diagram

Client: The client application starts by creating the connection (b) to the *Printer Service*-interface. We have not developed a cryptographic solution that exchanges keys and secure an encrypted communication channel here. Instead, in accordance with the assignment description, we assume connection (b) is secured in some other way. Before a user can perform any operations he needs to run operation 10[3.1] first. This authorization step is then executed in *Printer Servant*. The user will be noticed whether the login is successful or not.

Application Server: Whenever a server is started, a database will be created and populated with some hard-coded users (i), because no user creation is implemented. Furthermore three printers are created (k) from our *Printer*-class implemention and populated in a global ArrayList<Printer> *printers* in the *Server Servant*-class.

```java
public class Printer {
  ArrayList<String> queue = new ArrayList<String>();
  HashMap<String, Object> status = new HashMap<String, Object>();
  String printerName;
  boolean power;
  boolean isPrinting;
        ...

```

Note that our solution supports an arbitrary amount of printer creations. This class initialises an queue of type array-list, which is populated in the *Server Servant* upon server start:

```java
private void initialisePrinters() {
        // printer(boolean color, integer ink level)
        Printer office = new Printer(true, 90);
        office.setPrinterName("office");
        printers.add(office);
        ...

```

Every time a request to the server is made from the client, the server will check if the current session is valid. This is done by invoking *session.getSessionState()* in the method for any of the operations[3.1] the user has requested. Only a valid session can execute the operations. This session is created when a user has been authorized by the server. This is illustrated below. Observe that the method also invoke *writeLogEntry(...)*. Every time a user performs an request to the server, the responsible method logs this to the log-file (j). There exists a log for the server and each of the printers. See example of a log-file in appendix[A.3]. As with connection (b) there exists no encrypted channel on connection (j). This is again assumed secured in another way. The stored data on the log-file is in plaintext.

```java
public void print(String filename, String printer) throws ... {
    if(!session.getSessionState()) {
      // do nothing
    } else {
      for (Printer p : printers) {
        if(p.printerName.equals(printer)) {
          p.addToQueue(filename);
          writeLogEntry(filename, path + printer + ".log");
        ...

```

Method **authenticateUser(uid, password)** retrieves the users hashed password and salt from the database, from the user provided username (*uid*), on the form [*h(password+salt), salt*], by invoking *db.getCredentials(uid)*. Then hashes the user provided password with the database retrieved salt by invoking *crypto.hash(password, salt)*.

```java
public String authenticateUser(String uid, String password) ... {
    ...
    String[] credentials = db.getCredentials(uid);
    String h1 = credentials[0];
    String h2 = crypto.hash(password, credentials[1]);
    ...

```

ultimo comparing the hashes by invoking *crypto.compareHashes(h1, h2)*, and creating a user session if a match is found.

```java
    ...
    if(authAttempts < 3) {
        loggedIn = crypto.compareHashes(h1, h2);
        if(loggedIn) {
            session.beginSession(uid);
            loggedInUser = session.getUser();
    ...

```

The hashing method is responsible for producing a SHA-512 hash digest from provided password and salt. This is done with the help of the *Java Security*-library[2] which provides the *SecureRandom*-class that produces a strong random number, and *MessageDigest* that produces the one-way hash function. When this method is invoked with a password and salt it returns the hash.

```java
public String hash(String password, String salt) {
byte[] bSalt = new byte[16];
    MessageDigest md = null;
    if(salt == null) {
      SecureRandom random = new SecureRandom();
      random.nextBytes(bSalt);
    } else {
      bSalt = salt.getBytes(Charset.forName("UTF-8"));
    }
    try {
      md = MessageDigest.getInstance("SHA-512");
      md.update(bSalt);
    } catch (NoSuchAlgorithmException e) {
      e.printStackTrace();
    }
    byte[] hashedPassword = md.digest(password.getBytes(Charset.forName
  ("UTF-8")));
    String hash = new String(hashedPassword,Charset.forName("UTF-8"));
    return hash;
```

The *Database*-class is invoked from *Service Servant* (d) upon initialization of the server and when users are being authorized. The connection to the actual database (i) is, as with connection (b) and (j), assumed secured in some other way. The database initializes a table with users, password and salt. The hard-coded users are then inserted via an *SQL INSERT*-statement. As observed below, the username and salt is stored in plaintext, the password is being hashed before it is stored.

```java
public void initialiseDatabase() throws RemoteException {
    ...
    String sql = "create table users (user varchar(20), password
  varchar(200), salt varchar(200))";
    ...
    sql = "insert into users values ('jeff','" + crypto.hash("
  password22", "22-10-2021:21.18zz") + "','22-10-2021:21.18zz')";
    ...
```

When the database is queried with a given username it simply returns the hashed password and salt of that user. So the the plaintext password is never revealed between classes.

4 Evaluation

Whenever any of the following happens:

- Print

- Queue
- topQueue
- start
- stop
- restart
- readConfig
- setConfig
- User login (Authenticate a new session)
- Wrong user credentials triggered delay

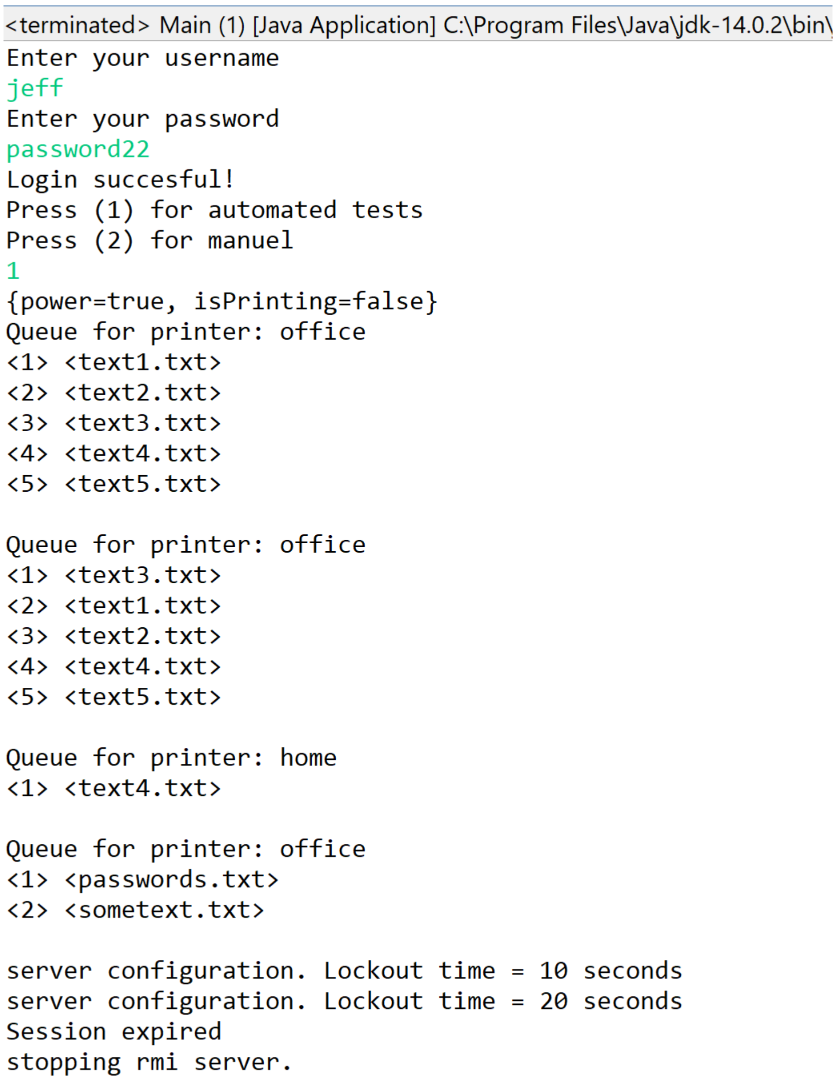
It is written to the log. Everything happens as described in previous sections including assumptions and exceptions. In order to help implementation and testing, once the client is authenticated, he will be presented and option to do automated tests to make sure everything is working as intended. The resulting terminal of a client authenticating and doing automated tests can be seen in illustration A.1 with the corresponding log file shown in illustration A.4 and printer queues in illustrations A.2 and A.3.

5 Conclusion

A simplified mock printer service has been implemented, showing how user authentication including password transfer, storage and verification is done. Future work includes removing assumptions about secure connections and explicitly implementing session authentication rather than emulating such. Furthermore it would be interesting to implement a pepper on top of the salt used in password verification and storage. The log file is currently public and non-secret and in future versions it might be something worth encrypting. Perhaps coupled with role-based access control.

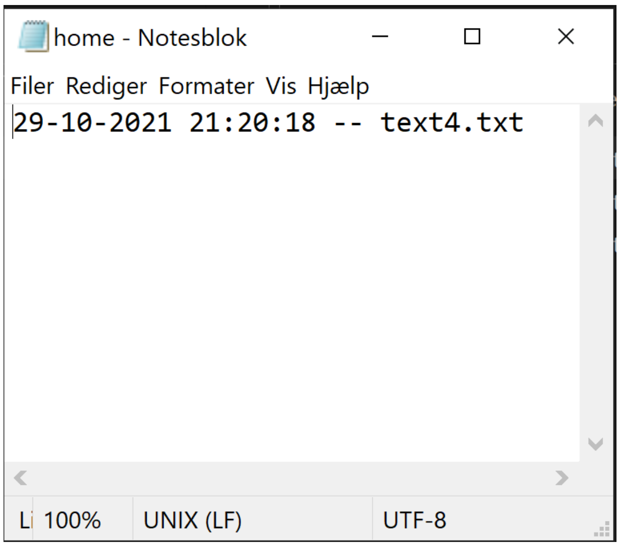
A Appendix

A.1 Illustrations



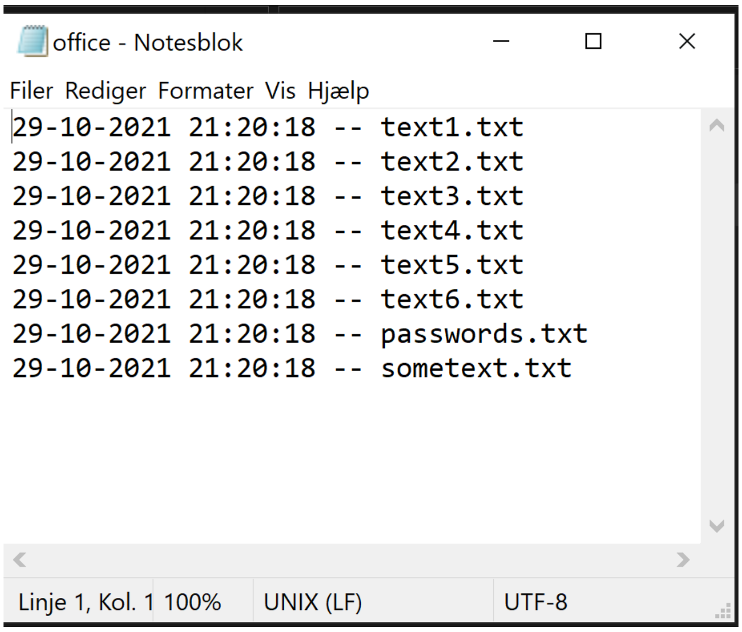
```
<terminated> Main (1) [Java Application] C:\Program Files\Java\jdk-14.0.2\bin\
Enter your username
jeff
Enter your password
password22
Login succesful!
Press (1) for automated tests
Press (2) for manuel
1
{power=true, isPrinting=false}
Queue for printer: office
<1> <text1.txt>
<2> <text2.txt>
<3> <text3.txt>
<4> <text4.txt>
<5> <text5.txt>

Queue for printer: office
<1> <text3.txt>
<2> <text1.txt>
<3> <text2.txt>
<4> <text4.txt>
<5> <text5.txt>

Queue for printer: home
<1> <text4.txt>

Queue for printer: office
<1> <passwords.txt>
<2> <sometext.txt>

server configuration. Lockout time = 10 seconds
server configuration. Lockout time = 20 seconds
Session expired
stopping rmi server.
```

Figure A.1: The resulting terminal when a client authenticates and chooses to test the functionality automatically.



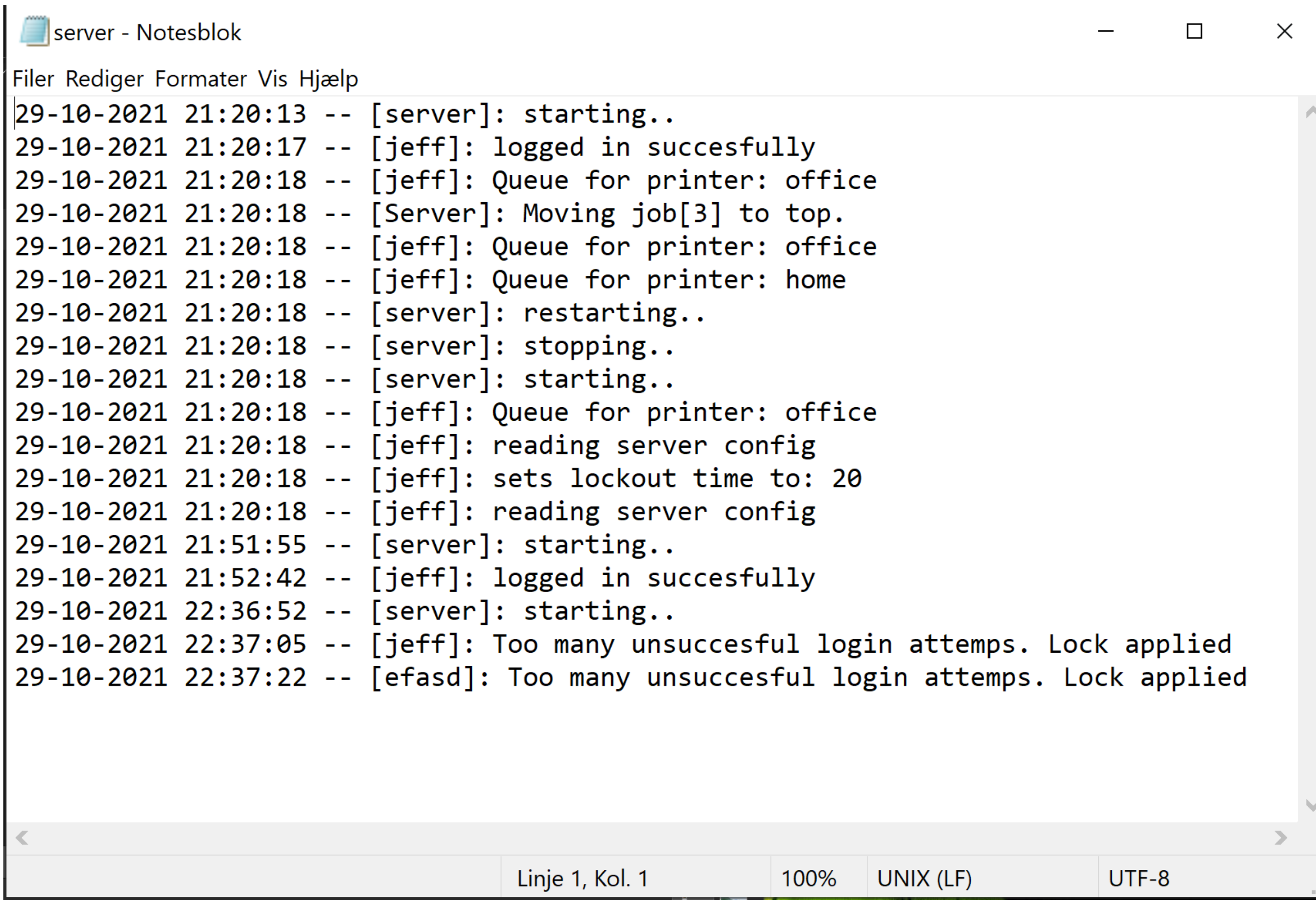
```
29-10-2021 21:20:18 -- text4.txt
```

Figure A.2: The resulting print-queue for printer "Home" when doing automated testing.


office - Notesblok
Filer Rediger Formater Vis Hjælp
Linje 1, Kol. 1 100% UNIX (LF) UTF-8

```
29-10-2021 21:20:18 -- text1.txt
29-10-2021 21:20:18 -- text2.txt
29-10-2021 21:20:18 -- text3.txt
29-10-2021 21:20:18 -- text4.txt
29-10-2021 21:20:18 -- text5.txt
29-10-2021 21:20:18 -- text6.txt
29-10-2021 21:20:18 -- passwords.txt
29-10-2021 21:20:18 -- sometext.txt
```

Figure A.3: The resulting print-queue for printer "Office" when doing automated testing.



```
29-10-2021 21:20:13 -- [server]: starting..
29-10-2021 21:20:17 -- [jeff]: logged in succesfully
29-10-2021 21:20:18 -- [jeff]: Queue for printer: office
29-10-2021 21:20:18 -- [Server]: Moving job[3] to top.
29-10-2021 21:20:18 -- [jeff]: Queue for printer: office
29-10-2021 21:20:18 -- [jeff]: Queue for printer: home
29-10-2021 21:20:18 -- [server]: restarting..
29-10-2021 21:20:18 -- [server]: stopping..
29-10-2021 21:20:18 -- [server]: starting..
29-10-2021 21:20:18 -- [jeff]: Queue for printer: office
29-10-2021 21:20:18 -- [jeff]: reading server config
29-10-2021 21:20:18 -- [jeff]: sets lockout time to: 20
29-10-2021 21:20:18 -- [jeff]: reading server config
29-10-2021 21:51:55 -- [server]: starting..
29-10-2021 21:52:42 -- [jeff]: logged in succesfully
29-10-2021 22:36:52 -- [server]: starting..
29-10-2021 22:37:05 -- [jeff]: Too many unsuccesful login attemps. Lock applied
29-10-2021 22:37:22 -- [efasd]: Too many unsuccesful login attemps. Lock applied
```

Figure A.4: The resulting log when a client authenticates and runs automated tests on the printer service.

A.2 Setup & Running the project

- **The software used in this project:**
  - **IDE:** Eclipse[1]. Version: 2020-12 (4.18.0).
  - **Database:** SQLite[4].

- **Run the project:**
  - Install Eclipse to your system.
  - Unzip the project file
  - In Eclipse: File > Import > General > Existing Projects into Workspace > Locate the unzipped project > Finnish
  - Run src/main/java/server/test/Main.java
  - *if localport is used pick a new port number* - When program runs

- **When application is running:**
  - Username: jeff

- Password: password22 - Press (1) for automated tests or (2) for manual tests.
- The session controller is locking user out after only 10 seconds for testing purposes. Otherwise the test would take too long to illustrate the lockout mechanism in work. Therefore, when testing manually, it is adviced to change this setting in: src/main/java/logic/Session.java: set *Time* to at least 60 seconds.
- The manually works, but not much work has been put into this, thus its not super user friendly. The automated test tests all the functionalities and can be found in: src/main/java/client/Client.java

References

[1] Eclipse. "IDE". In: (). URL: `https://www.eclipse.org/downloads/packages/installer`.

[2] Oracle. "Package java.security". In: (). URL: `https://docs.oracle.com/javase/7/docs/api/java/security/package-summary.html`.

[3] Oracle. "Remote Method Invocation". In: (). URL: `https://docs.oracle.com/javase/7/docs/technotes/guides/rmi/hello/hello-world.html`.

[4] SQLite. "database". In: (). URL: `https://github.com/xerial/sqlite-jdbc/releases`.