# Web SQL Injection Lab

In this lab we will learn about the vulnerabilities SQL injections can cause towards when launched in an interface talking with the web application and database servers. The users input isn’t validated correctly and this causes the user input to be sent to the database as SQL code, once this reach the database the database will interpret it as pure SQL code and answer the query sent to it, this can cause database leakage, both members personal information as well as the admin details being leaked. We will achieve this in a pre-built environment provided by the seed labs pre packed virtual machine, it contains a website which has an SQL injection vulnerability, we will launch attacks towards this website and try to accomplish an array of mischiefs.

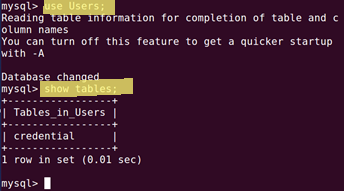
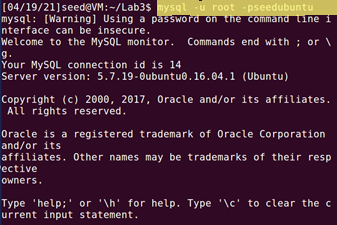
*Task 1*

For the first task we will simply get familiar with the SQL statement, a database exists called Users, and inside the Users database there’s a table called credential which stores personal information of every employee. In the first task we will play around with the database to get familiar with the SQL queries, in the first task we are going to launch 3 queries provided by the lab, followed by a query where we list all the profile information of the employee “Alice”, execution of the last query is up to us.   
The 3 queries given to us are as following:

mysql –u root –pseedubuntu (Logs into the mysql as username: root, password: seedubuntu

use Users; (Tells us to use the database Users)

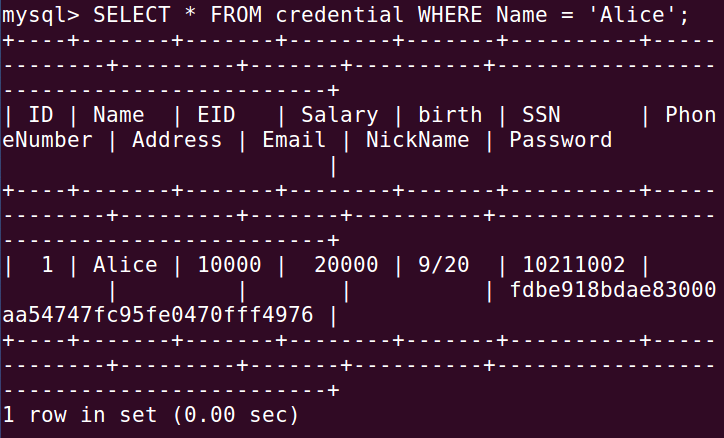
show tables; (Shows the tables present in database Users)



To show all the information of the employee Alice we need to match “name” with someone called “Alice” in the database, we know from the upcoming task (task 2) that name is a genuine entry and it’s also a widely used entry to store names. Knowing this we need to use a very simple SQL query.

SELECT \* FROM credential WHERE Name = ‘Alice’;

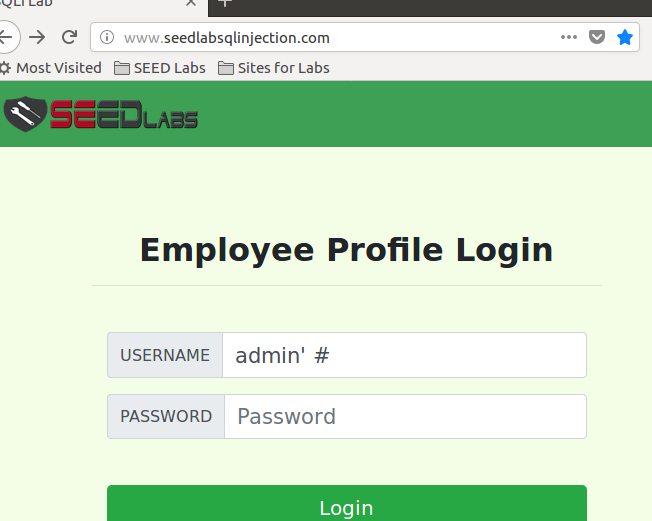
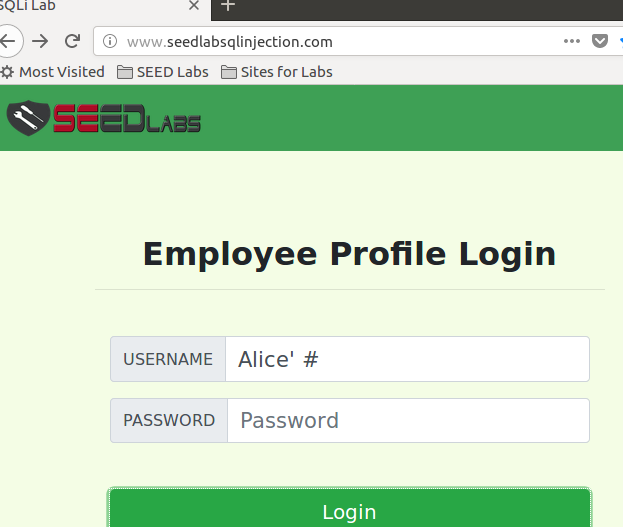
Here we tell the database to select everything from the table credential where Name == Alice. Testing this in our virtual machine gives us the following result:

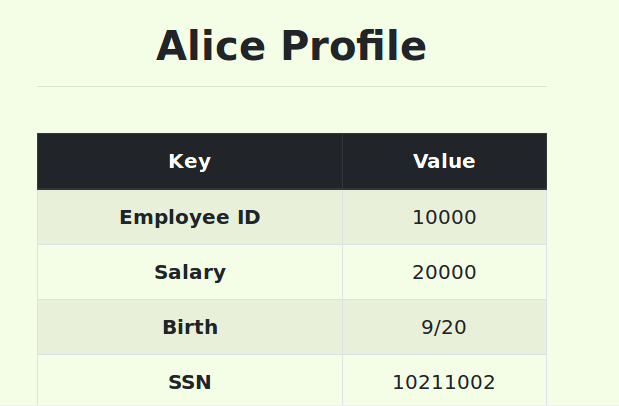
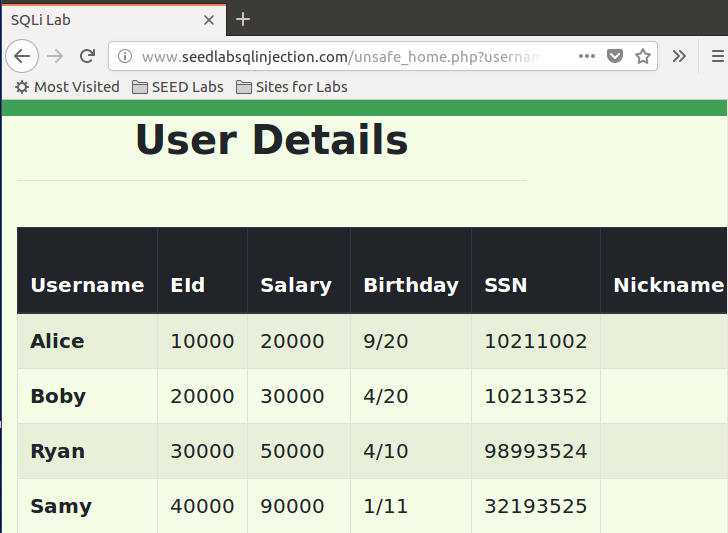
**

Here we can see that Alice’s password is encrypted in a hash.

*Task 2*

In task two we want to launch an SQL injection attack on the select statement, for the first task we are meant to login as an admin through the website ([www.SEEDLabSQLInjection.com](http://www.SEEDLabSQLInjection.com)), we know the admins account name is “admin” but we don’t know the password. So a first step is to enter “admin” in the username window and then something else as the password, to do this I decided to write “admin’ #” in the username field. This will make it interpret the code as end of SQL code and continue on with its execution, accepting whatever I write into the password field. It doesn’t matter what I write into the password field, or anything at all, I can leave it blank, because the database only requires the name to be admin for it to be able to show the entire table, the pound sign tells the SQL to comment out everything after the **admin’** and as we don’t know the password, it would be beneficial to us if the code never matched the username to the provided password. We can login as any user the same way, such as **Alice’ #**



The next part of this task wants us to repeat the previous task, but from the command line using tools such as “curl”. Curl sends HTTP requests, as an example to how curl works we’ve been given the following code snippet which sends a GET request to our web application with two parameters “username” and “Password”.

curl

’www.SeedLabSQLInjection.com/index.php?username=alice&Password=111’

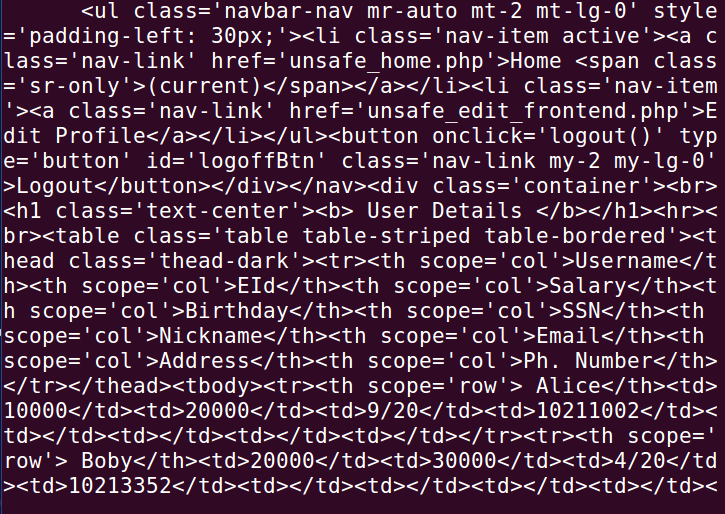
It should be quite straight forward to break into the database using curl as it’s basically repeating what we did on the website, but instead using the terminal. So let’s try to login to admin once more. From the lab description we need to use %20 for white space and %27 for single quotes (which we used on the website), however we want to use the pound sign as well, I found this thread which explains how to use Pound sign in curl: [https://stackoverflow.com/questions/27842462/getting-a-file-with-hash-in-path-by-curl](https://stackoverflow.com/questions/27842462/getting-a-file-with-hash-in-path-by-curl%20%20)

Knowing this we can conduct our string, from the lab description we know the php code which handles login attempts is called “unsafe\_home.php”, appending this to the website we get

**curl ‘http://www.SeedLabSQLInjection.com/unsafe\_home.php?username=admin%27%20%23&Password='**

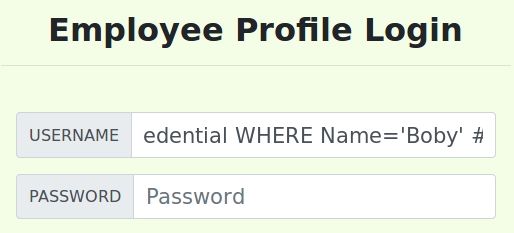
Just as with the previous task we leave Password empty as there’s no need to enter it as long as we provide the pound sign.

This prints us the whole HTML code but in the HTML code we can find the credentials of the employees.

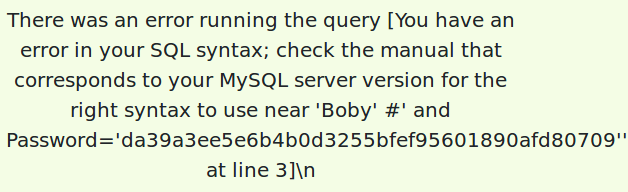


For the next part, task 2.3 we are going to append a new SQL statement, instead of just stealing information we want to modify the database and the idea is to use the same technique as previously but turn one SQL statement into two, second one being either update or delete. We are going to attempt to delete a record from the database. My reasoning behind this task is that we can use the previous query admin’ #, but adding more queries before the pound sign (which means comment out everything behind #). As noted by the lab description we use a semicolon (;) to separate two SQL statements, so far we haven’t used any statements when trying to login. However appending as mentioned should work, hence something like admin’ SELECT \* FROM credential WHERE name=’Alice’ should work. We want to delete or update something meaning we should use “UPDATE” or “DELETE”. Trying to delete something I’m using this string:

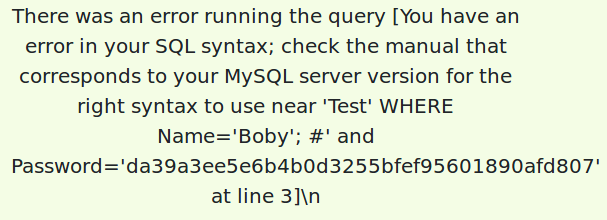
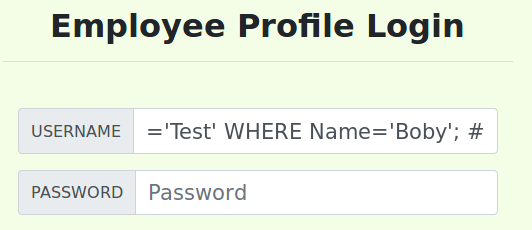
**admin’; DELETE FROM credential WHERE Name=’Boby’; #**

****

This however throws an error. We will attempt the same but instead try to update.

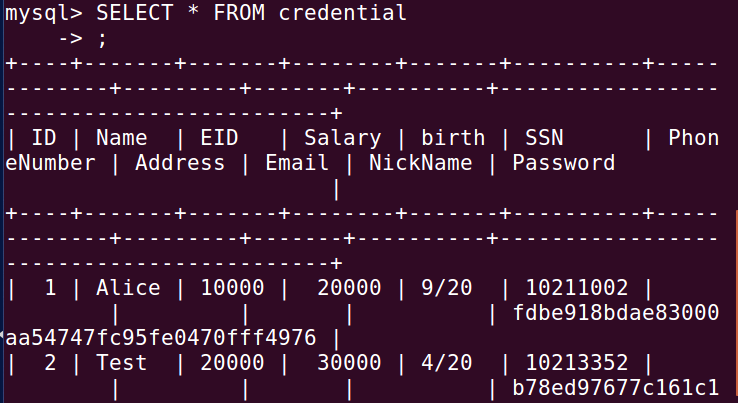
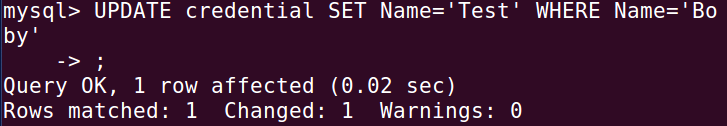


**admin’; UPDATE credential SET Name=’Test’ WHERE Name=’Boby’; #**

****

When we’re trying to update we want to update something in the credential table, we set our new name to Test and want to update the value where Name = Boby.

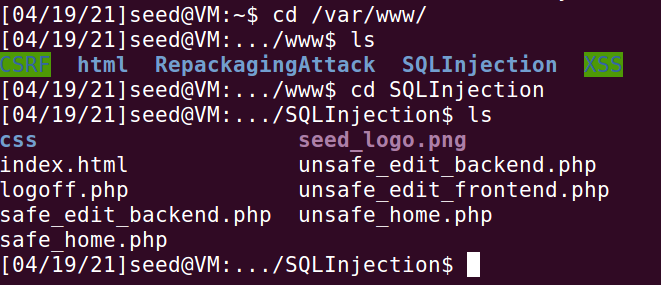
Both of those queries however give us errors, but when we launch those queries in the terminal it seems to be working as it should:

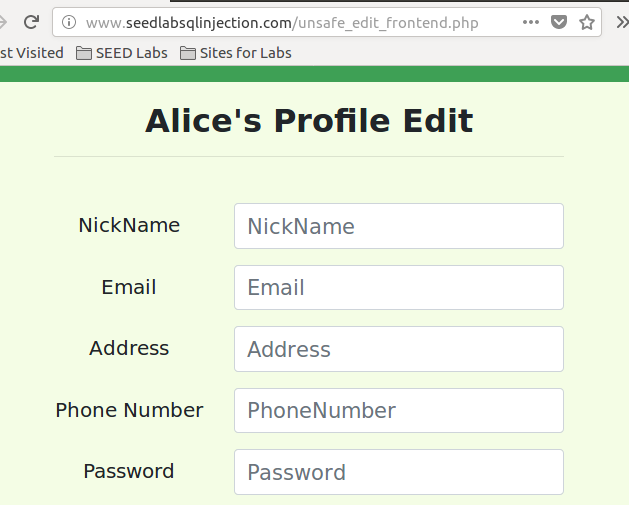
****

This indicates that it has something to do with the php and not the actual query code, I can’t seem to either update or delete a record through the website using multiple SQL statements. As we know from the lab video this is because of PHPs mysqli extensions which doesn’t support multiple queries.

*Task 3*

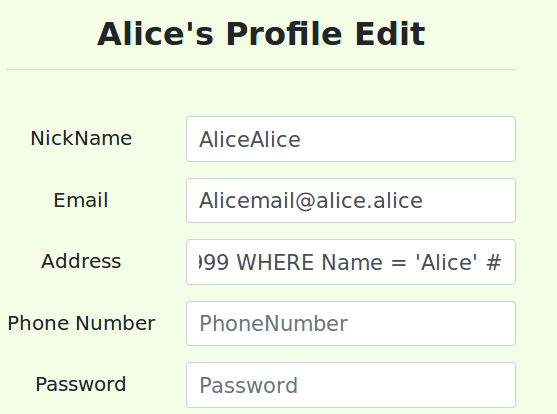
In task 3 we first want to modify your own salary, in this case we are the employee “Alice” who is mad at her boss, and wants to change her own salary, then her boss’s salary to 1, and lastly change the bosses password to something else. As we’ve seen in task 2 we can login as Alice already, but we can’t modify anything from her profile, and we can’t run two statements in the same line, therefore it would be wise to login as Alice, and then change her salary from her profile. As noticed by the lab description we want to exploit the vulnerability found in the Edit-Profile page, which has its’ PHP code implemented in “unsafe\_edit\_backend.php”.

From the lab description we know that unsafe\_edit\_backend.php is stored under /var/www/SQLInjection, and after trying to reach the backend website to no avail I checked the directory contents and found this:

We can make the conclusion that the edit profile page is stored in unsafe\_edit\_frontend.php, so we’ll login as Alice and navigate to this page by appending it to the URL. Which brings us to this page for Alice.

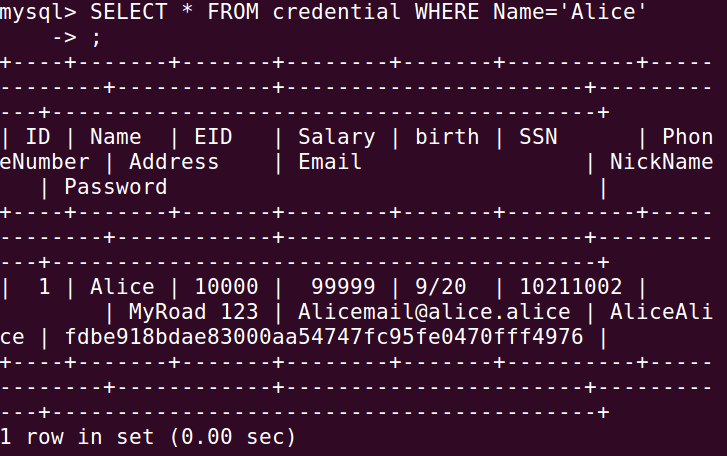
In here we know there exists an SQL vulnerability, and we want to update Alice’s salary, so logically we want to update that field, from previous task we already have a similar string.

I’m using this string in the address field **MyRoad 123', salary = 99999 WHERE Name = 'Alice' #** to set her salary to 99999, while also changing the road name to MyRoad 123.

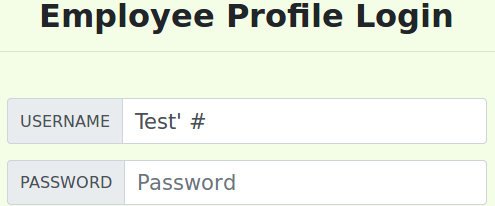


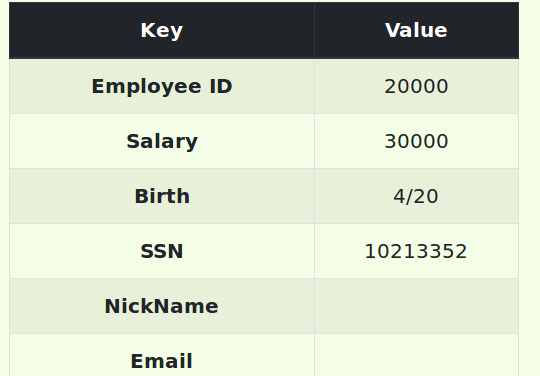
When we hit save...



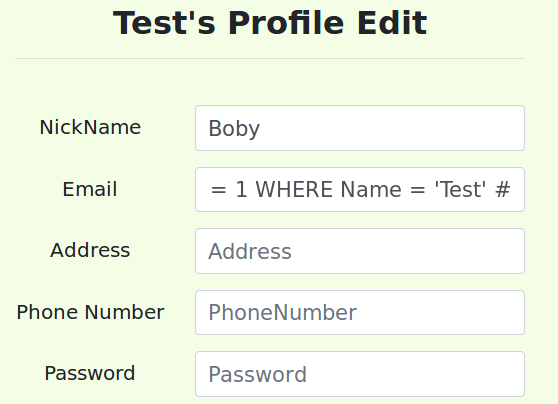
Double checking in the terminal we can see that her information is updated correctly, 

For the next task we are going to change the boss’s salary to 1, I actually changed the bosses name from “Boby” to “Test” earlier on in the task, hence his name is nowadays “Test” instead of “Boby”. We already know how to login as an admin, we know how to login as Alice, therefore we can easily login as Boby as well by using the exactly same method as for task 3.1 (But of course using Test as name instead of Boby!)

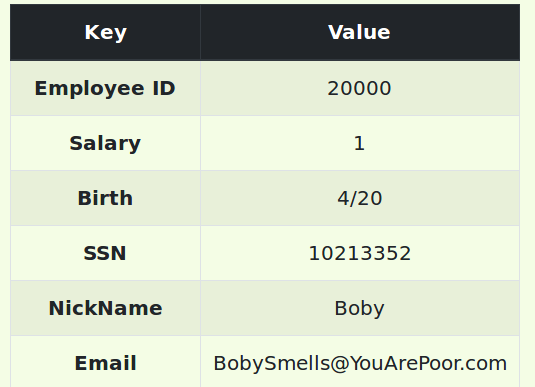




Appending the unsafe\_edit\_frontend to edit his information...

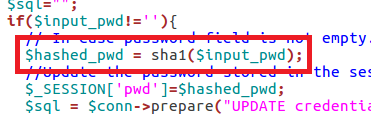


Using the SQL string **BobySmells@YouArePoor.com', salary = 1 WHERE Name = 'Test' #** in the Email field.

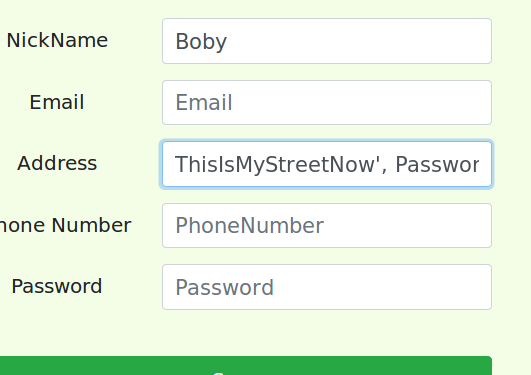
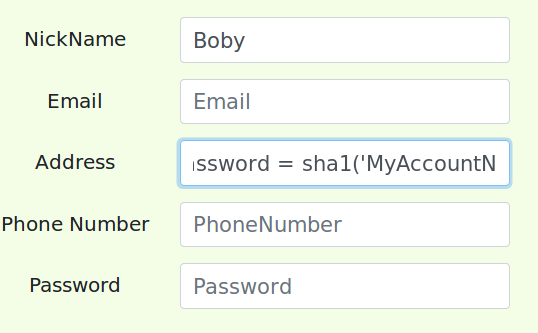
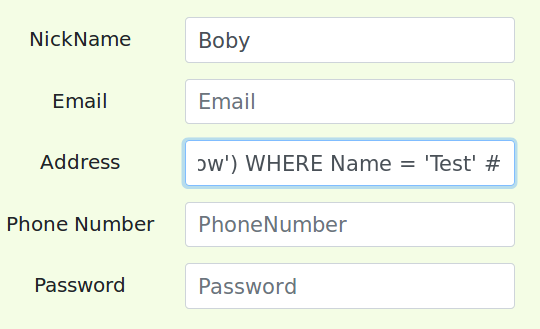


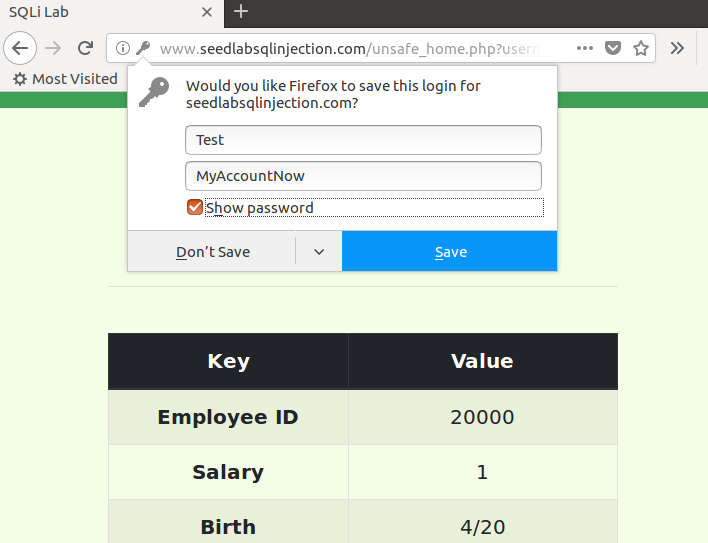
For task 3.3 we want to modify Boby’s password, we first need to change his password, then prove that we’re able to login as Boby (Without using the previous Test’ # method). The database store the hash values of the password instead of the plain password, we can look into the unsafe\_edit\_backend.php code to see how the password is stored, it’s using a SHA1 hash to generate the hash value, this means we can reverse hash the password in the end. I am going to use the same method as in task 3.1 and 3.2 for this task but change the password instead of the salary.

From the unsafe\_edit\_backend.php we can see that we need to change a field called “sha1”, and not the actual “password” field.

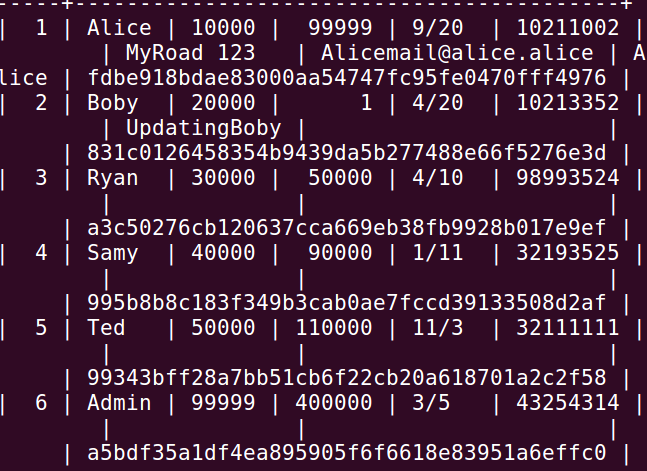
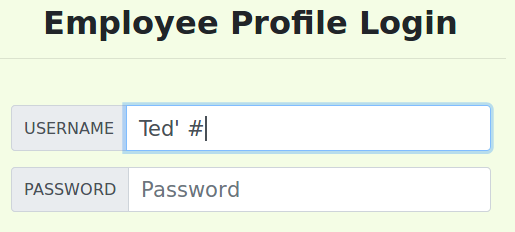
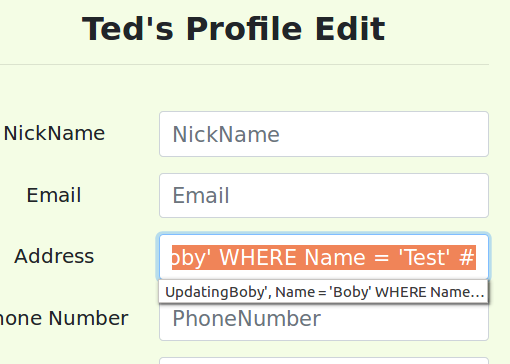


Putting this into works we conduct something as in the previous tasks, **ThisIsMyStreetNow’, Password = sha1(‘MyAccountNow’) WHERE Name = 'Test' #** as noted by the backend field we see that we need to place the new password inside parenthesis.

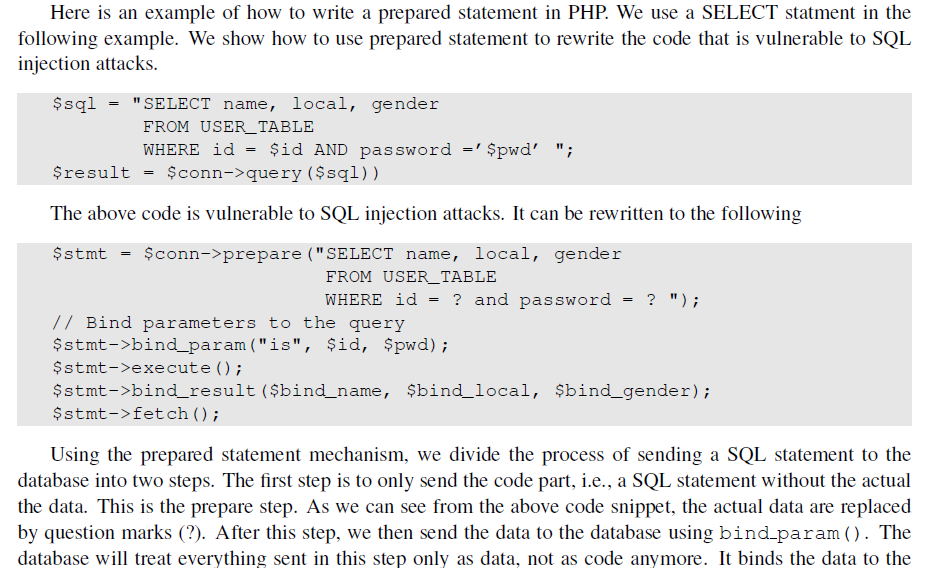


We can now successfully login as the user “Test” with password “MyAccountNow”. As one final thing I’ll change back Boby’s name from “Test” to “Boby”, I can do this through the website as well as any user, I’ll be logging in as Ted. Using the SQL query **UpdatingBoby', Name = 'Boby' WHERE Name = 'Test' #** in the address field.

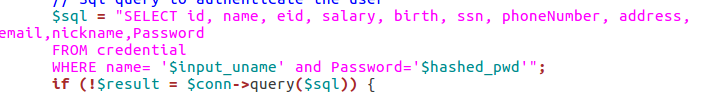


*Task 4*

In the last task we are going to find some sort of counter measurement to those vulnerabilities. Most of those vulnerabilities arise because the code isn’t correctly separated from the data, the PHP knows the difference but the database doesn’t, according to the lab description the most secure way is to use prepared statement. For this task we are going to create those prepared statements for our previous SQL exploits, we will re-write the code in the unsafe files and make them safer, first up is unsafe\_home.php. We get an example from the lab description which is as follows;

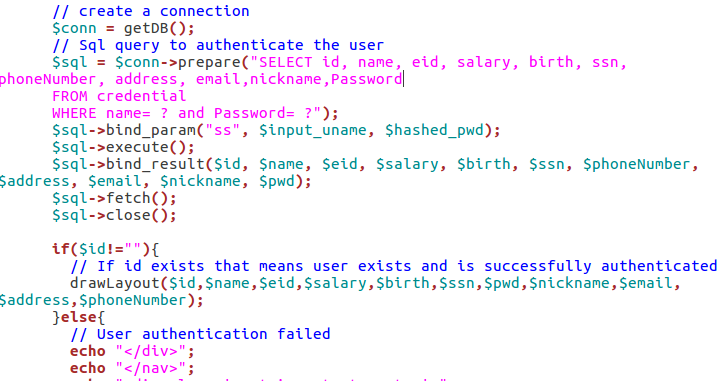
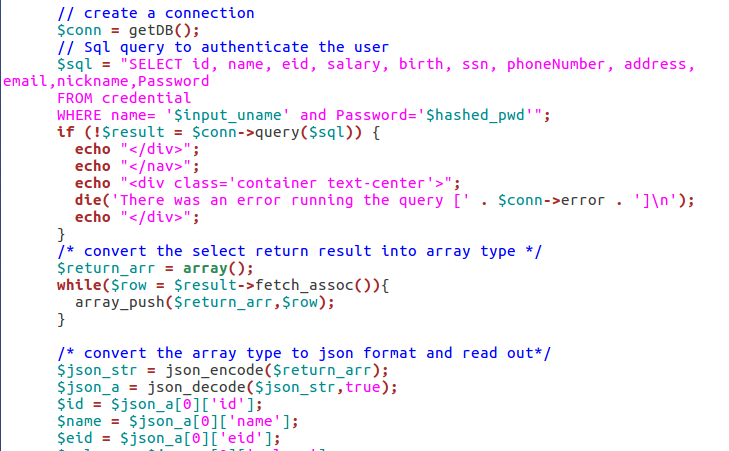


From this information we should be able to setup prepared statements in the current code. We can see such a flaw mentioned above here.

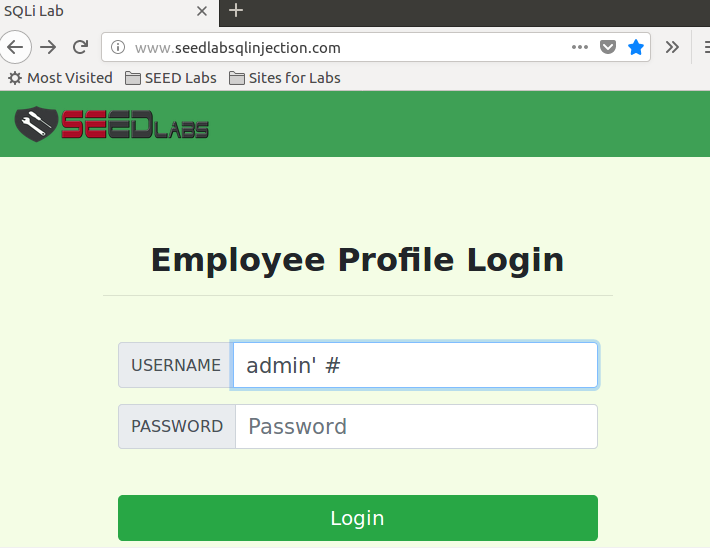


Luckily we have the “safe” versions provided to us by the lab, instead of being called “unsafe\_xxx” they are called “safe\_xxx”. For instance our unsafe home is called “unsafe\_home.php” while the safe implementation is stored in “safe\_home.php”, from here we can see the difference between the two versions.

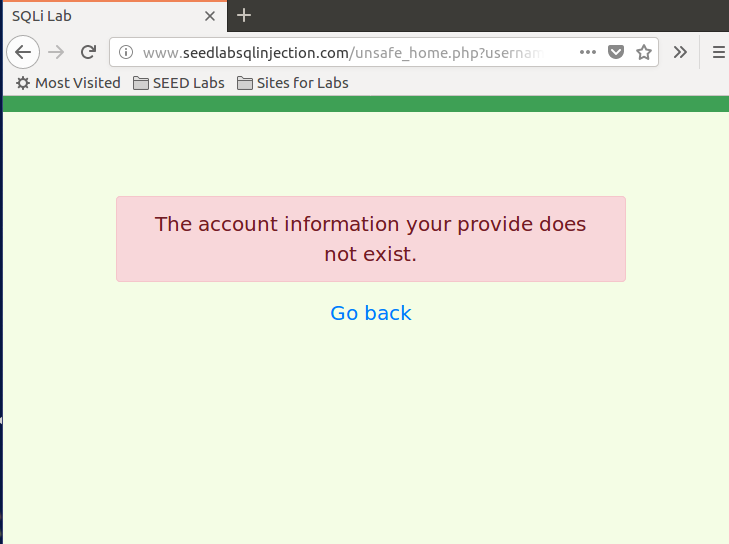
Safe version Unsafe version

I replace the unsafe version with the safe version and try our first attack once more.



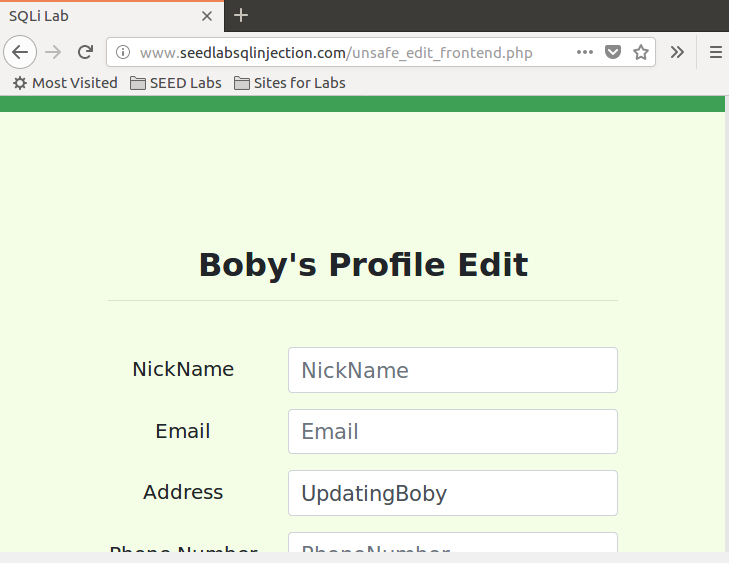
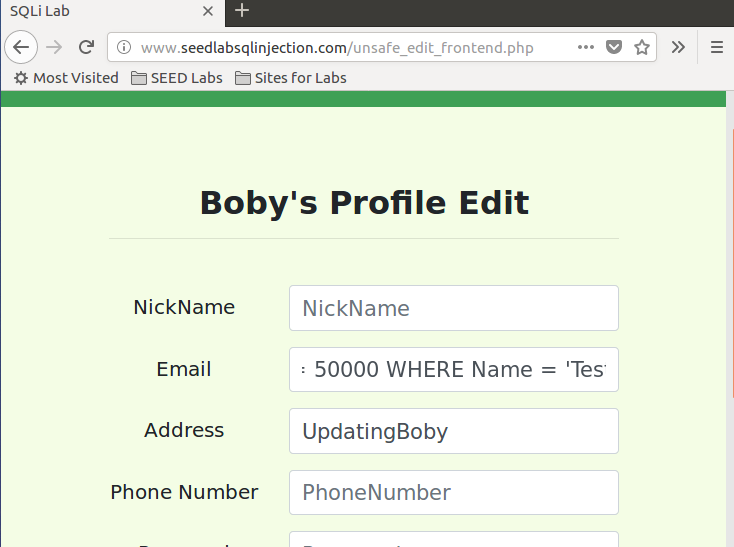
Upon clicking “Login” we now get an error instead,



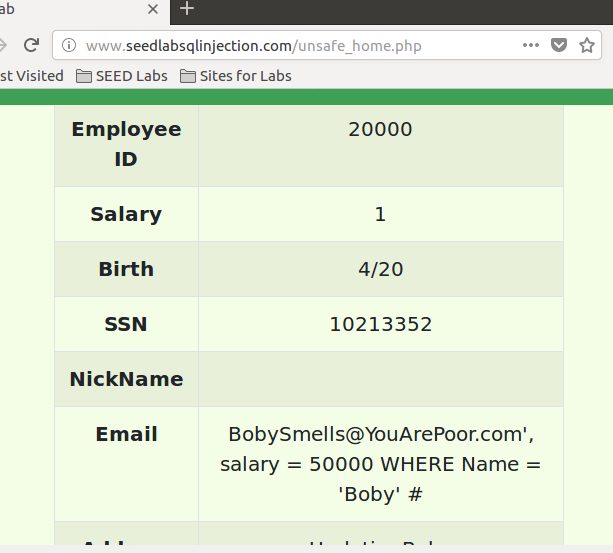
We will do the same for all the other unsafe files and replace them by the code found within their “safe” counterparts. We still know the login details for Boby so we will use them to login, and let’s try to change his salary again. Note that the frontend hasn’t been changed, but the unsafe\_edit\_backend.php has been changed which should remedy our attacks on the frontend.

User: Boby

Password: MyAccountNow

Using the query **BobyDoesntSmells@IAmNotPoor.com', salary = 50000 WHERE Name = 'Boby' #** once more, but using his actual name (as we changed that at the end of task 3) and trying to change his email to something nicer, and a more respectable salary.



Saving this now instead of executing the SQL query stores it as a full string in his email address, I guess his salary won’t change and his email is doomed to look strange.

In a “safe” frontend some kind of string validation would be in order to block such entries, such as limiting the specific fields to specific characters, for instance the email field would accept some text, an @, some more text and finally a domain name, such as “Boby@email.com”.

After those safe edits have been applied I can’t seem to execute any previous attack I’ve attempted, so as far as I can see the prepared statements remedy those problems.