**Assignment 9:- SQL Injection Attack**

**Name:- Aparna Krishna Bhat**

**ID:- 1001255079**

**Task 1:- Get Familiar with SQL Statements**

We start by logging into MySQL username **root** and password **seedubuntu** and changing the database to Users.

Text

Description automatically generated

When we look at all of the tables, we note that there is only one called credential.

Text

Description automatically generated

All of the details about the employee 'Alice' are printed using the command ***SELECT \* FROM credential WHERE Name=’Alice’;***

Chart

Description automatically generated with medium confidence

**Task 2:- SQL Injection Attack on SELECT Statement**

***Task 2.1: SQL Injection Attack from webpage***

Using admin’# as the username and admin as the password

Graphical user interface, application, website

Description automatically generated

The following is the output when we click on login.

Table

Description automatically generated

The username entered here causes the server to run the following query.

Graphical user interface, text, application

Description automatically generated

Since JavaScript can check if a field has been filled and, if it hasn't, it can request it by triggering a warning or error, preventing a successful SQL Injection. The # character causes everything after the word "admin" to be commented out, including the password. Using the admin ID, we were able to obtain all of the information about the employees.

***Task 2.2: SQL Injection Attack from command line***

I need to figure out the HTTP method is being used to send the Username and Password when the login form is submitted first. The data is sent to the unsafe\_home.php using the GET HTTP form, with username and password as parameters. This means I'll need to go to www.seedlabsqlinjection.com/unsafe\_home.php?username=admin'#&Password=

Graphical user interface, text, application, email

Description automatically generated

Graphical user interface, text, application, email

Description automatically generated

We use the curl command to send an HTTP request to the website and then login as before, and we get an HTML page in response.

Text

Description automatically generated

Text

Description automatically generated

All of the employee's information is returned in an HTML tabular format. As a result, we were able to repeat the attack from Task 2.1. Where the Web UI fails, CLI commands will assist in automating the attack. The curl command encoded the special characters in the HTTP request, which was a big change from the web UI. We use the following: Space - %20; Hash (#) - %23 and Single Quote (‘) - %27.

***Task 2.3: Append a new SQL statement***

In the username field, type the following to append a new SQL statement.

***admin'; UPDATE credential SET Name = 'Aparna' WHERE Name = 'Alice'; #***

At the web server, the ; distinguishes the two SQL statements. In this case, we're attempting to change the name of the entry from Alice to Aparna. When we click login, we see that an error occurred while running the query, and our attempt to run another SQL command failed.

Graphical user interface, text, application, email

Description automatically generated

To delete a record from the database table, we'll try something similar.

***admin'; DELETE FROM credential WHERE Name = ’Samy’; #***

Graphical user interface, application, website

Description automatically generated

When the query is modified to the one entered in username, we get a similar error.

Graphical user interface, text, application, email

Description automatically generated

Since the mysqli::query() API in PHP's mysqli extension does not enable multiple queries to run simultaneously in the database server, this SQL injection does not function against MySQL. Since the MySQL server does allow multiple SQL commands in a single string, the problem is with the extension rather than the server itself. This MySQLi extension restriction can be solved by using mysqli -> multiquery (). However, we should never use this API for security reasons, and we should avoid using SQL injection to run multiple commands.

**Task 3: SQL Injection Attack on UPDATE Statement**

***Task 3.1: Modify your own salary***

We can change Alice's salary by logging into her account and editing her profile. Using username Alice and password seedalice and then click on the Edit Profile button on the top menu

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

We enter the following information in the form. In the nickname field, type a string that will allow us to add salary to the list of fields that will be modified. I'll give it a shot by entering

***‘, salary=’900000***

A screenshot of a computer

Description automatically generated

We will see the profile as soon as we save the changes. The intention is that this will trigger the SQL query to be updated to this.

$sql=”UPDATE credential SET nickname=”, salary=’900000′,email=’$input\_email’,address=’$input\_address’,Password=’$hashed\_pwd’, PhoneNumber=’$input\_phonenumber’ WHERE ID=$id;”;

A screenshot of a computer

Description automatically generated

The SQL injection attack was successful

***Task 3.2: Modify other people’ salary***

Before any updates, we can see Boby's profile. Now, in the Phone number segment, we try to adjust Boby's salary from Alice's account using the following string.

***‘, salary=1 WHERE Name=’Boby’;#***

Table

Description automatically generated

Graphical user interface, application

Description automatically generated

We log into Boby's profile after saving the changes and verify that we have successfully adjusted his salary. Except for the password field, which is hashed, we could put the string in any of the other fields.

Graphical user interface, table

Description automatically generated

***Task 3.3: Modify other people’ password***

To change Boby's password, follow the same steps as before and edit the field "Phone number" in Alice's profile. Enter the below command in the Phone number field and save the changes.

***', Password = sha1('hacked1') WHERE name= 'Boby' #***

Graphical user interface, application

Description automatically generated

We log out of Alice's account and attempt to log in to Boby's account after saving the changes.

I've used the previously given password to demonstrate that it no longer functions, but Alice will not have this knowledge and thus will not be able to complete this step.

Graphical user interface, application, website

Description automatically generated

Graphical user interface, application

Description automatically generated

When we try to log in with the new password, we see that we are able to do so successfully. We are essentially performing the same steps as the program by using the sha1 function in our input. This indicates that our SQL injection attack to change passwords was successful.

Graphical user interface

Description automatically generated

**Task 4:- Countermeasure — Prepared Statement**

To address this problem, we'll build prepared statements based on the SQL statements that were previously exploited. In the unsafe\_home.php file, the SQL statement used in Task 2 is rewritten.

Change this part of the code

Graphical user interface, text, application

Description automatically generated

Rewritten as follows

Graphical user interface, text, application

Description automatically generated

Task 2.1 attack is being re-attempted. We can no longer access the admin account because we are no longer successful. There was no user with the credentials admin' # and password admin, according to the error.

Graphical user interface, application

Description automatically generated

Graphical user interface, application

Description automatically generated

Trying the attack from the terminal using curl

A picture containing text

Description automatically generated

Text

Description automatically generated

This is how the SQL injection attack fails as well.

The SQL statement in the unsafe\_edit\_backend.php file that was used in task 3 has now been rewritten.

I changed this part of the code

Graphical user interface, text, application

Description automatically generated

Rewritten as follows

Graphical user interface, text, application

Description automatically generated

We see that the salary does not adjust when we retry the same as in Task 3.1 and save the changes, so we are unable to perform SQL injection with prepared statements. The compilation phase transforms a prepared statement into a pre-compiled question with blank placeholders for details. We must provide data to run this pre-compiled query, but this data will not go through the compilation step; instead, it will be plugged directly into the pre-compiled query and sent to the execution engine. As a result, even if the data contains SQL code, the code would be considered as part of the data without any special meaning if the compilation phase is skipped. The prepared statement protects against SQL injection attacks in this way.

A screenshot of a computer

Description automatically generated