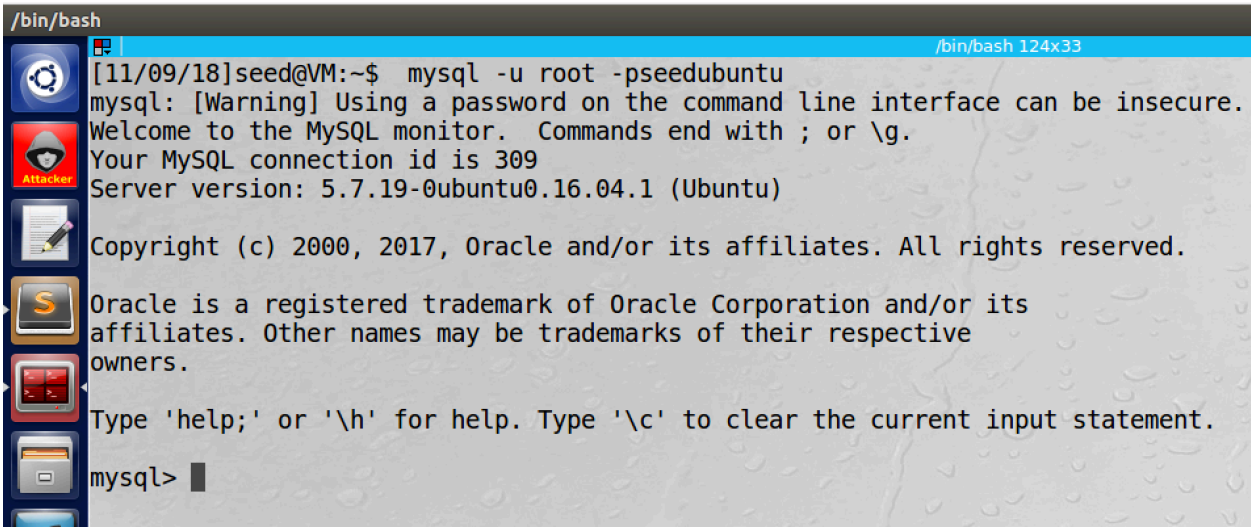
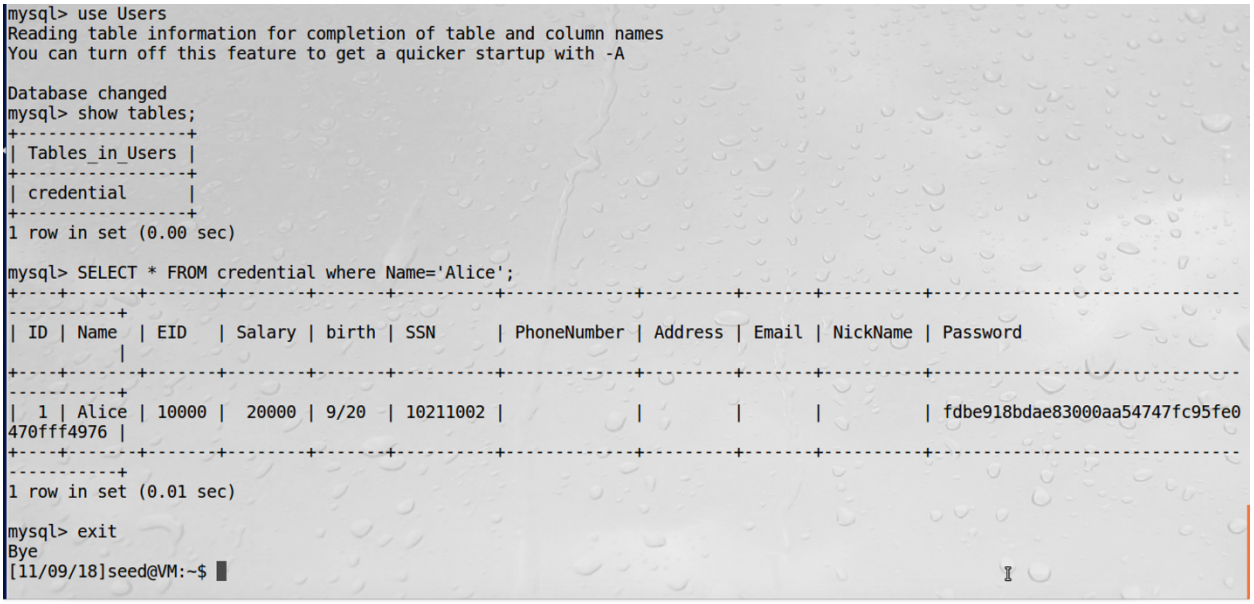
CSE643 Lab11
Yishi Lu
11/12/2018

Task1: Get Familiar with SQL Statements



screenshot1, we login to MySQL console



screenshot2, we run SQL query to fetch all information of Alice

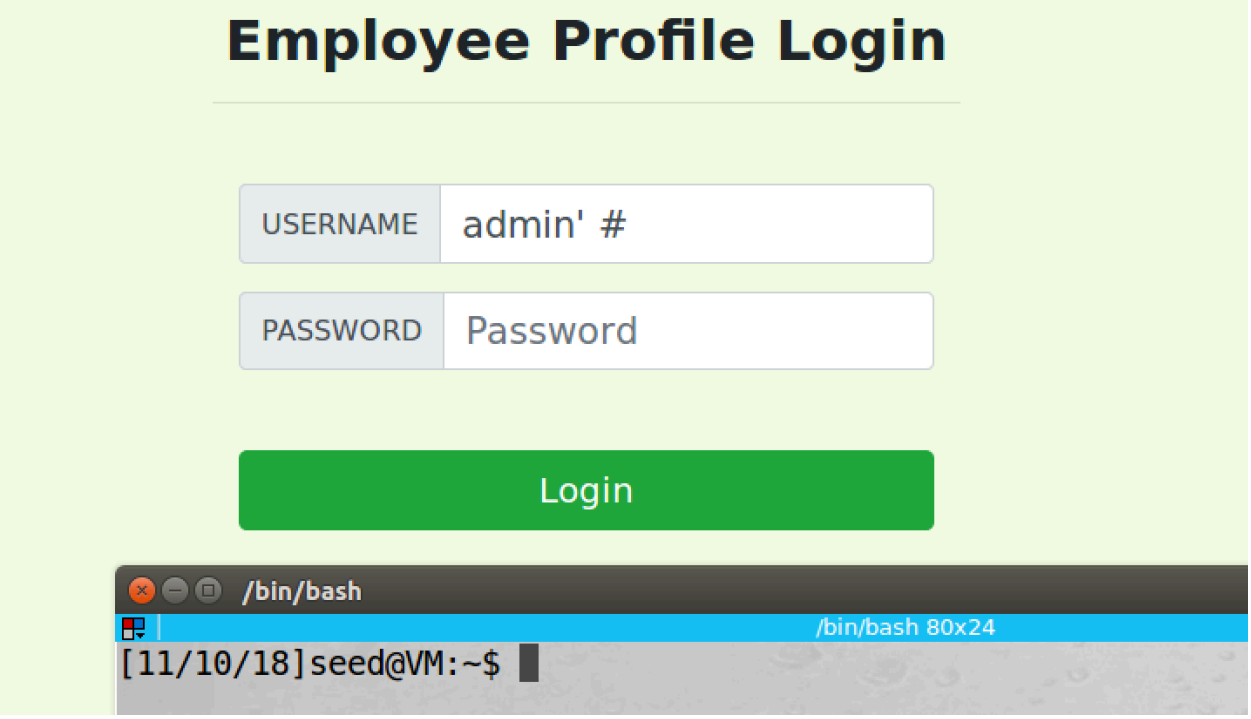
Observation and Explanation:

In this task, we are going to be familiar with MySQL database. In the VM, MySQL data base is already installed, and we login to its console (screenhsot1). There are one built database which called Users, and it contains one table: credential. We run SQL query “SELECT * FROM

credential where Name='Alice';", such command will print all information of "Alice" in table credential(screenshot2).

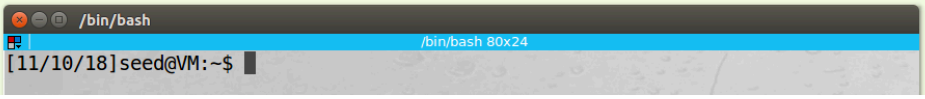
Task2: SQL Injection Attack on SELECT Statement

Task2.1: SQL Injection Attack from webpage



sereenshot1, we type "admin' #" in the username field, and we leave blank in the password field

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				

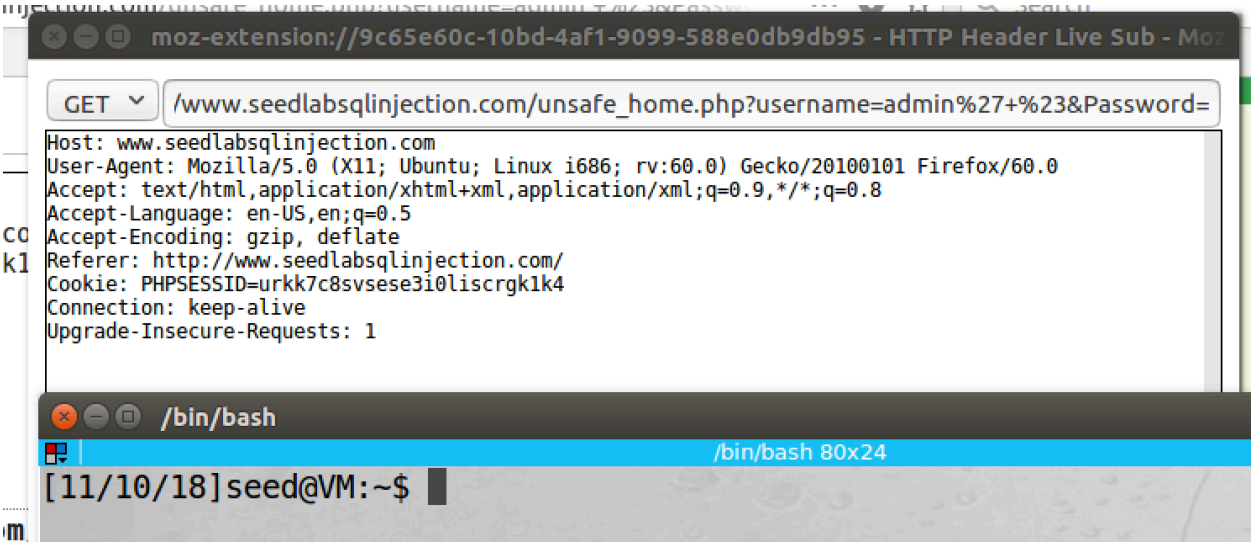


screenshot2, we successfully login to the database, and all employees' information are printed

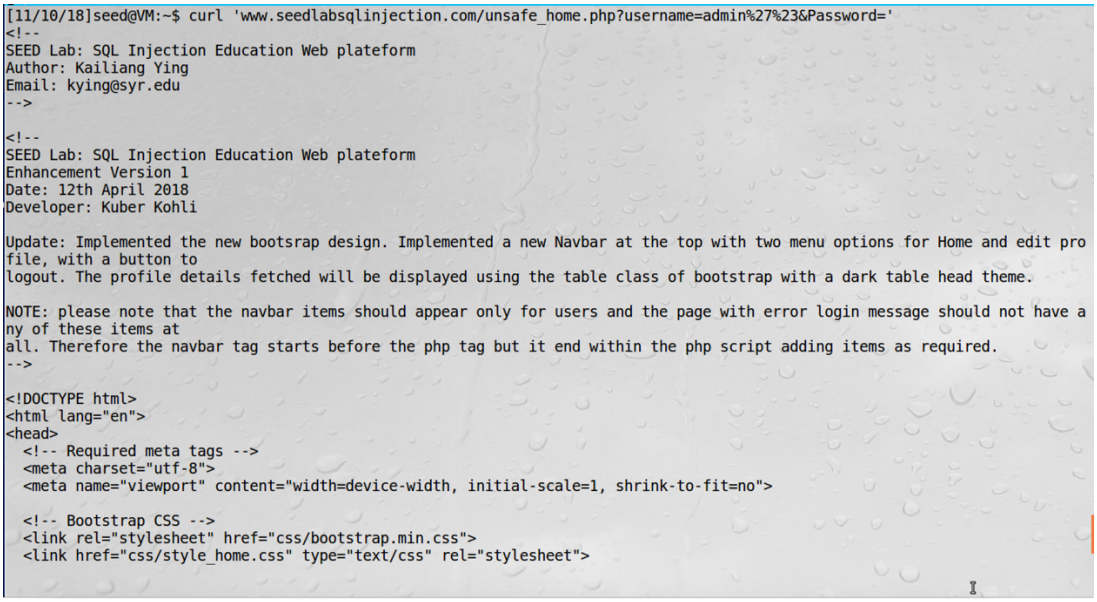
Observation and Explanation:

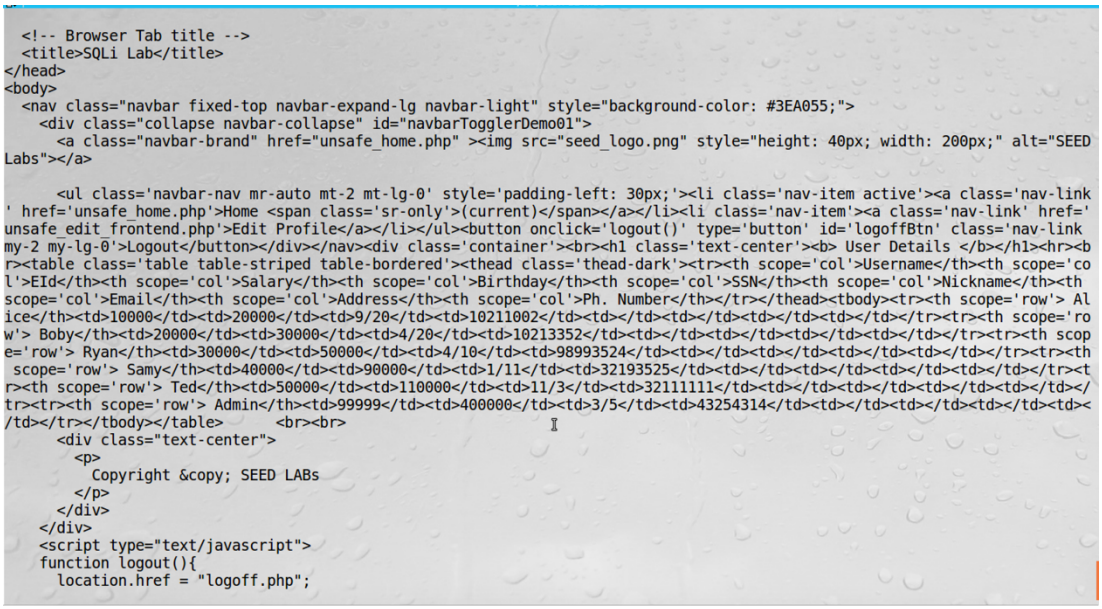
In this task, we perform SQL injection attack on SELECT statement. There are some special characters, such as (‘) and (#). We can use them to change the meaning of the SQL statement on the target website. In our case, we know the user name is “admin”. We use admin’ to end the username input field; then we append # at last, this is a comment, everything behind # will be treated as comment (screenshot1). So after we click login, the SQL statement becomes “WHERE name= ’**admin’ #**’ and Password=’$hashed_pwd’"; so the password field will be comment out. As a result, we can login to the system by only using username. As screenshot2 shows, we successfully login to the system.

Task2.2: SQL Injection Attack from command line



screenshot1, we capture the URL by inspection tool





screenshot2, we use curl to login to the system. And the result is sent back.

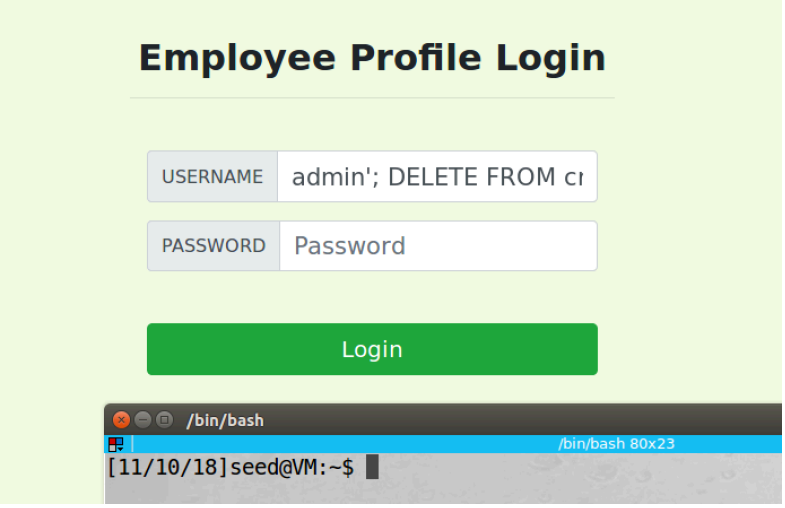
Observation and Explanation:

In this task, we perform same attack again, but this time we use command-line. Curl is a command-line tool; it can send HTTP request to target server. Then the server will send result back. We first need to know the URL, so we use inspection tool to get the URL (screenhsot1), then we construct the URL and send it to the website by curl.

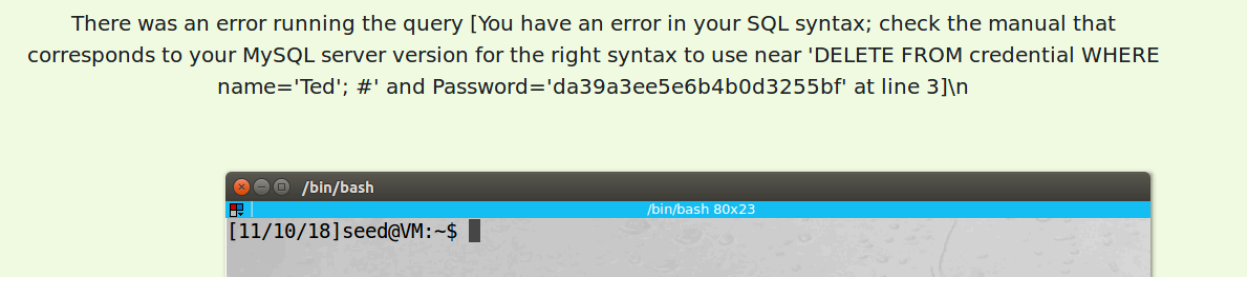
“curl 'www.seedlabsqlinjection.com/unsafe_home.php?username=admin%27%23&Password='”

It contains three parts, first the server URL. Second the username field, which is same as last attack. Because we send it via HTTP request, special characters must be encoded. In our case, we encode (‘) to %27 and (#) to %23. The last part is the password field, we just leave it blank because it will be comment out by (#). As screenshot2 shows, after we sent out the request. The result is sent back, and the information of the whole credential table is included in the result. So our attack is successful.

Task2.3: Append a new SQL statement



screenshot1, we construct the input in username field, such command can login as admin, and it also delete all information of Ted in the database



screenshot2, our attack is not successful

Observation and Explanation:

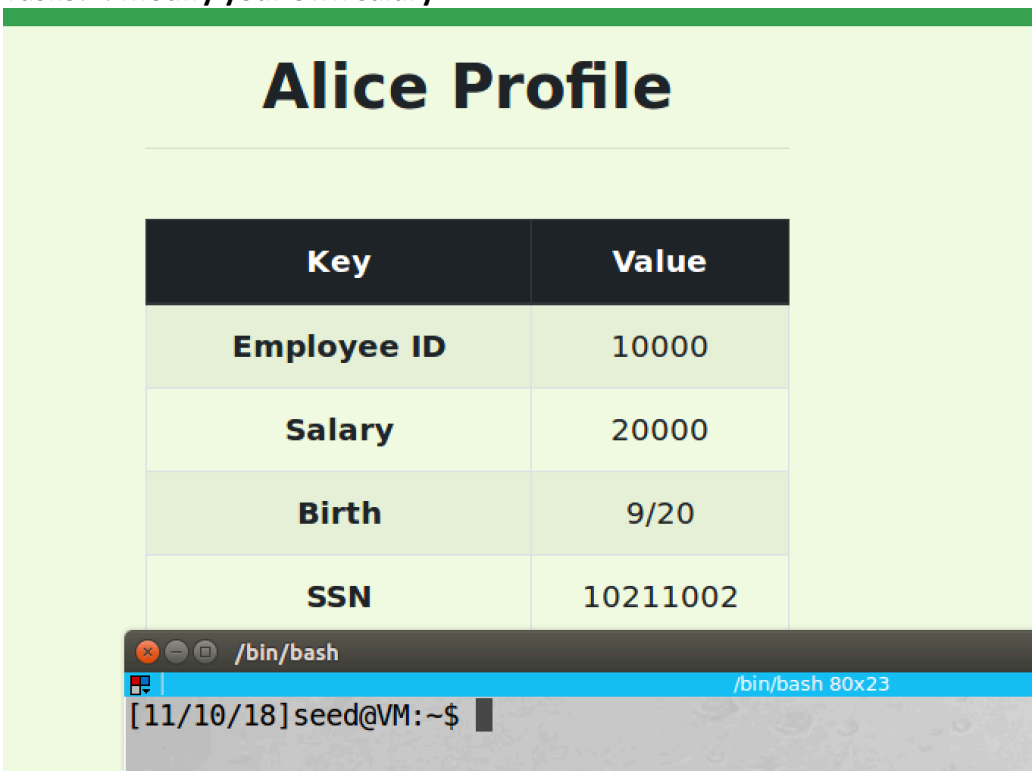
In this task, we want to run multiple SQL statements at once. Instead of login to admin account, we also want to delete account of Ted at same time. The following is our input:

admin'; DELETE FROM credential WHERE name='Ted'; #

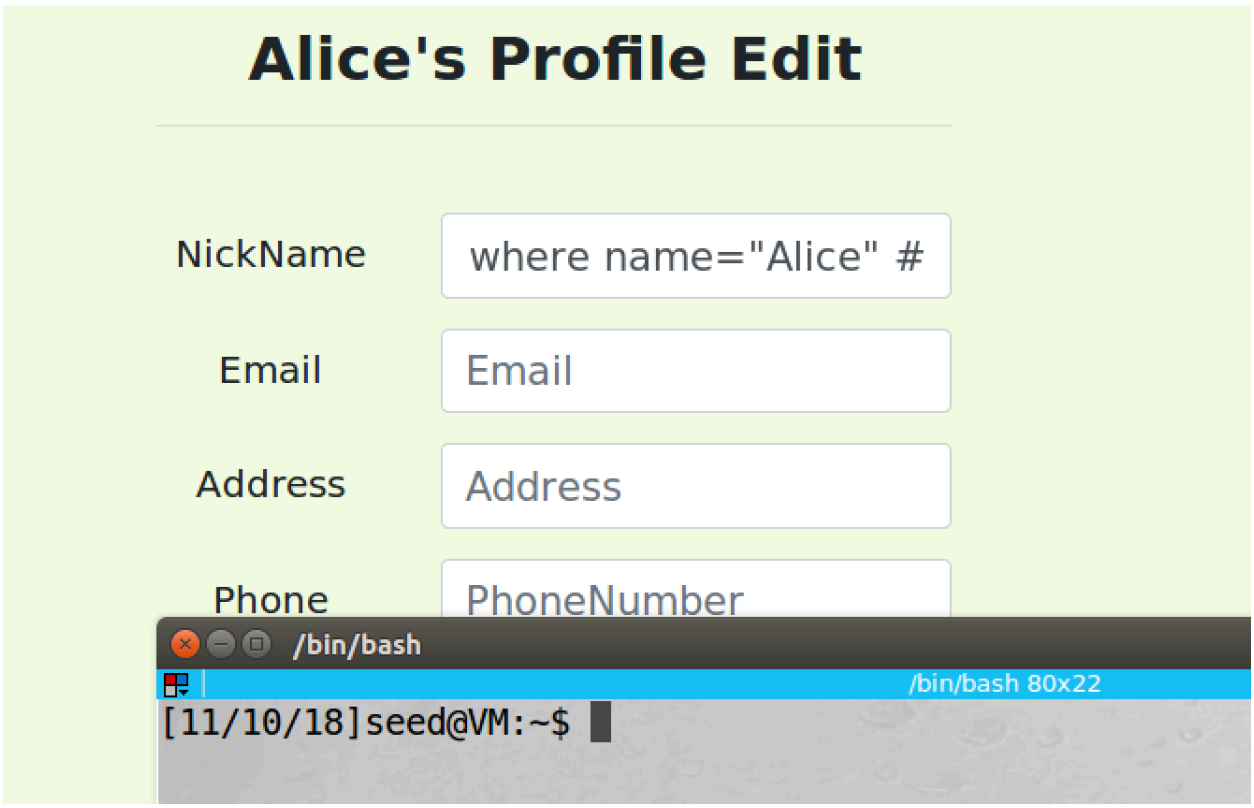
SQL statement is separated by (;), so after append (;) on admin', we can start new statement, this statement can delete a row in the credential table with name Ted. Then we use (;) to end this statement, and we also need to append (#) at last to comment out the password field. However, our attack fails (screenshot2). The reason is that the server uses "mysqli" to create database connection, and such extension does not allow multiple SQL statements, so our attack fails. But if the server uses multi_query(), this allows multiple SQL statements, then our attack will succeed.

Task3: SQL Injection Attack on UPDATE Statement

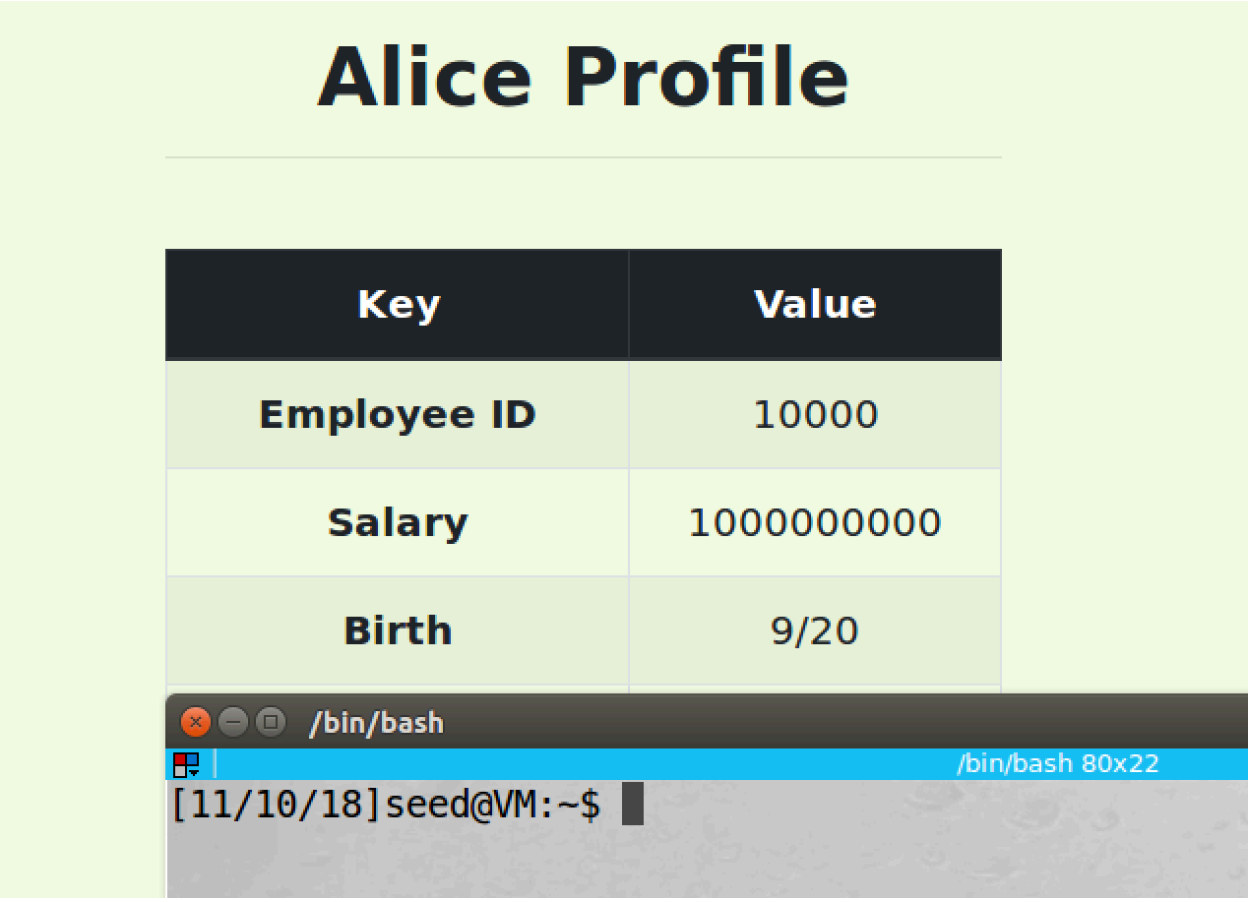
Task3.1: Modify your own salary



screenshot1, we login to Alice account



screenshot2, we enter input in NickName field, and we leave other field blank. The full input is **', salary=1000000000 where name="Alice" #**

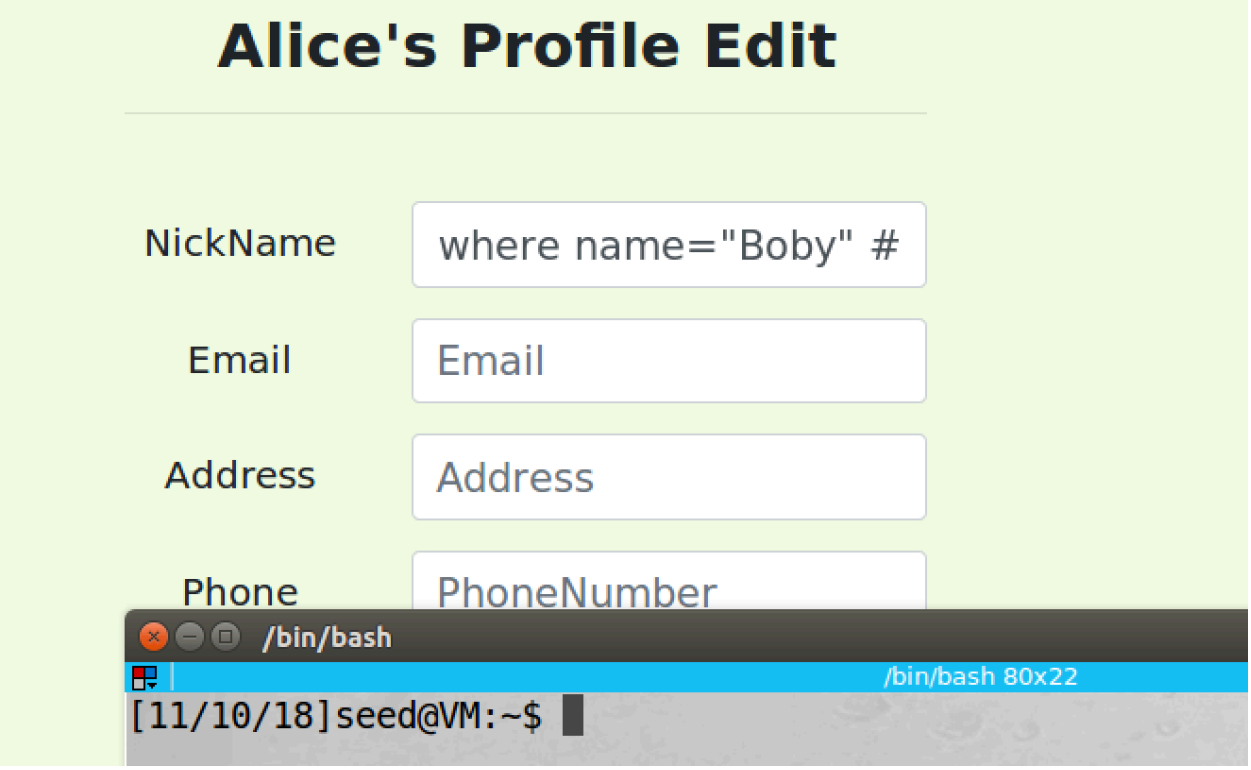


screenshot3, we successfully change Alice's salary to 1000000000

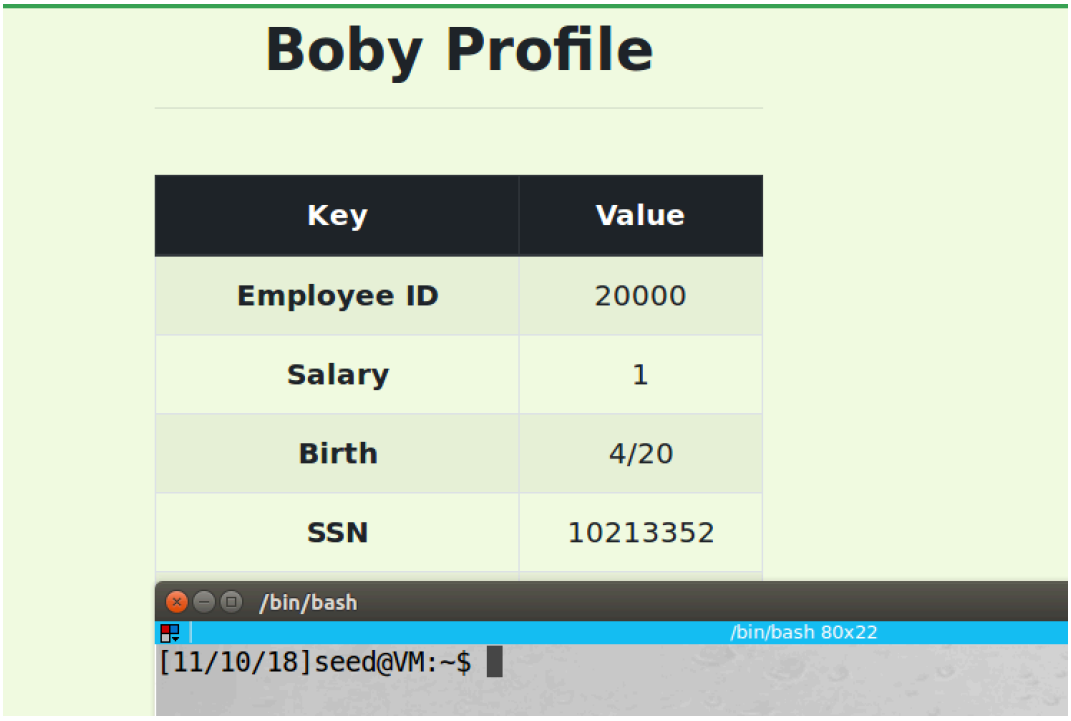
Observation and Explanation:

In this task, we want to perform SQL injection attack on UPDATE statement. If we already know the target column name in the table, we can inject this column name in a UPDATE statement, then the target column will be updated. In our case, we want to update Alice's salary which has name salary in the table, so we can add this field to the UPDATE statement in the profile editing page. The UPDATE statement in the profile editing page contains several field; as screenshot2 shows, we enters: **', salary=1000000000 where name="Alice" #** in the nick name field. (') is used to close the input field of nick name field. (,) is used to separate two field. salary= 1000000000 is new field which we want to update. We know that Alice's name is Alice in the table, so we add where statement to make sure we only change her account. (#) is used to comment out everything after our statement in the same line. As screenshot3 shows, we successfully modified Alice's salary.

Task3.2: Modify other people's salary



screenshot1, we modify Boby's salary on Alice's account, the full input is **', salary=1 where name="Boby" #**

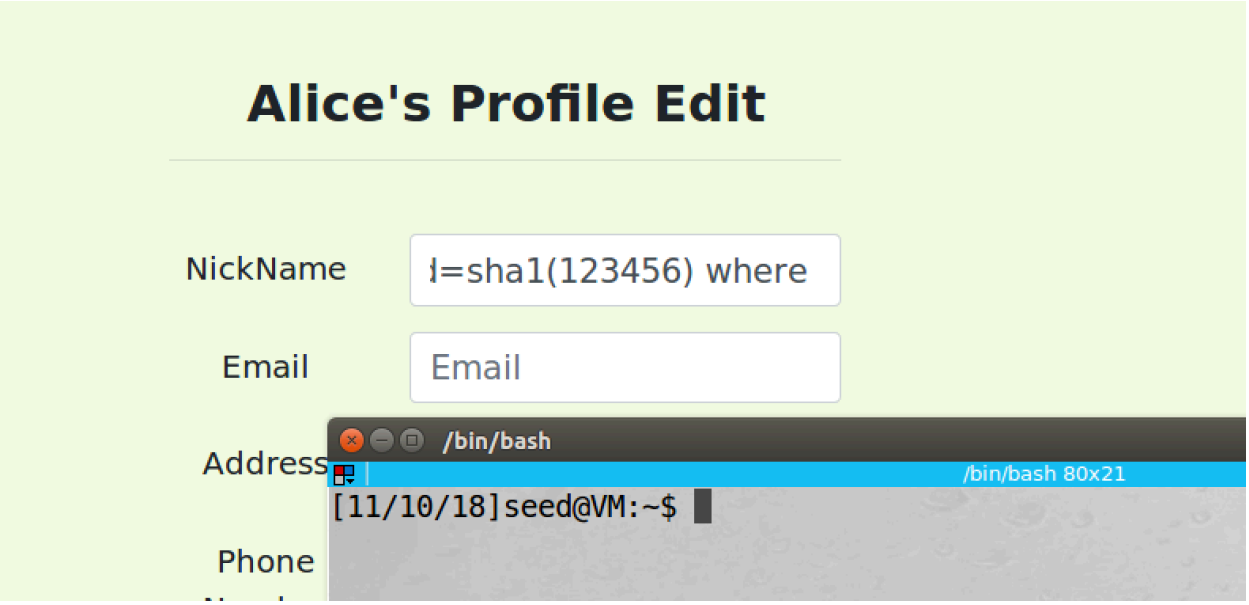


screenshot2, we login to Boby's account, and the salary is changed to 1

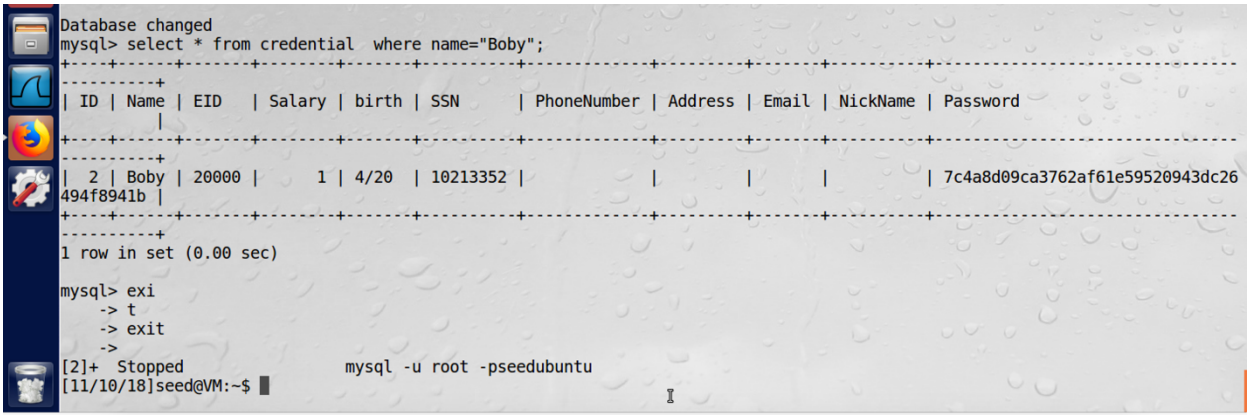
Observation and Explanation:

In this task, instead of changing Alice's salary, we change Boby's salary. The input statement is very similar to last task. There are two difference; first, salary value is 1; second, in where statement, we change the name to Boby, so only Boby's account will be modified (screenshot2). As screenshot1 shows, after editing on Alice's account, we login to Bob's account, and his salary becomes 1. So our attack is successful.

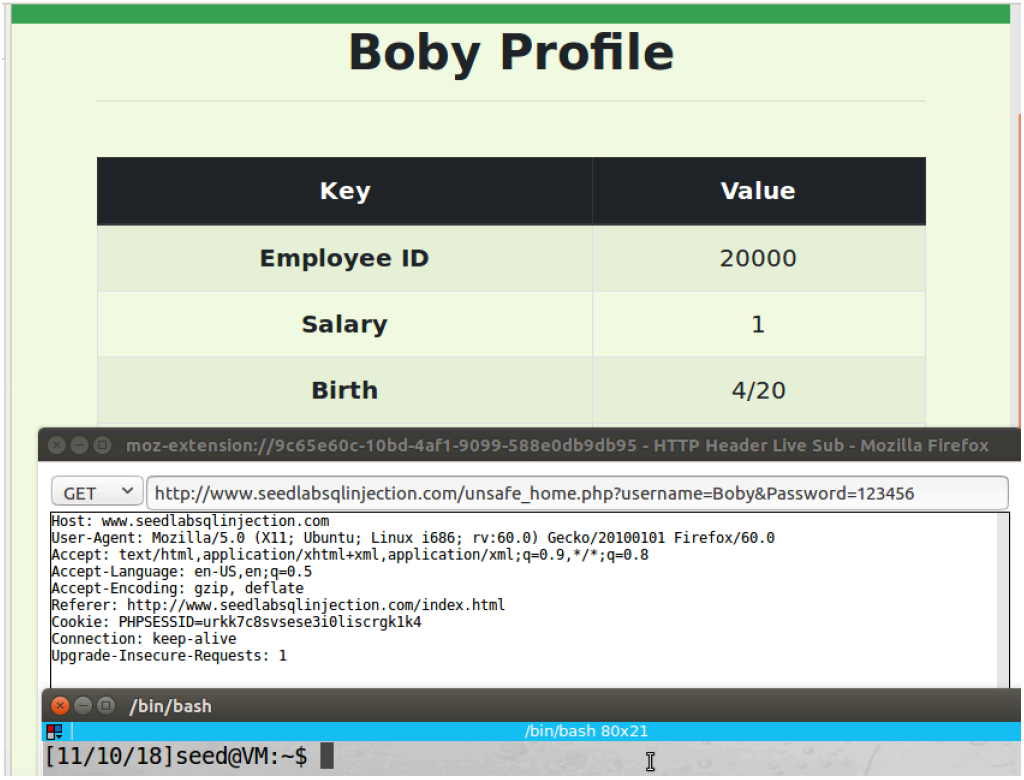
Task3.3: Modify other people' password



screenshot1, we change Boby's password to 123456 on Alice's profile editing page



screenshot2, we check Boby's password on the MySQL console, the hash value changed



screenshot3, we login to Boby's account by the new password.

Observation and Explanation:

In this task, we want to modify Boby's password. In fact, this attack is still similar to last attack, we just need to change salary field to Password field in our input. The input is following:

', Password=sha1(123456) where name="Boby" #

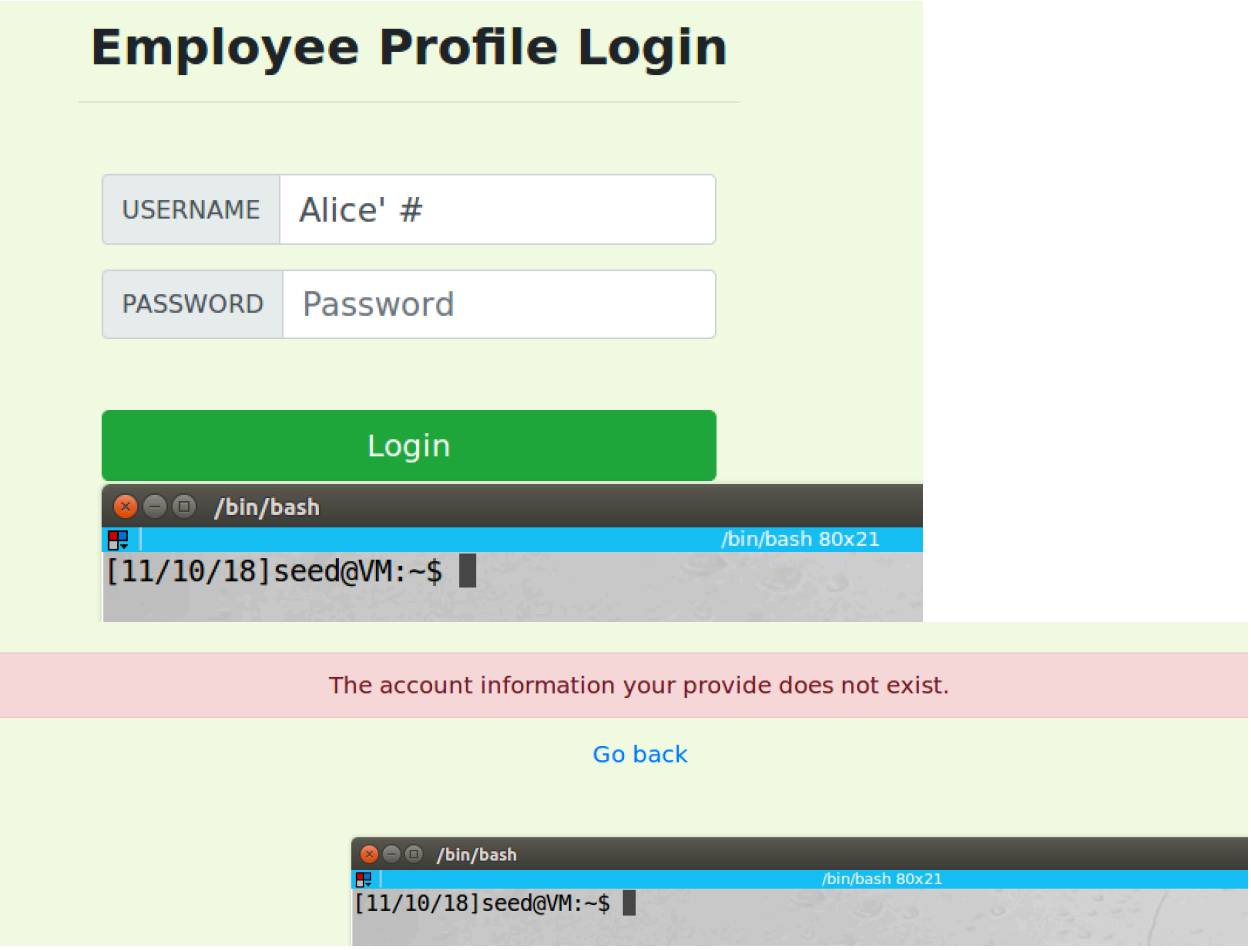
Because on the server, the password is encrypted by hash function before it is saved in the database. When we login to the account, our input password will also be hashed, and it will be compared to the password in the database. If we do not hash the new password, the plaintext will be stored in the database. As a result, when we use the new password to login, even the password is correct, it will be compared to its hash value; obviously, they are different. Therefore, when we update the password in the profile editing page, we use function sha1() to encrypted the new password. As screenshot 1 and 2 show, after we update the password, the hash value of the new password is stored in the database. And then we successfully login to Boby's account by the new password, the new password is also captured by the inspection tool (screenshot3).

Task4: Countermeasure-Prepared Statement

```
<div class="container  col-lg-4 col-lg-offset-4" style="padding-top: 50px; text-align: center;">
  <h2><b>Employee Profile Login</b></h2><hr><br>
  <div class="container">
    <form action="safe_home.php" method="get">
      <div class="input-group mb-3 text-center">
        <div class="input-group-prepend">
          <span class="input-group-text" id="uname">USERNAME</span>
        </div>
        <input type="text" class="form-control" placeholder="Username" name="username" aria-label="
      </div>
      <div clas
        <div cl
          <span
        </div>
        <input
```

```
/bin/bash
/bin/bash 80x21
[11/10/18]seed@VM:~$
```

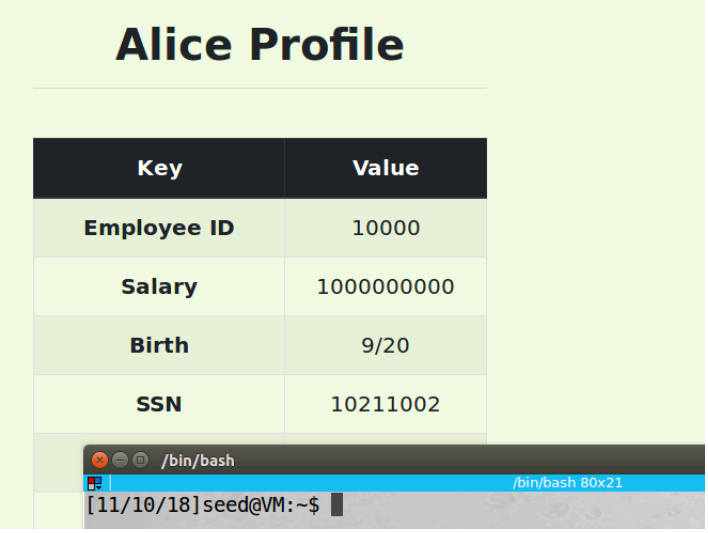
screenshot1, we open index.html file, and we change value in action file of a form to “safe_home.php”, this php page use prepared statement to prevent SQL injection attack on login page



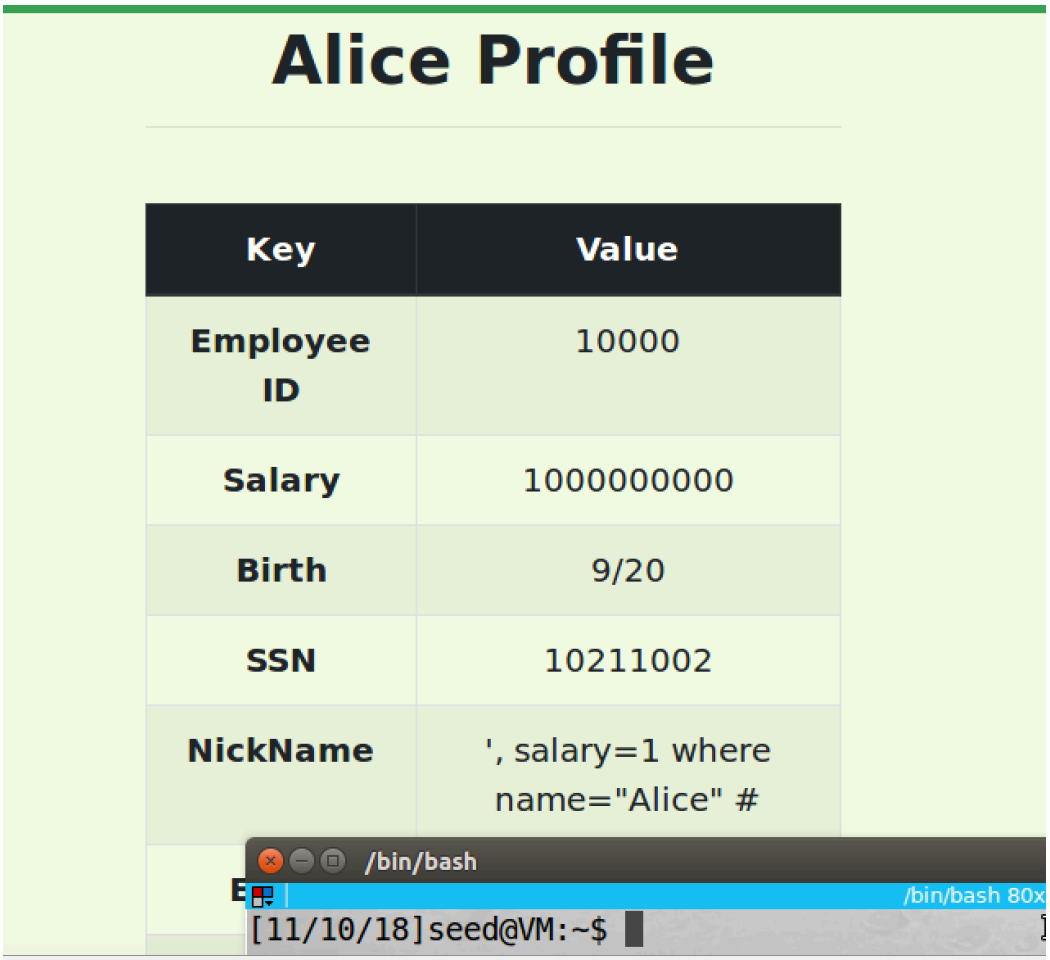
screenshot2, we try to login to Alice account again with same method in task2, but this time our attack fails

screenshot3, we login to Alice account by correct username and password, it works

screenshot4, in the unsafe_edit_frontend.php file, we change the action field of the Profile Edit form to safe_edit_backend.php, which implements prepared statement to prevent SQL injection on profile editing page



screenshot5, this time we try to change Alice's salary to 1 dollar by the same method used in task3



screenshot6, after we submitted, the nick name filed become our input. So our attack fails

Observation and Explanation:

In this task, we learn how to prevent SQL injection. There are several ways to prevent SQL injection, here we use prepared statement. As the previous attacks show, the fundamental problem of SQL injection is that program code and user date are mixed together, and we cannot separate them or see the boundary between them after the SQL statement sent to the database. If the data come from user which contains some keywords (i.e. (') (#)), the parser cannot separate them from original program; as a result, these date can be a part of original program, and they will be executed and cause damage. Prepared statement solves this problem by separate code and date. We first create SQL statement template and send it to database, then the template will be compiled. Now the template becomes pre-compiled query statement, and it left some unspecified parameters (?), these parameters need to be filled in by user. Later, when data arrive, they will not go through compilation; instead they will be directly plugged into the pre-compiled query statement. So data and code are totally separated. As screenshot1 and 2 show, we use safe_home.php to login, this php file uses prepared statement; and then we login to Alice's account by same method used in task2 again, but this time we fail. We also try to login by account and password, we successfully login to Alice account (screenshot3). As

screenshot4 shows, we also use safe_edit_backend.php to edit profile, again this php file uses prepared statement. As screenshot5 and 6 show, we want to modify Alice's salary to 1 dollar; with the same method which is used in task3, but such method cannot work anymore. After we submit, the nick name field becomes our input, and Alice's salary does not change. So our attack fails.