COSC 647 - Fall 2015 - Logan Bair, Harold McGinnis, & Mary Snyder
Lab 4 - SQL Injection Lab

1. **What were the environment and attack set up? What is/are the ultimate goal(s) for this lab?**

   The ultimate goal of this lab is to teach the basics of SQL Injection Attacks and how to prevent them. Vulnerabilities are present when user input is not strongly typed or SQL statements are not correctly filtered for embedded string literal escape characters. The lab runs on an Ubuntu box on which a local web server is run, connected to a MySQL server backend. The following outlines the general setup steps for this lab:

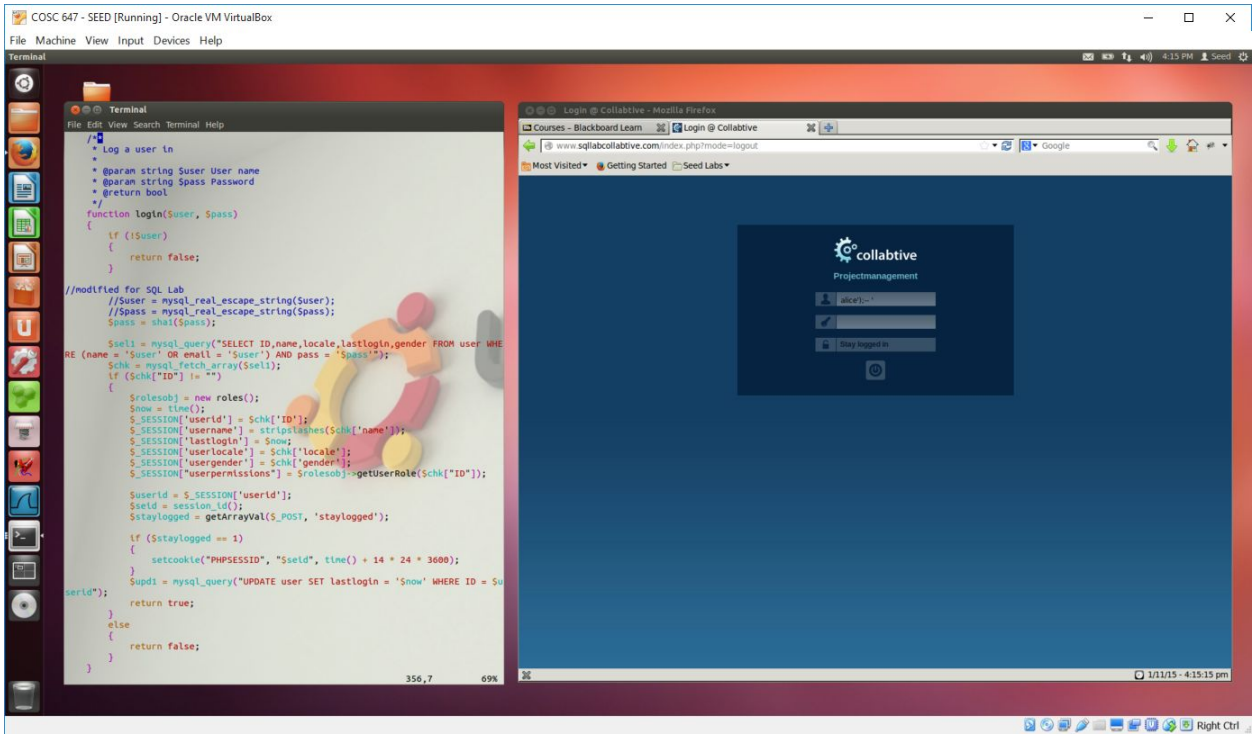
   Initial setup steps:
   a. Start the apache server:
      i. sudo service apache2 start
   b. Turn off the countermeasures:
      i. 'sudo gedit /etc/php5/apache2/php.ini' (or vi instead of gedit)
      ii. Update the line "magic_quotes_gpc = On" to magic_quotes_gpc = Off"
      iii. Save the file and exit the editor
   c. Restart the apache server:
      i. sudo service apache2 restart

   Steps to determine exploit information
   a. Open the source code file of Collative:
      i. /var/www/SQL/Collabtive/include/class/user.php
   b. Open website and play around:
      i. http://www.sqllabcollabtive.com/

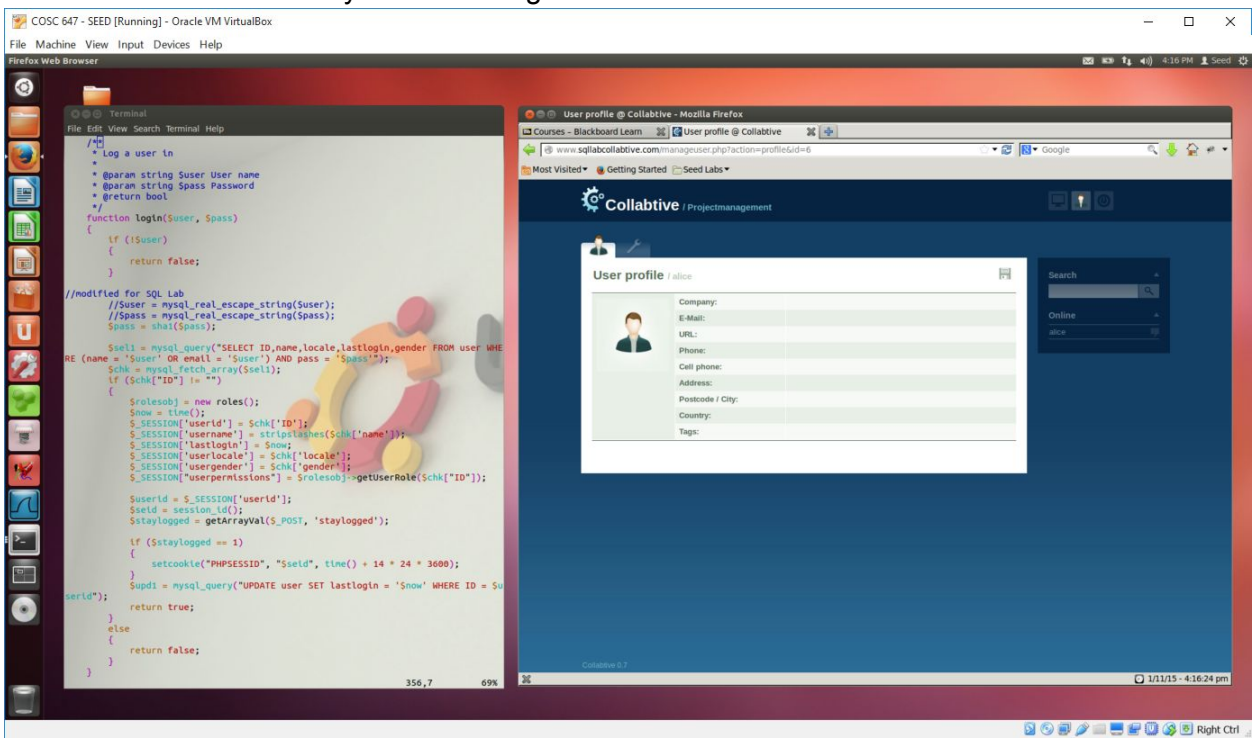
2. **What were the steps that you take in order to launch the attack? (Note: Make sure your include the shell commands, GDB debugger commands and screenshots of your computer to demonstrate it.)**

   Steps to determine exploit information
   a. Open the source code file of Collative:
      i. /var/www/SQL/Collabtive/include/class/user.php

   For exploit 1 (Login as any user account without knowing/entering the password):
   a. In the login function, determine what information is necessary for the query to succeed without a password.
      i. When we examined the source code for the web application, we realized that the query, "SELECT ID,name,locale,lastlogin,gender FROM user WHERE (name = '$user' OR email = '$user') AND pass = '$pass'", is vulnerable to SQL Injection. User input will be directly added to the query string without escaping, meaning that it can change the syntax of said query. Specifically, the query can be changed such that the password check is ignored.
      ii. For alice, use the username "alice');-- '" will bypass the check. The apostrophe will complete the input and the comment will ensure the password check will be ignored.
   b. Open website and login using the :
      i. http://www.sqllabcollabtive.com/
      ii. for alice, use the username "alice');-- '" to bypass the password check



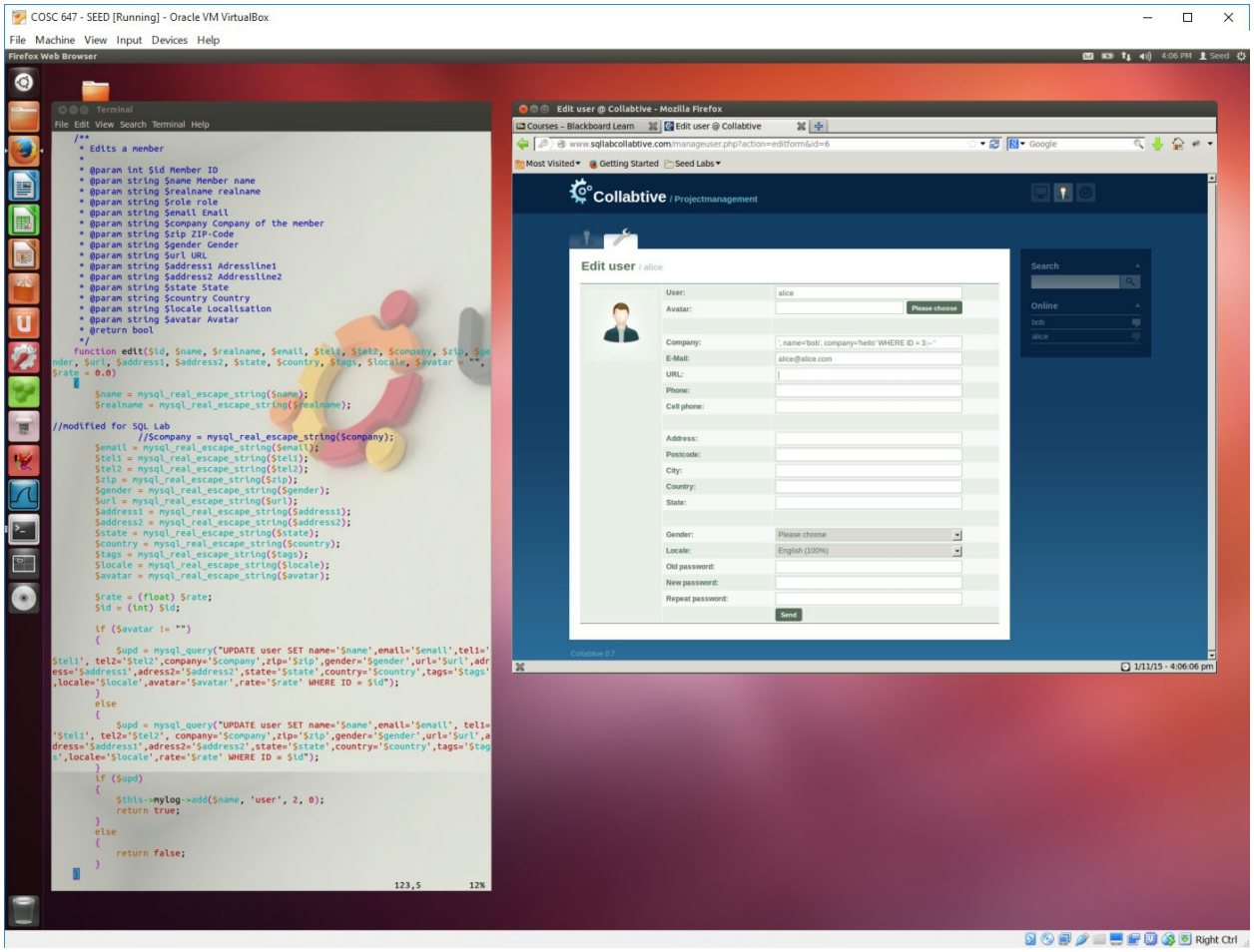
iii. verify access was granted as user alice.



For exploit 2 (From alice's account, update bob's info):

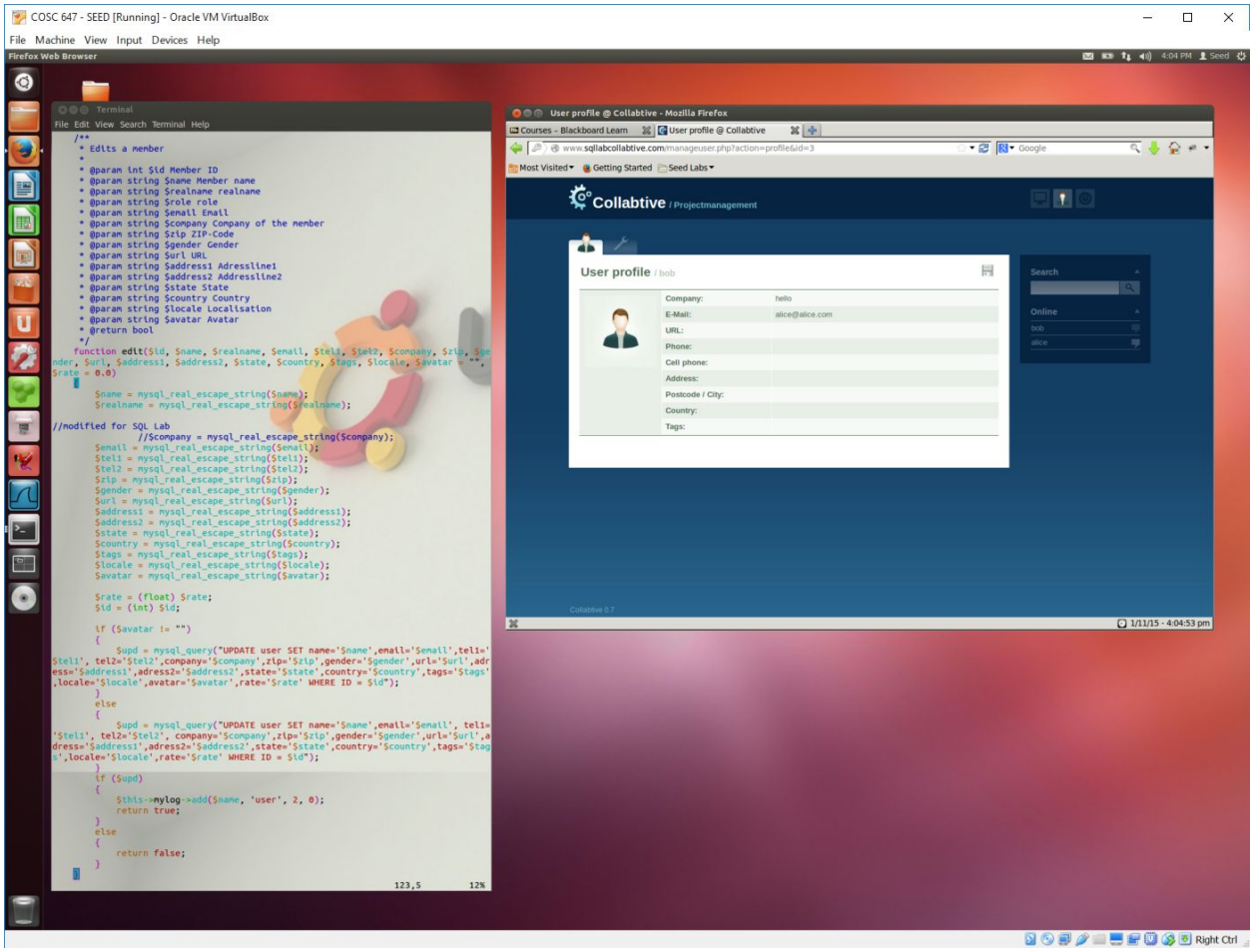
a. In the edit function, find the field that does not use the "mysql_real_espace_string" to sanitize the input for "company"
a. Open website and login as alice
b. Open my account for alice and click the wrench "edit"

c. Update the “Company” field with one of the following exploit strings:
    i. ', name='bob', company='hello' WHERE ID = 3;-- '
        1. 3 is the ID for ‘bob’, as determined by inspecting the element of the home page that contained Bob's name.
        2. name=’bob’ overwrites the first part of the update staring which sets name=’alice’. It is worth noting that the query can be shortened by omitting the name section so long as the 'User' field is changed to 'bob' in the UI. If both name=’bob’ is excluded and the user field is not changed to ’bob’, the exploit will fail due to the uniqueness constraint of the key (name).
    ii. ', company='hello' WHERE name=’bob’;-- '
        1. the ID for ‘bob’ is not necessary because the update is done only for records where name=’bob’. Since names are unique, there it is guaranteed that only the desired account will be affected.
d. Enter an e-mail address (mandatory for any edit).
    i. example: alice@alice.com
e. Press send to commit updates



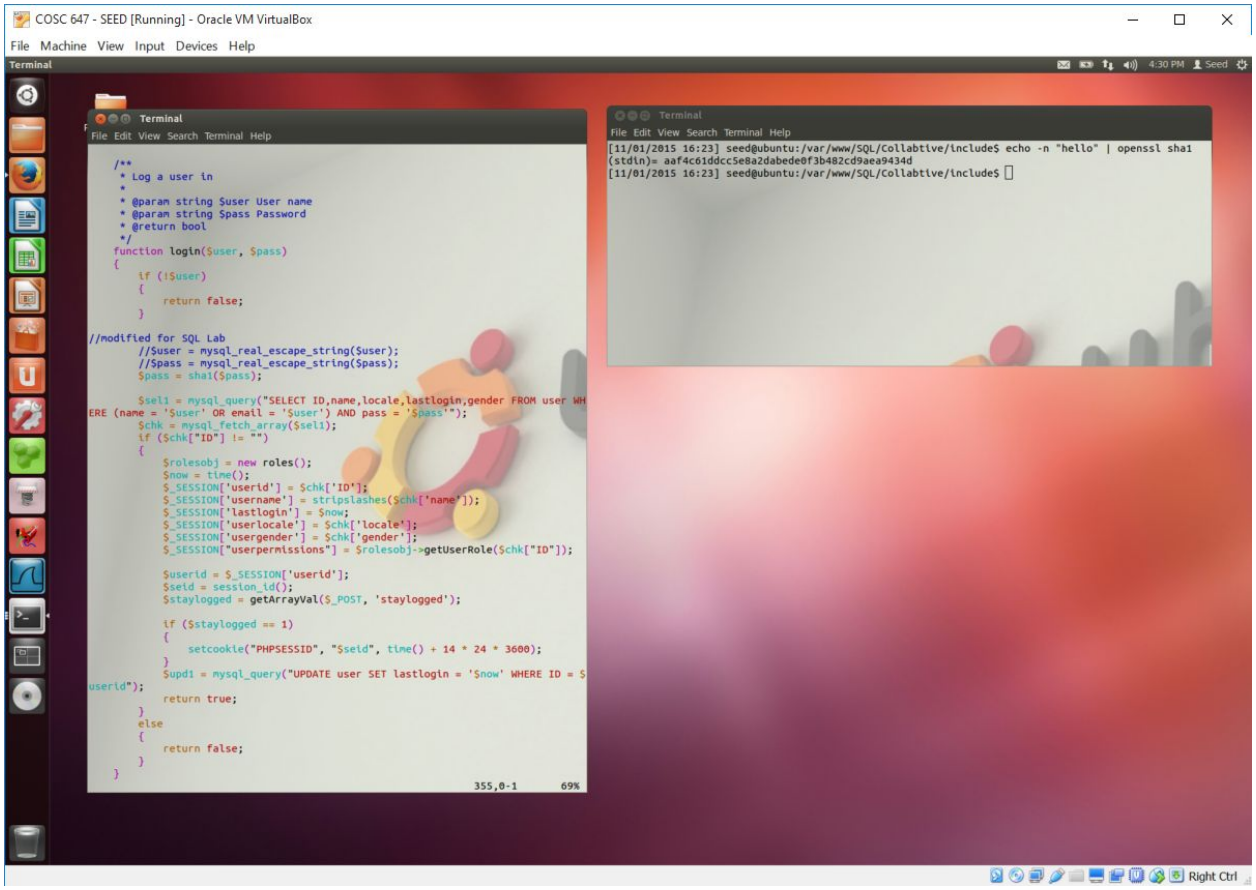
f. Logout and login as bob
g. Verify bob’s information has been changed:
    i. Company: ‘hello’

ii. E-Mail: ‘alice@alice.com’

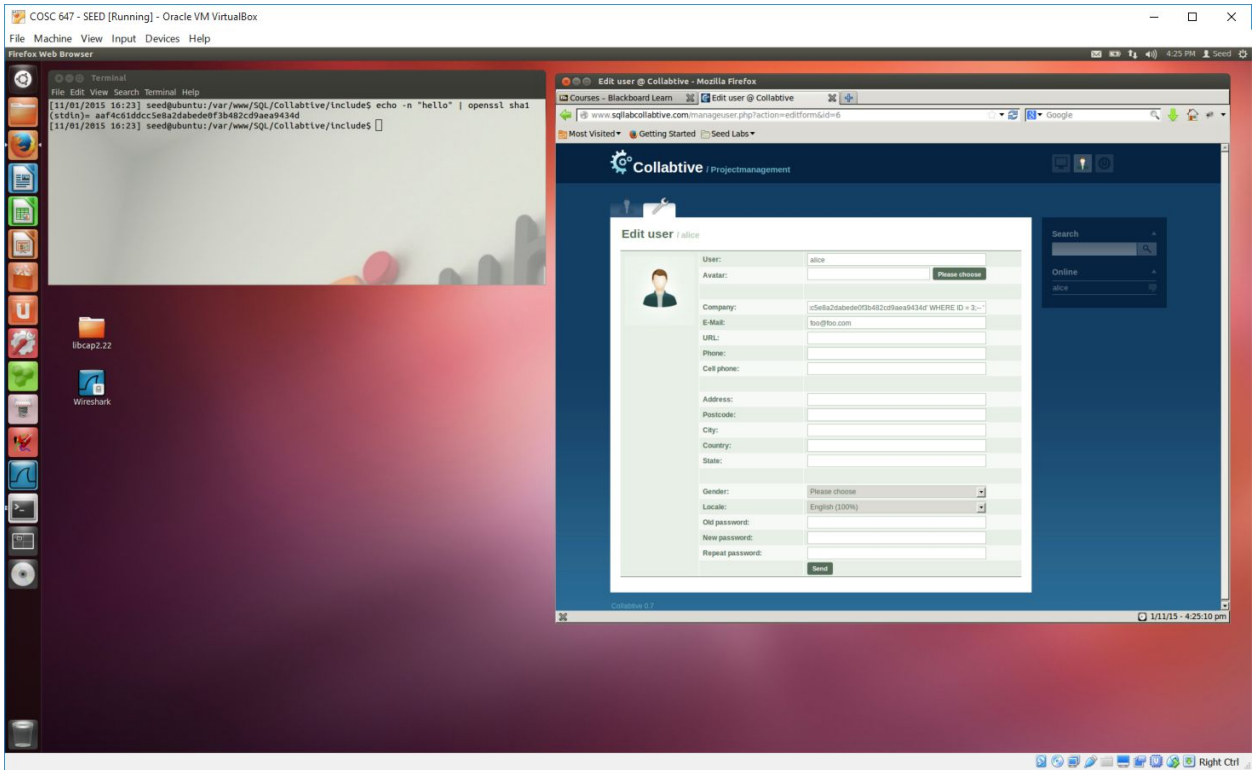


For exploit 3 (Update bob’s password from alice’s account):

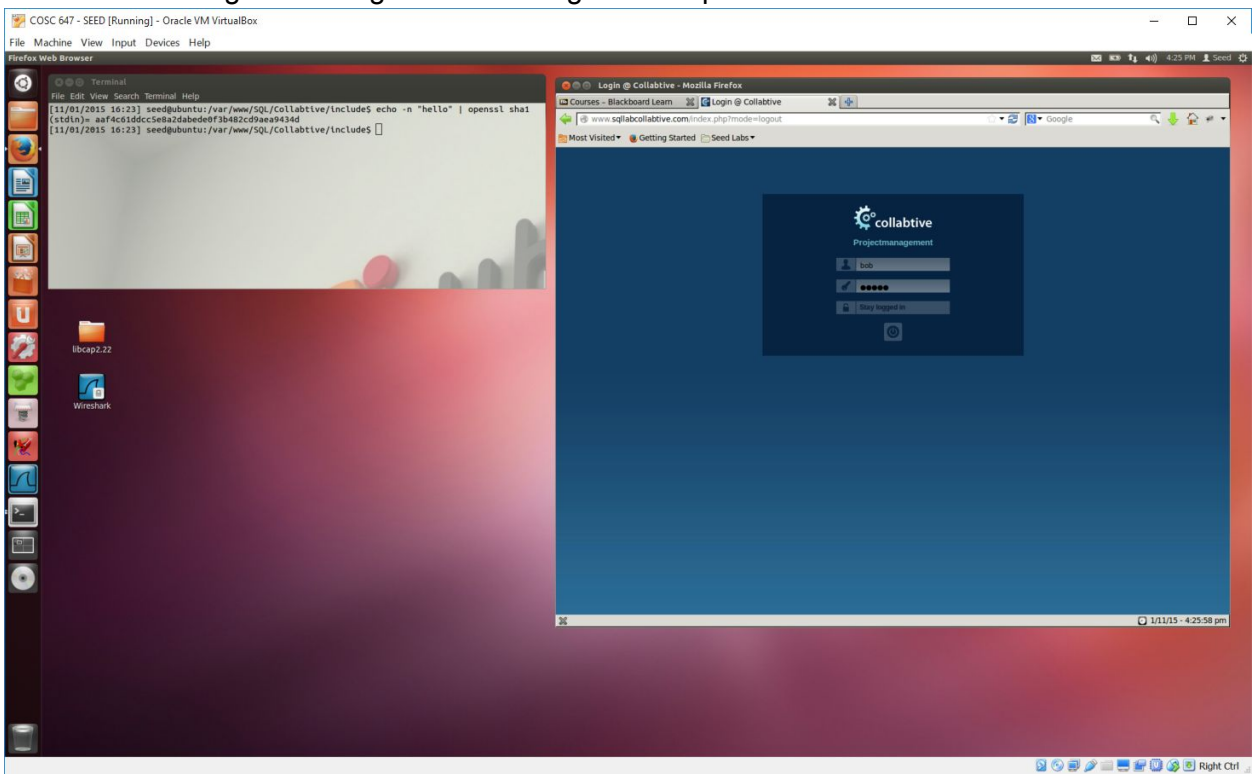
a. In the login function, notice the password is verified using the sha1 of the password string.
b. In a terminal determine the sha1 of the new password string (“hello”):
    i. echo -n “hello” | openssl sha1



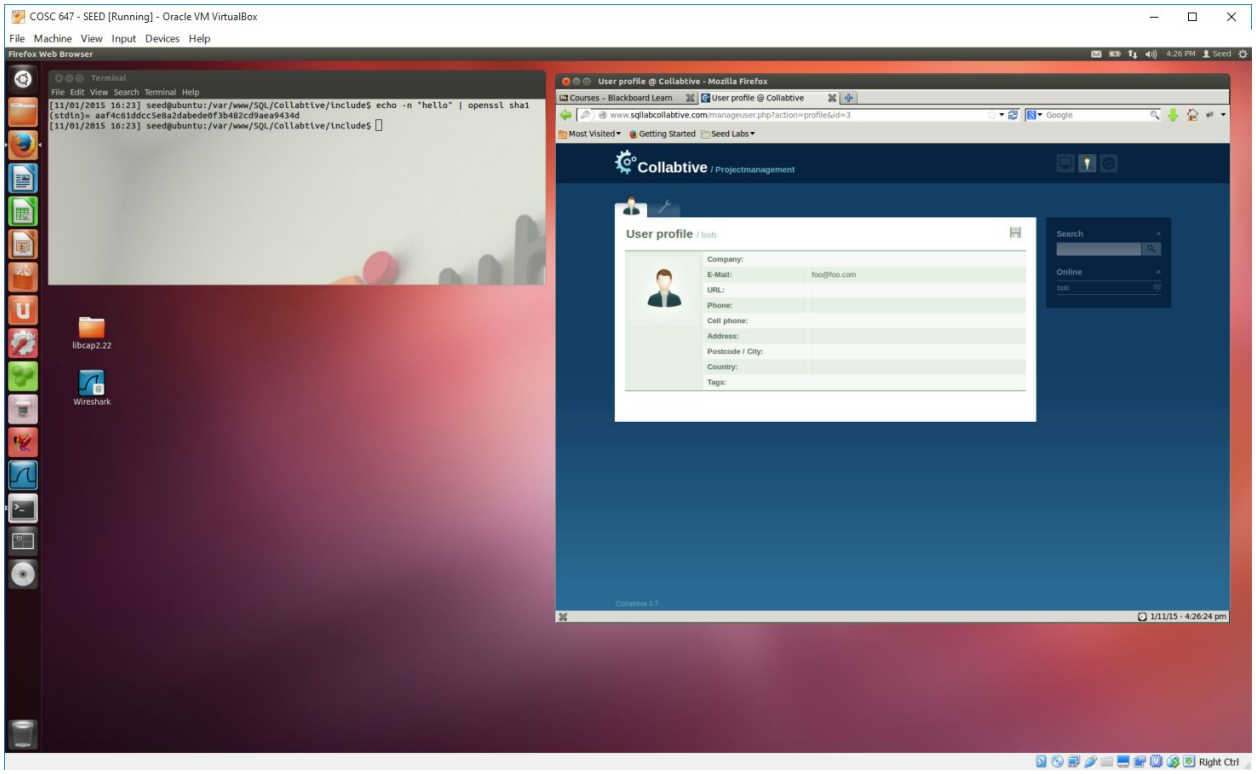
c. Open website and login as alice
d. Open my account for alice and click the wrench “edit”
e. Update the “Company” field with the following exploit strings:
    i. ', name='bob', pass=’aaf4c61ddcc5e8a2dabede0f3b482cd9aea9434d’ WHERE ID = 3;-- '
        1. 3 is the ID for ‘bob’
        2. name=’bob’ overwrites the first part of the update staring which sets name=’alice’
        3. pass is the sha1 of the password string
f. Enter an e-mail address (mandatory for any edit).
    i. example: foo@foo.com
g. Press send to commit updates



h. Logout and login as bob using the new password ‘hello’



i. Verify bob’s information has been changed:
   i. Company: ‘’
   ii. E-Mail: ‘foo@foo.com’



3. **What have you learned from this lab? Make at least 3 bullets.**
    - Unsanitized inputs can be used to change the meaning or impact of a SQL query.
    - Numerical ID's make for terrible obfuscation - they're easily guessable - and no ID should be exposed in the UI.
    - Even without access to the source code, SQL Injection Attack vectors (vulnerable inputs) can be found by causing the program to error, in this case rendering a blank screen.