Authentication Lab

Jens Leysen (s191908)
Maud Leclerc (s191975)

October 2019

1 Introduction

As stated in the lectures, Authentication is the process of asserting (confirming) an identity. It's not the same as Identification which is the act of asserting who a person is. For this lab we will be designing and implementing a simple authentication mechanism for a client/server application.

We will not consider the enrollment of users, as stated in the lab description. We will consider the problems of password storage, password transport and password verification. Since we're only a group of two students, we were not able to implement a secure channel by cryptographic means (although we would have liked to).

Some of the problems that we will consider in our solution are:

1. General password storage problems such as identical hashed passwords for different users.
2. SQL Injection attacks.

The general outline of our solution is as follows:

1. Access control by using a DBMS.
2. Prepared statements to combat SQL Injections.
3. Hashing passwords and using salts.
4. Implementing an authenticated session mechanism.
5. Storing the client state server-side and using a timeout mechanism.

2 Authentication

In this section we will describe problems related to authentication and modern authentication solutions. Not every method mentioned here will be included in our implementation.

2.1 General problems related to password authentication

We will regularly cite [1] in the following section. We will discuss the most relevant problems related to passwords in the section below. The normal operation mode is as follows: the OS authenticates the user by asking a name and password which it in turn has to validate, most likely by comparing a value stored in some sort of table.

1. Storing the password in plain text: When the table is leaked, the attacker has direct access to all passwords and corresponding usernames. The identity can be public but the password shouldn't be readable by just anyone. This problem can be solved by hashing the password and saving this hashed version in the table. When the user wants to authenticate, the same hashing function is applied to the entered password and this digest is compared to the hashed password that was saved in the table.

2. Brute force active authentication: A system should lock out a user who fails a small number of successive login attempts. [1] This prevents an attacker from brute forcing his way through an active authentication session. Note: this gives the attacker a way of locking out legitimate users. (Availability issue)

3. Rainbow table: A rainbow table is a precomputed list of popular values, such as passwords. [1] An attacker can make a rainbow table (list of concealed forms of the common passwords) and compare it to an intercepted password table, and gain access to these commonly used passwords.

4. Identical hashed passwords for different users: When using a simple hash function, two users having the same password will have the same hashed password. It is then easy for an attacker to find this password with a rainbow table. A way of preventing this is by adding a salt to the password before hashing it. A salt is an extra data field different for each user [1] (e.g. the date the account was created or a part of the username).

5. Brute force: A brute force attack on an intercepted table will have the attacker try all possible password combinations (from the full character set). In theory (given an unlimited amount of time), this method would always find every password in the table. The only way to make it hard(er) for the attacker to find a password is by choosing a good password.

2.2 Discussion on password storage methods

Here we shortly discuss the implementation and issues of the three possible solutions: passwords stored in a public file, system file and database.

1. Public file: We will discuss how system 'password' files are implemented on the UNIX filesystem and analyse the security of the UNIX implementation. As mentioned in the following book on Unix security [2]: "Traditionally, Unix uses the /etc/passwd file to keep track of every user on the system. The /etc/passwd file contains the username, real name, identification information, and basic account information for each user."

   The traditional UNIX implementation of the password file was public to read but not necessarily to write for everyone.

   Unix systems use the one-way function crypt(), to encrypt your password. The result of the calculation was traditionally stored in the /etc/passwd file. When you try to log in, the OS will encrypt your password using the same function and compare the two values. (As described in section 2.1)

   Additionally, Unix makes use of a SALT. The reason for using a SALT was explained in section 2.1. When you change your password, the /bin/passwd program selects a salt based on the time of day. The salt makes it possible for a user to have the same password on a number of different computers and to keep this password secret, even from somebody who has access to the /etc/passwd files on all of those computers. [2]

2. System file: Modern Unix systems have adopted an approach called shadow password files to deal with the problem of password files leaking. A shadow password file is a secondary password file that shadows the primary password file. The shadow password is usually stored in the file /etc/shadow and contains the encrypted password and a password expiration date. This file is protected so that it can be read only by the superuser. Thus, an attacker cannot obtain a copy to use in verifying guesses of passwords. [2]

3. DBMS: Storing a password file on a database will often make use of the same techniques as described already in section 2.1 and the "System file" subsection above. That is they will make use of cryptographic one-way hash functions and SALTs. A DBMS can however provide confidentiality and integrity of the stored passwords by use of it's security architecture, eg. the access control mechanism.

   The database should not be located at the same place as the server, because if an attacker manages to get into the administrator account of the server, they will be able to access the database. This means that the data transmitted between the two must be secured, by using a secure protocol (for instance SSL). To protect the database against unwanted access, one should implement some access control (for users), and a firewall, that will protect it from unwanted access, and from initiating outbound connections. To prevent SQL injections attacks (that can be used to obtain or delete data), the best way is to use input validation and parametrized queries including prepared statements. The developer must sanitize all inputs. It is also a good idea to turn off the visibility of database errors on production sites. Database errors can be used with SQL Injection to gain information about the database. Finally, there should be some backup system (encrypted as well), to ensure the data can be recovered, should the database be compromised.

2.3 Discussion on password transport methods

Once a client has logged in, a developer can choose to work with individual request or authenticated sessions. The difference lies in where the user state is stored.

Individual Request Authentication With individual request authentication, the user-state is stored on the client side. The client will authenticate itself on each request to the server, by for example using a token. A token is a text string, containing a digitally signed information, named payload. This payload can for example contain the client identity and an expiration time. The signature enables the server to trust the sender, and the client doesn't have to authenticate itself on a password every time. If the token is not recognized by the server, it will not answer the request.

Authenticated Sessions With authenticated sessions, the user state is stored on the server side. The server assigns a unique session id to the client. This session id is then sent along with every other request. A popular example is the use of cookies in HTTP. The server will authenticate a client (on a password for instance), and create a session number that is stored in a database. A cookie with the session ID is then stored on the client's side. On every request, the IDs from each side are compared in order to process the request.

Advantages and disadvantages of both methods When using sessions, the ID needs to be stored both on the client and on the server sides, as opposed to tokens, where the token is only stored on the client's side. It can then be more efficient to use tokens when there are lots of users for the service, to save time (there is no need to look up the sessions ID in a database, which can be quite time-consuming). Another advantage of the token is that, where a session is just reduced to an ID, it can store a lot more information, for example roles and permissions for the user (although this information is just signed, not encrypted, and thus should not be sensitive). On the other side, it takes more space to store a token than a session ID, especially if it contains a lot of data.

2.4 Discussion on choosing good passwords

One of the main liabilities with passwords is that people tend to choose existing words that are easy to remember, making these passwords weak and easy to guess. One other problem is that a user can write down, or even share their password, making any encryption useless. Here, we will discuss some general rules for good passwords. We will not implement any password strength checking mechanism ourselves.

1. Use a variety of characters: A password should make use of lowercase and uppercase letters, and digits.
2. Choose a long password: The longer the password, the more characters, and thus the more possible combinations.

3. Do not choose existing words: Using a random, or meaningless, combination of characters. Or use a passphrase.
4. Use different passwords for different authentications: If a user uses the same password on multiple websites or services and that password is compromised on one of the services, the attacker can then log in other services.
5. Use a password that one can remember: A good strategy is to use a passphrase that is easy to remember, modify it to include some lowercase and uppercase characters, add some digits, and then add a small variation for each account.
6. Change the password often: This protects a user account even if an attacker manages to obtain an old list of passwords. A system can force users to change their password from time to time.

3 Design and Implementation

3.1 Authentication Mechanism

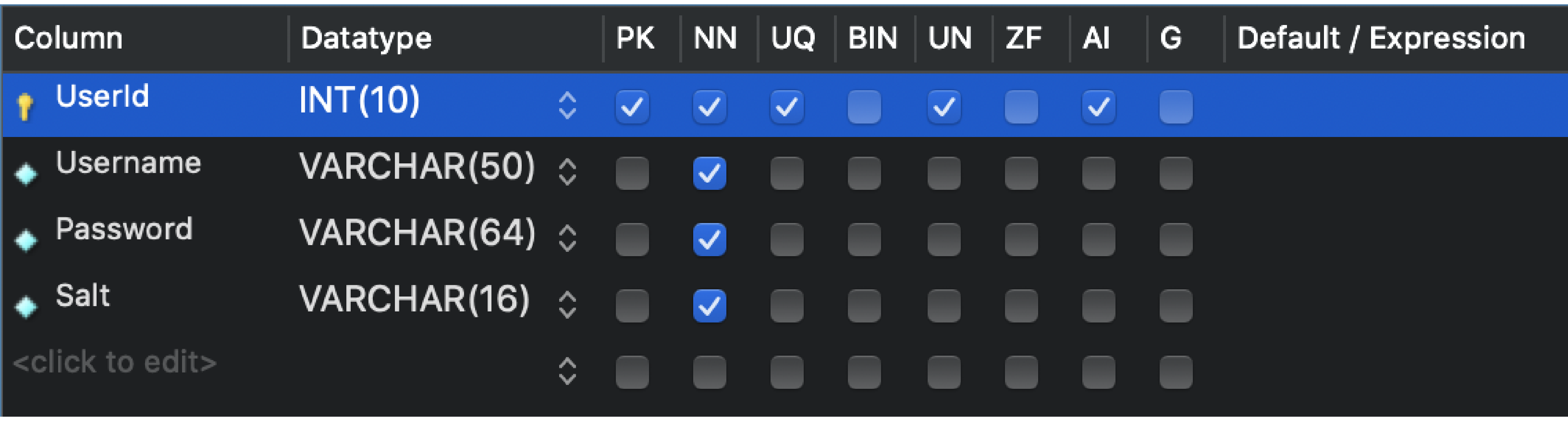
We didn't implement any channel security between client and server since we're a group of only two people. We assume full channel integrity, confidentiality and availability. Additionally, per section 2.4 we assume the user has respected the best practices for the creation of a strong password. We propose the following design:

1. DBMS: Using a database for password storage, allows for use of access control mechanisms.
2. Prepared Statements: Using prepared statements to prevent SQL Injection attacks.
3. Active Client Objects: Saving the client state on the server side by using a custom client object. Clients can be timed out after a specified time, they will then need to re-authenticate.
4. Authenticated sessions: Letting the user login once and assign it a SID for invoking server functions.
5. Logging: Logging of all invoked functions and users that invoked these functions in a log file to ensure accountability.
6. SHA-2: Using a SHA-2 hashing algorithm, the current hashing standard for SSL.
7. Salt: Generating a random 16 byte salt for generating the salted and hashed password.

3.2 Implementation of the designed authentication mechanism

DBMS We used a local MySQL database as our DBMS. We created a database 'PWD' with one table 'Users'. This USERS table will hold the usernames and encrypted passwords of the clients. Note: If you want to test our code, you will have to run a local MySQL instance and create a database 'PWD'. The used port is 3306.

The columns were initialised as shown in the following picture:



Column	Datatype	PK	NN	UQ	BIN	UN	ZF	AI	G	Default / Expression
UserId	INT(10)	✓	✓	✓		✓		✓		
Username	VARCHAR(50)		✓							
Password	VARCHAR(64)		✓							
Salt	VARCHAR(16)		✓							
<click to edit>										

UserDatabase This class is just used for testing and populating the database. All the queries executed in this class are done by a user with enough rights to create/delete tables and add/remove users on the local database. (Eg. Root or other Admin) The admin username and password are set at the top of the class. We will discuss the most important functions:

1. AddPrinterAccount: Will create an account for the printer on the local database. It will set a username (Printer) and password (password) for the printer. We make a separate account for the printer in order to apply access control mechanism to this account.
2. AddGrantsPrinterAccount: Will give the Printer account global 'SELECT' rights and nothing else. This ensures that the printer can only request information and cannot alter or delete any data in the table.
3. AddUser: Will add a User (Client) to the Users table. As noted before, we will not worry about enrollment of users.

SHA256Hasher This class has a public function 'HashSHA256' that accepts a salt and password. It also has a function 'getSalt' which will form a random 12 byte array. Additionaly, it has a 'byteToString' function that will use Base64 encoding (e.g. encode a 16 character string from the 12 bytes). It will use this 16 character string as a SALT with the provided password to create a SHA256 digest.

Client This class implements the behavior of a normal user. It will first prompt for username and password via the console. The password is not masked by asterisks, normally we would use the 'System.console' class for masking the user input. When using an IDE, IntelliJ runs java code without association with current console, and since there is no associated console window 'System.console()' returns null. [4] This class uses remote method invocation to access the printer functionalities.

ClientObject The active clients are stored on the server side in an ArrayList of type ClientObject. The ClientObject class holds the SID and Timeout values and calculates the active duration for active clients. The Timeout variable holds the time the client is allowed to be active. The 'checkSession' function in the printServant class will check the duration between the time the client was created and the current time using the 'timeElapsed' function. The 'checkSession' function will be discussed further in the next paragraph.

PrintServant The PrintServant class implements the PrintService interface and holds some additional functionality. It holds an ArrayList of ClientObject named 'activeClients' and an ArrayList of String objects named queue (the printer queue), which simply holds the name of the files that should be printed. Additionally, it holds the printer on/off state in a boolean variable named printServerOn. We won't discuss all of the full implementation of the printService interface, some of the more interesting functions are discussed below.

1. initiateSession: Accepts a username and password, it will use the getUserID function to get the UserId from the database using username and password. If a valid ID is returned (ID>0), it will create a UUID object as an SID for the user. Next it will create a ClientObject object and add it to the 'activeclients' ArrayList. The function returns the SID (a UUID object).

2. getUserId: Will connect to the local database with the printer account credentials. It will first query for the salt using the username of the client. Next it will create a SHA256Hasher object and create a digest using the inserted password and retrieved salt. Afterwards, it will query for the UserID using username and the salted+hashed password. If the UserId is not found, the function will return -1.

3. checkSession: Will iterate through the 'activeClients' ArrayList. It will check all clients to see if their time has expired and remove if so. It will also check the active client SID's and see if they are equal to the provided SID. If the provided SID matches an active client SID, it will log the name of the invoked function and username of the client to a file named "printServer.log".

4. print: The print function is a good example of how the PrintService functions are implemented. It will first check whether the print server is on. If the printer is off, it will return "Print server off". Note: It's not clear what the start/stop function should do. Next it will check the validity of the SID using checkSession. If not valid it will return "Session expired or invalid SID" to the client. If the SID was valid, it will return "File: " + filename + " added to queue" and add the file name to the queue ArrayList.

4 Evaluation

The goal of this project is to authenticate users of a print service using a password. The print service is set in a way that when a service is called, the answer of the printer is returned as strings that can then be printed by the user. Additionally, every method invocation done by a user with a valid SID is logged in a file 'printServer.log'. This enables us to check that a request is successful, or on the contrary, that the user session has expired and the request couldn't be answered.

The chosen solution to store user passwords is a MySQL database. To avoid the biggest problems related to passwords (see section 2), the passwords are salted and then hashed. This protects them against rainbow table attacks, and having the same stored hash for different users with the same password.

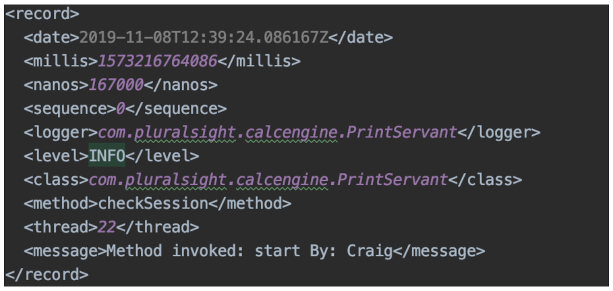
The hashed version of the password is then stored in a database, that can only be modified by a user having Admin privileges. It is assumed that the communication between the print service and this database is secure. To protect the database against SQL injection attacks, the print service only uses prepared statements.

When accessing the service, the user authenticates himself with a password, that is salted and hashed and then compared to what is stored in the database. If the two hashed digests match, the service then issues a session ID, that is stored on the printer side and transmitted to the client. The client will then use this session ID to authenticate himself on each request. This means the user is always authenticated. For more security, the sessions ID expires after a certain amount of time, after which the user needs to enter his password again.

The full process answers to the following requirements: password authentication, safe password storage, limited password transport, authentication on every request, answers from the print service.

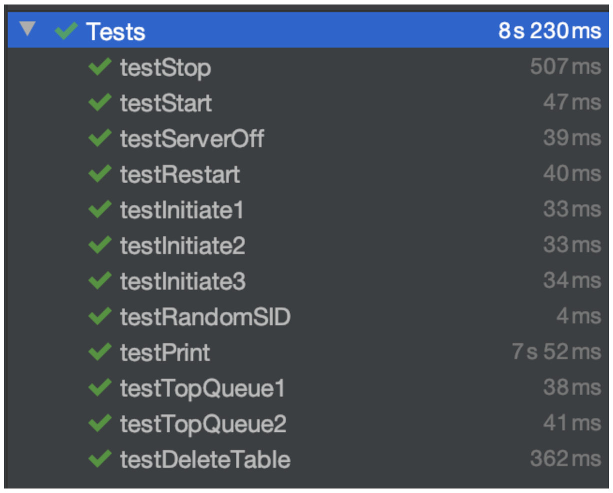
Unit testing A set of 12 tests were implemented using Junit in order to evaluate the basic printing functionalities, the session and password authentication mechanisms and the access control of the DBMS.

Logging Logging functionality was added using 'java.util.logging.Logger'. Using 'StackTraceElement', the calling method name is found and printed to the 'printServer.log' file together with the username of the client requesting the service. The log file gives additional information such as the date and time the function was executed and the thread handling the method execution. The client class implements the behavior of a 'normal' client, trying different functionalities of the server. It has been validated that the client actions are logged. Note: all actions are appended to the log file.



```
<record>
  <date>2019-11-08T12:39:24.086167Z</date>
  <millis>1573216764086</millis>
  <nanos>167000</nanos>
  <sequence>0</sequence>
  <logger>com.pluralsight.calcengine.PrintServant</logger>
  <level>INFO</level>
  <class>com.pluralsight.calcengine.PrintServant</class>
  <method>checkSession</method>
  <thread>22</thread>
  <message>Method invoked: start By: Craig</message>
</record>
```

(a) Example of a record in the log file



```
Tests              8s 230ms
  testStop           507ms
  testStart           47ms
  testServerOff       39ms
  testRestart         40ms
  testInitiate1       33ms
  testInitiate2       33ms
  testInitiate3       34ms
  testRandomSID        4ms
  testPrint        7s 52ms
  testTopQueue1       38ms
  testTopQueue2       41ms
  testDeleteTable    362ms
```

(b) Unit tests

5 Conclusion

The user authentication works as expected, assuming that the communication channel between the print service and the database is secure. If the user inputs the correct login and password, a session ID is created, enabling him to make requests to the print server. But if the password doesn't match the user login, or if someone tries to log in with an empty login and password fields, no session ID will be created. The user will not be considered logged in and thus any request he sends to the printer will be answered by a "Session expired or invalid SID" message.

The database is protected against SQL injection attacks by use of prepared statements. Moreover, only the administrator of the database is able to access and modify it, meaning that, for example, a printer can not modify the database.

Future work Some of the main password problems described in section 2 have not been addressed in this project. The first one is that the user will not be locked out after too many failed login attempts. It would be a good idea to implement this security steps in a future version of this project. The second one is dependant on the user itself, as it consists in choosing a strong password. We are not considering the enrollment of users here, so the quality of the password is just assumed to be good. But we could imagine a program checking that the password is of a certain minimal length, and contains a variety of lowercase and uppercase characters, plus digits and special characters for example. It could also ask the user to change his password every few months. This would be a way of making sure the password is not too weak.

From a more technical point of view, we would have liked to implement more unit tests to test some more interesting scenarios. For example, clients with the same username will lead to the salt of the first user being returned and the second user never being able to login. This is part of user enrollment however.

In the current implementation, the print server has only one queue (for one printer). A normal print server would have more than one printer connected to it. We would implement a printer class that is stored by the server, a print job is then submitted to the queue of the distinct printer object.

References

[1] Pfleeger&Pfleeger. (5th edition) Security in computing.
`Chapter 2.1: Authentication`

[2] Gene Spafford,Alan Schwartz&Simson Garfinkel. (3rd Edition) Practical UNIX and Internet Security
`Chapter 4: Users, Passwords and Authentication`

[3] The Linux Information Project. The /etc/passwd File.
`http://www.linfo.org/etc_passwd.html`

[4] Stackoverflow
`https://stackoverflow.com/questions/26470972/trying-to-read-from-the-console-in-jav`