Justin Gallagher

CSC 482

SQL Injection Lab

In this lab, we will exploit a SQL Injection vulnerability in a web application and see how to mitigate such vulnerabilities. SQL injection is a code injection technique that exploits the vulnerabilities in the interface between web applications and database servers. The vulnerability is present when user's inputs are not correctly checked within the web applications before being sent to the back-end database servers.

Task 1: Get Familiar with SQL Statements

In this task, we familiarize ourselves with SQL Statements. The VM has a database set up called `Users`, which has a table called `credential`. `credential` stores personal information of employees like employee ID, password, social security number, salary etc. We will use MySQL along with this database. Using the username `root` and the password `seedubuntu`, I logged into MySQL using this command: `mysql -u root –pseedubuntu`. Once logged in, the Users database can be loaded using the following SQL statement: `use Users;` . The console outputs "`Database changed`" to let you know your statement was successful. Now that we are in the `Users` database, we can use the following SQL statement to show the tables in the Users database: `show tables;` . This query shows us one table named credential. This is the table we expect to see within the database, so we know out SQL statement was successful. Now, the lab gives us the task of printing out all the information pertaining to the user Alice. This can be done by using the following SQL statement: `SELECT * FROM credential WHERE Name = "Alice";` . The statement matches any record in the `credential` table that has the field `Name` 'Alice'. The result is pictured below:

```
mysql> SELECT * FROM credential WHERE Name = "Alice";
+----+-------+-------+--------+-------+----------+-------------+---------+------
-+----------+------------------------------------------+
| ID | Name  | EID   | Salary | birth | SSN      | PhoneNumber | Address | Email
 | NickName | Password                                 |
+----+-------+-------+--------+-------+----------+-------------+---------+------
-+----------+------------------------------------------+
|  1 | Alice | 10000 |  80000 | 9/20  | 10211002 |             |         |
 |          | fdbe918bdae83000aa54747fc95fe0470fff4976 |
+----+-------+-------+--------+-------+----------+-------------+---------+------
-+----------+------------------------------------------+
1 row in set (0.00 sec)
```

Task 2: SQL Injection Attack on SELECT Statement

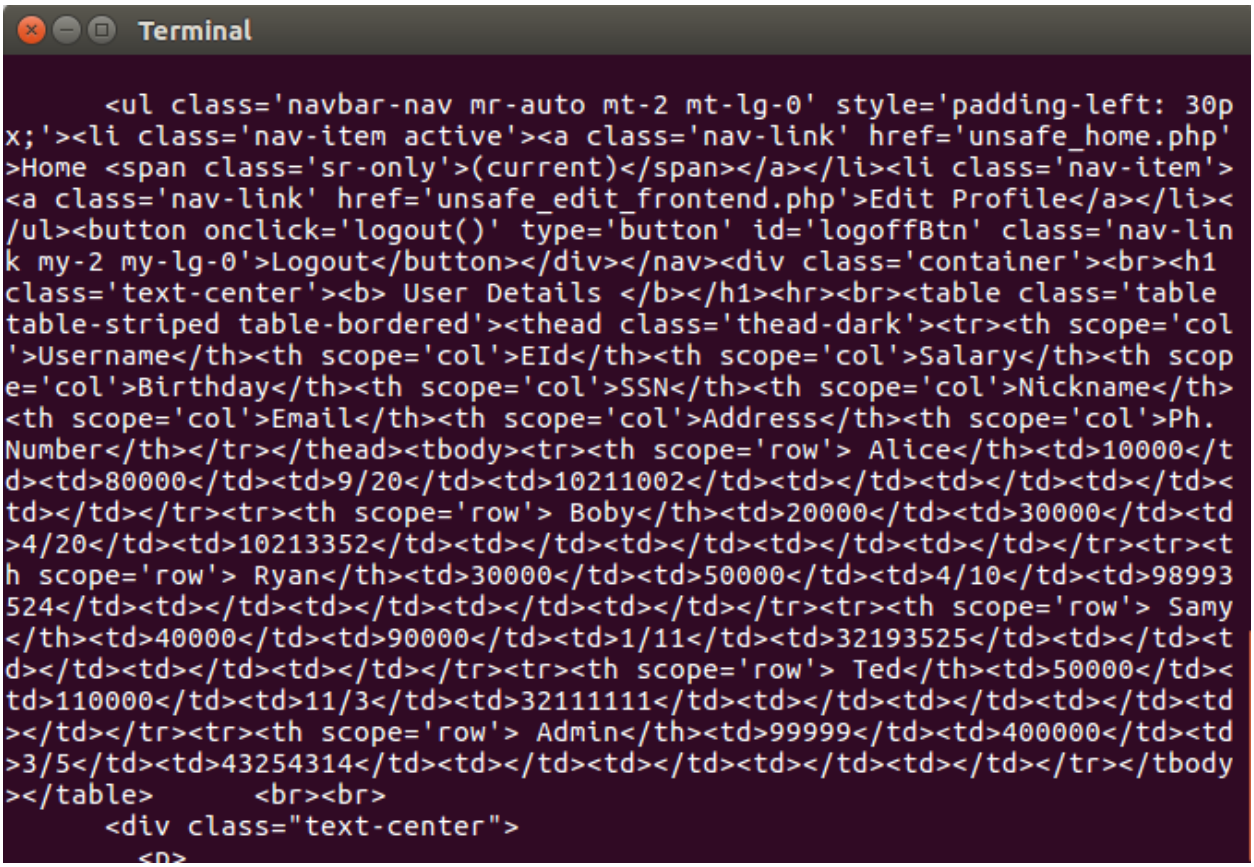
In this task, we use the webpage "http://www.seedlabsqlinjection.com/" to test out SQL injection attack on. This page asks users to provide a username and password. The web application authenticates users based on these two pieces of data, so only employees who know their passwords can log in. The goal of this task is to log into the web application without knowing any employee's credential. The lab gives us the location of the php code pertaining to the SQL query made when a user attempts to provide the

webpage with a username and password. In the /var/www/SQLInjection directory we can find the `unsafe_home.php` file which contains said code. Upon inspection, we discover that a SQL statement is constructed using the two pieces of data from the webpage; username and password. The following is the SQL statement from the code: `SELECT id, name, eid, salary, birth, ssn, address, email, nickname, Password FROM credential WHERE name = '$input_uname' and Password = '$hashed_pwd'` . If the username and password were designed in such a way, we could add additional statements to this SQL statement. To start this task (2.1), we first want to access the admin user account via the webpage. We assume that we do not know the password to admin, but we do know the username is admin. We load up the webpage and for the username I type `admin' #` and for the password I type `pass`. When I hit the enter key to login, the webpage directs me to a screen that lists out the credential table with all its records and fields. This happens because when entering the following data into the username and password, the SQL statement gets built and becomes: `SELECT id, name, eid, salary, birth, ssn, address, email, nickname, Password FROM credential WHERE name = 'admin' #' and Password = 'pass'`. The statement now enables us to access information pertaining to all the employees in the database only using the administrator id. The webpage that contains all the employee information is pictured below:

User Details

Username	Eid	Salary	Birthday	SSN	Nickname	Email	Address	Ph. Number
Alice	10000	80000	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				

The second part (2.2) of this task asks us to do the same thing as before with the admin id, but this time only using the terminal and the curl command. To start, we will take the URL from when we logged into the webpage using the username admin' # and password pass. The URL is as follows: 'http://www.seedlabsqlinjection.com/unsafe_home.php?username=admin'+#&Password=pass'. We can use this with the curl command in the terminal, but we must change a few things. Using the URL directly with the curl command will fail, because encoding will take place and it will not be recognized. To fix this, we replace the ' after admin with %27, we replace the space represented in the URL by the + with a %20, and we replace the # with a %23. The URL now becomes the following: 'http://www.seedlabsqlinjection.com/unsafe_home.php?username=admin%27%20%23&Password=pass'. We then run the following command within the terminal using the new URL: `curl 'http://www.seedlabsqlinjection.com/unsafe_home.php?username=admin%27%20%23&Password=pass'` Once the command is run, the following is output to the terminal:



The output shows the different rows and columns within the credential table along with the information pertaining to all the employees. This shows that we were successful in running the first attack but this time only using the terminal.

The last part (2.3) of this task requires us to use the semicolon to separate the SQL statement into two and use the new statement to delete a record from the table. We will be doing this from the login webpage. To construct our statement, I first enter `admin';` for the username, then for the password I enter `DELETE FROM credential WHERE Name = 'Alice'`. Upon pressing the enter key to log in, I am given an error: "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 '' and Password='0bfb7b469ca494fd9df2bcc34663bd259ac0a6ee'' at line 3]\n". Running a few more tests, it seems as though we are unable to successfully delete from the table. Upon further investigation, it turns out that php does not allow for multiple SQL statements to be ran at once, which is why last park of this attack throws an error.

Task 3: SQL Injection Attack on UPDATE Statement

In this task, we perform the SQL injection attack on the UPDATE statement via the edit page which is a part of the vulnerable webpage http://www.seedlabsqlinjection.com/. The code for this page is implemented in the file `unsafe_edit_backend.php` in the /var/www/SQLInjection directory. This gives us access to the SQL statement used to edit an employee's profile: `UPDATE credential SET`

nickname='$input_nickname',email='$input_email',address='$input_address', Password='$hashed_pwd',PhoneNumber='$input_phonenumber' WHERE ID=$id;. Using this, we can modify this statement to produce our own results.

The first part (3.1) of this task has us modifying Alice's salary. To do this, we assume that we are Alice trying to modify her own salary. We first login to her account using her username and password (Alice and seedalice). Next, we navigate to the edit profile page. In the given forms, I enter the following: 'Alice' for Nickname, 'alice@nku.edu' for Email, '' for Address, '555-5555', salary = 100000 WHERE name = 'Alice' #' for Phone Number, and '' for password. This makes it so that the SQL statement now reads: UPDATE credential SET nickname='alice',email='alice@nku.edu',address='', Password='',PhoneNumber='555-5555',salary=100000 WHERE name='Alice' #' When I clicked save changes, her account screen pops up with the changes made to the nickname, email, phone number and the salary. This shows that you can embed UPADTE statements into the SQL Injection attack.

The second part (3.2) of this task asks us to modify the salary of another employee. We will change Boby's salary to $1. We will be doing this from Alice's account. Once logged in to her account, I navigate to the edit profile page. I then type out all the same information as we did in the first part of this task, but instead of using '555-5555', salary = 100000 WHERE name = 'Alice' #' for the Phone Number, we will now use the following: '555-5555', salary = 1 WHERE name = 'Boby' #'. We then hit save changes and Alice's account page is loaded back up. To see if we were successful, I log into Boby's account to see that his salary is now $1 confirming that we can change the salaries of other employee's accounts.

The last part (3.3) of this task asks us to change Boby's password using Alice's account. To do this, we will use the same method as part two of this task. In Alice's edit profile page, we enter the following into the Phone Number field: '555-5555', Password = sha1('alice') WHERE name = 'Boby' #'. This will make it sho Boby's password is now 'alice'. After saving changes, I attempt to login to Boby's account using the new login information and was successful. This shows that we can change Boby's password from another employee's account.

Task 4: Countermeasure – Prepared Statement

In this task, we encounter the countermeasure for SQL injection attacks. The main problem with SQL injection attacks is the failure to separate code from data. This vulnerability allows for the manipulation of said data through malicious code being injected into the data. Prepared statements will allow for separation of the data from the compilation step meaning that anything contained within the data will now be unrecognizable to the program and instead just be a part of the data itself rendering this attack unsuccessful. This task asks us to fix the issues within the files for the webpage used in this lab. The files we will be changing are 'unsafe_home.php' and 'unsafe_edit_backend.php'. The following changes are implemented into the code:

```
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 passowrd 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_phonenu
mber);
  $sql->execute();
  $sql->close();
}
$conn->close()
```

and

```
// create a connection
$conn = getDB();
// Sql query to authenticate the user
$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();
$sql->close();
```

After making the following changes to the code, I check to see if the previous attacks will work. Upon trying the first attack from task 2 and the first attack from task 3, the error message arises: 'The account information your provide does not exist.' This shows that the malicious SQL statements are now being interpreted as just data and not as SQL queries.