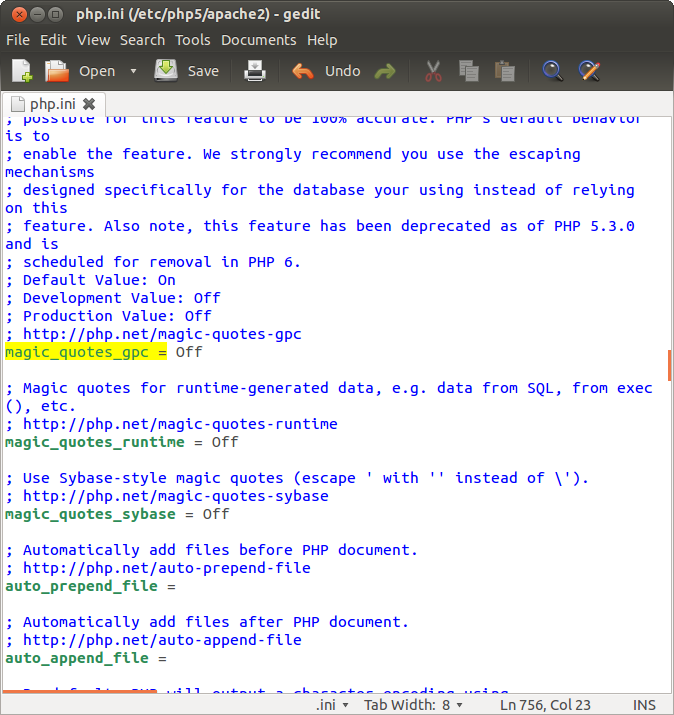
**SQL Injection Attack Lab Report**

**Setup:**

****

**Observations and Explanation:**

1. Turn off magic\_quotes\_gpc and restart the apache server.

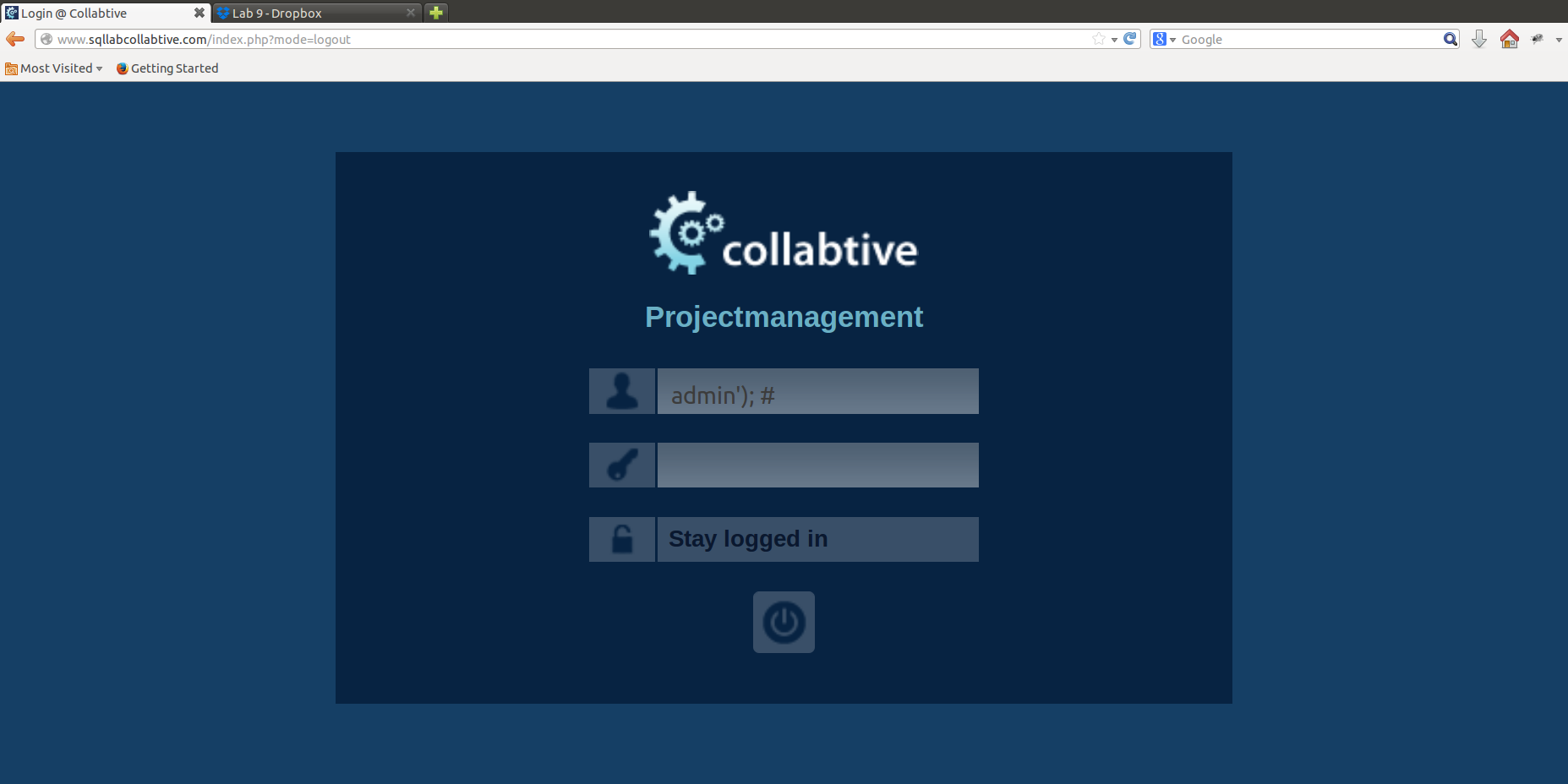
**Task 1:  
**

Figure 1.1

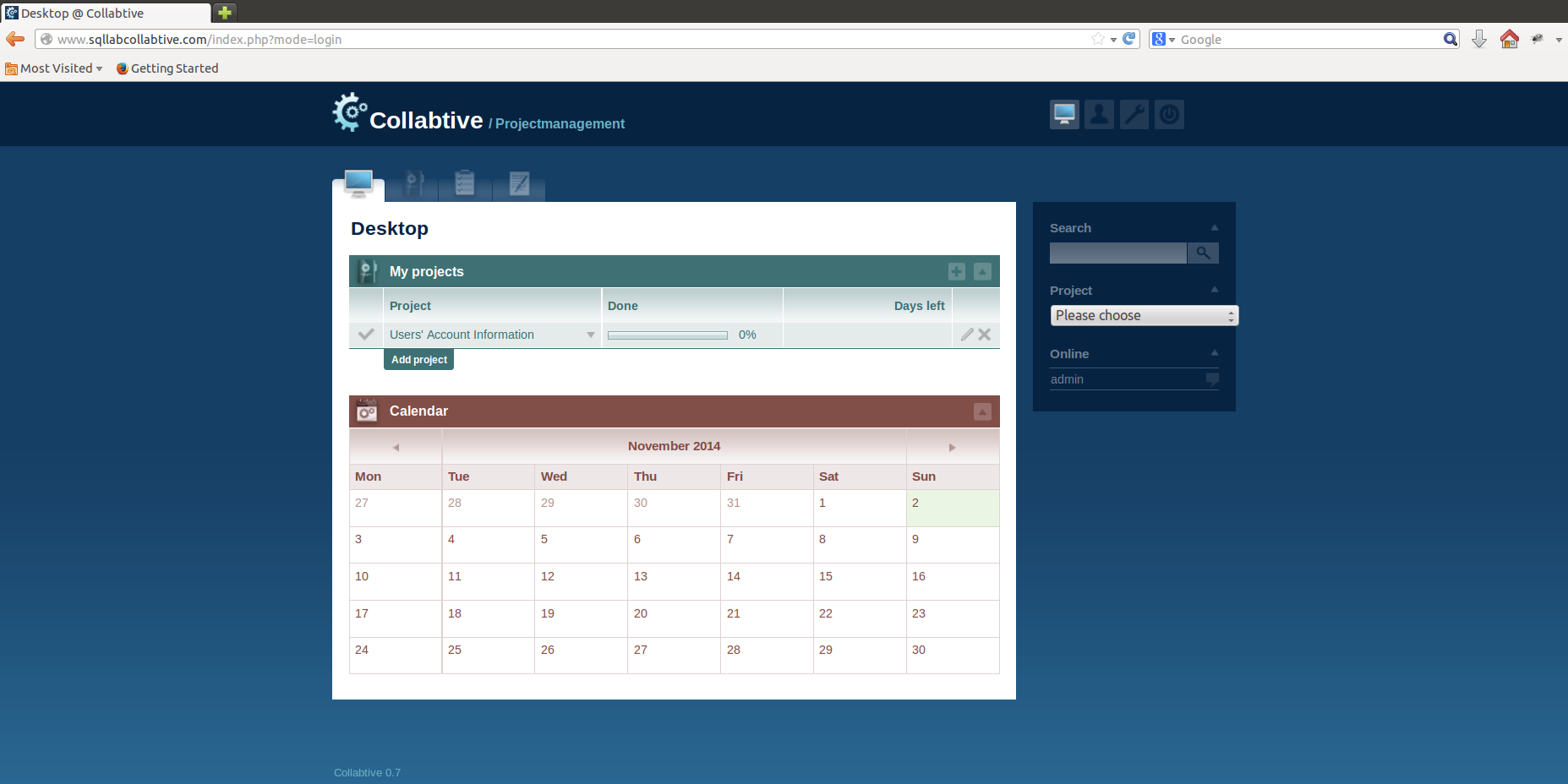


Figure 1.2

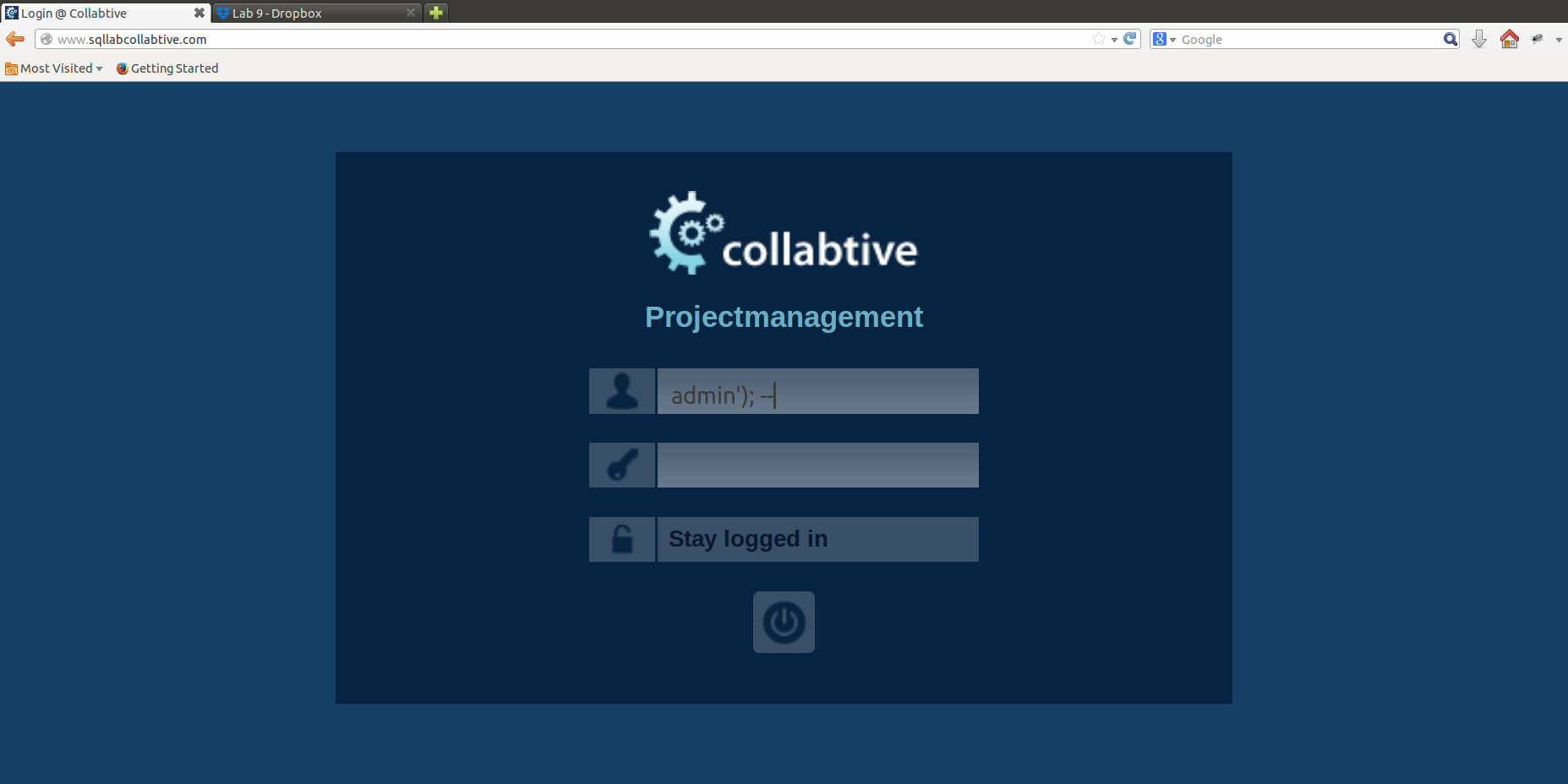


Figure 1.3

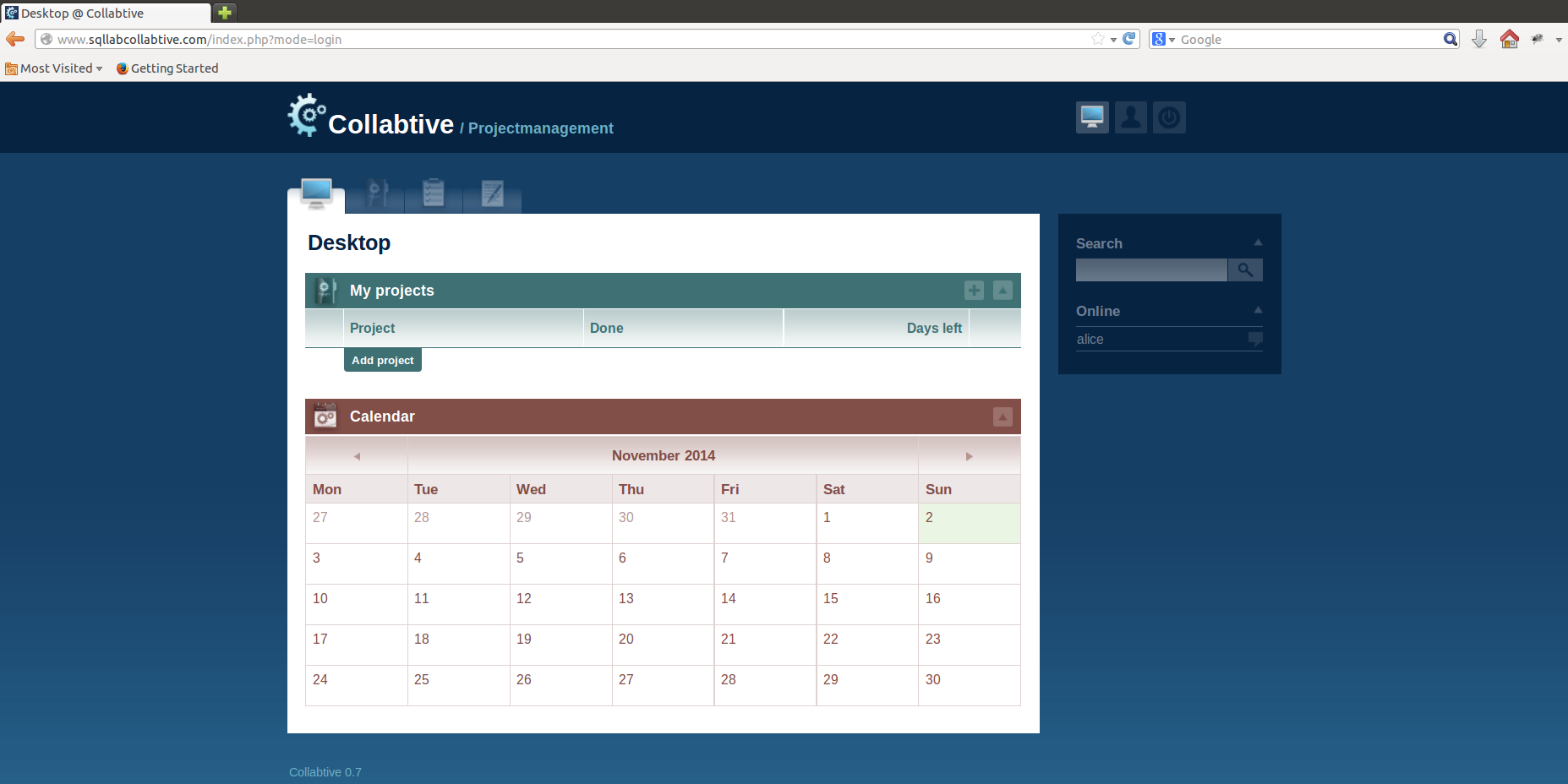


Figure 1.4

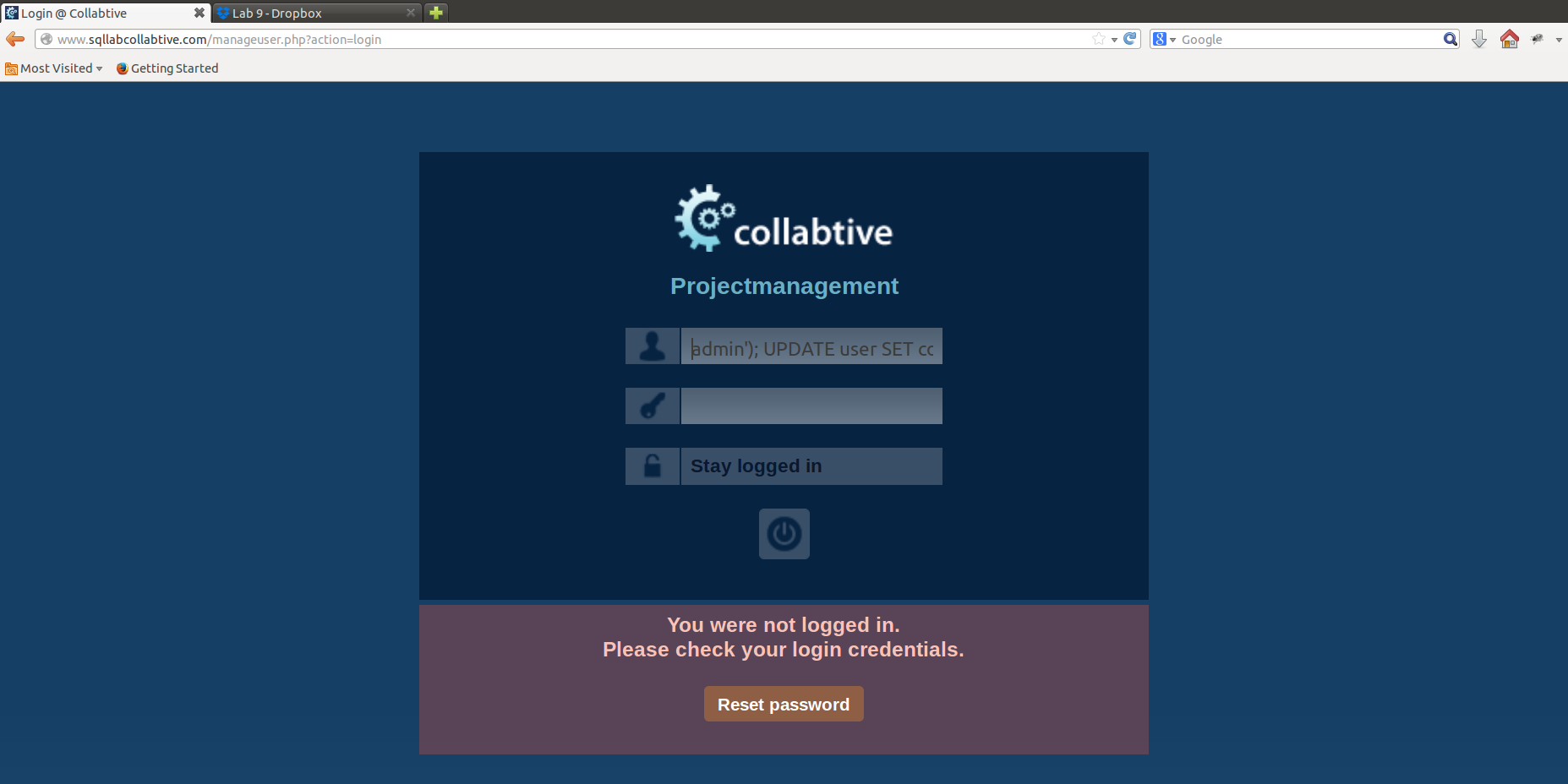


Figure 1.5

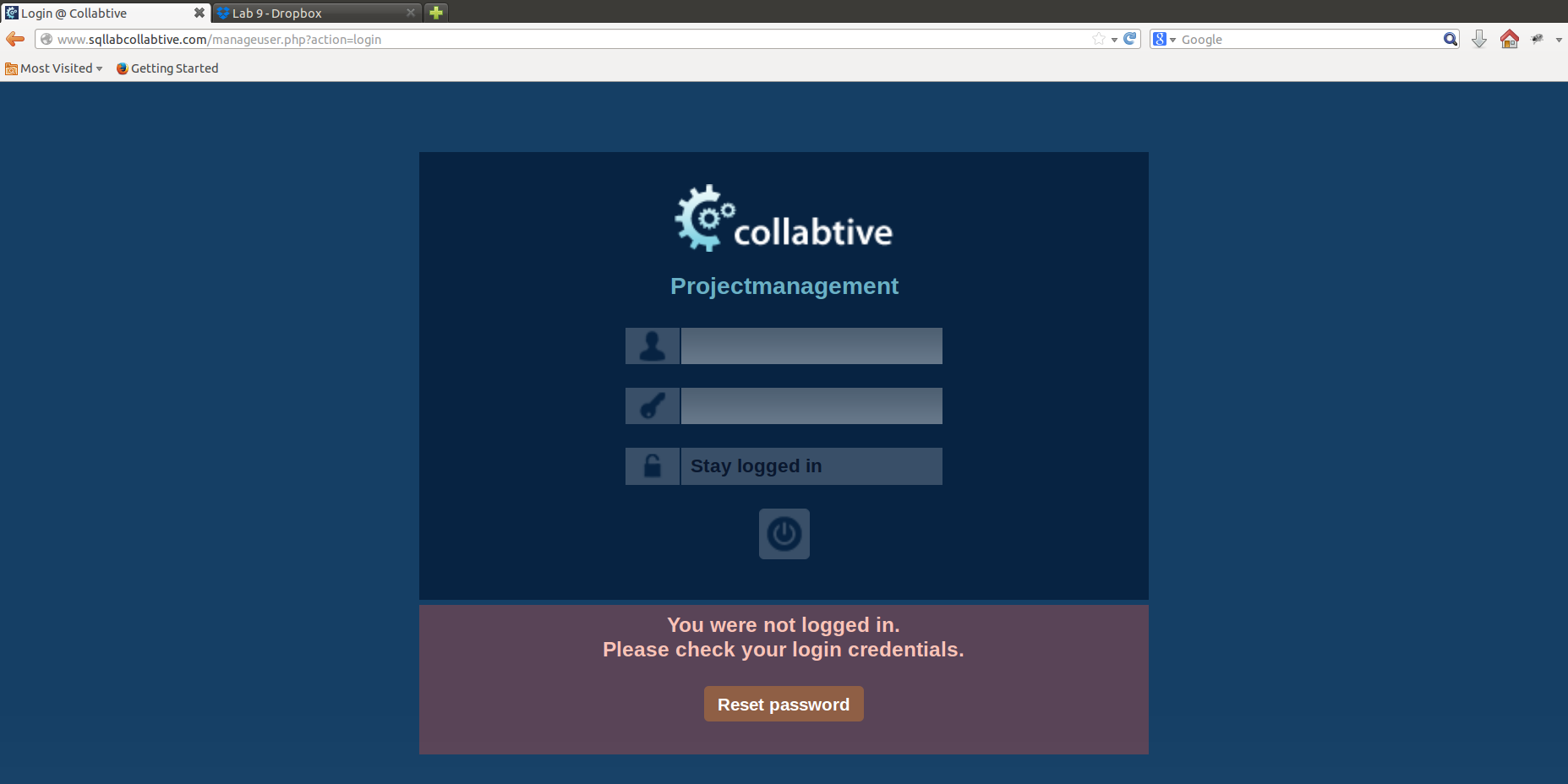


Figure 1.6

**Observations and Explanation:**

1. To log into the admin account without knowing the password, we take advantage of the SQL injection vulnerability. The code has the ***mysql\_real\_escape\_string($user)*** function commented, so no input sanitization is done, which allows us to inject code.

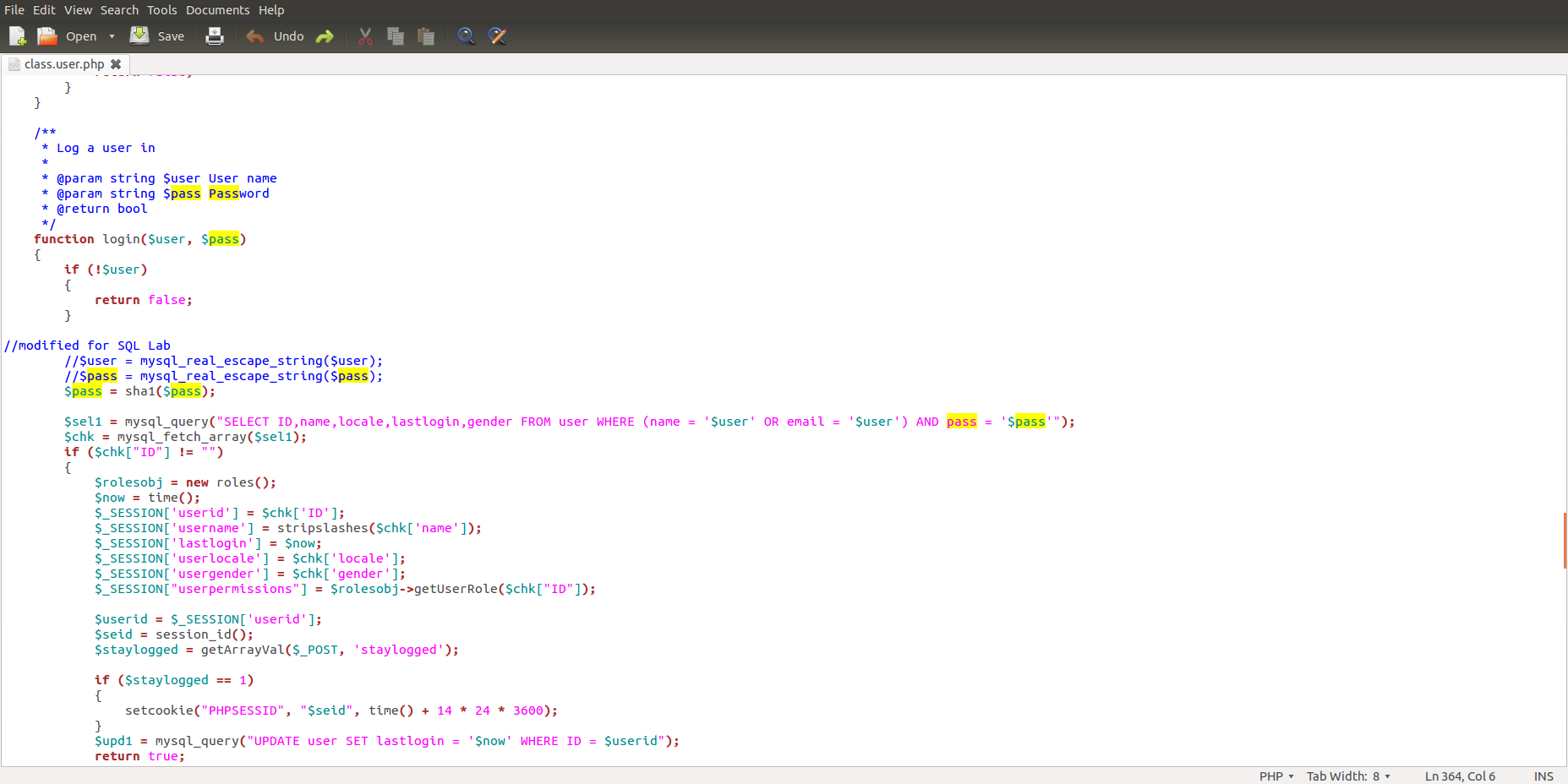
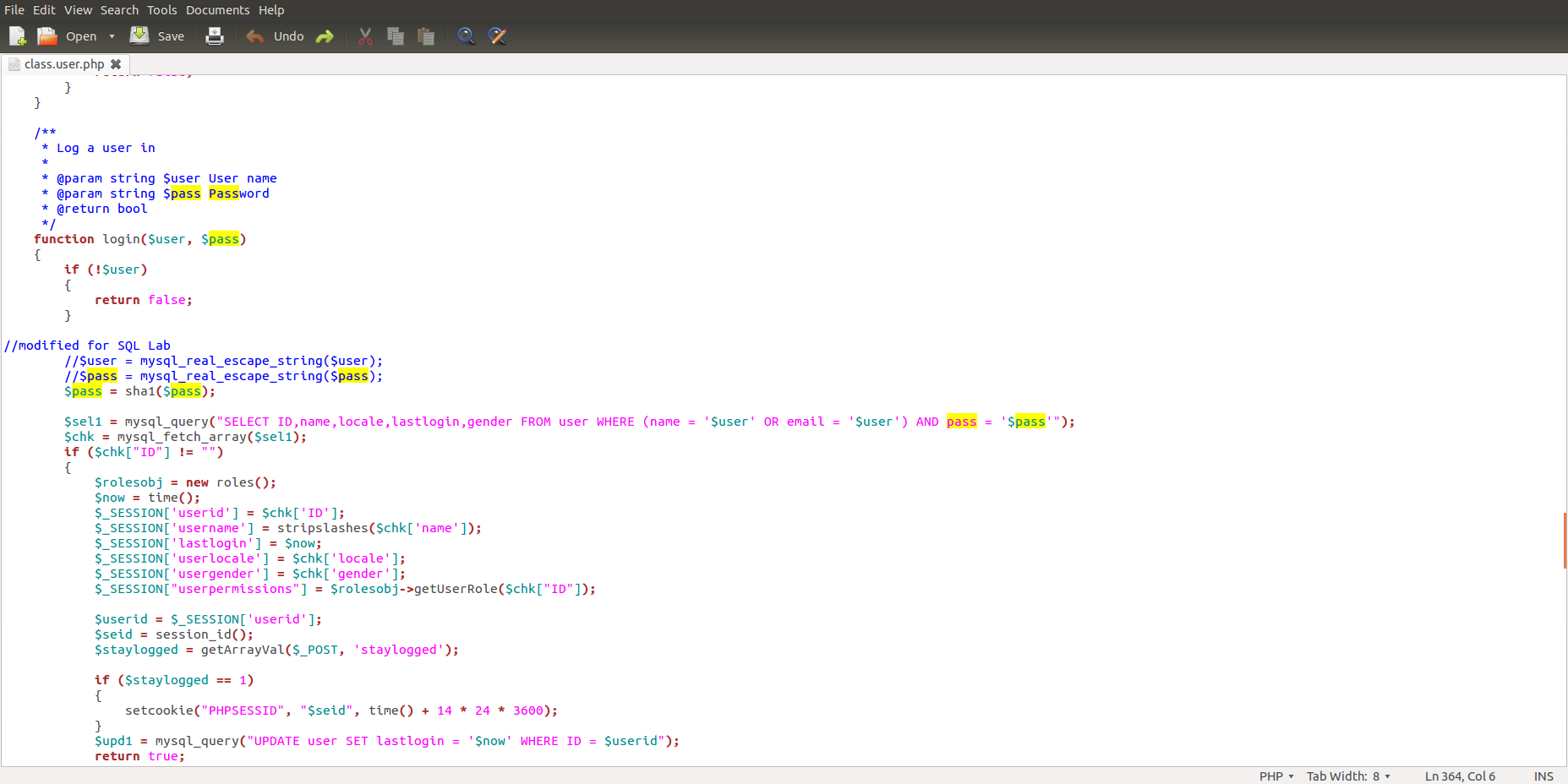
****

Figure 1.7

1. We use this line of SQL code to the username field to get access to the account, without knowing the password.

SQL injection code,

**admin’); --**



We add the **‘** to end the input to the name field, and then add “**)**” to end the WHERE clauses open bracket, we then supply the delimiter **; AND** then comment out the rest of the SQL query, by the use of  **“#”** or “ **--** ”, for – the trailing space is necessary, or it will not work.

**#** is used for single line comment in SQL server. We can use  **-- or #**  ro perform the attack.

1. On trying to login to the account as well as update the account, the attempt fails, I tried the following variations,

* admin'); UPDATE user SET company='task1.2' where id=4; --
* admin’); UPDATE user SET pass='7sa7afbcn78as8bp332o9b322bp923b' WHERE ID = 3; --
* admin'); UPDATE user SET company='task1.2' where id=1; #

each of the attempts resulted in the same output as Figure 1.6.

By using the delimiter ;, we can send 2 SQL queries to the server, and expect both to be executed, but when we see the code we see that the query is executed using the **mysql\_query**(), which doesn’t allow multiple queries to be sent to the server. Query Stacking is not allowed in the mysql\_query() function. This causes the attack to fail when MYSQL executes the functions.

**Task 2:**

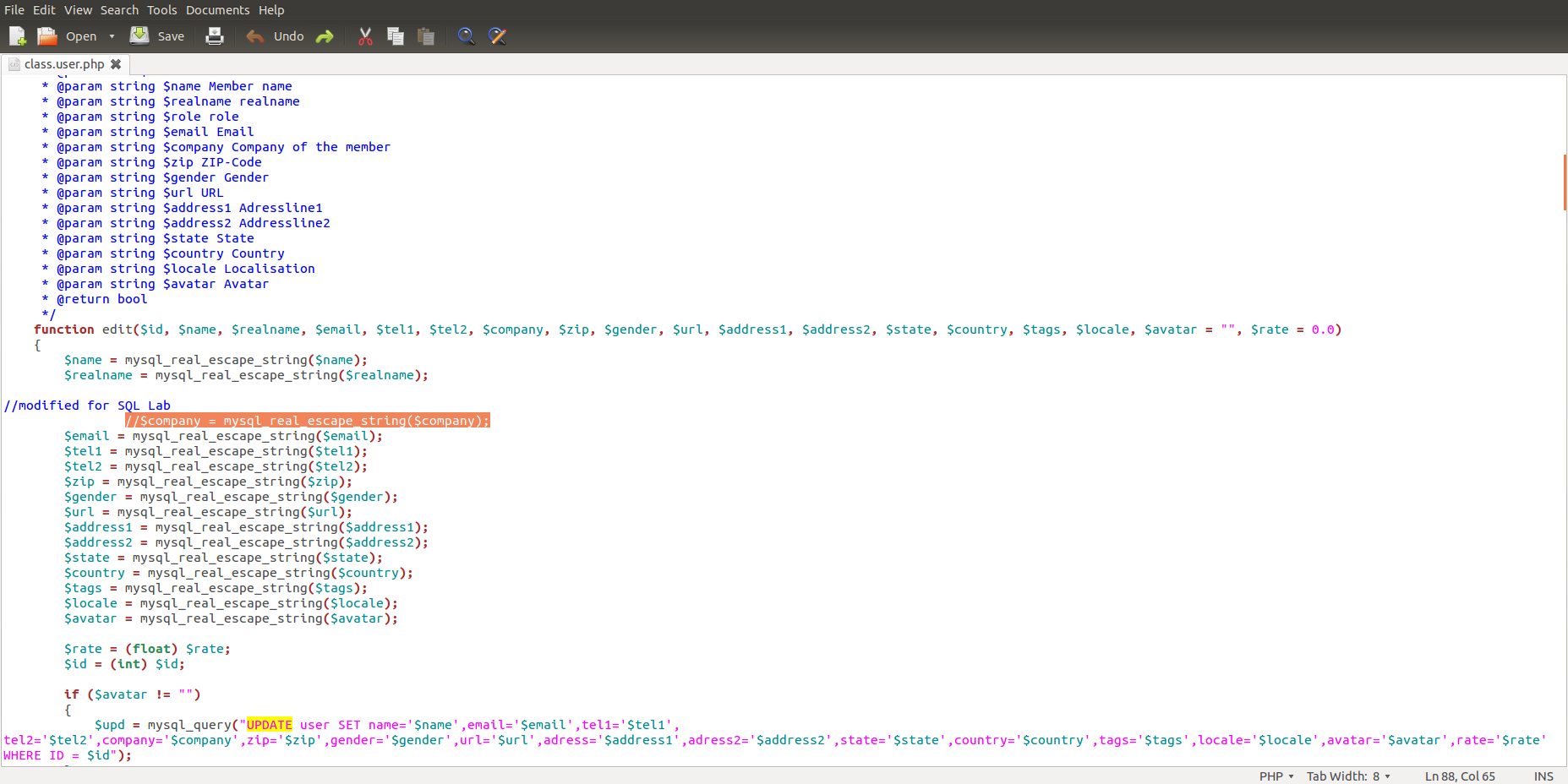


Figure 2.1

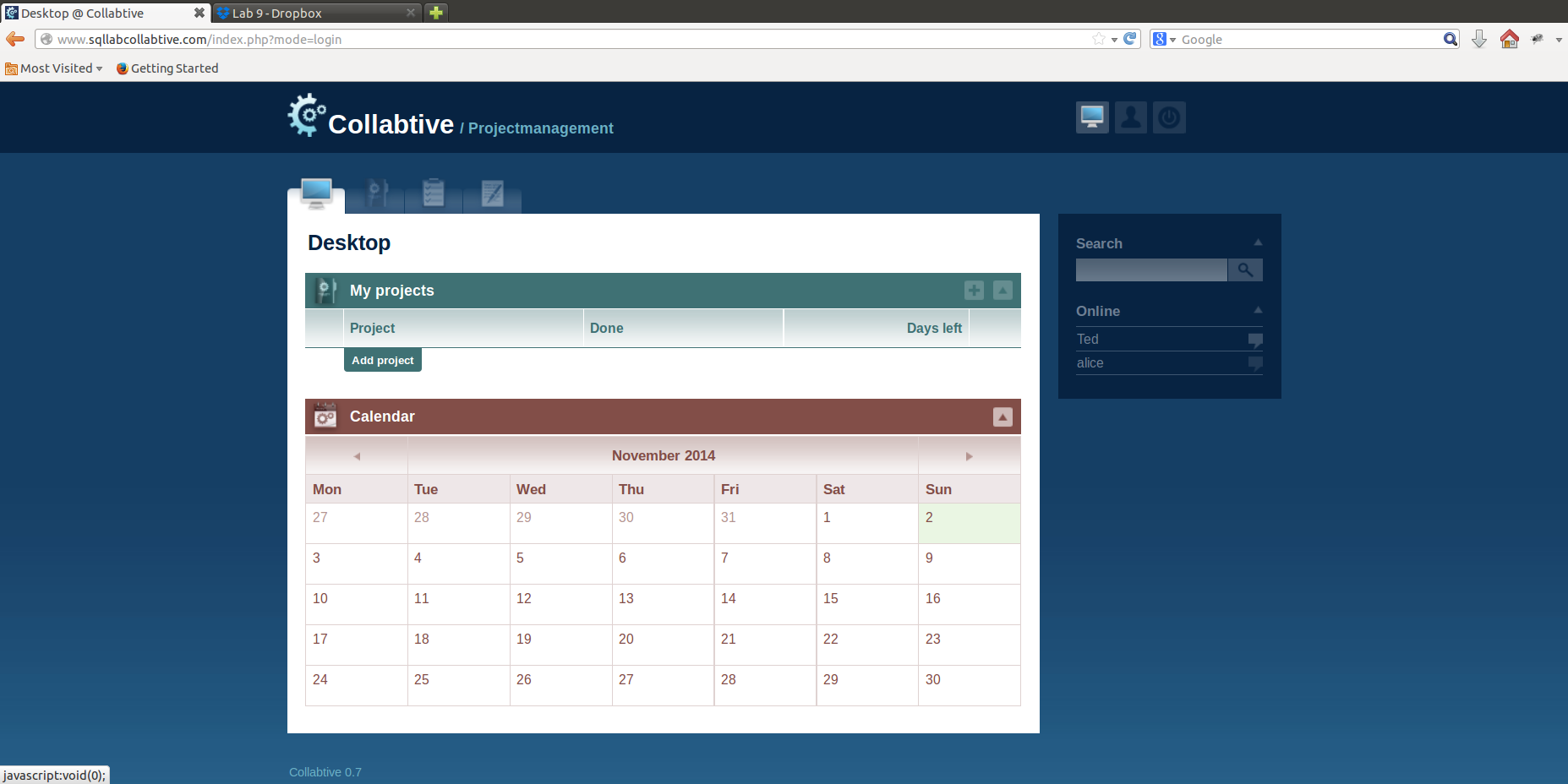


Figure 2.2

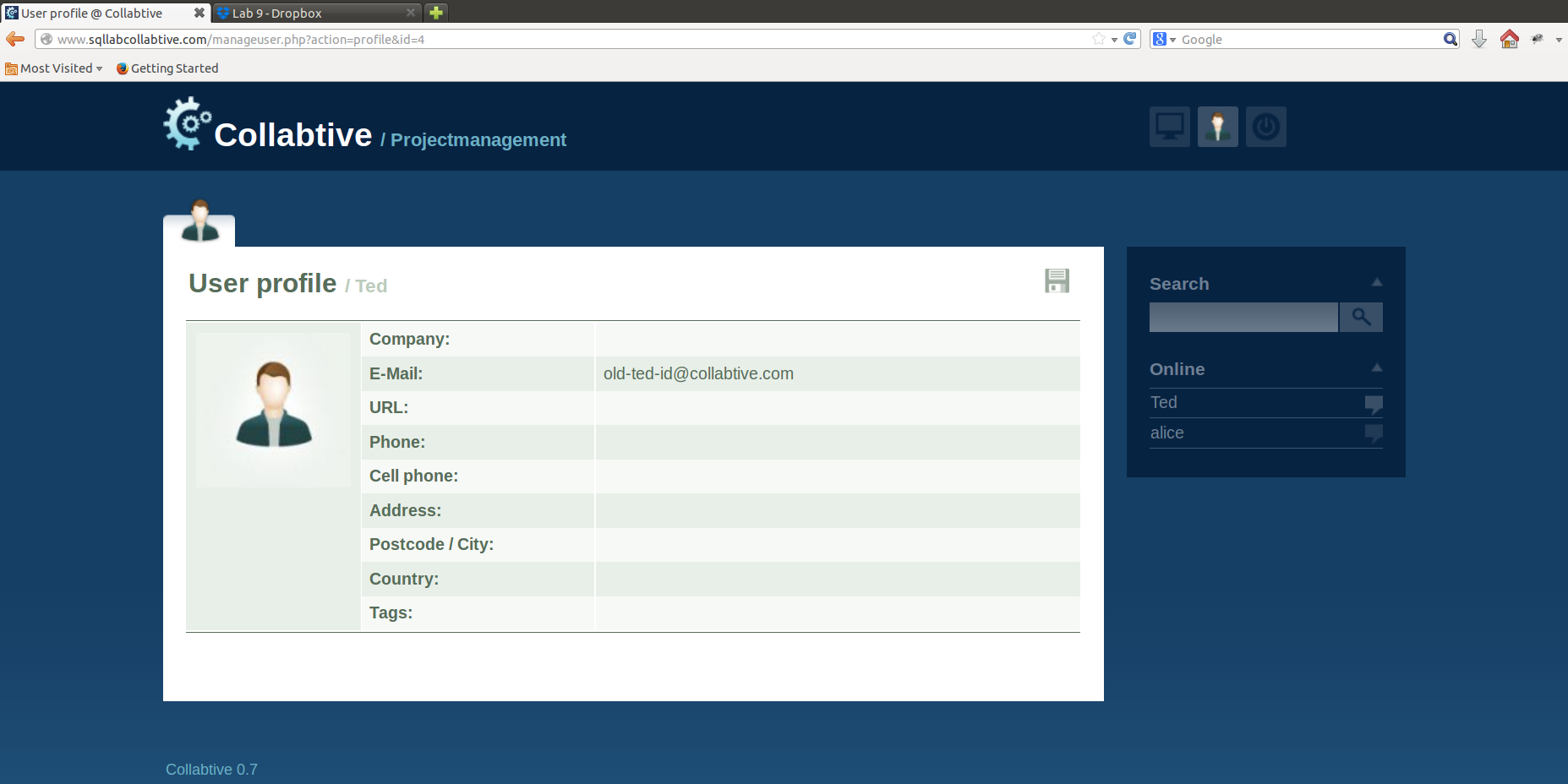


Figure 2.3



Figure 2.4

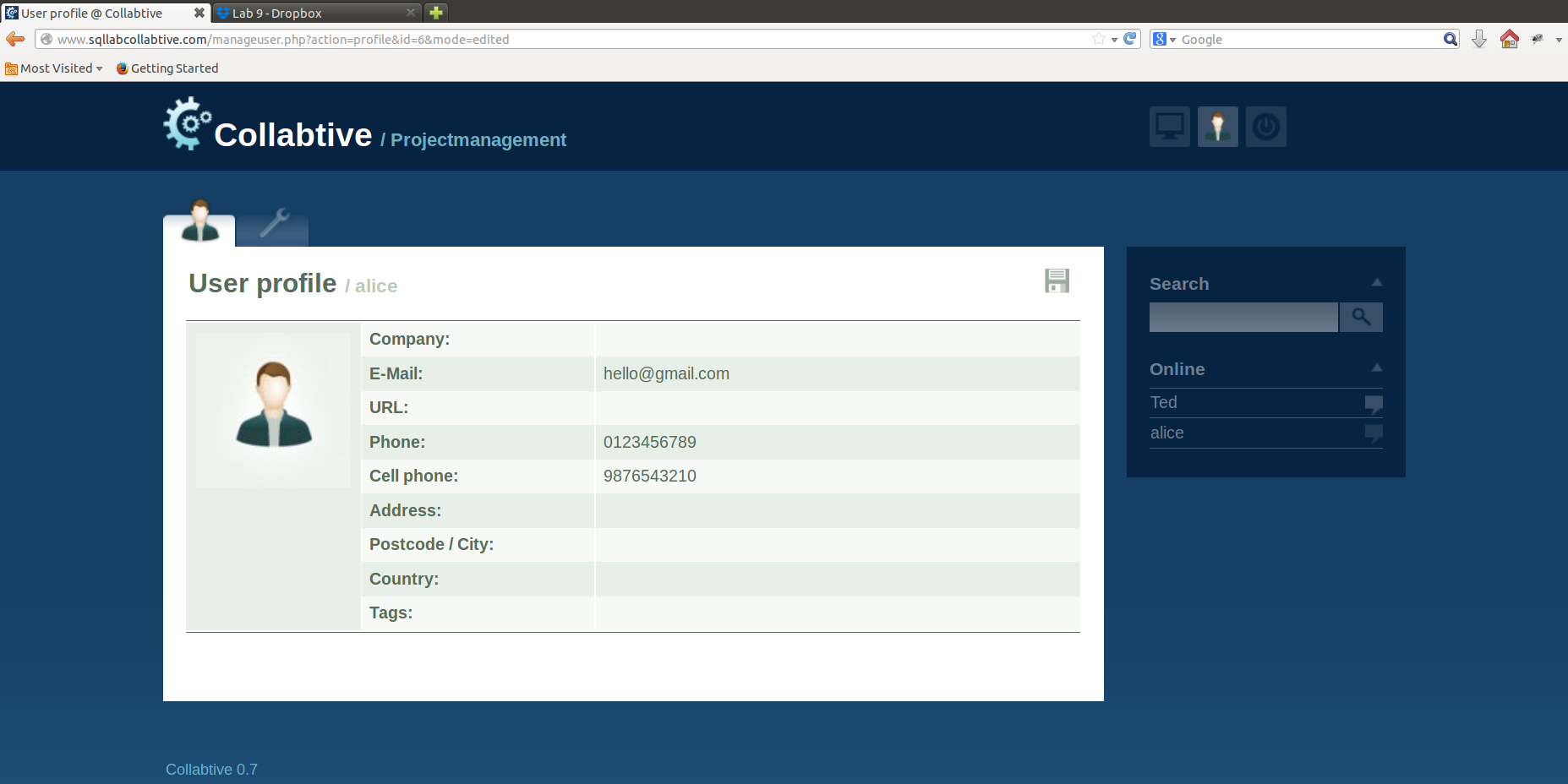


Figure 2.5

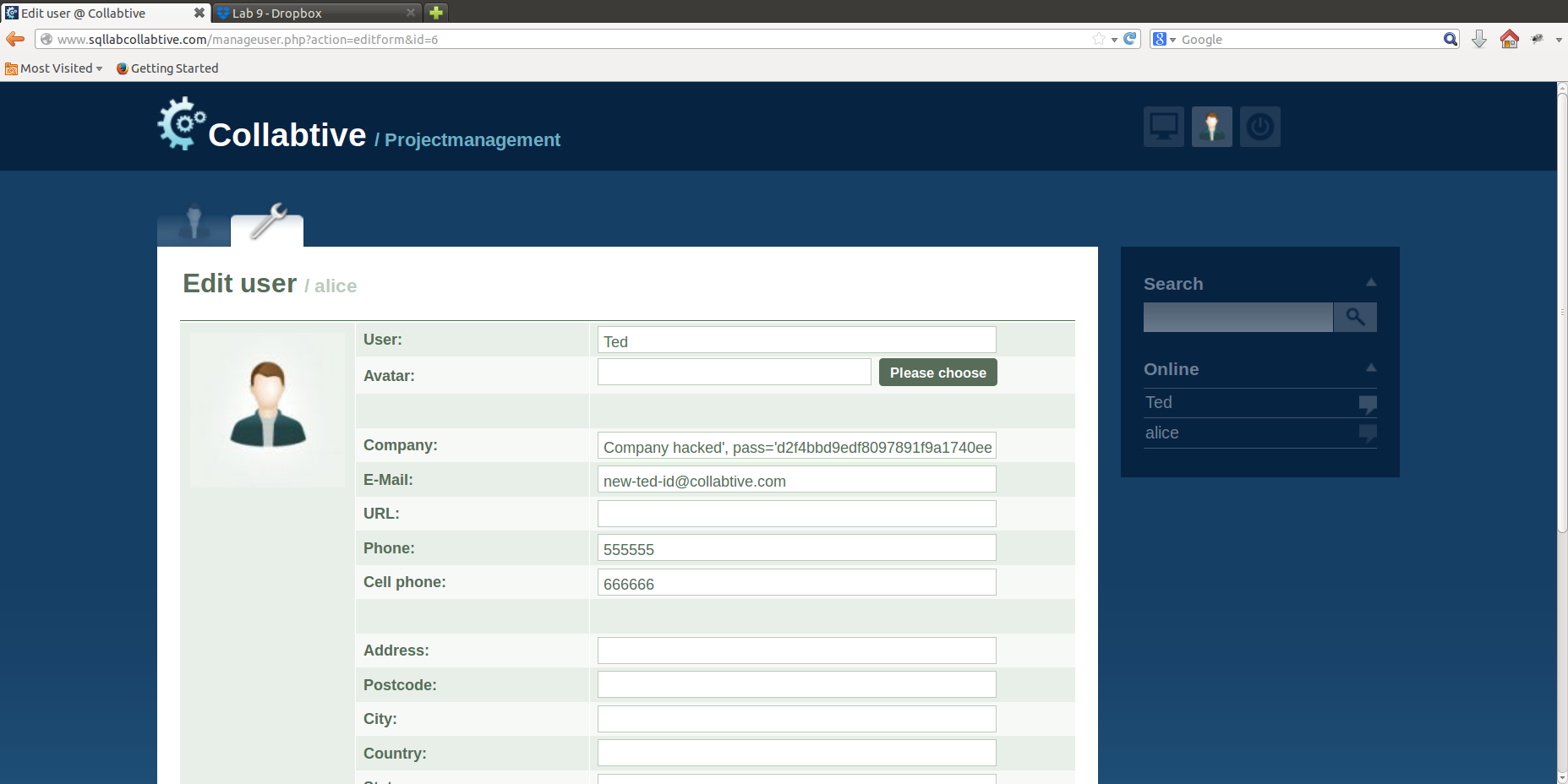


Figure 2.6

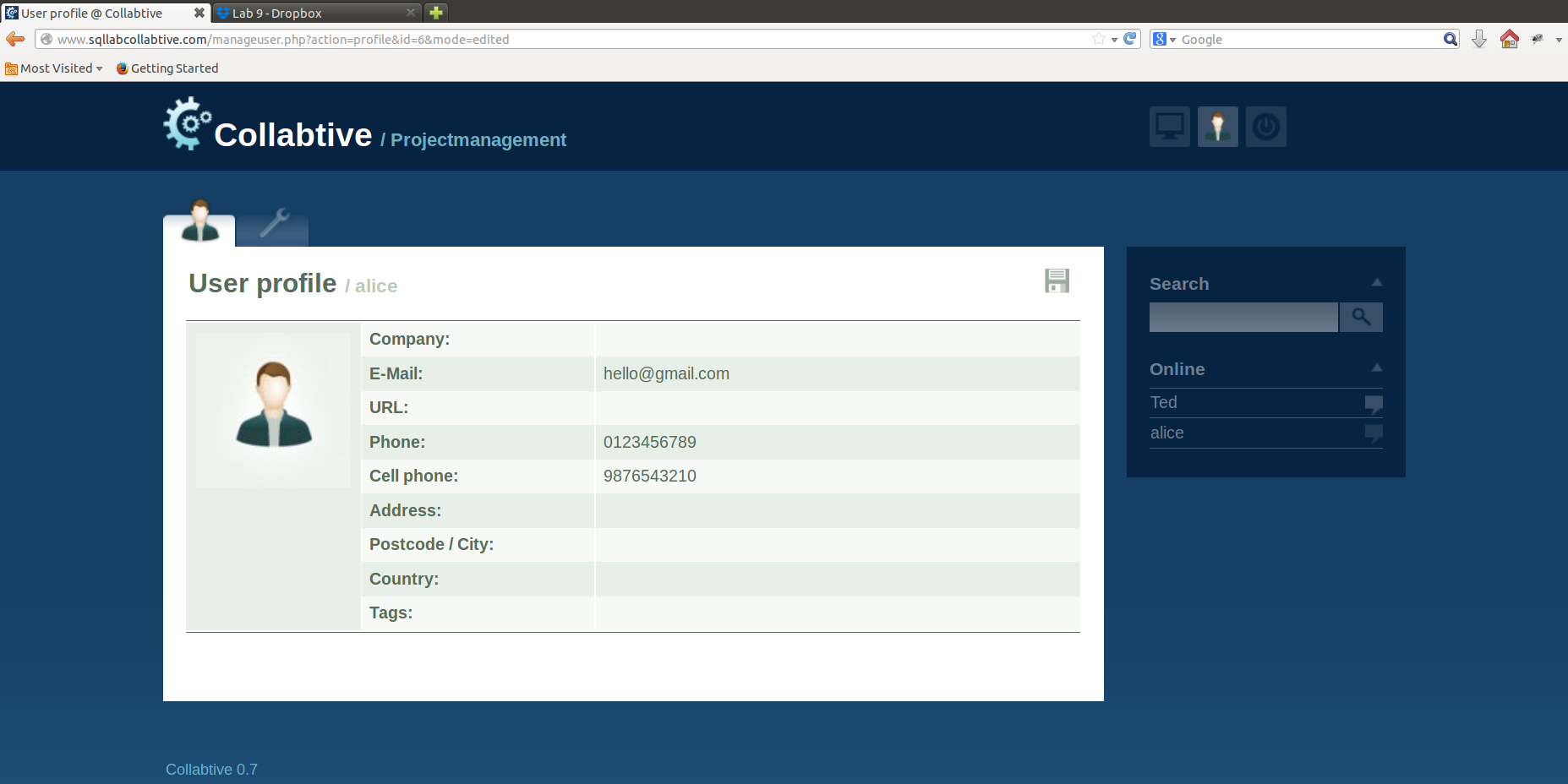


Figure 2.7

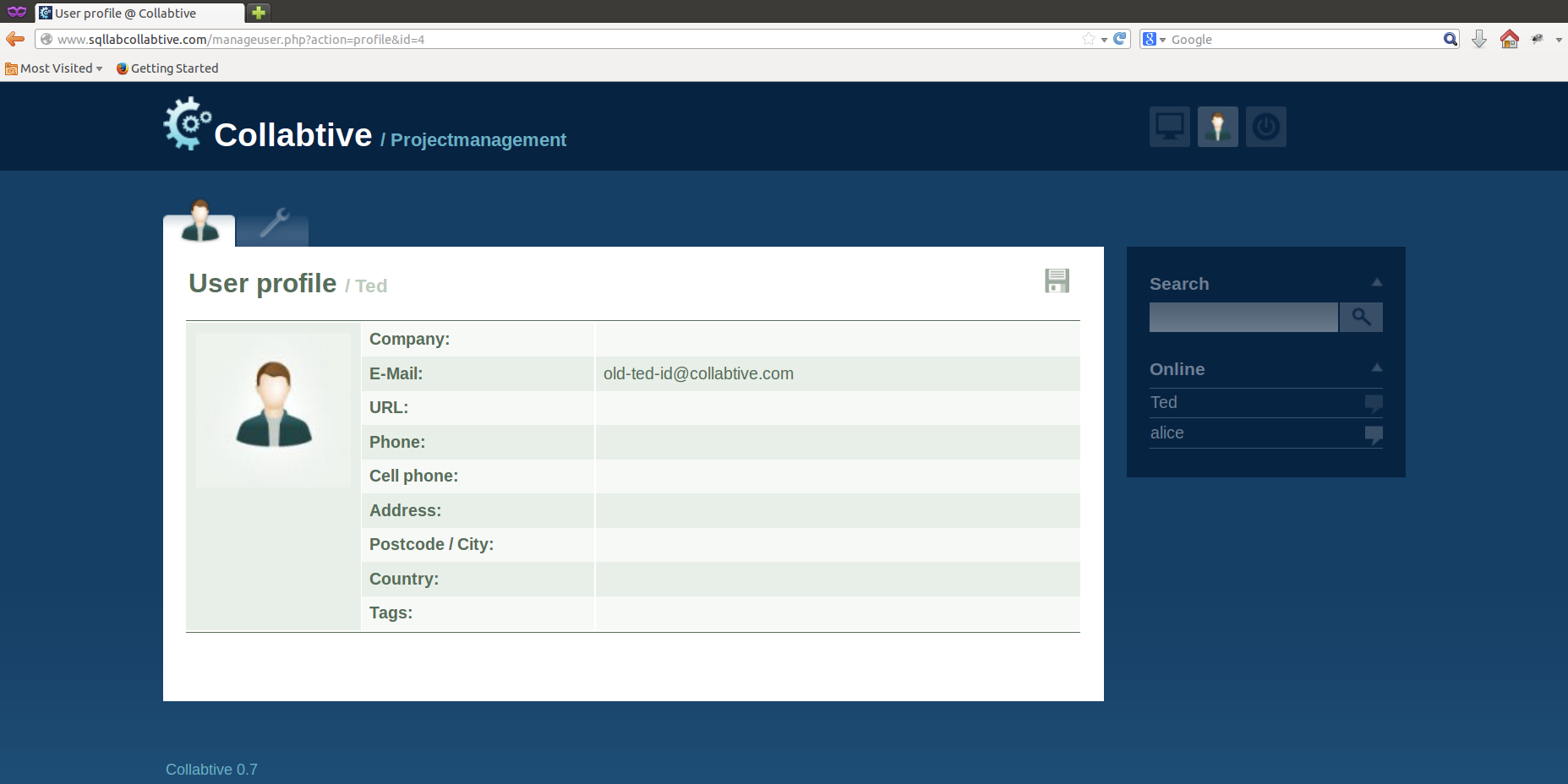


Figure 2.8

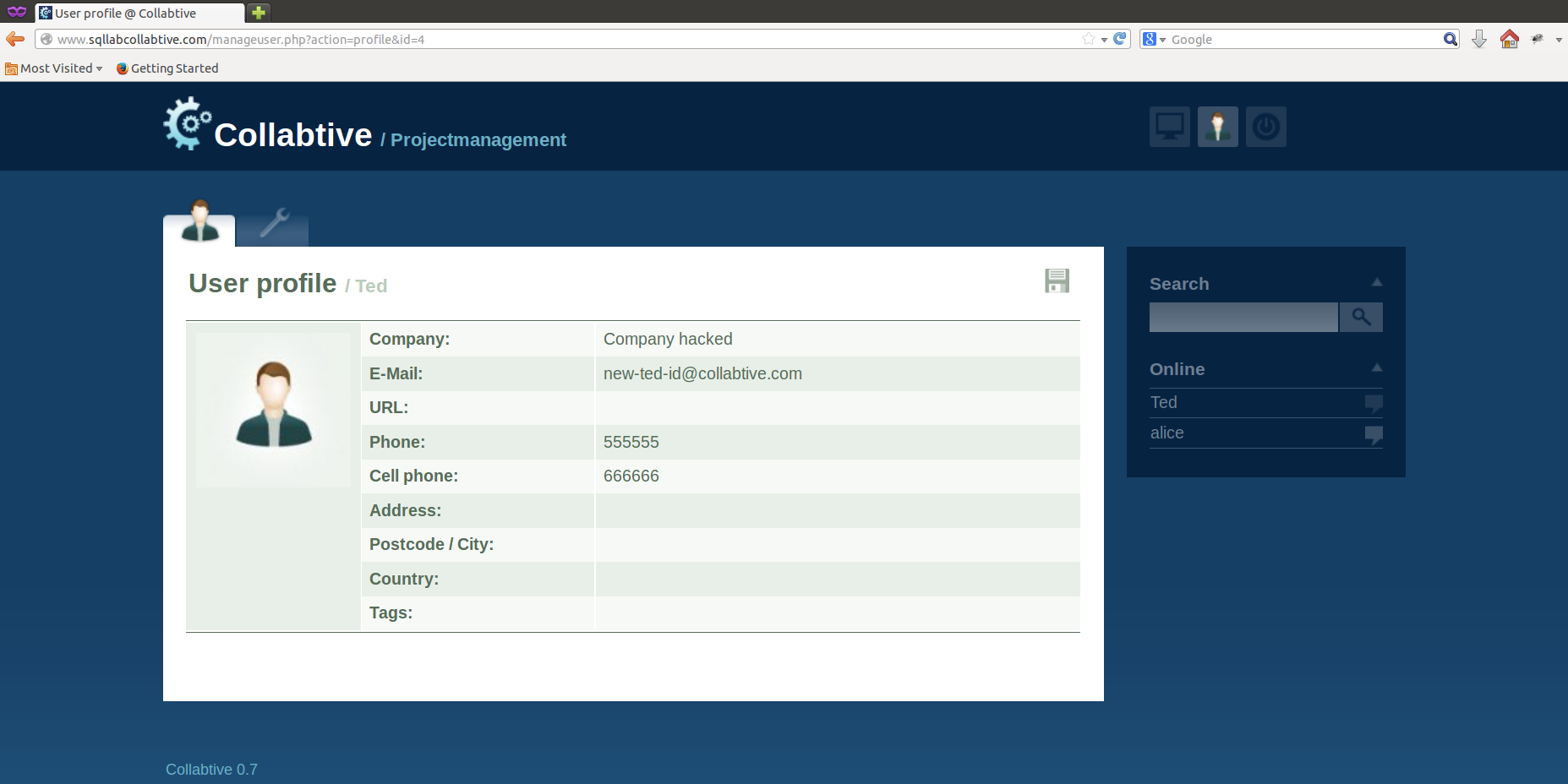


Figure 2.9

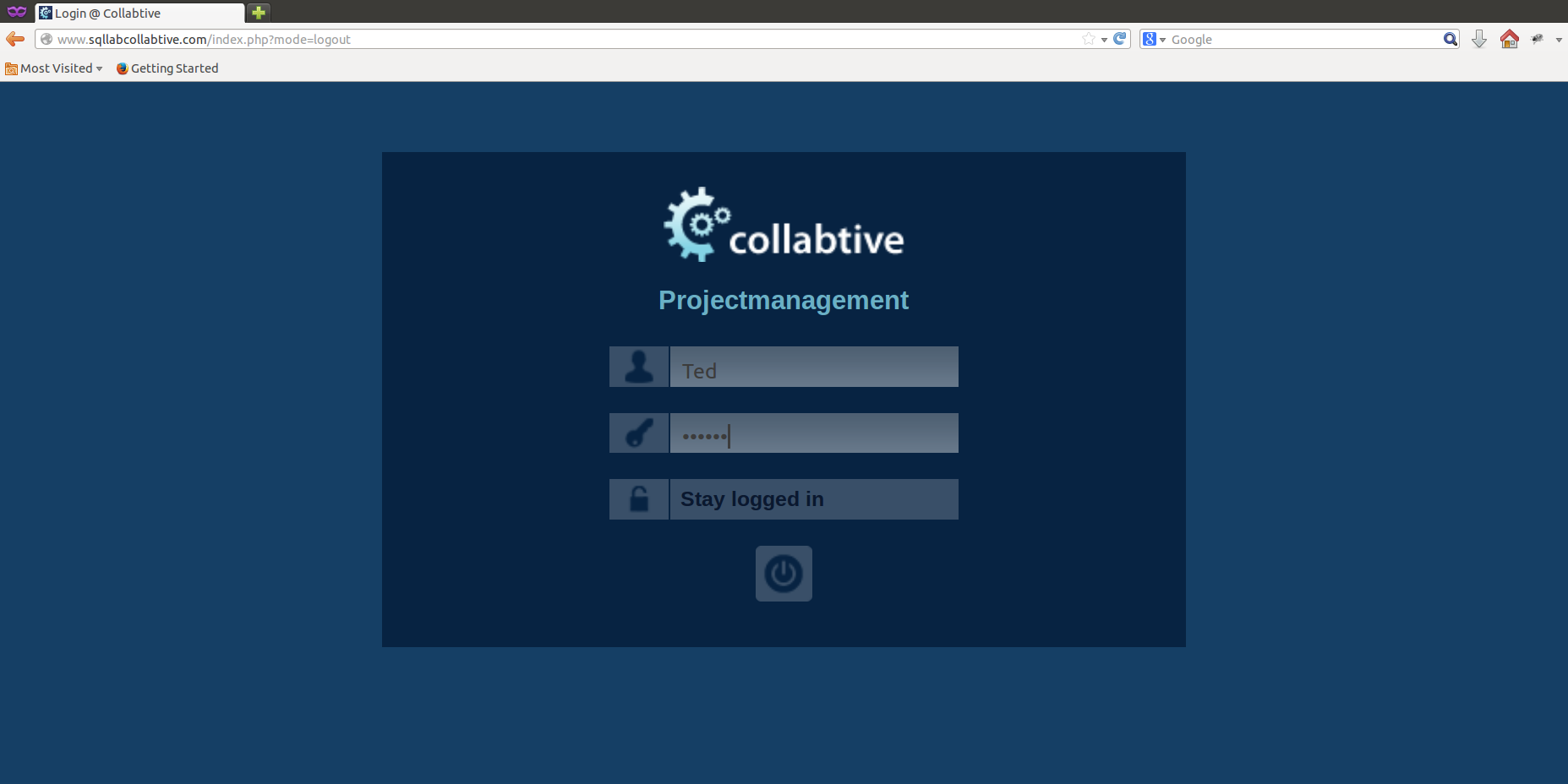


Figure 2.10



Figure 2.11

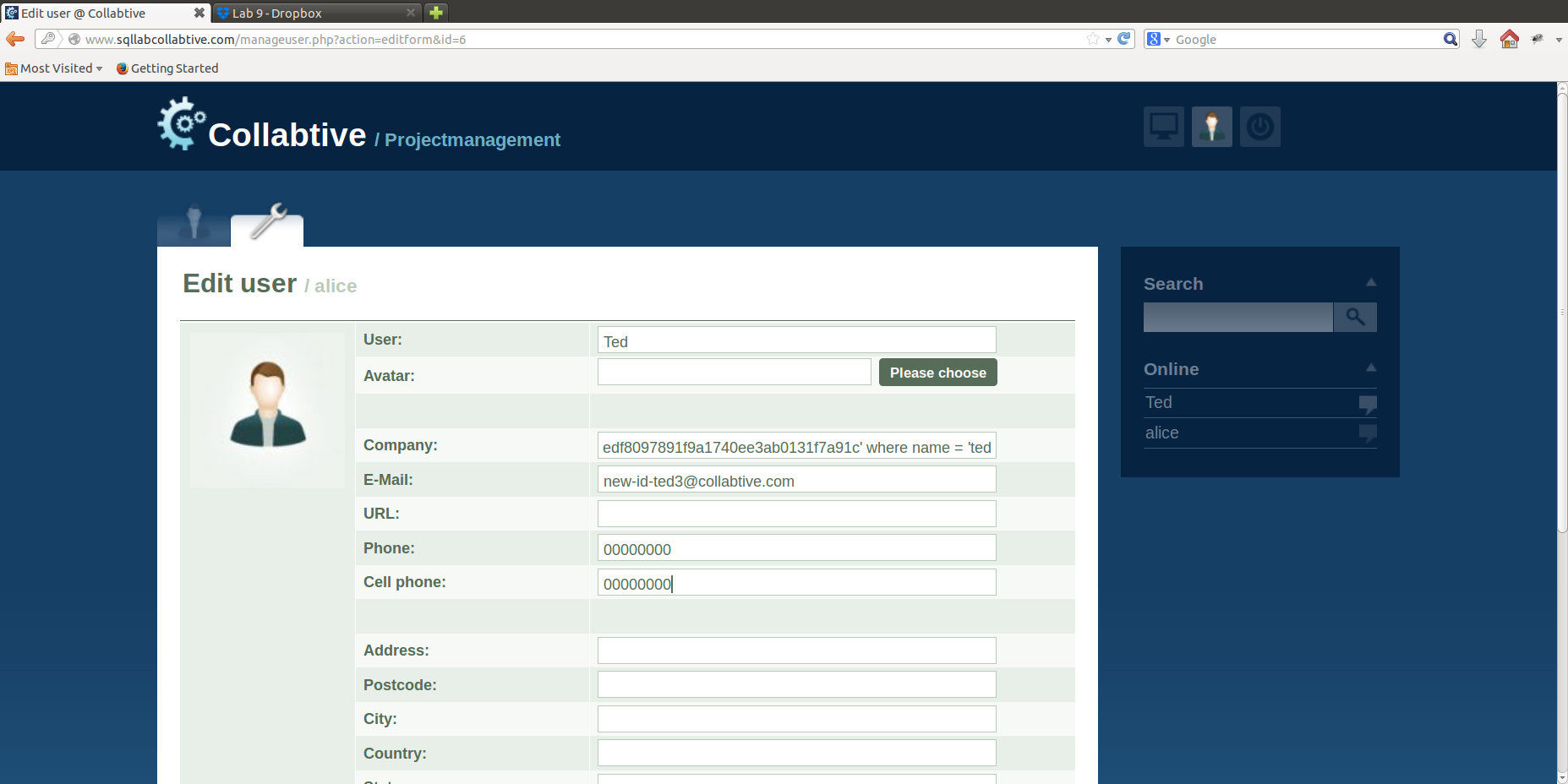


Figure 2.12

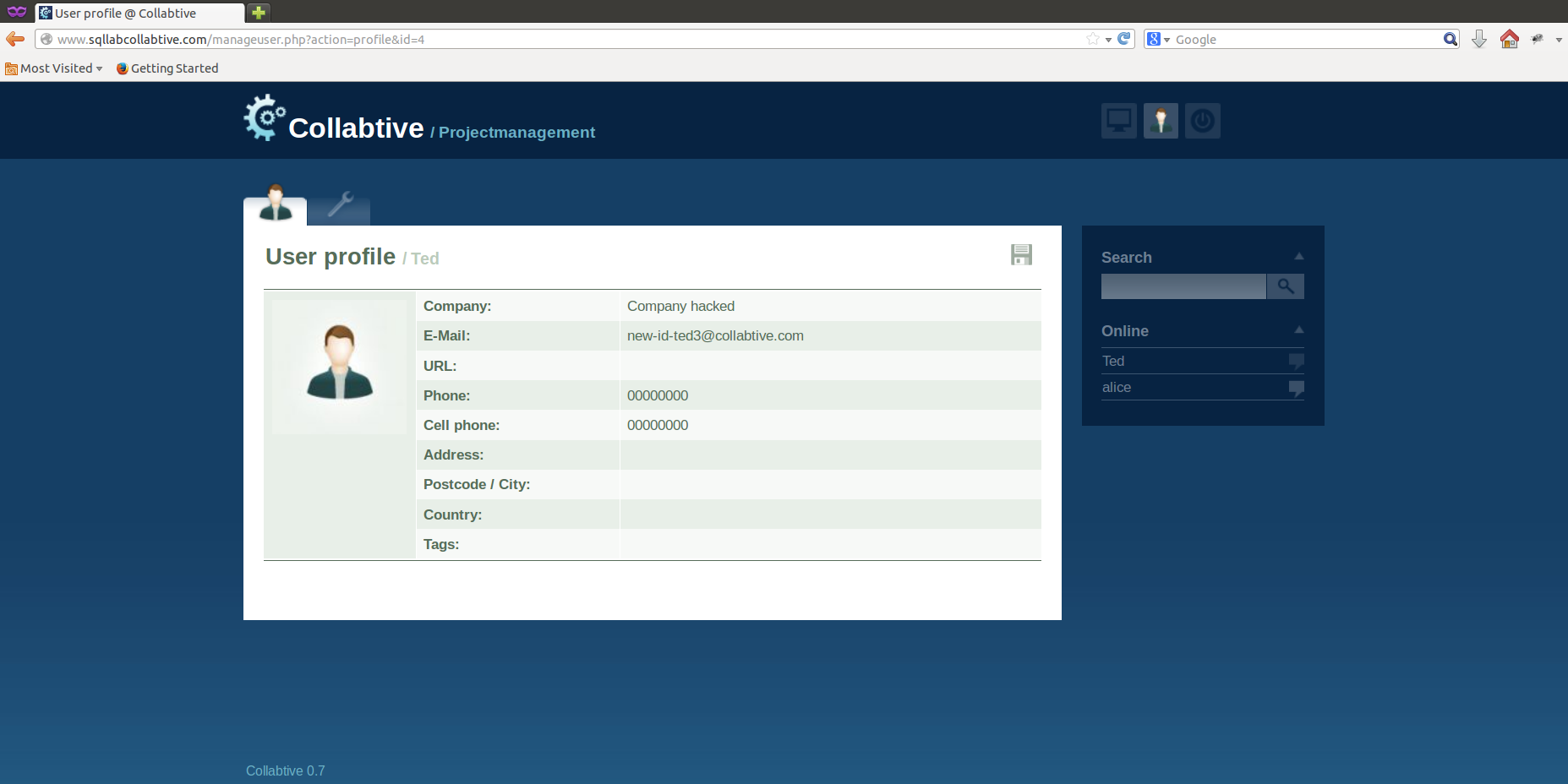
****

Figure 2.13

**Observations and Explanations:**

1. The vulnerability is in the Company field of the profile edit page of collabtive as can be seen from the php code.
2. To get the id of ted so that we can change the password, we visit his profile page and examine the Live HTTPHeader, we get the id as 4 and we will use this in SQL injection attack to modify his page by just modifying Alice’s page. Figure 2.2, 2.3 are accessed using Alice(the attackers) login.
3. Figure 2.4 has the id of Ted, which we will use in our sql, query to perform the attack.
4. Figure 2.5 has Alice’s account before we perform the attack.
5. In Figure 2.6 we modify Alice’s account and change the User field to Ted and from our finding of the vulnerability in the php code, we add the SQL query to modify Ted’s account in the compay field. The query we add is,

***Company hacked', pass='d2f4bbd9edf8097891f9a1740ee3ab0131f7a91c' where id = 4; --***

From looking at the code we can find out that, the password is not stored as plain text but instead, it is stored with SHA1 encryption, but since we can’t call the php function **sha1()** in the SQL query statement, so we use online tools to get the output of applying the function.

So we add a password **Alice!** to Ted’s profile and the SHA1 hash of the string is used above.

1. Once Send is clicked, the SQL query is executed and Ted’s profile is edited, also his password is changed, as can be seen from Figure 2.9.
2. We log into Ted’s account using the newly set password. See Figure 2.10 and 2.11.
3. We can also log into Ted’s account by using the following query,

***Company hacked', pass='d2f4bbd9edf8097891f9a1740ee3ab0131f7a91c' where name = ‘ted’; --***

1. Figure 2.12 and 2.13 demonstrate the attack in the alternative way.

**Task 3:**

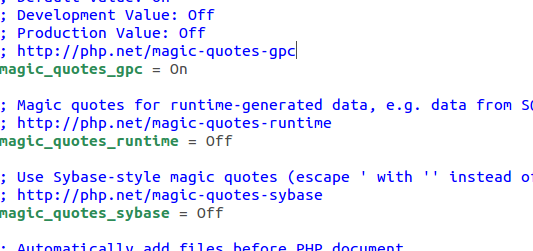
****

Figure 3.1

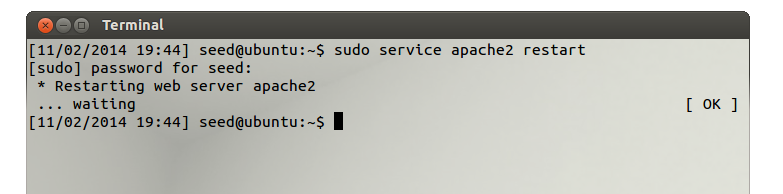


Figure 3.2

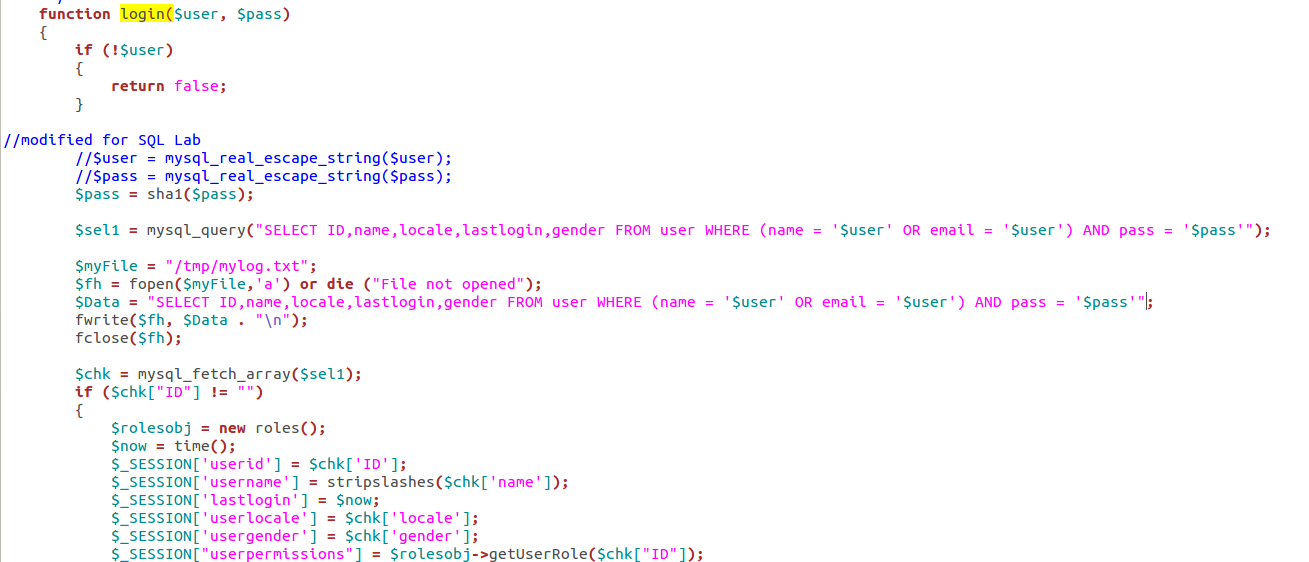


Figure 3.3

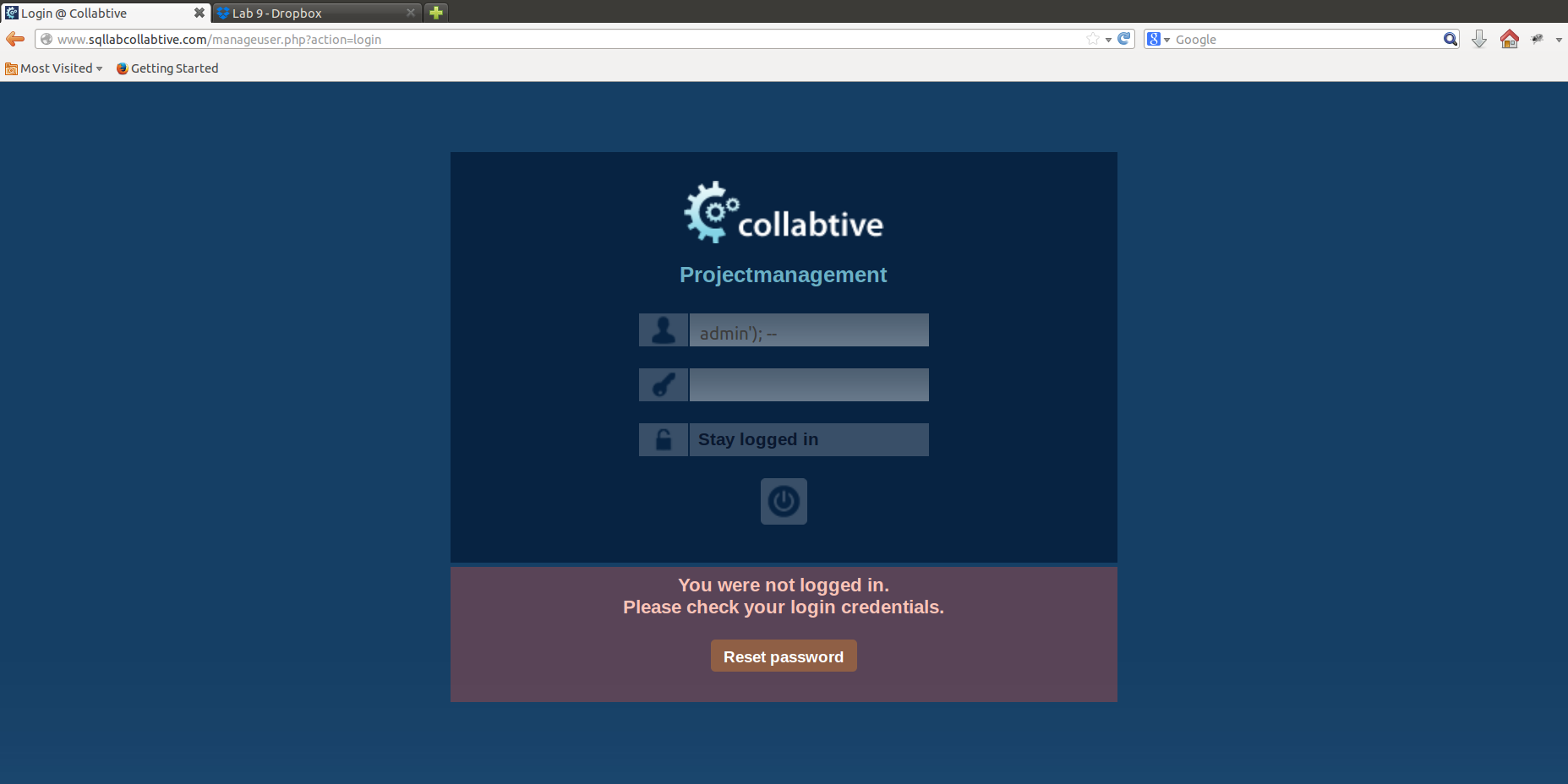


Figure 3.4

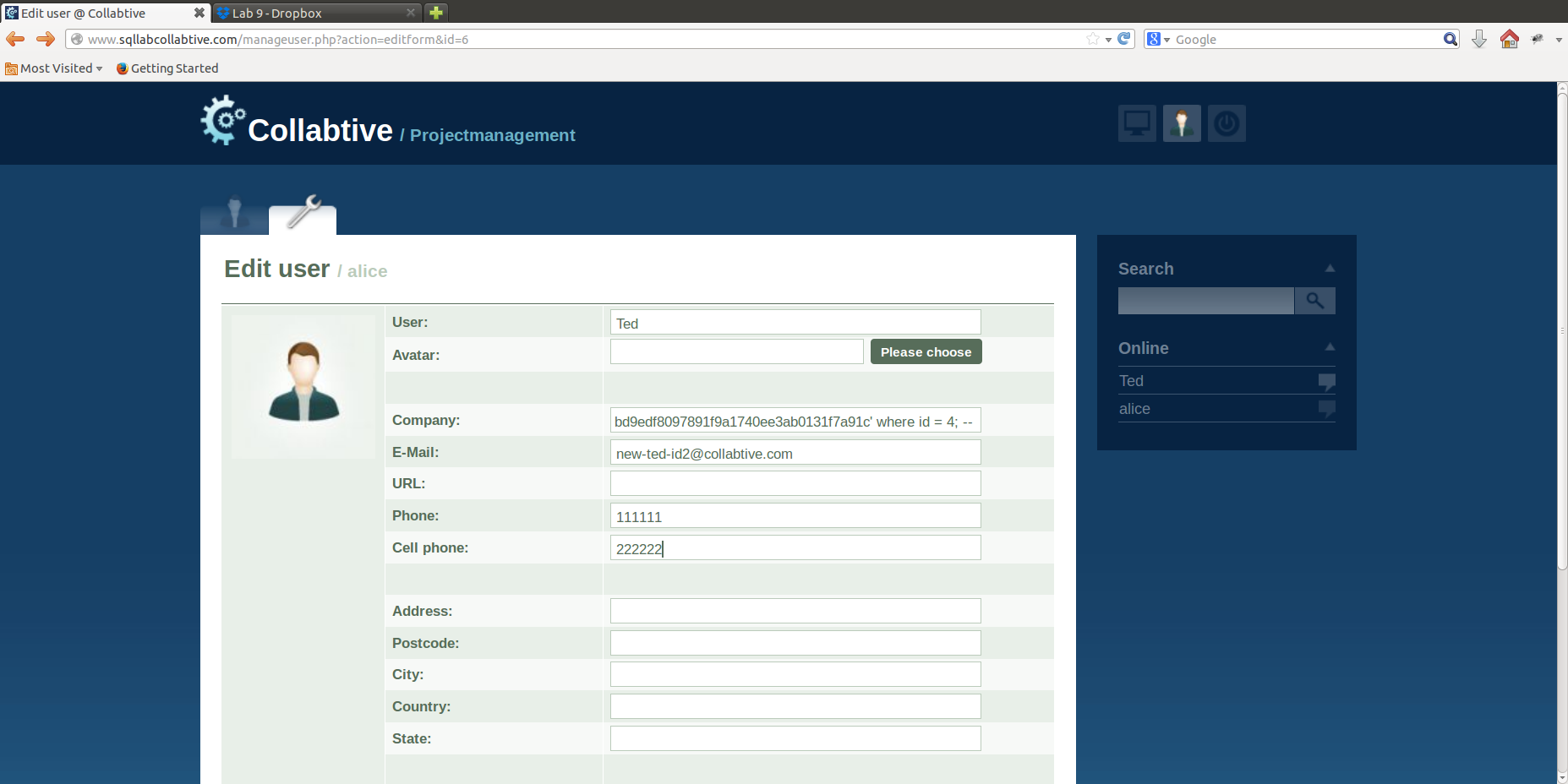


Figure 3.5

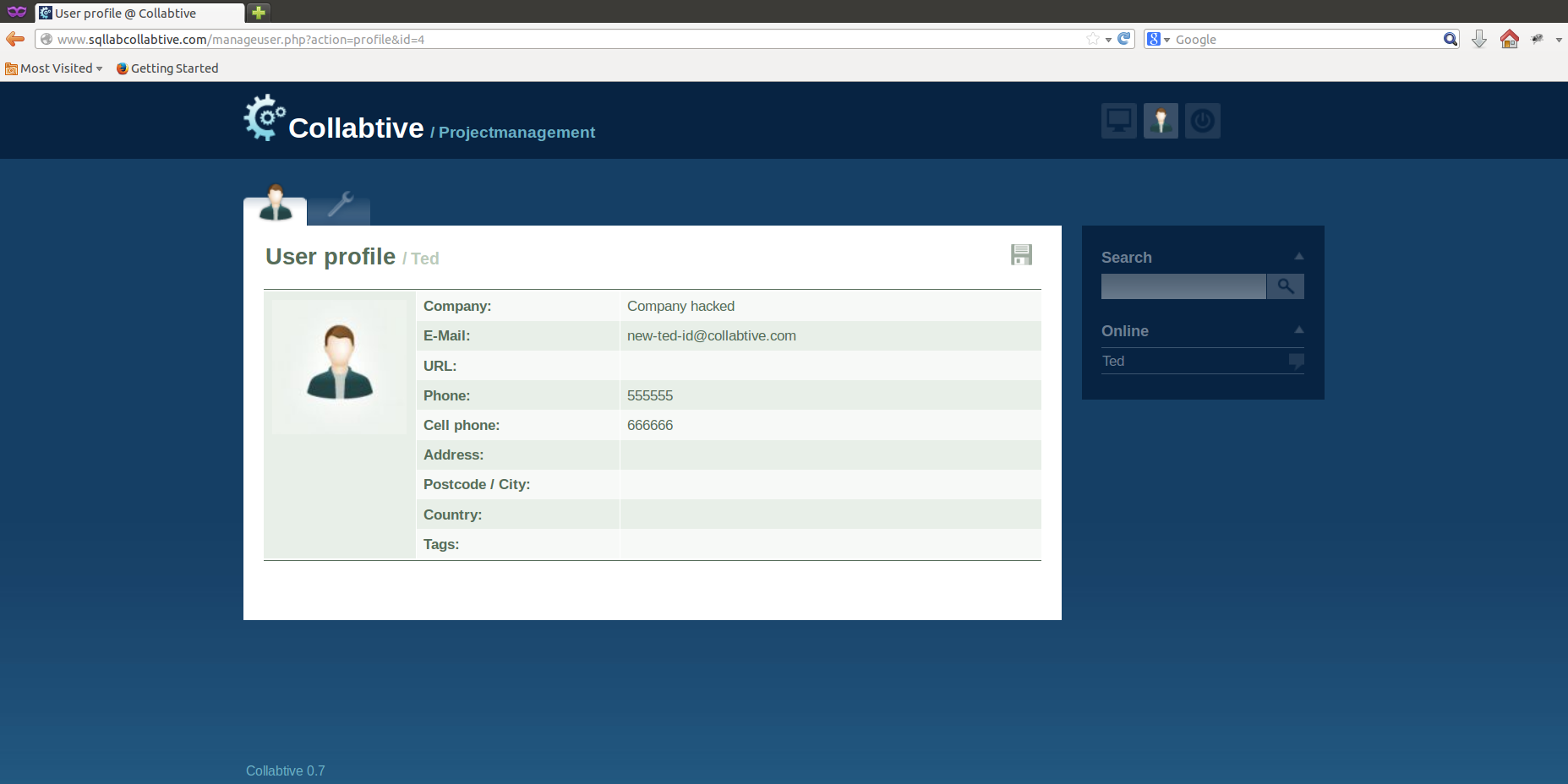


Figure 3.6

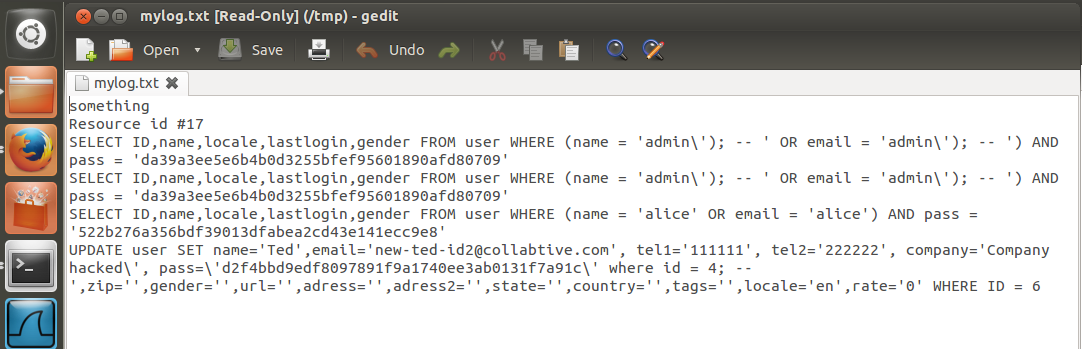


Figure 3.7

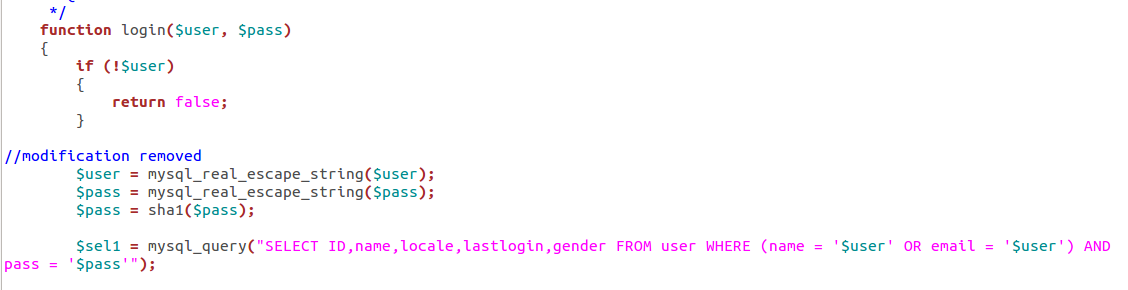


Figure 3.8

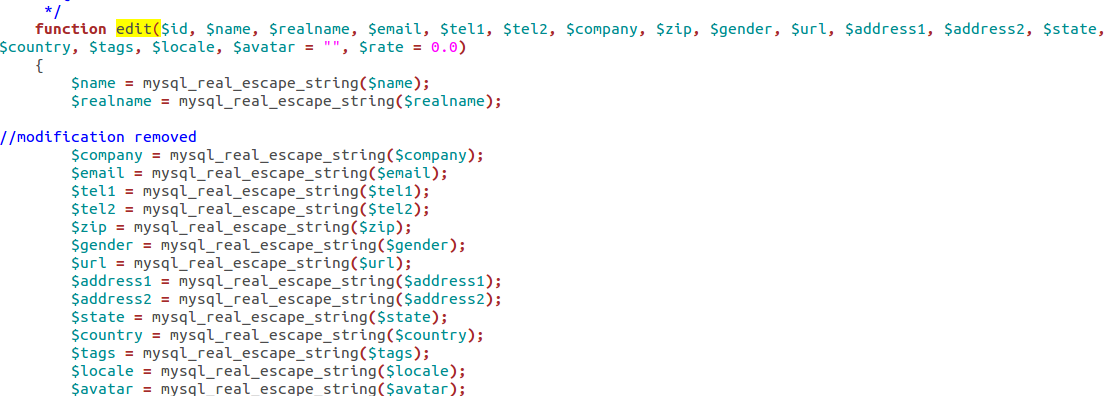


Figure 3.9

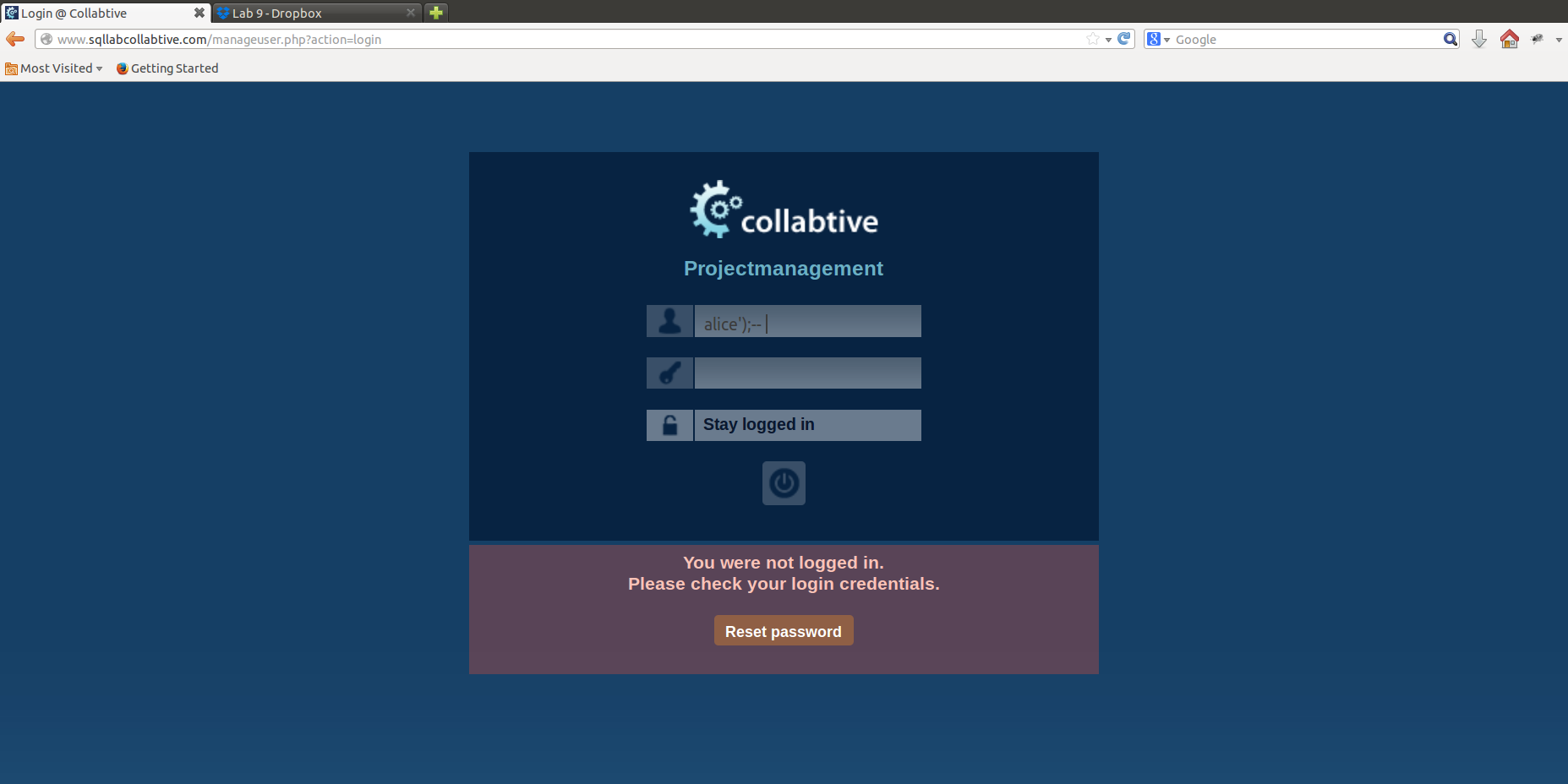


Figure 3.11

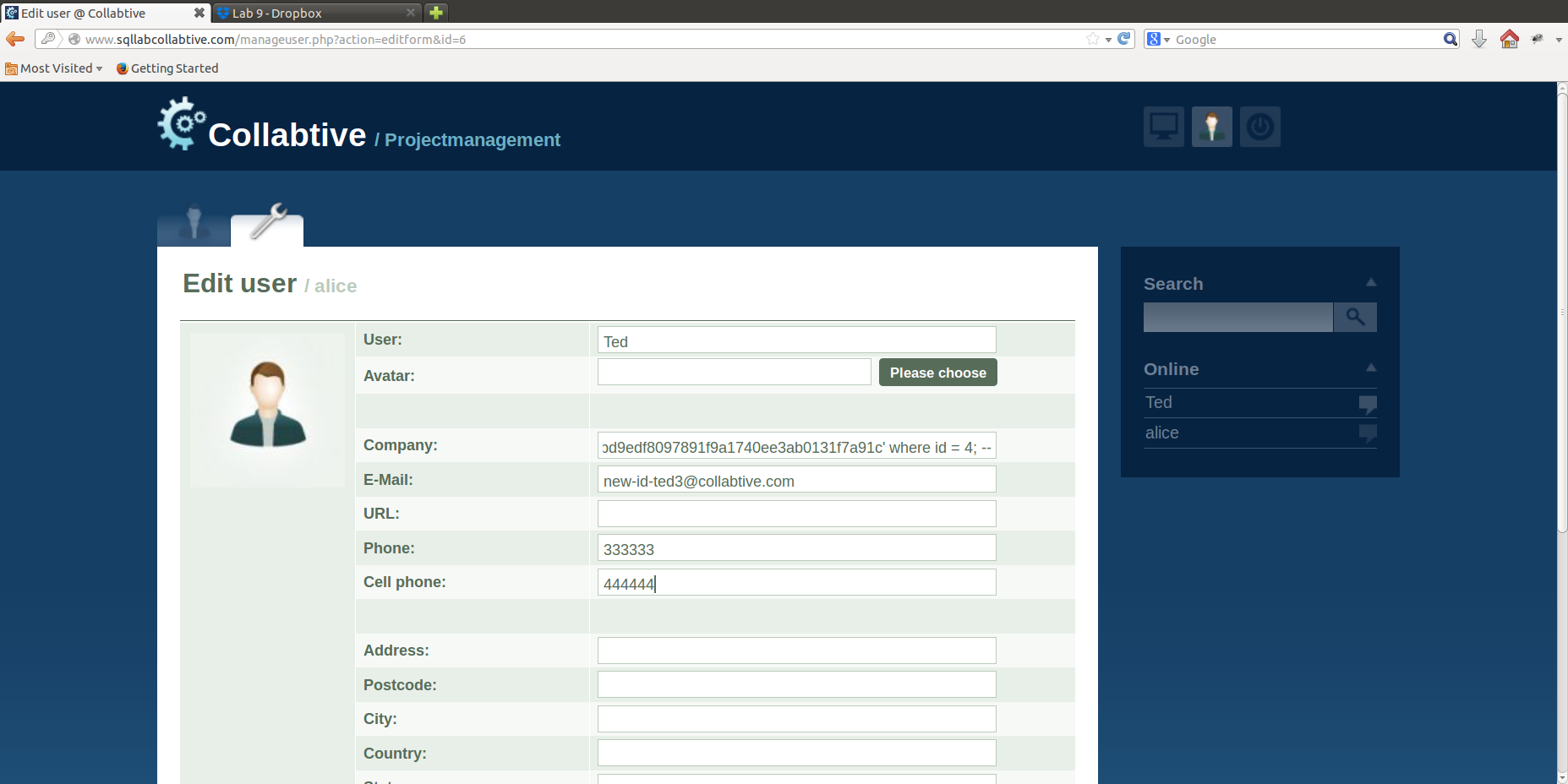


Figure 3.12

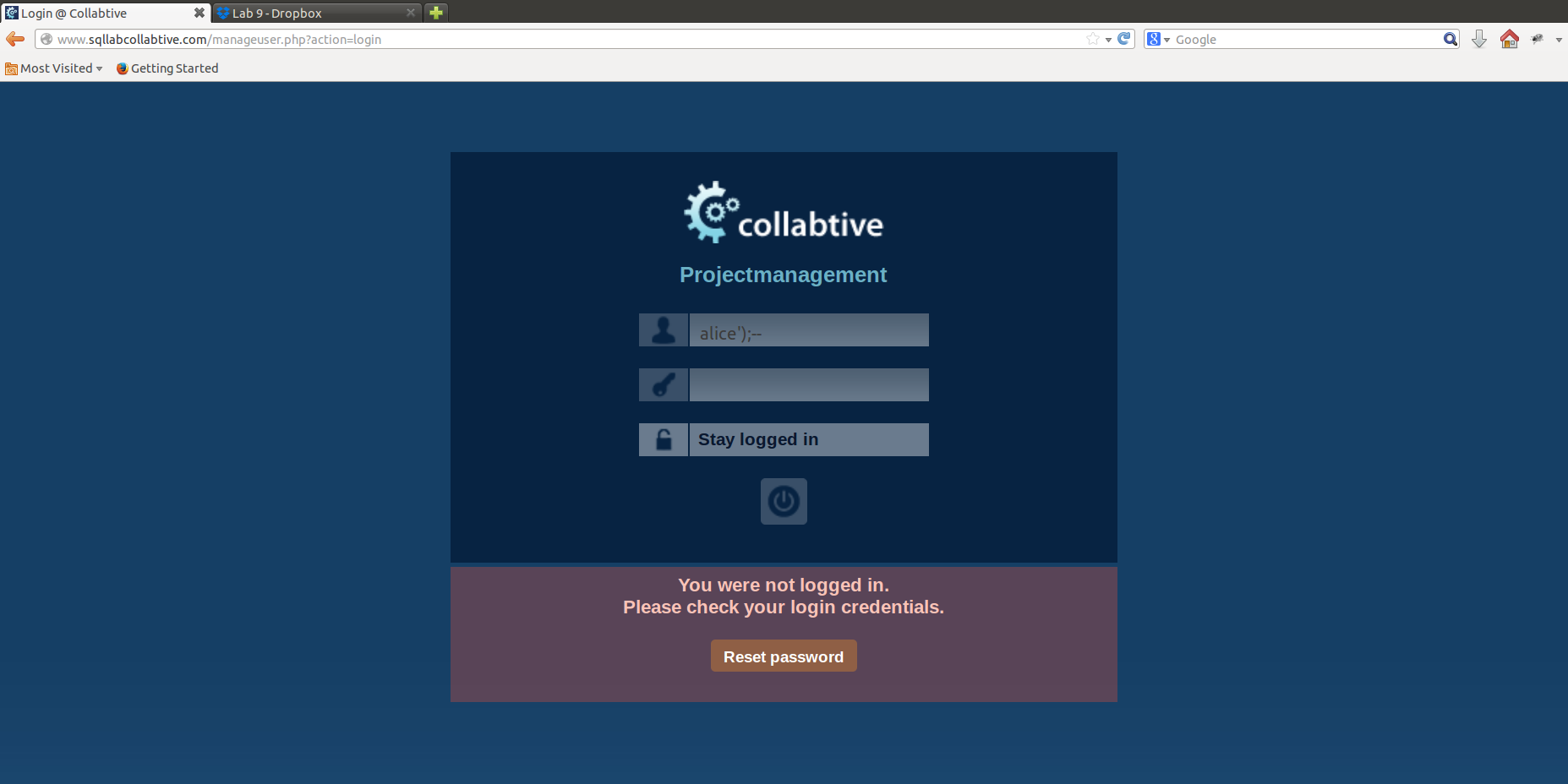


Figure 3.13

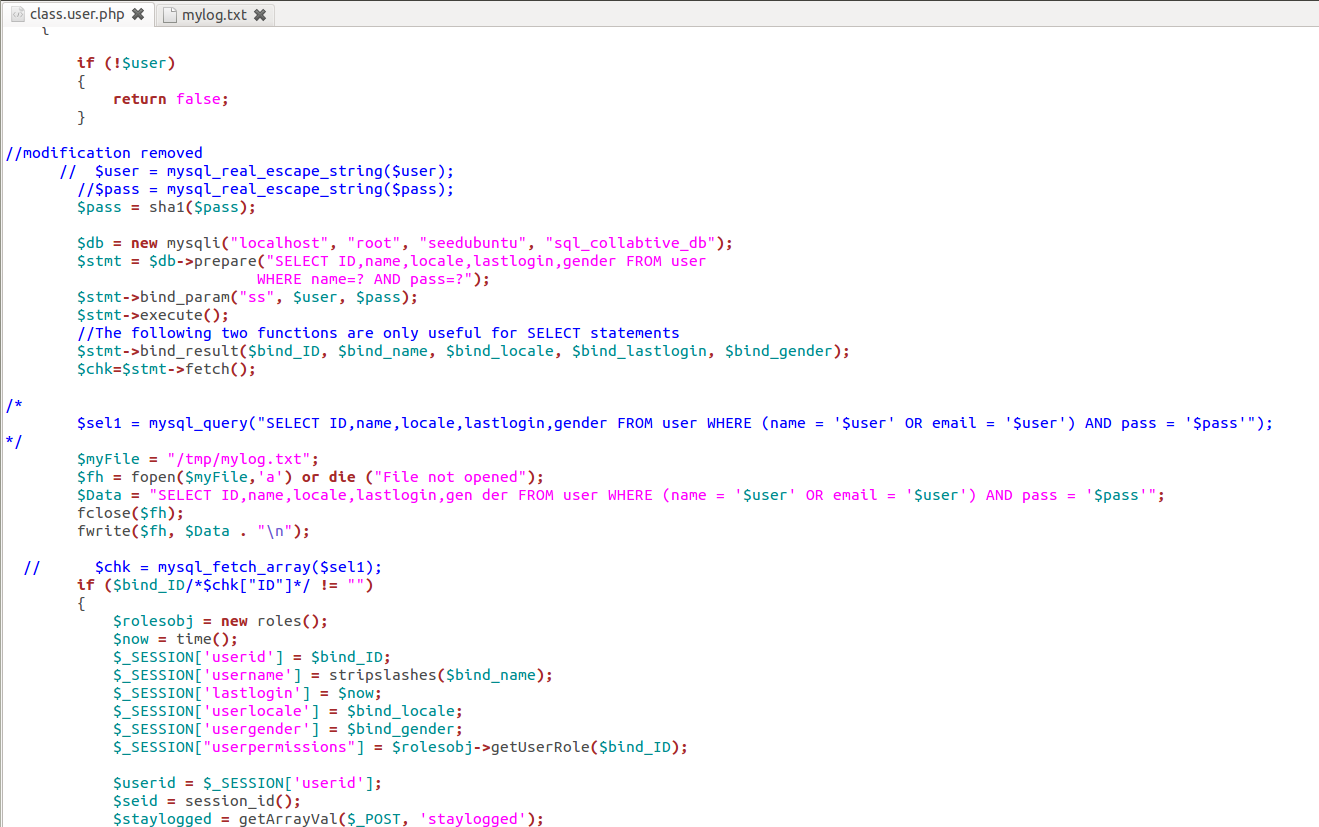


Figure 3.14

**Observations and Explanation:**

1. We turn on the countermeasure that we had turned off at the start of this lab, **magic\_quotes\_gpc=On** and restart the apache server.
2. Add debug code to the php file in the login function and the edit function.
3. On running the login attack, it fails and we can see that the countermeasure is turned on by seeing the log, Figure 3.8.
4. Now, that the countermeasure is turned on, we can see from the logs that all the single quotes and double quotes have been escaped using the escape character (\).
5. This countermeasure has been deprecated as portability of code is affected, because the code ill function depending on whether this countermeasure is being used or not. Another reason that this is deprecated is that, not all user input in liable to SQL injection and so performance is affected and also user is not able to input required text.
6. We turn off the magic\_quotes\_gpc protection and restart the apache server before analyzing the **mysql\_real\_escape\_string()** function.
7. The function calls were disabled in edit () and login functions of the php code. We uncomment the code and the countermeasure is now in place.
8. The login attack as well as the edit profile attack both fail when the function calls are uncommented, as seen from Figure 3.13, the log, User inputted single quotes have been commented out and this leads the attack to fail. The back slash (\) is prepended to the special characters to avoid SQL injection attacks.
9. For task 3.3, we use the prepared statement to make sure that the attacker is not able to inject any SQL code into the form for user input.
10. Here, the issue is that the browser doesn’t know the distinction between data and code, to ensure that this distinction is not lost, we have to make sure that the code part is treated as code and data part is treated as data. So instead of directly providing the user input to the query in the code and losing this distinction, we use a positional placeholder. This placeholder will ensure that the data part that is input y the user is treated as data, the prepared statement ensures that the code part is parsed separately and the data is bound to it.
11. Figure 3.13 shows that the SQL injection attack has failed and Figure 3.14 shows the modification to the code that has been done to enable the prepared statement countermeasure in the login function.