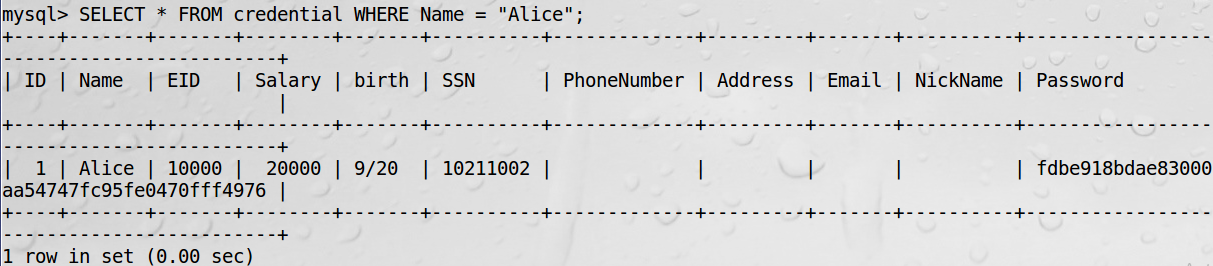
Q1: None

Q2:

SELECT \* FROM credential WHERE Name = “Alice”;



Q3.1

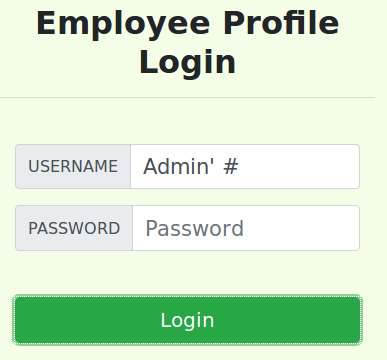
1. The exact content typed into USERNAME and PASSWORD:

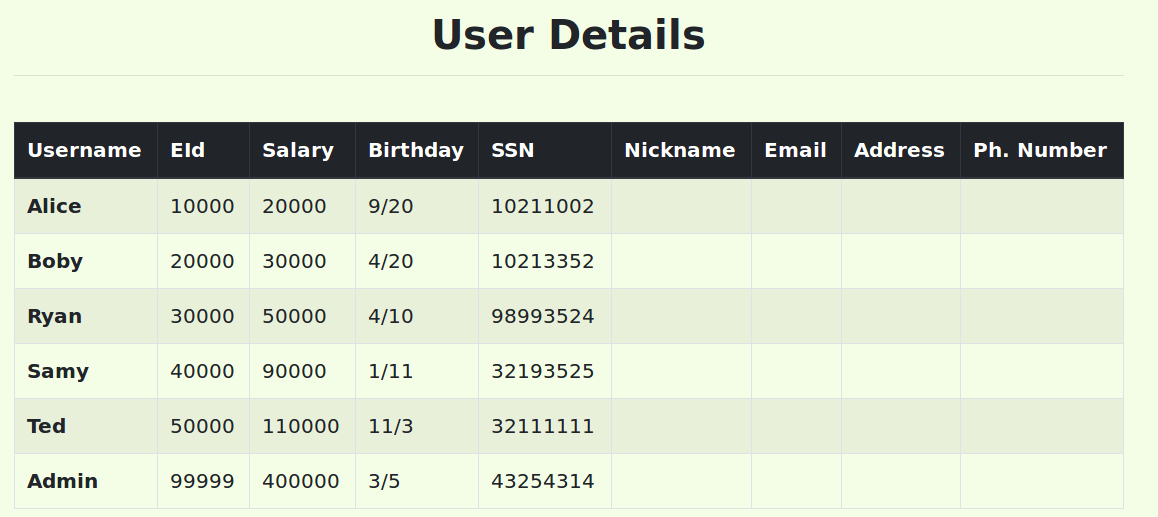
USERNAME: Admin’ #

PASSWORD:

Note: Nothing is typed into PASSWORD for this attack.

1. Screenshots:





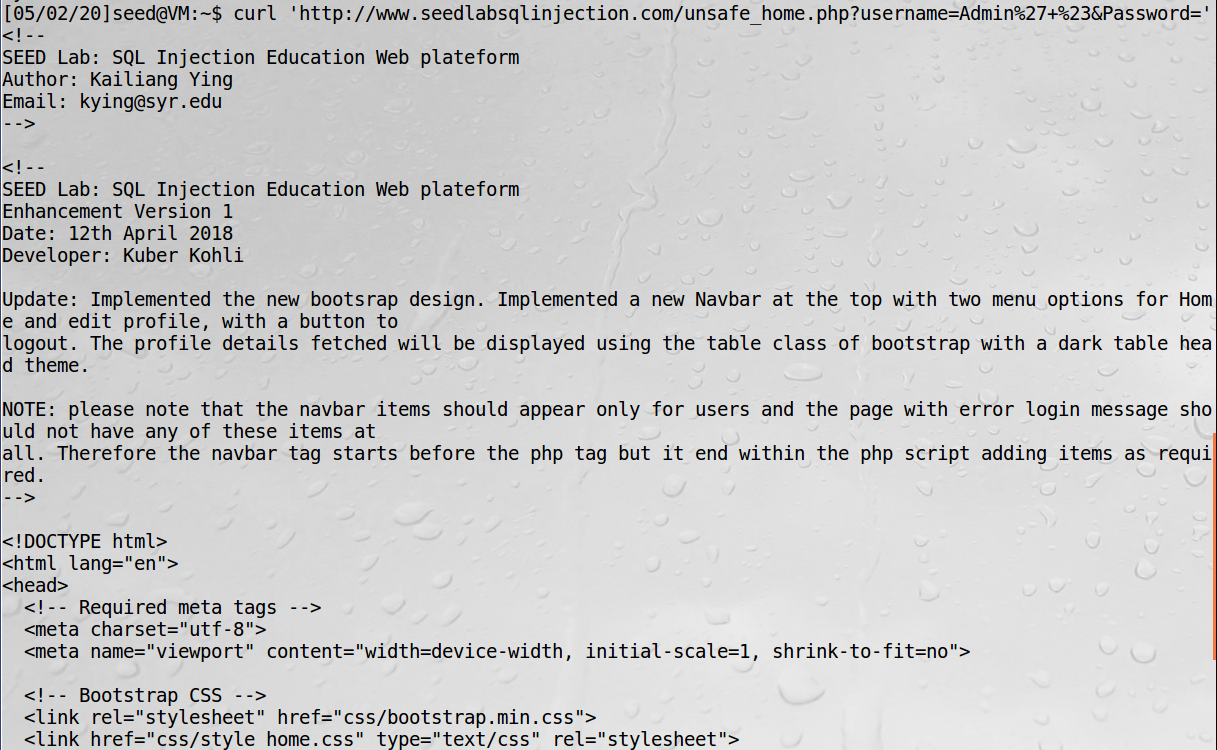
Observation: Given that I know the admin’s account name but not the password. I constructed an SQL injection attack by using the method above which exploits the vulnerability in the dynamic string evaluation in *unsafe\_home.php*. In this attack, I entered the admin’s account name in the username field, 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:

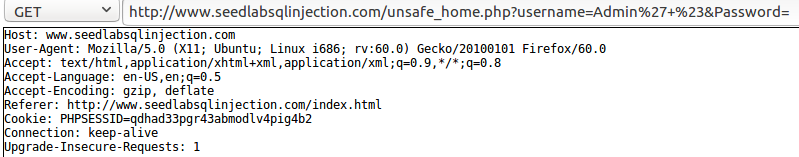
curl ‘http://www.seedlabsqlinjection.com/unsafe\_home.php?username=Admin%27+%23&Password=’

1. The output of the command line:





1. HTTP header used for constructing the attack from the command line:



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:

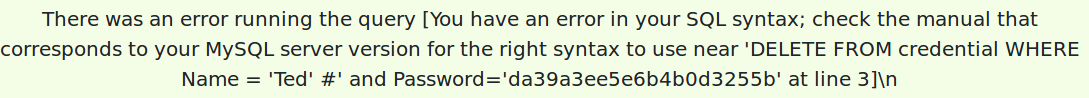
USERNAME: Admin’; DELETE FROM credential WHERE Name = ‘Ted’ #

PASSWORD:

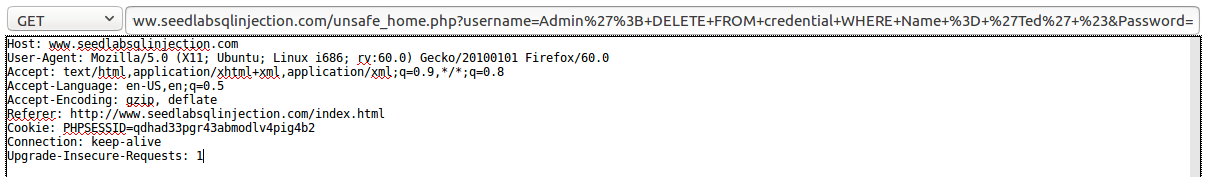
* The password is empty.

Screenshots:

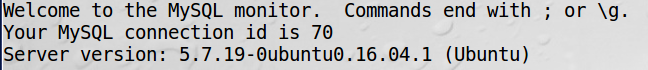
* Error message:



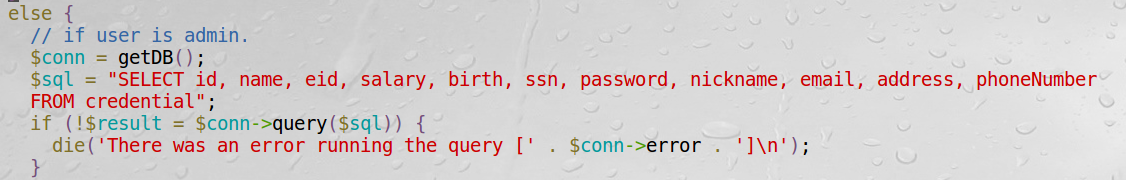
* GET request when submitting the form:



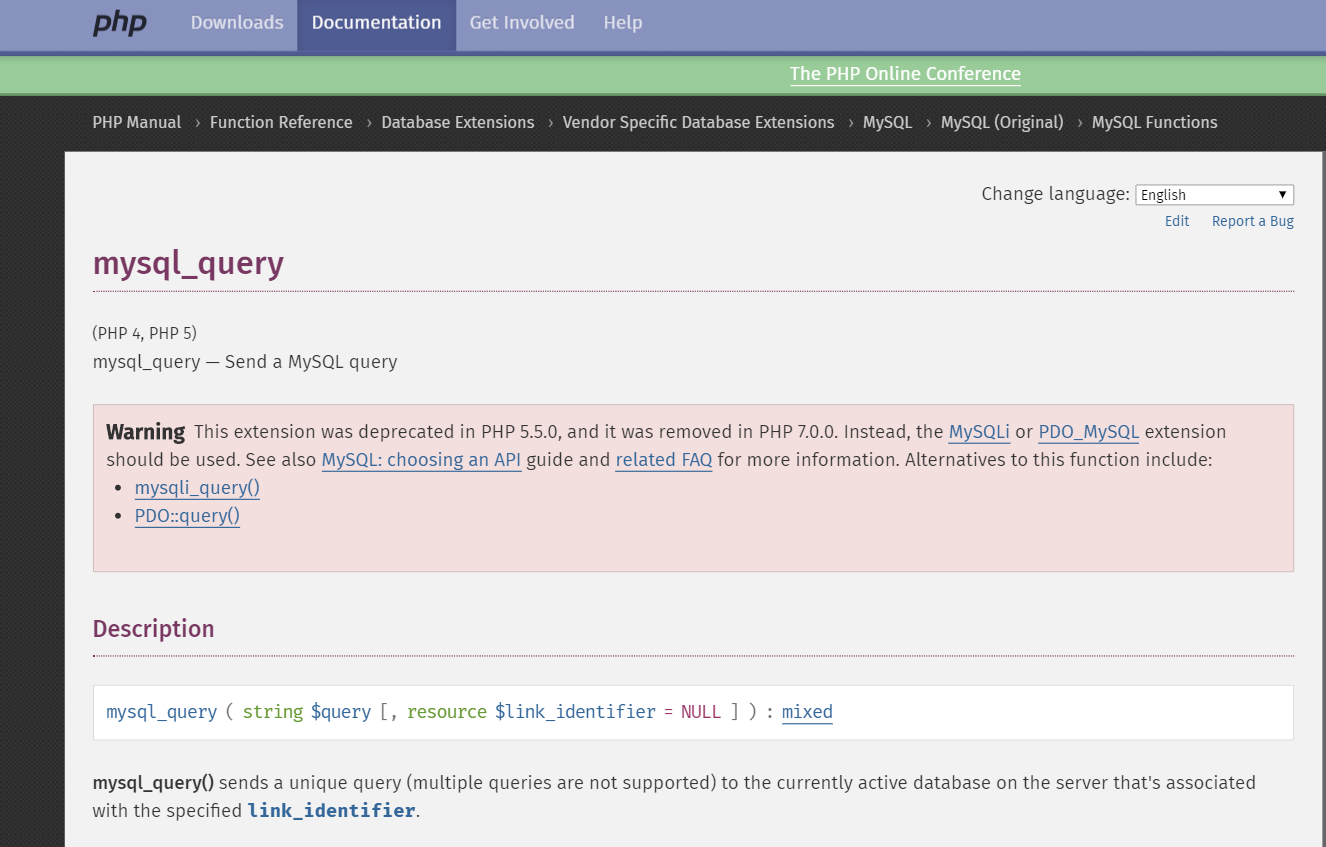
* MySQL version:



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



* Documentation for the PHP *query* function:



Explanation for failure to append a new statement:

The attack was not successful because this is a feature of PHPs 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.

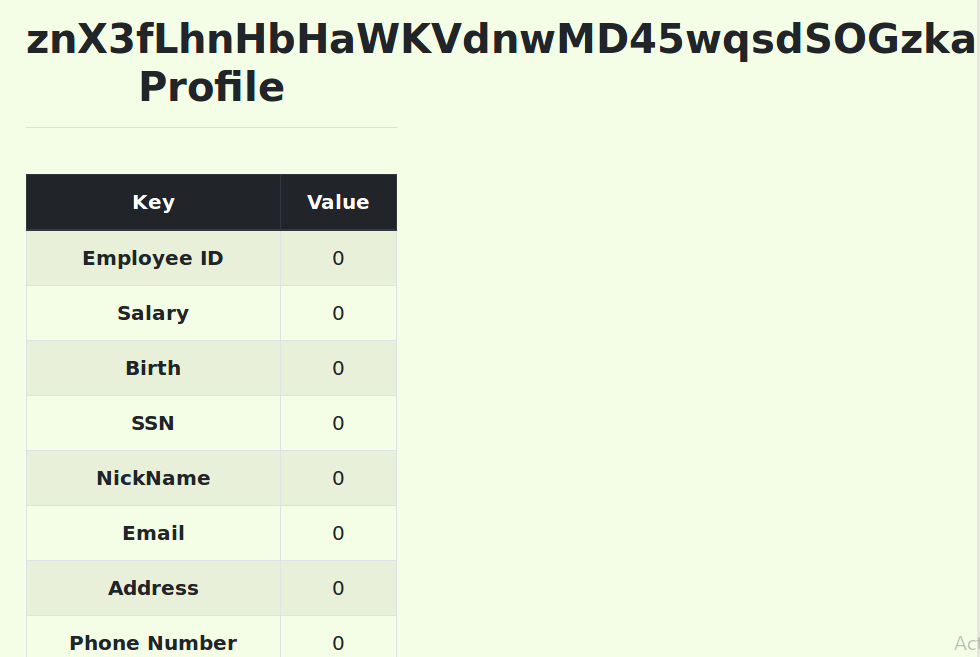
1. The exact content typed into USERNAME and PASSWORD:

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:

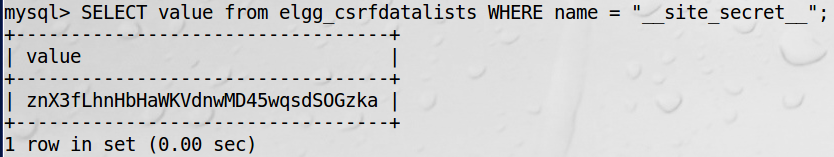
* The PASSWORD field is empty.

1. The output of the successful attack:



1. Screenshots:

The content of the site secret:



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:

1. Find the table name.

USERNAME: ‘ UNION SELECT 0, TABLE\_NAME, 0, 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.

1. Find the database name.

USERNAME: ‘ UNION SELECT 0, TABLE\_SCHEMA, 0, 0, 0, 0, 0, 0, 0, 0, 0 FROM information\_schema.TABLES WHERE TABLE\_NAME = ‘elgg\_csrfdatalists’ #

PASSWORD:

* + The PASSWORD field is empty.

1. Find the column names.

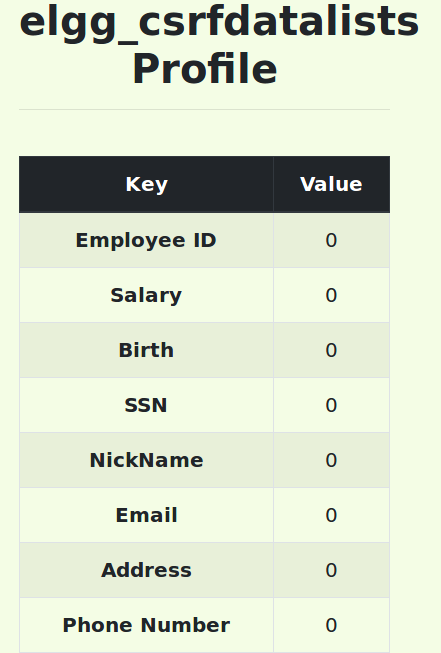
USERNAME: ‘ UNION SELECT 0, COLUMN\_NAME, 0, 0, 0, 0, 0, 0, 0, 0, 0 FROM information\_schema.COLUMNS WHERE TABLE\_NAME = ‘elgg\_csrfdatalists’ #

USERNAME: ‘ UNION SELECT 0, COLUMN\_NAME, 0, 0, 0, 0, 0, 0, 0, 0, 0 FROM information\_schema.COLUMNS WHERE TABLE\_NAME = ‘elgg\_csrfdatalists’ LIMIT 1, 1 #

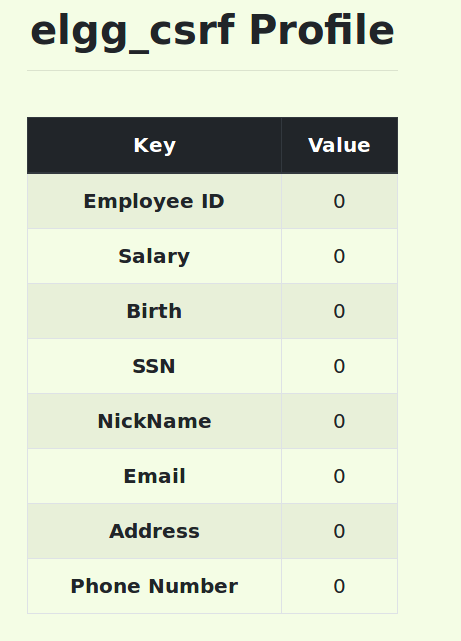
PASSWORD:

* + The PASSWORD field is empty.

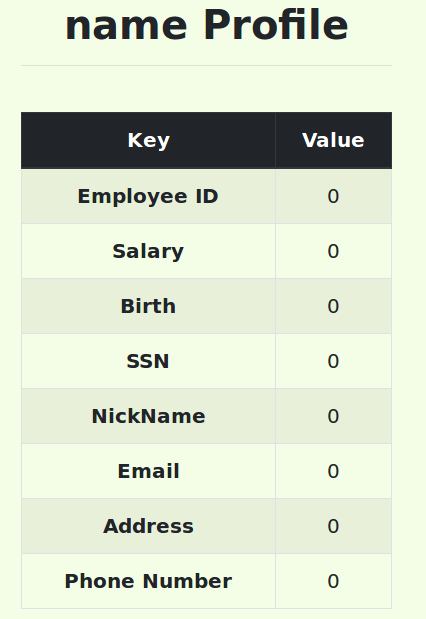
1. The output of the successful attack
2. Find the table name

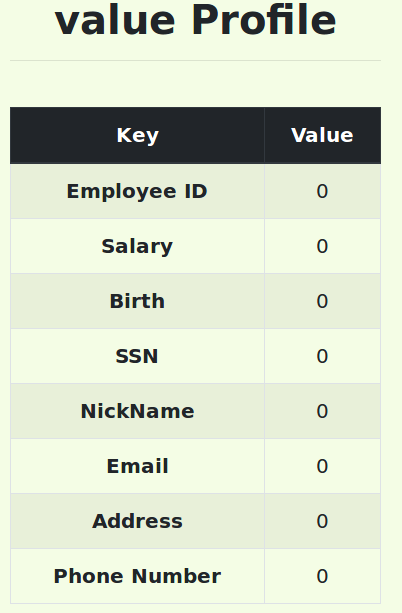


1. Find the database name



1. Find the column names





1. 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.

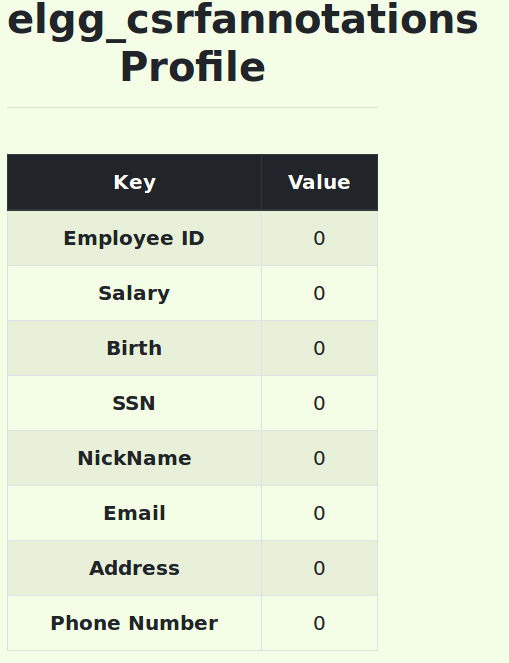
1. Additional screenshots

Trial-and-error for finding the table name.

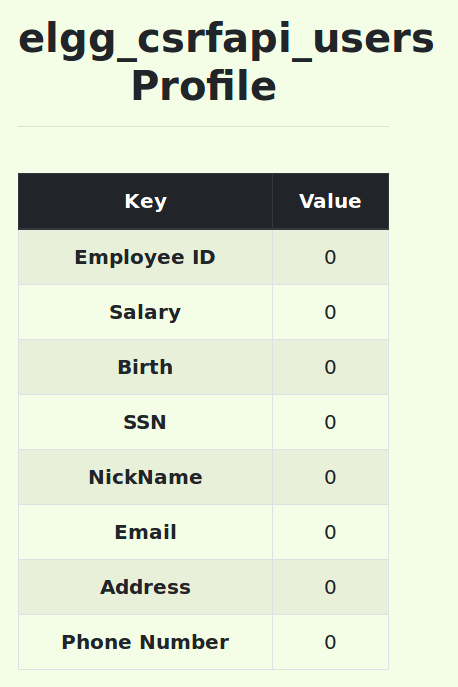
Attempt 1: (fail)



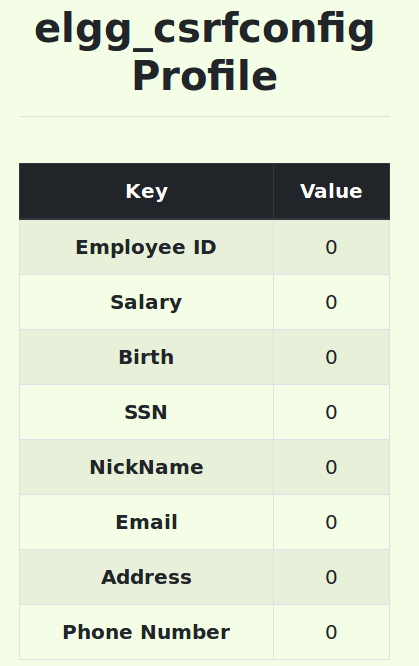
Attempt 2: (fail)



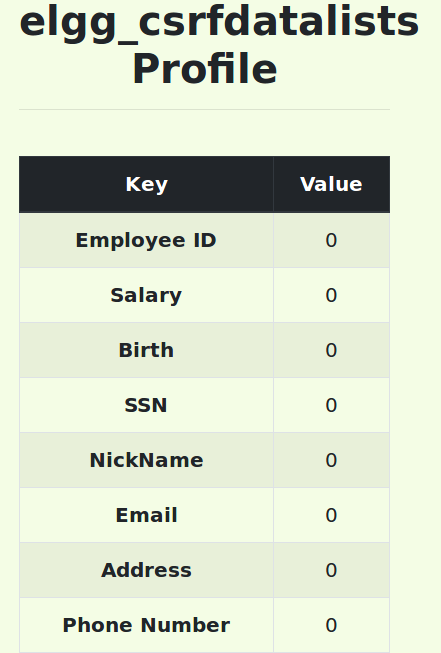
Attempt 3: (fail)



Attempt 4: (fail)



Attempt 5: (success!)



Q 4.1

1. The exact contents filled into each field.
2. 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, 0 FROM information\_schema.COLUMNS WHERE COLUMN\_NAME = ‘Nickname’ #

PASSWORD:

* + The PASSWORD field is empty.

1. 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, 0 FROM information\_schema.COLUMNS WHERE TABLE\_NAME = ‘credential’ LIMIT 3, 1 #

PASSWORD:

* + The PASSWORD field is empty.

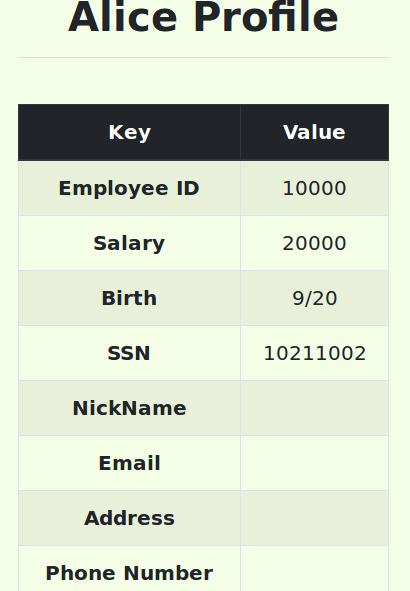
1. Next, using the column name *salary* obtained in part a, perform another SQL injection attack on the ‘Edit Profile’ page.

NICKNAME: ‘, salary = ‘90000’ WHERE EID = ‘10000’ #

All other fields are empty.

1. Screenshots

Alice’s profile before the attack:

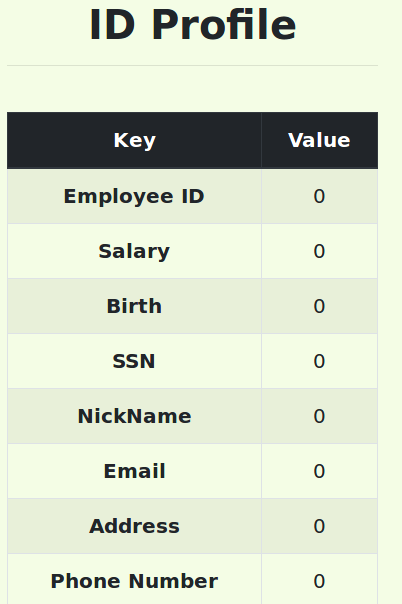


Attack in step a:



Trial-and-error in step b:

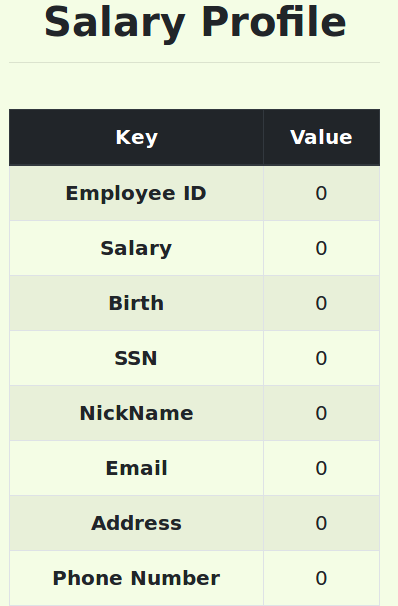
Attempt 1: (fail)



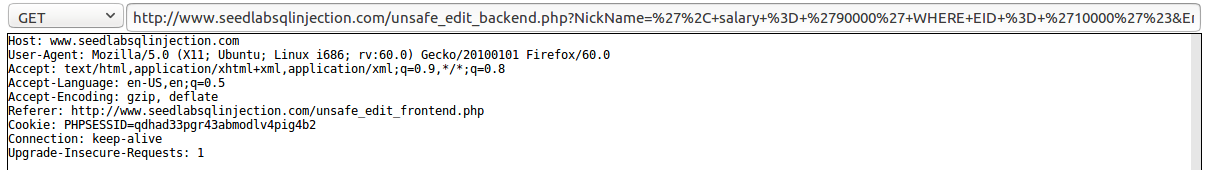
Attempt 2: (fail)



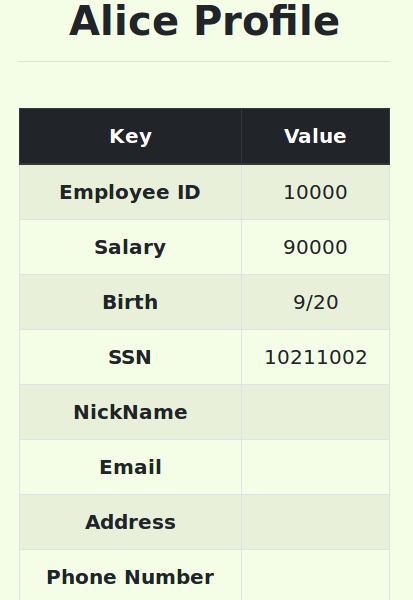
Attempt 3: (success!)



HTTP GET request when performing the attack:



Alice’s profile after the attack:



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

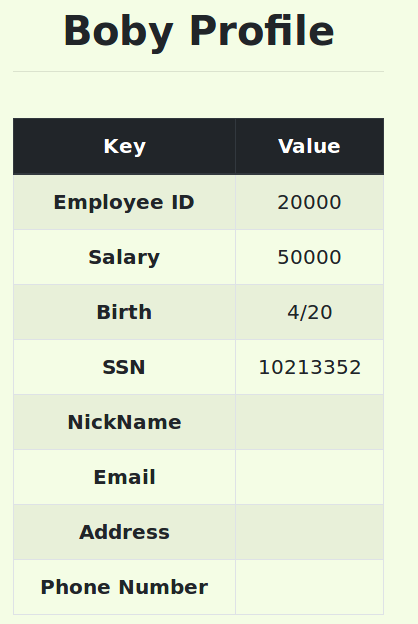
1. The exact contents filled into each field:

NICKNAME: ‘, salary = ‘1’ WHERE NAME = ‘Boby’ #

All other fields are empty.

1. Screenshots

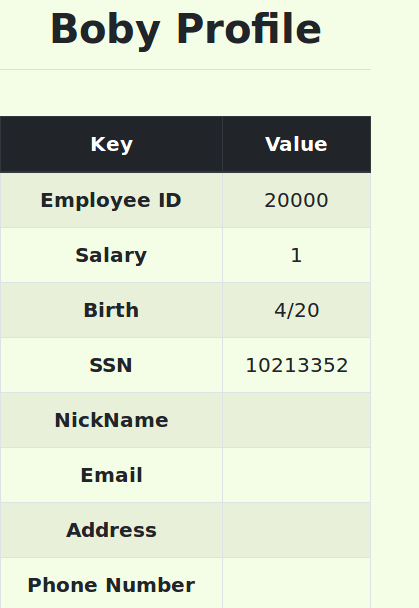
Boby’s profile before the attack:



HTTP GET request when updating Alice’s profile:



Boby’s profile after the attack:



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:
2. Alice plans to change Boby’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, 0 FROM credential WHERE EID = ‘10000’ #

PASSWORD:

* + The PASSWORD field is empty.

1. Using the hashed value of the password obtained in step a, Alice performs an attack on her “update profile” page to change Boby’s password to her own password.

NickName: ‘, password = ‘fdbe918bdae83000aa54747fc95fe0470fff4976’ WHERE NAME = Boby’ #

All other fields are empty.

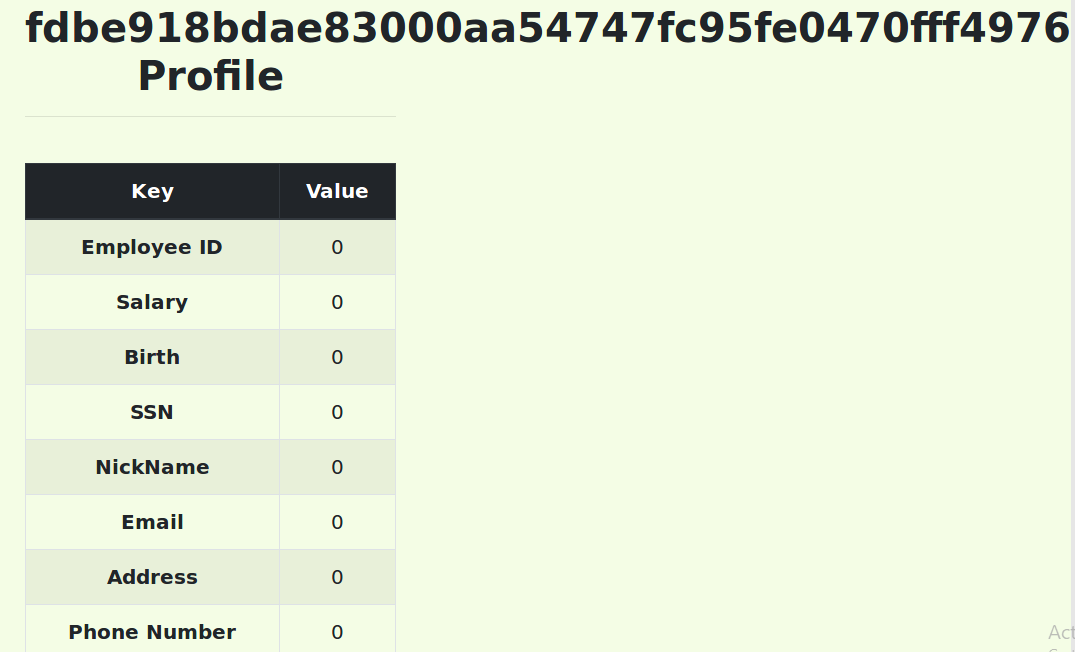
1. Alice uses the new password for Boby (i.e. Alice’s own password) to log into the web application as Boby.

USERNAME: Boby

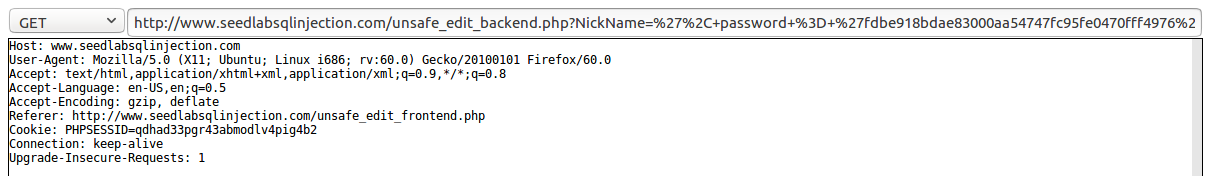
PASSWORD: seedalice

1. Screenshots

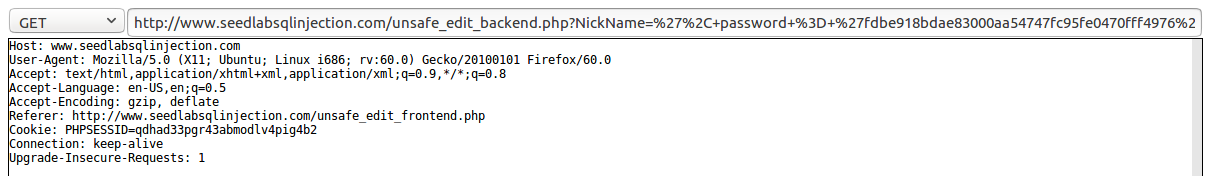
The hashed value of Alice’s password obtained from step a:



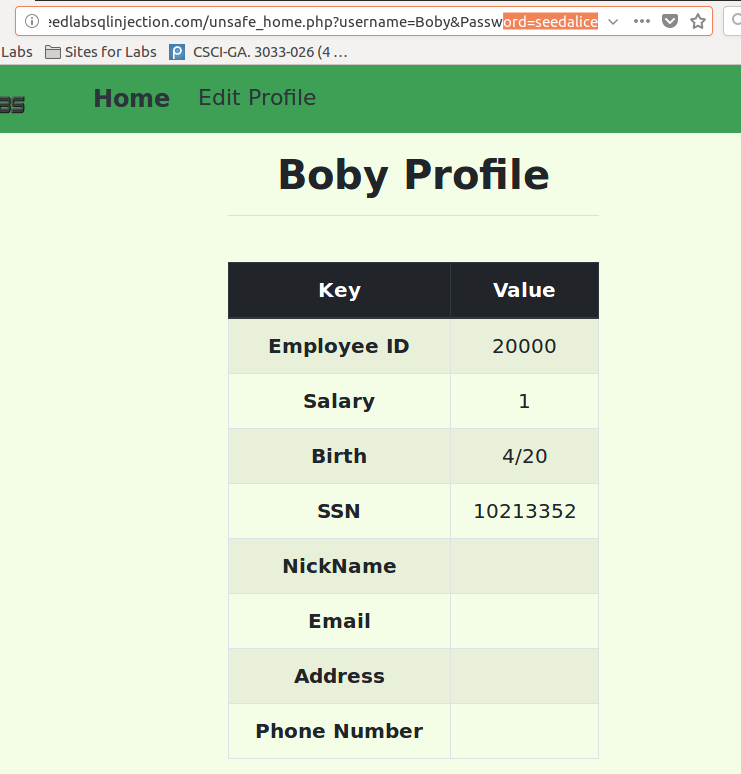
HTTP GET request when updating Alice’s profile:



HTTP GET request when Alice successfully logs in as Boby:



Alice logs in Boby’s profile using the updated password:



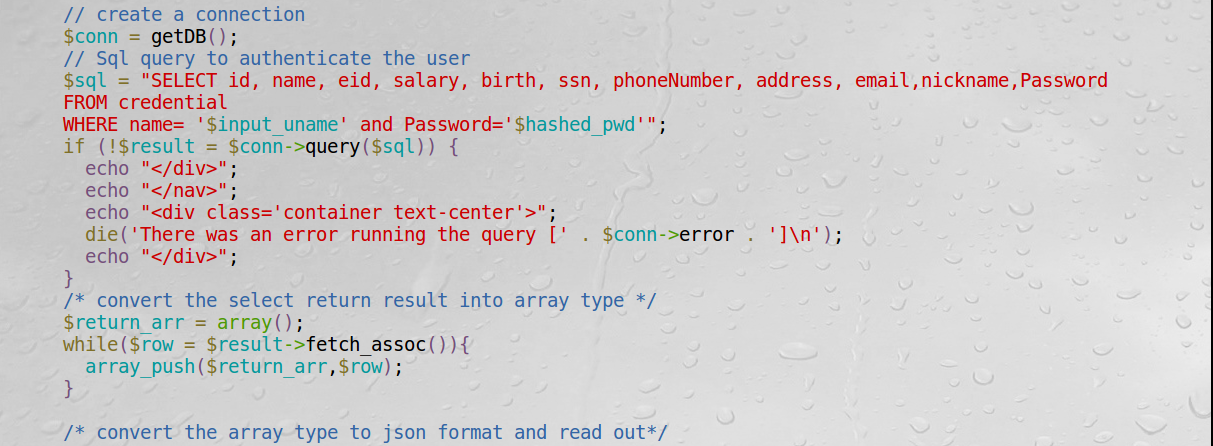
Observations: The goal is to modify Boby’s password. In this case, Alice decides to change Boby’s password to her own password. Since the database stores the hash value of the passwords instead of the plaintext password string, Alice has to find out the hash value of her own password first. The attack 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 Boby’s password with her own using the hash value just obtained. Finally, Alice can log in Boby’s profile with the updated password, i.e., her own password.

Notice that if Alice wants to change Boby’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 Boby. 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 Boby using the new password.

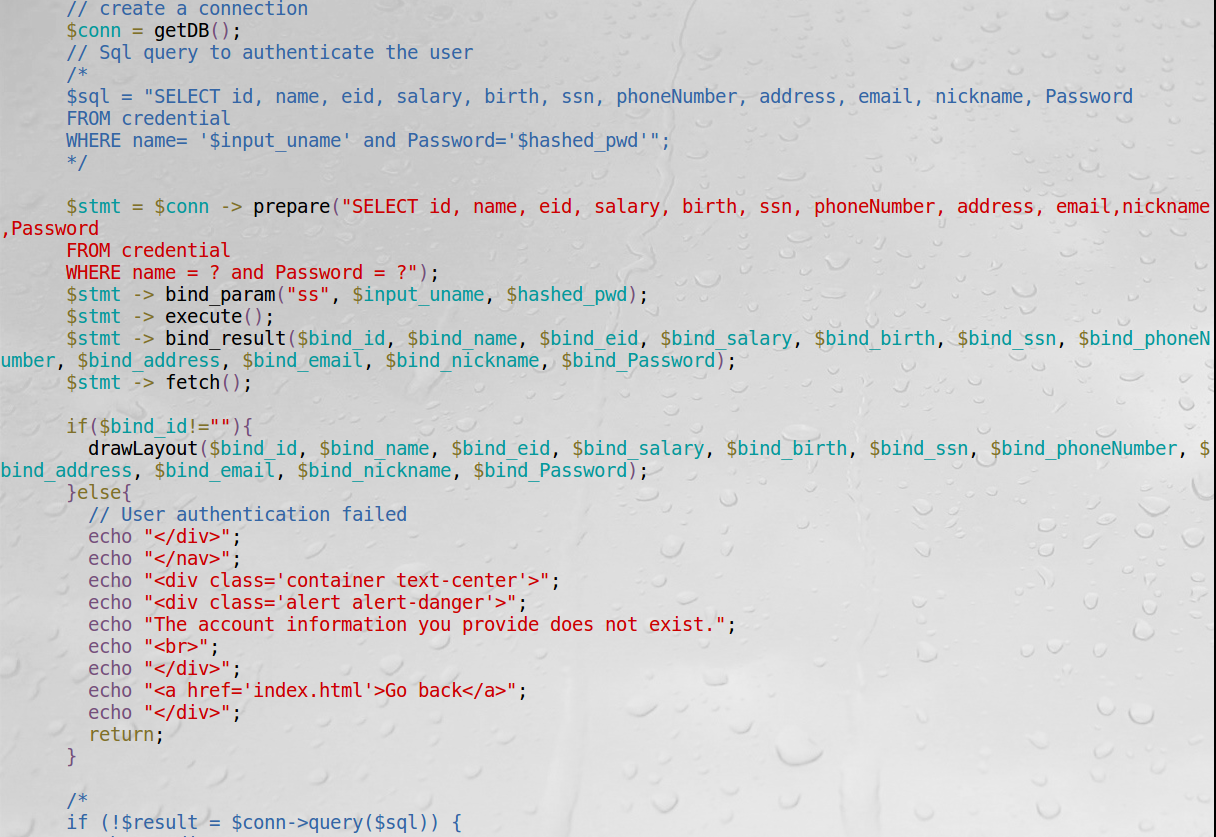
Q5

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.

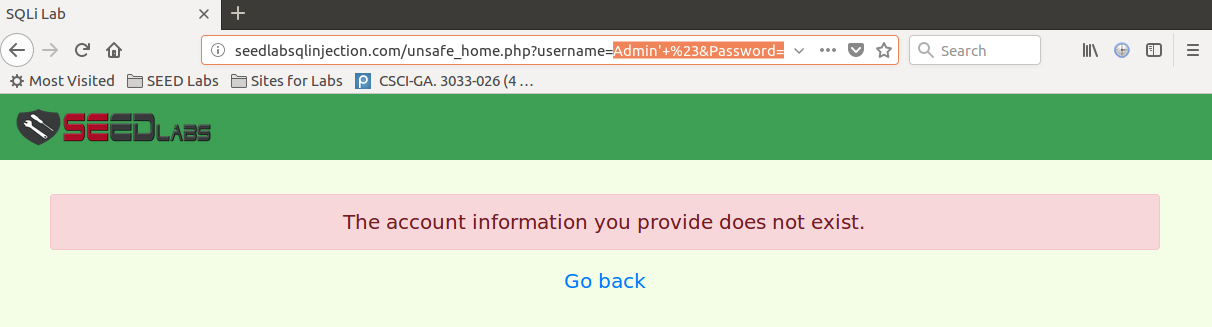
The PHP code in *unsafe\_home.php* before modifying:



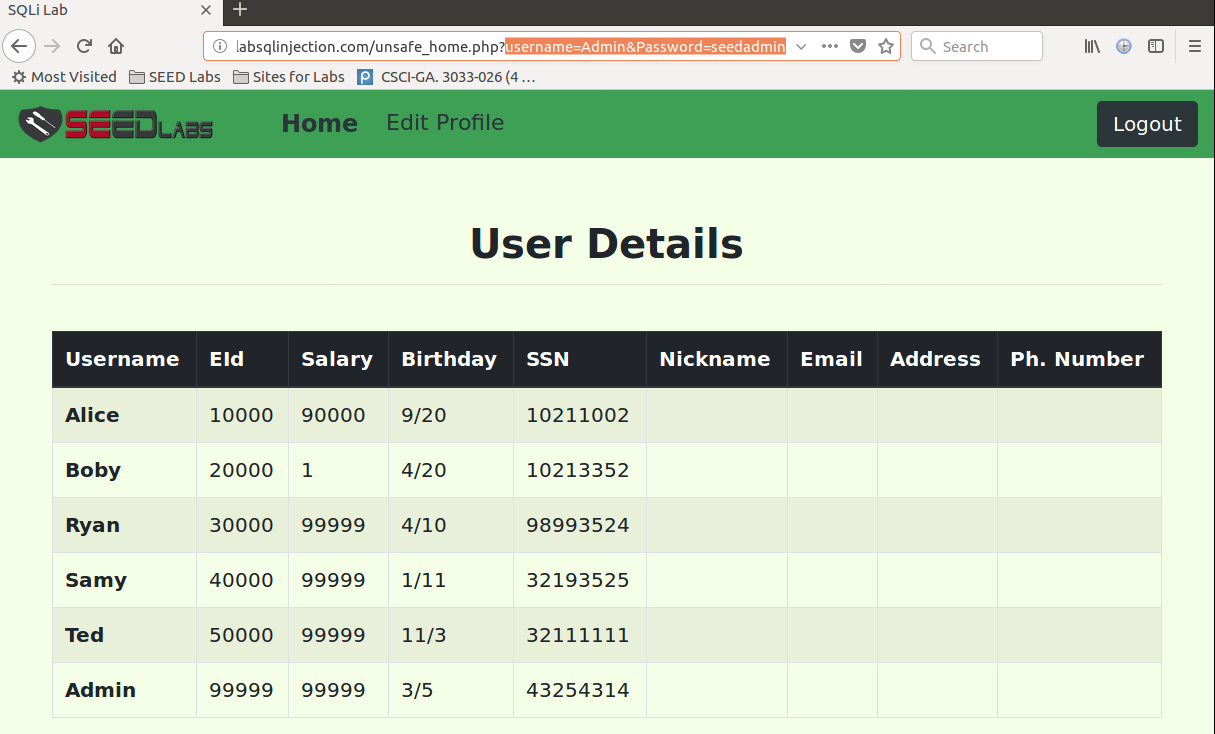
The PHP code in *unsafe\_home.php* after modifying:



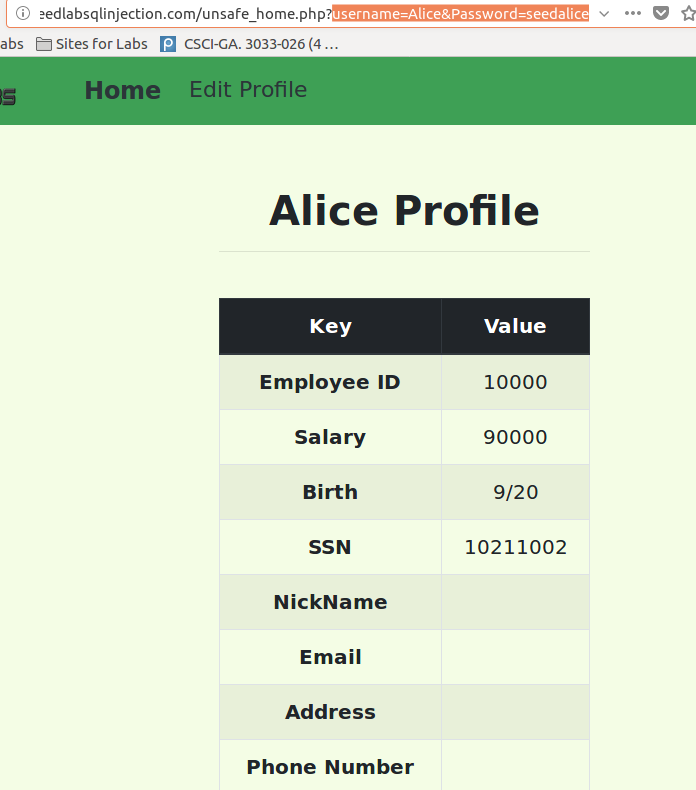
An attack attempt (Same instruction as Task 2.1) failed:



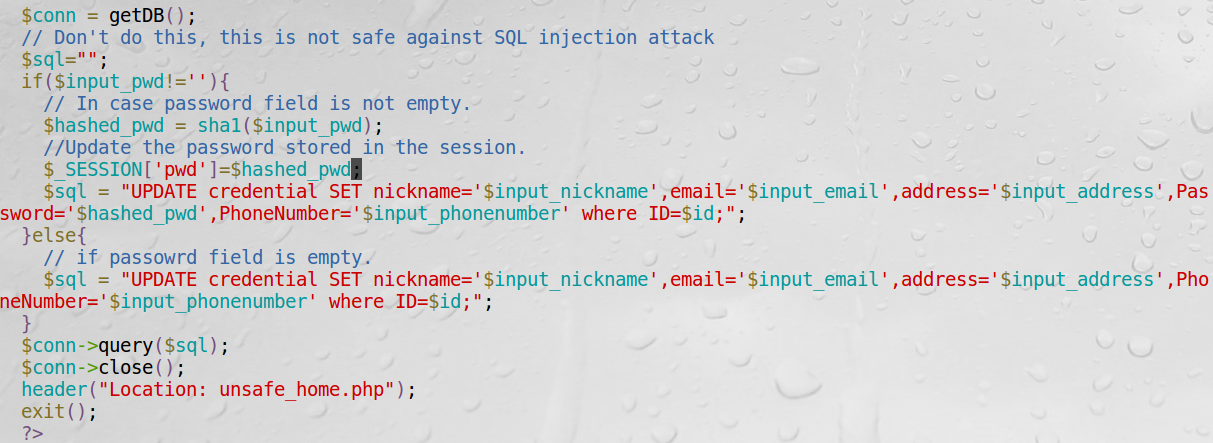
A normal admin login succeeded:



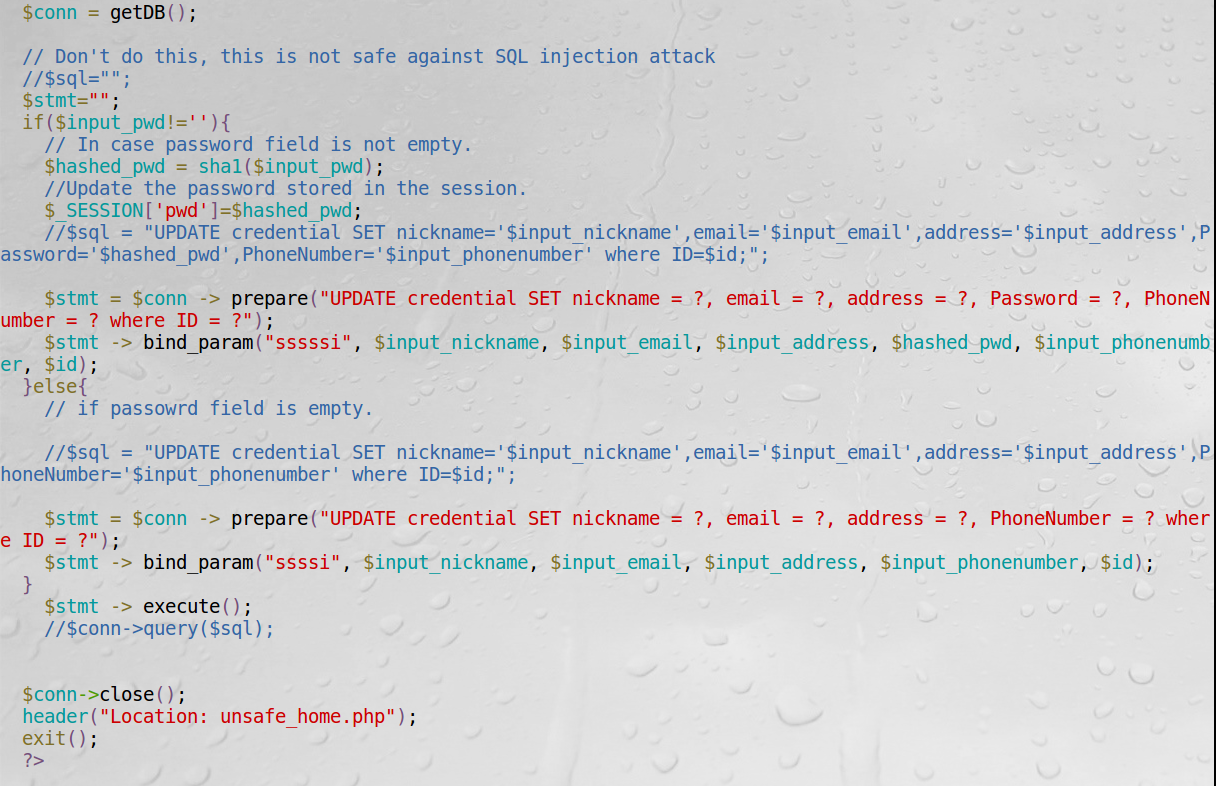
A normal user login succeeded:



The PHP code in *unsafe\_edit\_backend.php* before modifying:



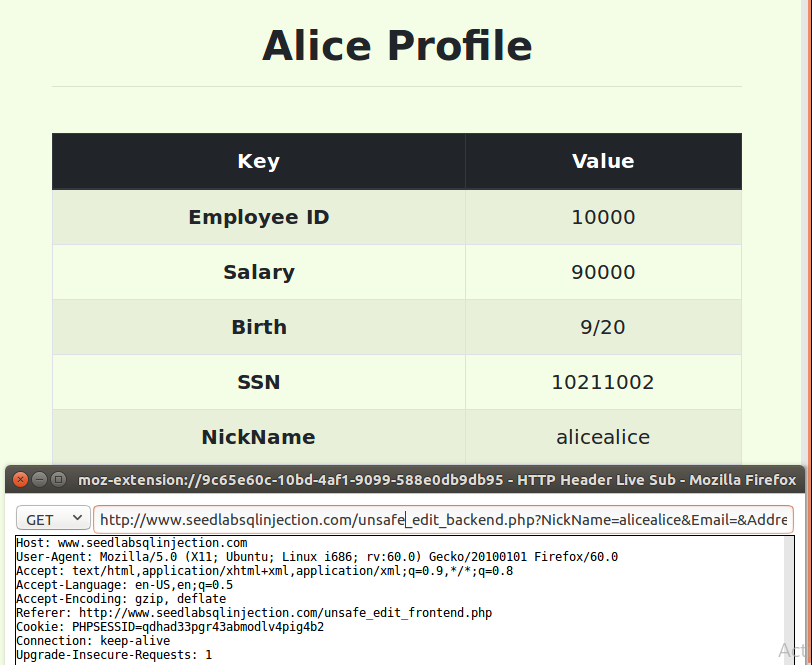
The PHP code in *unsafe\_edit\_backend.php* after modifying:



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



A normal profile update succeeded:



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.