Project 1: SQL Injection

**Task 1**

The objective of this task was to implement SQL commands in order to see the employee Alice’s information. The line of code used was: SELECT \* From credential WHERE name = ‘Alice’; Figure 1 below presents the results.

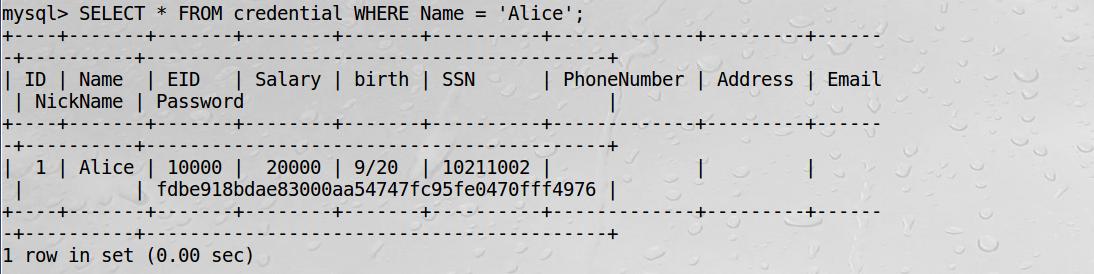


Figure 1. Employee Alice information table.

**Task 2**

In task 2, SQL injection attacks were performed using the login page as well as through the command prompt window by sending HTTPs requests.

**Task 2.1**

In order to login as the administrator without the password, ‘admin’#’ was used as the username and the password field was left completely blank. The # symbol allows an attacker to comment out a piece of code. Since the password portion is now commented out using SQL code, the database ignores the password input and grants the attacker access. Thus, appending this to the end of the username provides the attacker access to an account without knowing the user’s true password. Figure 2 presents what was typed into the login page and Figure 3 shows the results of logging in with those credentials.

**Task 2.2**

During this part of the task the curl statement was introduced. Instead of obtaining all employee information as the administrator through the login page, the URL was used in the command prompt window. The first attempt shown in Figure 4 used curl with the www.SeedLabSQLInjection.com and the correct parameters; however, the index.php file did not exist. Then, index.php was changed to index.html and this provided a clue that it was located in the file unsafe\_home.php from the clause <form action="unsafe\_home.php" method="get"> as shown in Figure 5. Using this information, the following command was typed: curl 'www.seedlabsqlinjection.com/unsafe\_home.php?username=admin%27%20OR%201=1%20%23&Password=';. In addition, %27 was used to replace apostrophes, %20 for white space, and %23 for the pound symbol. This only provided the information for employee Alice, but not the others as highlighted in Figure 6. Finally, it was observed that the OR 1=1 part of the encoding clause could be removed leaving just admin%27%20%23&Password=. The command: curl 'www.seedlabsqlinjection.com/unsafe\_home.php?username=admin%27%20%23&Password='; produced the results in Figure 7 which shows all employee information.

**Task 2.3**

An attempt to try and modify employee information by using two SQL statements, but it failed. Figure 8 presents the following command typed into the login page: admin'; SELECT TABLE credential DROP COLUMN birth; #. The logic behind it was to trick the SQL code into deleting the birth column. However, this attempt was unsuccessful seen in Figure 9.

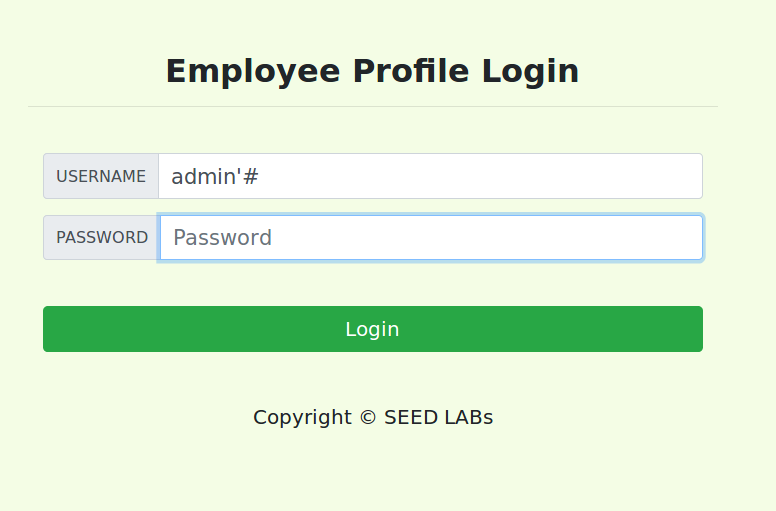


Figure 2. Login as administrator.

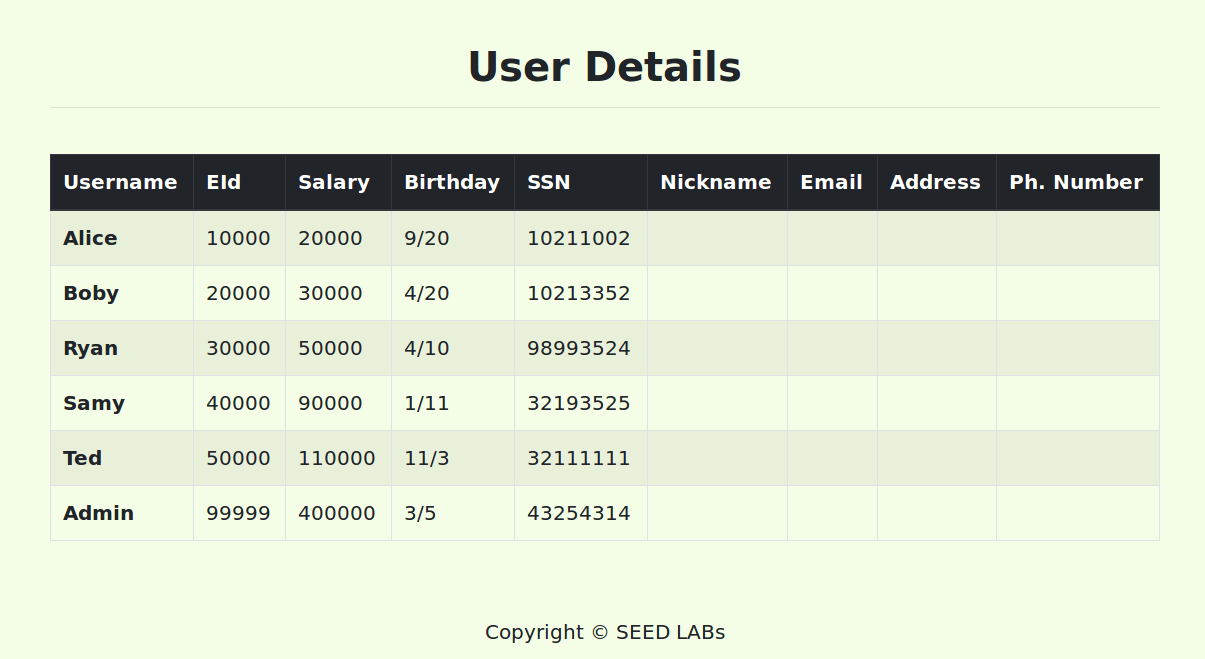


Figure 3. Result of Task 2.1.

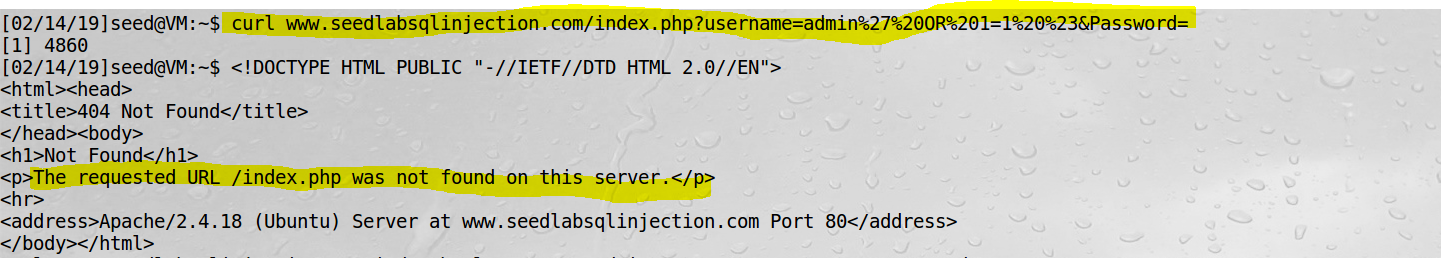


Figure 4. Attempt one using curl statement.

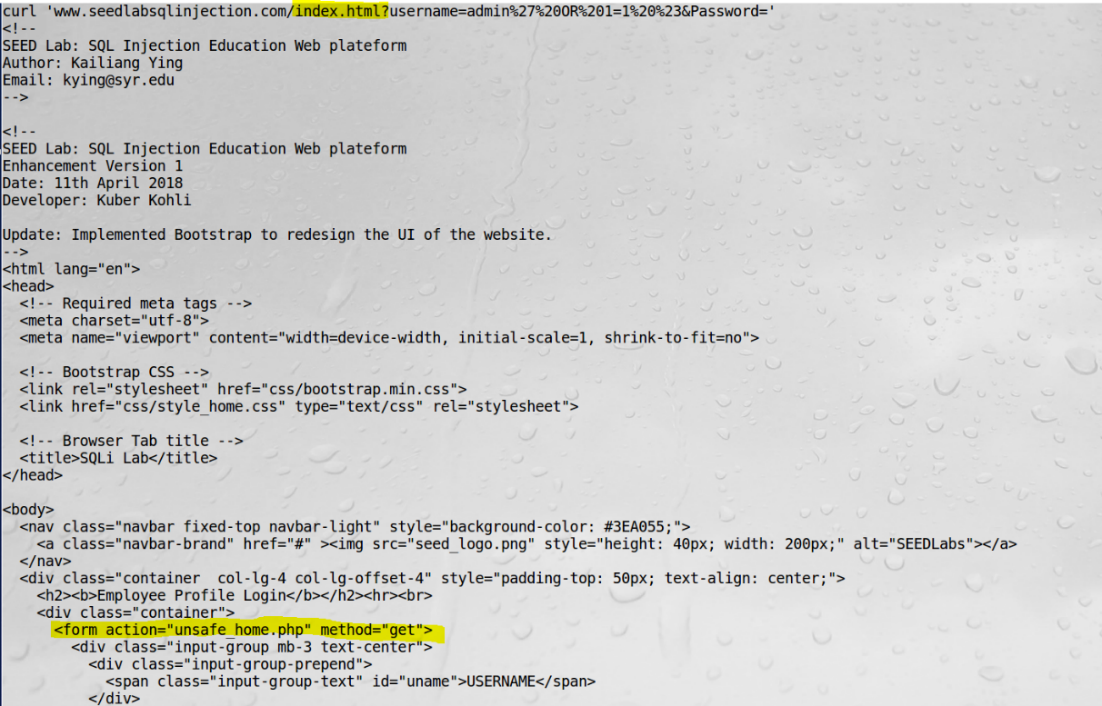


Figure 5. Unsafe\_home.php hint.

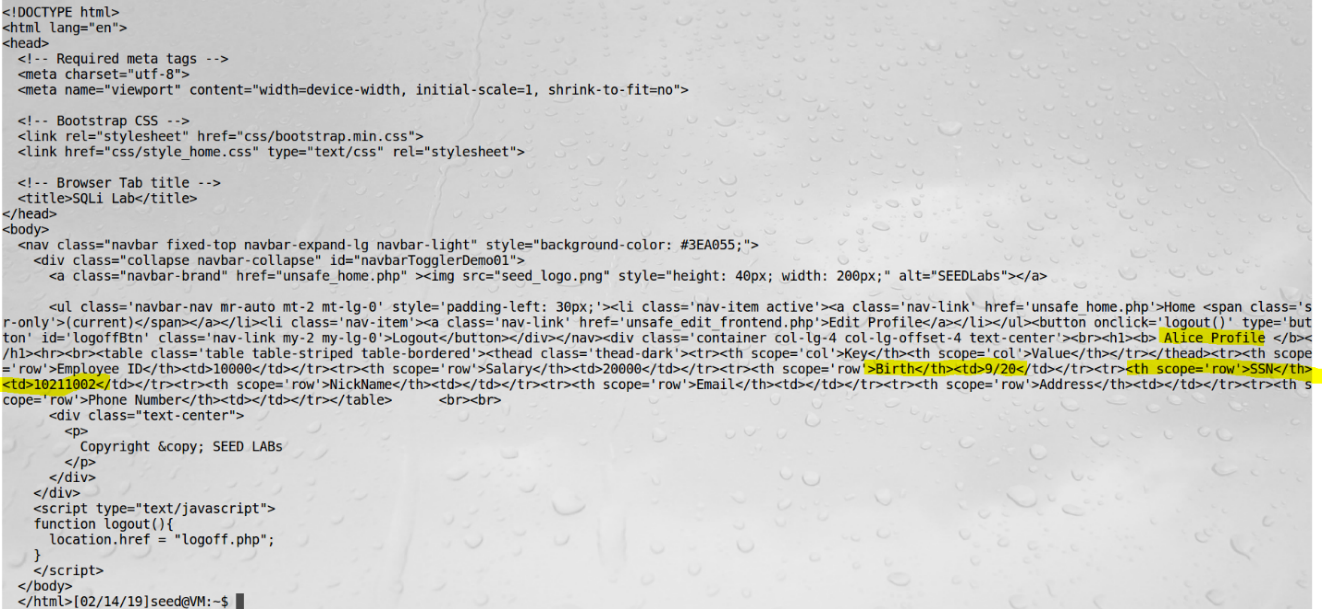


Figure 6. Employee Alice information accessed.

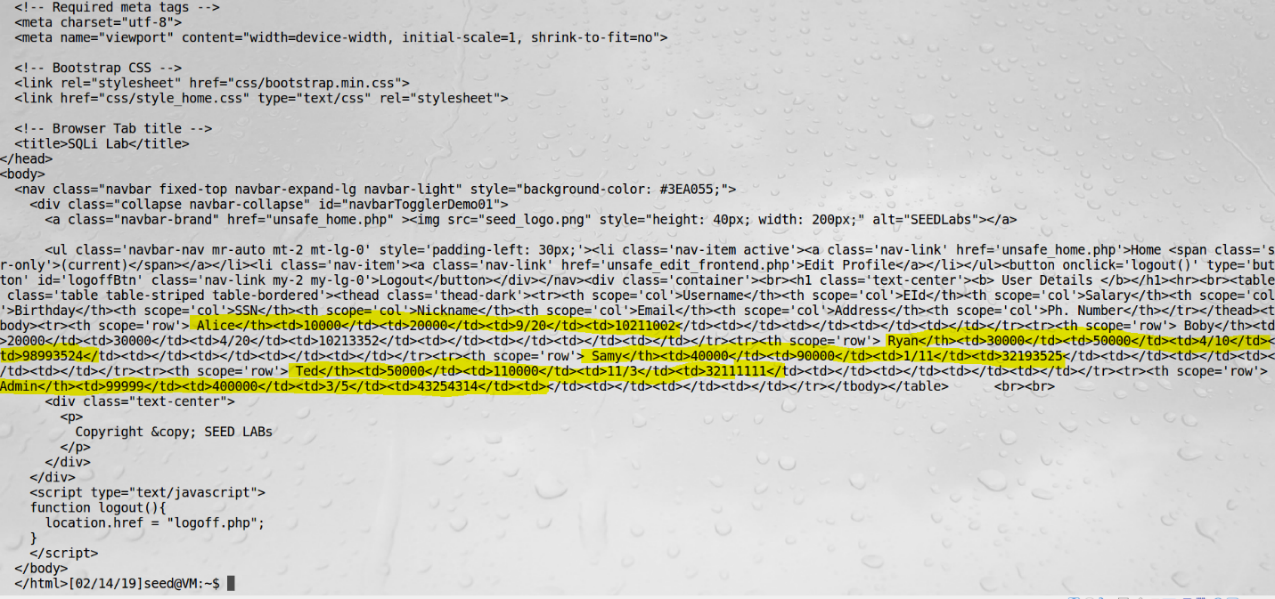


Figure 7. All employee information can be viewed.

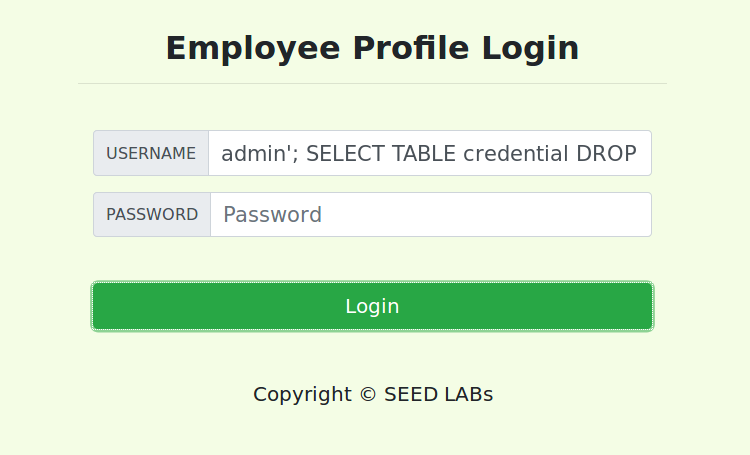


Figure 8. Attempt to modify using login page.

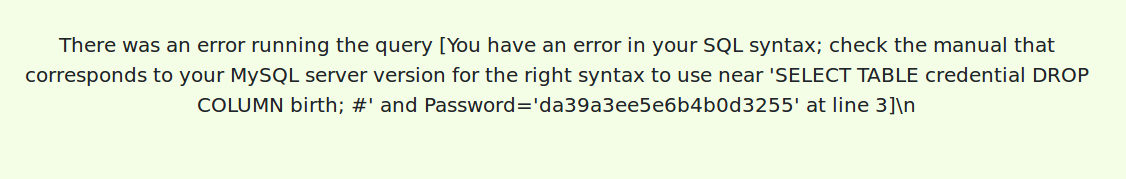


Figure 9. Results of Task 2.3.

**Task 3**

Task 3 focused on performing an SQL injection attack by gaining unauthorized access to modify information in the database.

**Task 3.1**

This task aimed to modify an employee’s salary as an employee who does not have such clearance. To do so, the Edit Profile page was used by logging in as Alice. This was done in the same manner as logging in as the administrator without a password. Thus, in the login page under username alice’# was typed. When editing the profile, the following was typed into the password field: whocareswhat’, Salary=120000 # which is shown in Figure 10. This did not work so the same statement was typed in the phone number text box. The results of this action are shown in Figure 11.

**Task 3.2**

Now, the salary of Boby was modified to a dollar. The steps taken to accomplish this was to login as Boby using boby’# in the username field of the login page demonstrated in Figure 12. After navigating to edit profile tab, the statement: whocareswhat’, Salary=1 # was typed into the phone number field shown in Figure 13 and the results of this are given in Figure 14.

**Task 3.3**

In order to modify Boby’s password, access to his account was gained by using boby’# in the username field of the login page as done previously. On the edit profile page displayed in Figure 15, it was very easy to update his new password to any string of characters. The password was changed to whocareswhat and could be used to gain access to the account from the login page from there on. Figure 16 demonstrates logging in with the new password and Figure 17 shows access granted to the account.

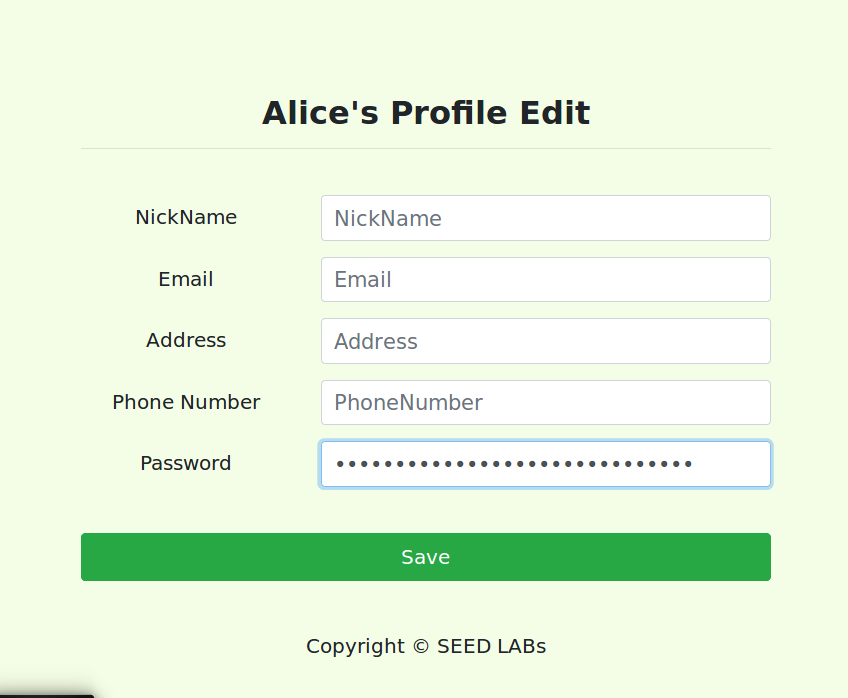


Figure 10. First attempt to modify Alice’s salary.

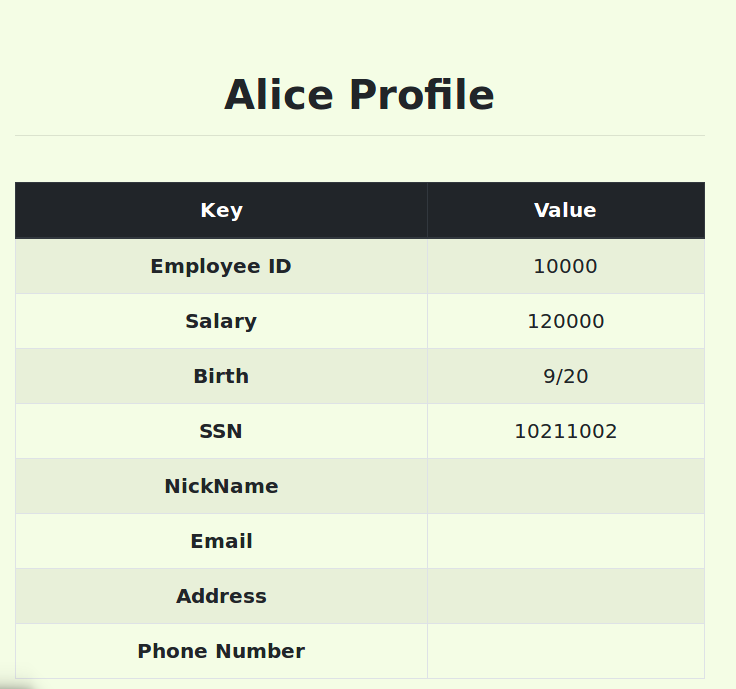


Figure 11. Results of first attempt.

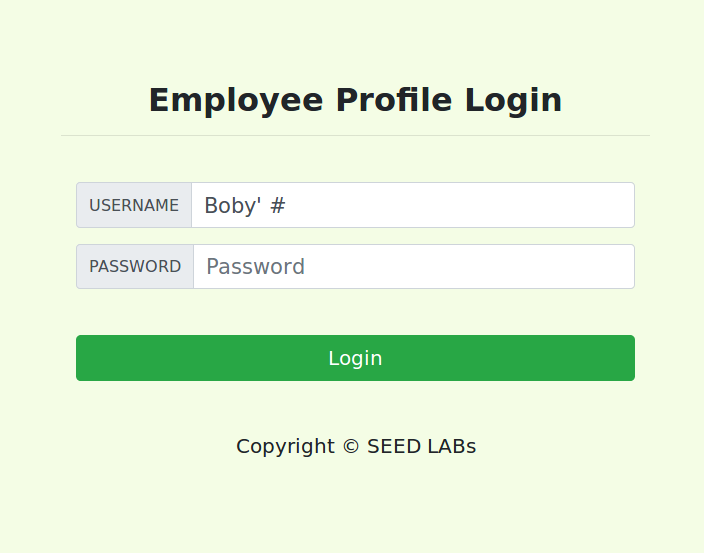


Figure 12. Logging in as Boby.

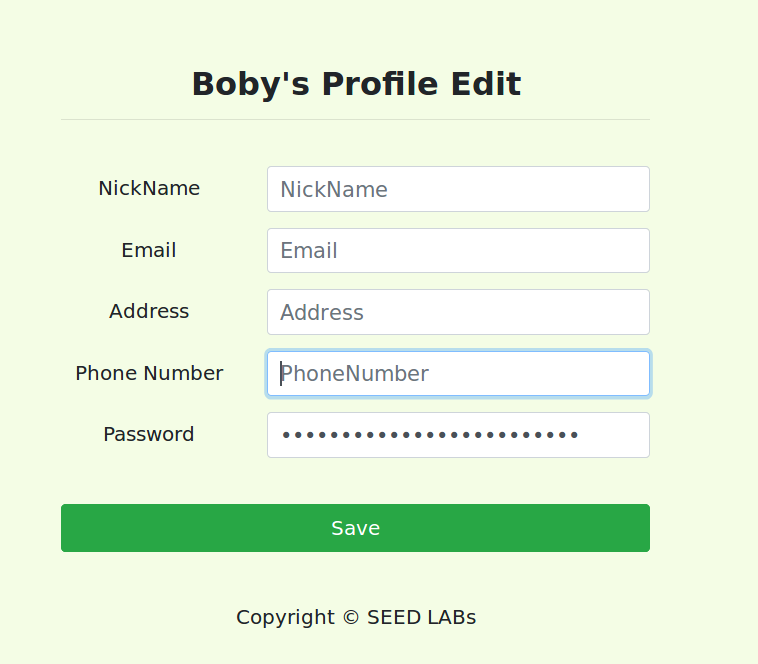


Figure 13. Update Boby’s salary.

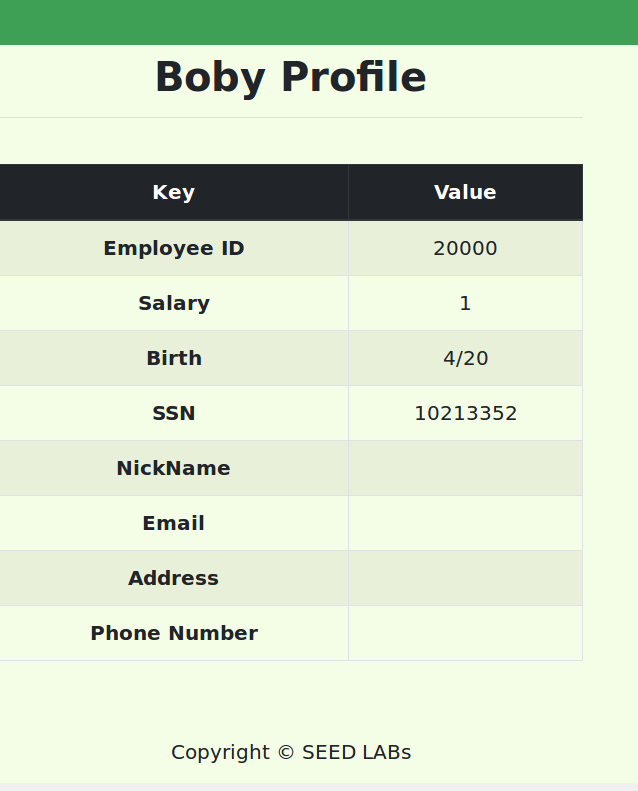


Figure 14. Results of Task 3.2.

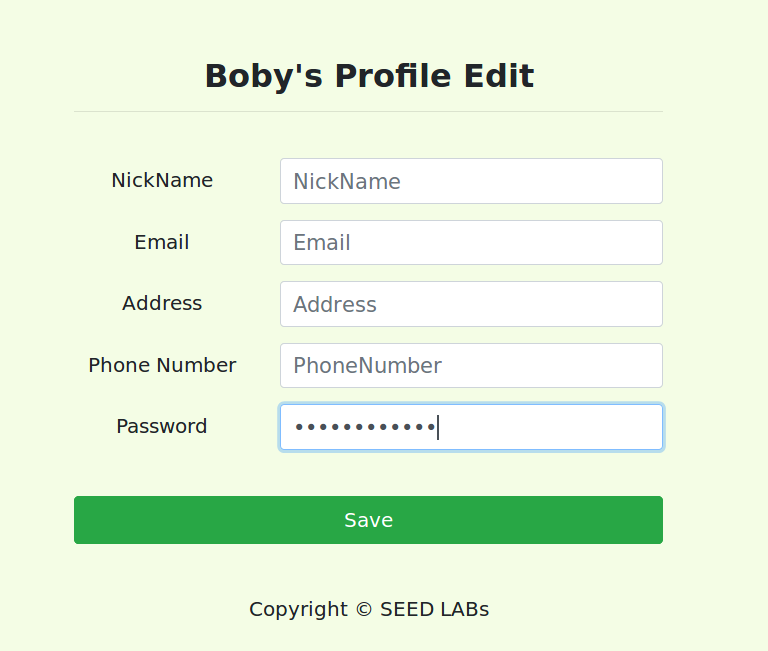


Figure 15. Change Boby’s password.

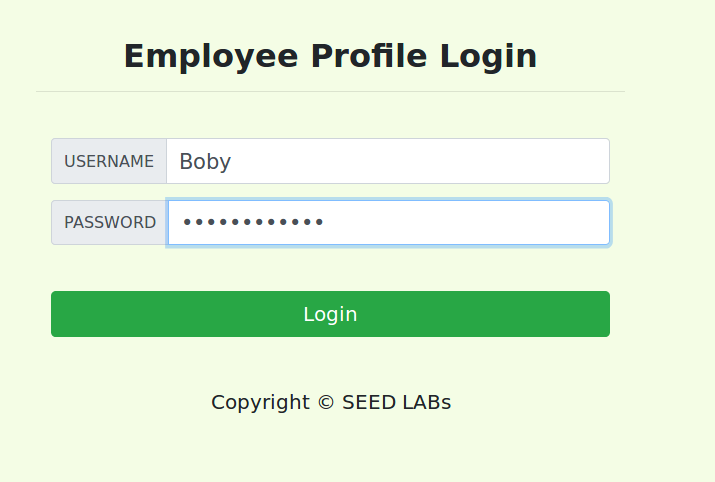


Figure 16. Sign in as Boby using password “whocareswhat”.

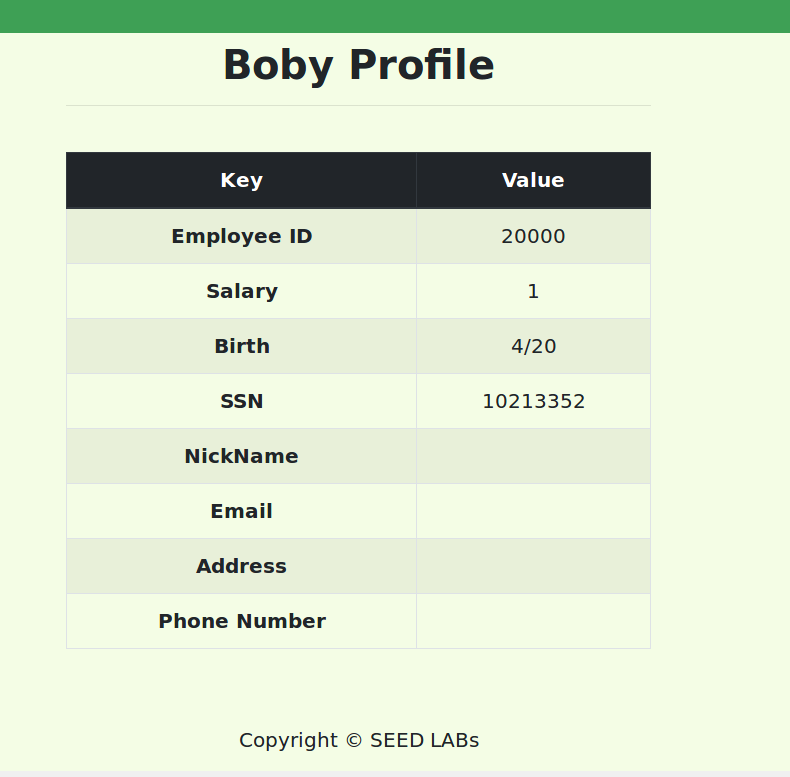


Figure 17. Result of Task 3.3.

**Task 4**

The final portion of the project explained why SQL injection attacks work and how to prevent the use of SQL code to gain unauthorized access to information.

First, the differences in the code between the files unsafe\_home.php and safe\_home.php were observed to understand where and how the prepared statement should be implemented. The bind-param function appears in safe\_home but not unsafe\_home shown in Figure 18 and Figure 19. It was seen that the code in safe\_home.php contained the prepared statement that eliminates a hacker's ability to use SQL injection attacks. The code in unsafe\_home.php does not contain the prepared statement, thus allowing the simple exploitation used in the previous tasks to gain unauthorized access. Figure 20 displays the terminal showing the sudo command paired with the gedit command was used to alter the which .php file was loaded by the .html file. Under the directory: var/www/SQLInjection, index.html was opened with gedit to update the code. The name of the .php file was edited to safe\_home.php and this took the file safe\_home.php instead of unsafe\_home.php. This action is demonstrated in Figure 21. Now, the prepared statement is in effect and blocks the exploitation used to gain access from previous tasks. The prepared statement was tested by trying to login using the same technique as seen in Figure 2. As expected, the previous attacks no longer worked and the error produced is given in Figure 22.

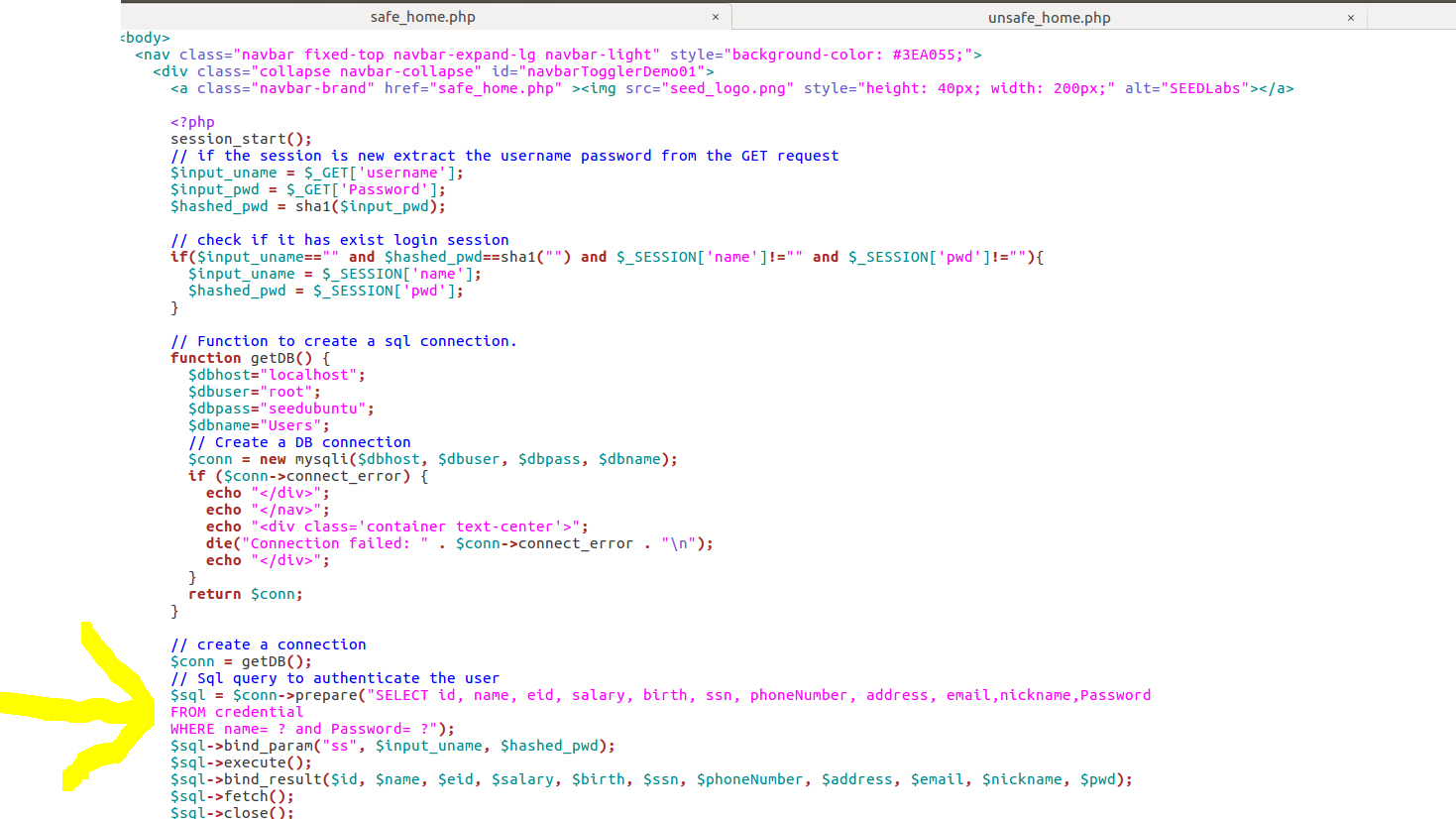
****

Figure 18. Code in safe\_home.php.

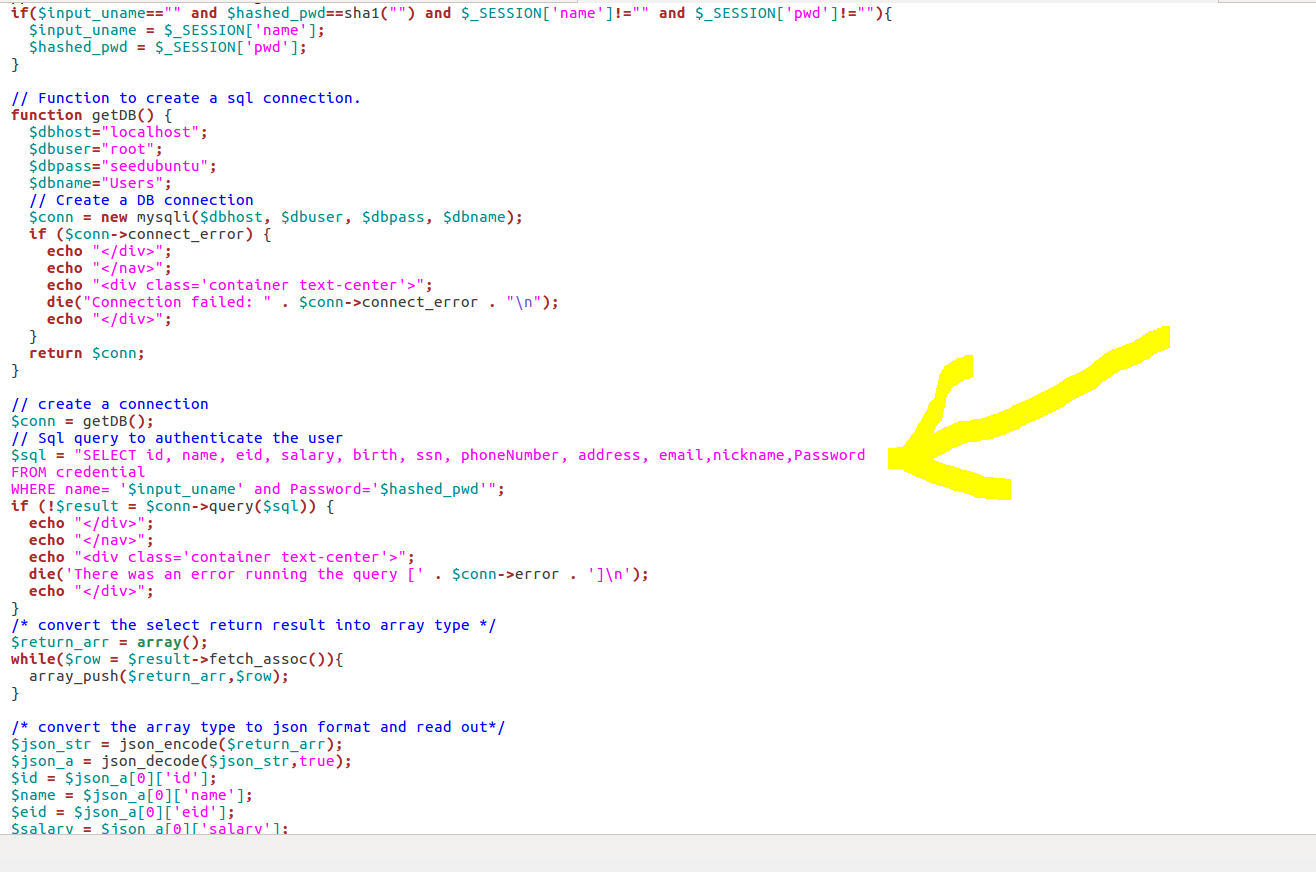
****

Figure 19. Code in unsafe\_home.php.

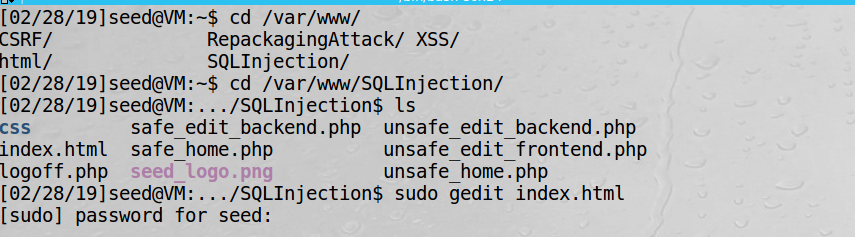
****

Figure 20. Implementing the sudo and gedit command.

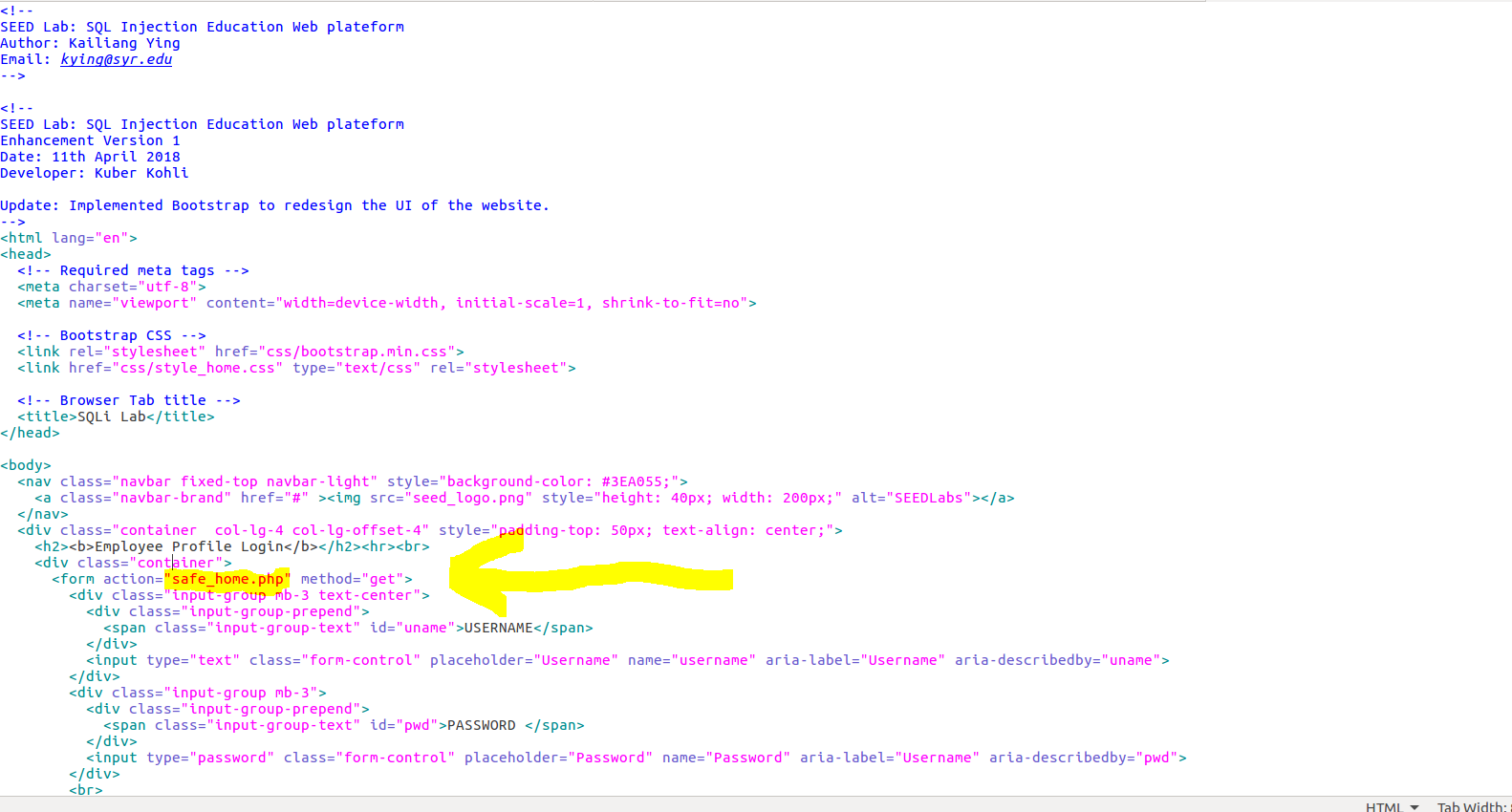
****

Figure 21. Updating the .php file.

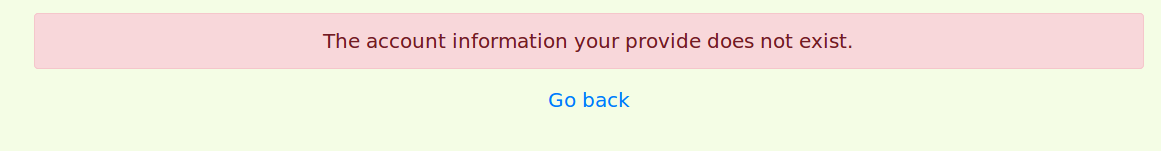
****

Figure 22. Attempt to implement SQL attack.