Matthew Austin

Web Development

Yi Yang

December 2018

Lab #5

Tasks

Turn off PHP Magic Quote Countermeasure

Prior to version 5.3.0, PHP enabled a countermeasure to SQL injection attacks called “Magic Quote”. To disable this, we simply edit the PHP configuration file /etc/php5/apache2/php.ini and change the line magic quotes gpc = On to magic quotes gpc = Off; finally, to make these changes live, we execute the command sudo service apache2 restart. (Du 3)

Lab Task 1: Exploiting the login prompt

The Collabtive web app on our virtual machine implements its user authentication using the file/var/www/SQL/Collabtive/include/class.user.php, which executes a SQL query using the specified user and password parameters to determine if a record exists which matches the pair of inputs. If so, it authenticates the user. Unfortunately, the function looks like this:

function login($user, $pass)  
{  
 if (!$user)  
 {  
 return false;  
 }  
   
  
 $pass = sha1($pass);  
   
 $sel1 = mysql\_query("SELECT ID,name,locale,lastlogin,gender FROM user  
 WHERE (name = '$user' OR email = '$user')  
 AND pass = '$pass'");  
 $chk = mysql\_fetch\_array($sel1);  
 if ($chk["ID"] != "")  
 {  
   
 return true;  
 }  
 else  
 {  
 return false;  
 }  
}

Note specifically that the authors of the lab disabled the input sanitization of the variables $user and $pass by commenting out the execution of mysql\_real\_escape\_string() on them. This leaves us with the following SQL query executed by the PHP script, where both $user and $pass are susceptible to an injection attack:

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

Task 1.1: Can you log into another person’s account without knowing the correct password?

Yes, The key is being able to inject SQL code we want into the $user parameter, which is ahead of the “AND” conditional password check in the WHERE clause. Hence, entering ​admin'); # '​ into the user field bypasses the password check and logs us in. This is because we complete the name predicate with the user admin followed by a ​'​followed by a ), which matches the ​'​ and ( before them in the query, then add a ; to complete the query, followed by a #, which comments out the rest of the query in the original statement from the PHP code, and finally followed with a ​'​to match the trailing one surrounding our user input. We can enter anything (or nothing) for the password, and be logged in as the admin user, because the SQL code that was supposed to check the password was disabled by our attack.

Task 1.2: Can you find a way to modify the database (still using the above SQL query?)

No, we were not able to modify the database through this particular query. Attempted injecting variations of ​admin'); UPDATE user SET pass=​'pass' WHERE name=​'admin'; # '​, where we added an UPDATE or DELETE FROM query after the first, completed with a ;, and still followed by a “#”, thus splitting the original query into two separate queries. The MySQL database in particular does not allow query stacking in the mysql\_query() function. execute two queries sequentially in the same mysql\_query() function call, MySQL itself causes the call, and therefore our attack, to fail. Note that this is not necessarily true of other PHP database extensions, which may or may not permit such query stacking. Since the vulnerable query begins with a SELECT statement, and the attack is only injectable further down in the query’s string, MySQL’s defense mechanism limits us to only exploiting the SELECT query.

Lab Task 2: SQL Injection on UPDATE Statements

The Collabtive web app also has a SQL injection vulnerability elsewhere in the /var/www/SQL/Collabtive/include/class.user.php file. On the “My Account” page, there is an “Edit” link that presents the user with a form for editing their profile’s information. This form calls the following function:  
  
function edit($id, $name, $realname, $email, $tel1, $tel2, $company,  
 $zip, $gender, $url, $address1, $address2, $state,  
 $country, $tags, $locale, $avatar = "", $rate = 0.0)  
{  
 $name = mysql\_real\_escape\_string($name);  
 $realname = mysql\_real\_escape\_string($realname);  
  
 $email = mysql\_real\_escape\_string($email);  
  
  
  
 $rate = (float) $rate;  
 $id = (int) $id;  
  
 if ($avatar != "")  
 {  
 $upd = mysql\_query("UPDATE user SET name='$name', email='$email',  
 tel1='$tel1', tel2='$tel2', company='$company',  
 zip='$zip', gender='$gender', url='$url',  
 adress='$address1', adress2='$address2',  
 state='$state', country='$country',  
 tags='$tags', locale='$locale',  
 avatar='$avatar', rate='$rate' WHERE ID = $id");  
 }  
 else  
 {  
  
 }  
 if ($upd)  
 {  
 $this->mylog->add($name, 'user', 2, 0);  
 return true;  
 }  
 else  
 {  
 return false;  
 }  
}

the variable $company, which had previously been escaped, has been modified by this lab’s authors so that it is susceptible to an injection attack, as it is no longer escaped. the previously mentioned MySQL defense against query stacking, instead of using the statement ​SELECT ID WHERE name=​'ted'​ as a subquery to return the ID of the user “ted” in the attack itself (so that we can UPDATE the table row corresponding exactly to his account), we had to manually experiment to find that his ID is 4. Third, since passwords should never be stored as plaintext, we had to further inspect the source code of the web application to determine that it is using an unsalted SHA1 hash of the password to represent it somewhat securely in the database.to set the password to ‘pass’, since we could not call the PHP sha1() function within the query, we had to manually hash the string ‘pass’, to obtain the SHA1 hash of ‘9d4e1e23bd5b727046a9e3b4b7db57bd8d6ee684’.

Combining this information with that learned from our previous attack (such as matching quotes where needed, and commenting out parts of a query we wish to overwrite), we were able to develop a successful attack, with these steps:

1. Log in as a user other than ted, since we want to change his password without his permission
2. Navigate to “My account” and click “Edit” to bring up the vulnerable form
3. Replace the user’s name in the form with ‘ted’ (so that the attack retains his name in the database)
4. Type the following query into the vulnerable “company” field: ​', `pass` = '9d4e1e23bd5b727046a9e3b4b7db57bd8d6ee684' WHERE ID = 4 # '​
5. Log out of the current user
6. Enjoy successfully logging into ted’s account with the password ‘pass’

Lab Task 3: Countermeasures

Magic Quotes

Re-enabling magic quotes by setting it to “On” in the file /etc/php5/apache2/php.ini (in the same manner that we originally used to turn it off) successfully prevents our SQL injection attacks: we were unable to replicate the login prompt exploit, nor the edit user form exploit. This is because it “magically” (read: automatically) escapes any single quote (‘), double quote (“), backslash (\), or null characters found in an input string. Although an effective countermeasure against our attacks, it has its drawbacks. Most notably, since magic quotes may or may not be enabled on any particular server, developers of PHP applications must include a check of this setting, and incorporate logic to handle both cases, lest their application not be portable to different environments. Additionally, magic quotes introduces performance penalties by escaping every input string (not just those used for user input), and can cause programming headaches when some data is not supposed to be escaped, but is regardless. For at least these reasons, magic quotes was deprecated in PHP version 5.3.0, and will not exist in PHP 6.

mysql\_real\_escape\_string()

As noted above, the SQL injection attacks were made viable specifically because of alterations made to the code by the lab’s authors. Below we present the fixed code that properly uses mysql\_real\_escape\_string() on all user inputs. Both these functions come from /var/www/SQL/Collabtive/include/class.user.php.

The mysql\_real\_escape\_string() function will escape these characters, \x00, \n, \r, \, ', " and \x1a, by prepending them with a backslash (\) in the string it returns. By including code similar to $input = mysql\_real\_escape\_string($input) for all variables that hold user input, such input will be properly escaped and, when passed into a SQL query, not cause said query to be susceptible to the attacks outlined in this lab. However, this function was deprecated as of PHP version 5.5.0, and should be replaced with prepared statements.

function login($user, $pass)  
{  
 if (!$user)  
 {  
 return false;  
 }  
   
   
 $user = mysql\_real\_escape\_string($user);  
 $pass = mysql\_real\_escape\_string($pass);  
 $pass = sha1($pass);  
   
 $sel1 = mysql\_query("SELECT ID, name, locale, lastlogin, gender  
 FROM user WHERE (name = '$user' OR  
 email = '$user') AND pass = '$pass'");  
 $chk = mysql\_fetch\_array($sel1);  
 if ($chk["ID"] != "")  
 {  
   
   
 return true;  
 }  
 else  
 {  
 return false;  
 }  
}

function edit($id, $name, $realname, $email, $tel1, $tel2, $company,  
 $zip, $gender, $url, $address1, $address2, $state,  
 $country, $tags, $locale, $avatar = "", $rate = 0.0)  
{  
 $name = mysql\_real\_escape\_string($name);  
 $realname = mysql\_real\_escape\_string($realname);  
  
  
 $company = mysql\_real\_escape\_string($company);  
 $email = mysql\_real\_escape\_string($email);  
  
 $rate = (float) $rate;  
 $id = (int) $id;  
  
 if ($avatar != "")  
 {  
 $upd = mysql\_query("UPDATE user SET name='$name', email='$email',  
 tel1='$tel1', tel2='$tel2', company='$company',  
 zip='$zip', gender='$gender', url='$url',  
 adress='$address1', adress2='$address2',  
 state='$state', country='$country',  
 tags='$tags', locale='$locale',  
 avatar='$avatar', rate='$rate' WHERE ID = $id");  
 }  
 else  
 {  
   
 }  
 if ($upd)  
 {  
 $this->mylog->add($name, 'user', 2, 0);  
 return true;  
 }  
 else  
 {  
 return false;  
 }  
}

Prepare Statement

The modern technique to preventing SQL injection attacks is the use of prepared statements, which allow a developer to separate SQL logic from user input logic. With this separation, user input types can be explicitly specified, making them strongly typed as far as the database is concerned. This process is somewhat similar to a format string in other languages.

1. The first step is to “prepare” the SQL query itself, which is done by sending a fully constructed SQL query to the database via the $stmt = $db->prepare($query) function (where $db is the database connection). Within the prepared query, possible user inputs are declared using a question mark ​input=​?​.
2. The next step is to bind those specified parameters, using bind\_param("si", $string, $int), which declares the type (string and int) for the list of parameters ($string, $int) present in the prepared query.
3. With the parameters bound, next the developer must call $stmt->execute(), to execute the prepared query.
4. To retrieve the results of the query, they must also be bound: $stmt->bind\_result($output\_1, $output\_2, ..., $output\_n), where the bound variables match the data expected to be returned from the query.
5. Finally, actually getting the query’s results requires calling $results=$stmt->fetch().

function login($user, $pass)  
{  
 if (!$user)  
 {  
 return false;  
 }  
   
  
 $stmt = $conn->prepare("SELECT ID,name,locale,lastlogin,gender FROM user  
 WHERE (name=? OR email=?) AND pass=?");  
 $stmt->bind\_param("sss", $user, $user, sha1($pass));  
 $stmt->execute();  
 $stmt->bind\_result($bind\_ID, $bind\_name, $bind\_locale, $bind\_lastlogin,  
 $bind\_gender);  
 $chk = $stmt->fetch();  
 if ($bind\_ID != "")  
 {  
  
 return true;  
 }  
 else  
 {  
 return false;  
 }  
}

function edit($id, $name, $realname, $email, $tel1, $tel2, $company, $zip,  
 $gender, $url, $address1, $address2, $state, $country, $tags,  
 $locale, $avatar = "", $rate = 0.0)  
{  
   
 $rate = (double) $rate;  
 $id = (int) $id;  
  
 if ($avatar != "")  
 {  
   
  
 $stmt = $conn->prepare("UPDATE user SET name=?, email=?, tel1=?,  
 tel2=?, company=?, zip=?, gender=?, url=?,  
 adress=?, adress2=?, state=?, country=?,  
 tags=?, locale=?, avatar=? rate=?  
 WHERE ID = ?");  
 $stmt->bind\_param("sssssssssssssssdi", $name, $email, $tel1, $tel2,  
 $company, $zip, $gender, $url, $address1,  
 $address2, $state, $country, $tags, $locale,  
 $avatar, $rate, $id);  
 $upd = $stmt->execute();  
 }  
 else  
 {  
 $stmt = $conn->prepare("UPDATE user SET name=?, email=?, tel1=?,  
 tel2=?, company=?, zip=?, gender=?, url=?,  
 adress=?, adress2=?, state=?, country=?,  
 tags=?, locale=?, rate=? WHERE ID = ?");  
 $stmt->bind\_param("ssssssssssssssdi", $name, $email, $tel1, $tel2,  
 $company, $zip, $gender, $url, $address1,  
 $address2, $state, $country, $tags, $locale,  
 $rate, $id);  
 $upd = $stmt->execute();  
 }  
 if ($upd)  
 {  
 $this->mylog->add($name, 'user', 2, 0);  
 return true;  
 }  
 else  
 {  
 return false;  
 }  
}