

Website Security Research Project

Instructional Documentation

CS 467 - Summer 2021

Team Members:

Cody Medaris

Ray Franklin

Emanuel Ramirez

Team Members:
Cody Medaris
Ray Franklin
Emanuel Ramirez

Introduction

The purpose of this project is to demonstrate common web based vulnerabilities and to teach defenses against those vulnerabilities. The website is hosted locally and should be restricted from being hosted on the internet due to the many vulnerabilities.

To start: You should navigate to localhost after setting up the environment as instructed in the Installation Documentation. You will be greeted by a listing of the various vulnerabilities as different links. Simply navigate to the page of your choice and read about that vulnerability below.

Example of the index page for navigating:

Vulnerable Site

Vulnerability 1: Injection

Vulnerability 2: Broken Authentication

Vulnerability 3: Sensitive Data Exposure

Vulnerability 4: XML External Entities

Vulnerability 5: Broken Access Control

Vulnerability 6: Security Misconfiguration

Vulnerability 7: Cross-Site Scripting - Vulnerable

Vulnerability 7: Cross-Site Scripting - Secured

Vulnerability 8: Insecure Deserialization

Vulnerability 9: Using Components With Known Vulnerabilities

Vulnerability 10: Insufficient Logging & Monitoring

Introduction Image 1: Index home page:

Vulnerability 1: Injection

HTML INJECTION:

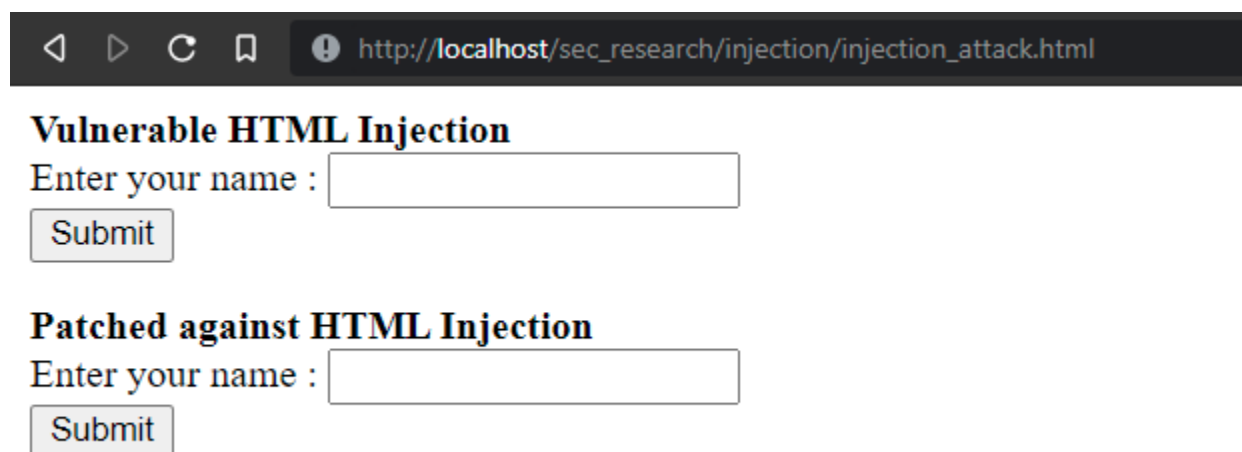
How it works:

Infection attacks work by inserting external data, often malicious data, to an application which then interprets and executes that data. Applications that do not validate or sanitize user input are vulnerable to injection attacks. In this project we are going to be focusing on SQL and HTML injection attacks. In SQL injection attacks, the attacker will try to manipulate the SQL queries used in the web application in order to gain direct access to the data stored in the database. This can put at risk private information from the user of the web application. On the other hand, HTML attacks are typically used to manipulate the web application interface by inserting unwanted images or messages into the main interface of the web application.

In the site

In the injection section of the website there is a html page that asks for the user's name. One form (the top-most one in *image1*) is a vulnerable form that does not validate nor sanitizes the user's input. Hence the form is vulnerable to any kind of attack, especially HTML injection attacks. In this case the user can insert the following html line `<p style=%22color:red;margin-left:20px;font-size:80pt;>YOU HAVE BEEN HTML INJECTED</p>`, shown in *image2*, in the vulnerable form. The result of this query can be seen in *image3*. The vulnerable code is shown on *image4*.

HTML Injection Image 1: Injection attack dashboard



Vulnerable HTML Injection
Enter your name :

Patched against HTML Injection
Enter your name :

HTML Injection Image 2: attacker input containing and HTML <p> tag



Vulnerable HTML Injection
Enter your name :

Patched against HTML Injection
Enter your name :

HTML Injection Image 3: HTML injection result



HTML Injection Image 4: Vulnerable piece of code

```
injection > submit.php
1  <?php
2  // This is vulnerable to an HTML attack, as it doesn't validate
3  // nor sanitizes user's input.
4  echo "Your name is: " . $_GET['name'];
5  ?>
```

Defenses:

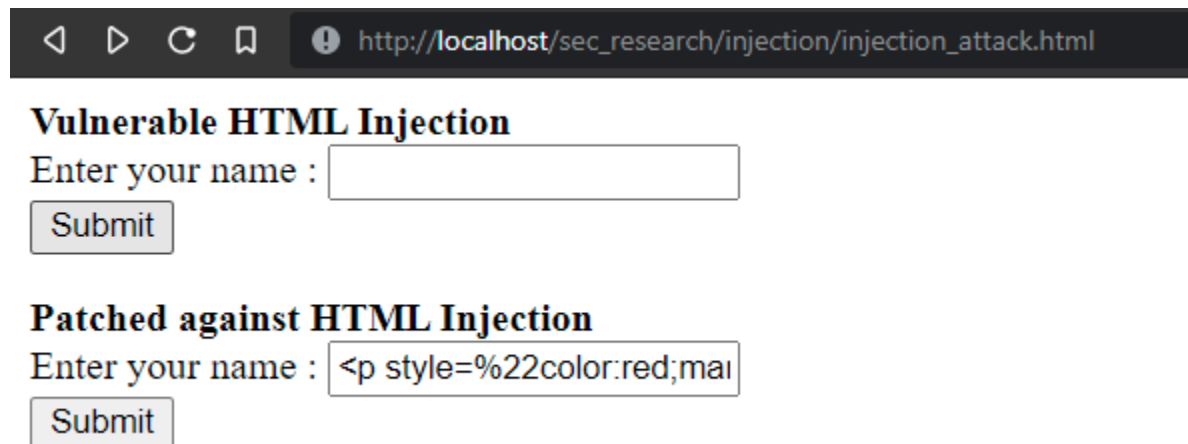
In order to prevent HTML injection attacks the developer of the web application must validate and sanitize any user input. Having a helper function to remove any possible HTML tags present in the user input will suffice in this case. *Image 5* shows the code behind the patch for

the above presented HTML injection attack and *image 6* and 7, show the patched input validation result.

HTML Injection Image 5: Patched HTML injection code

```
injection > submit_patched.php
1  <?php
2      $name = $_GET['name'];
3      echo "Your name is: ".remove_tags_from_name($name);
4
5      // Helper function that validates and sanitizes the user input
6      function remove_tags_from_name($name) {
7          $fixed_name = strip_tags($name);
8          if (strlen($fixed_name) == 0) {
9              return "ERROR PARSING NAME";
10         } else {
11             return $fixed_name;
12         }
13     }
14  ?>
```

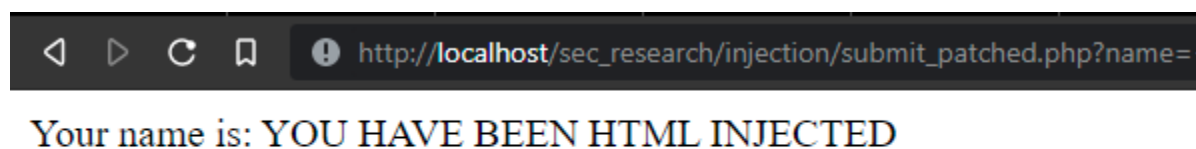
HTML Injection Image 6: HTML user input in patched HTML vulnerability form



Vulnerable HTML Injection
Enter your name :

Patched against HTML Injection
Enter your name :

HTML Injection Image 7: Query result showing the user HTML input as a string which doesn't render the website usability like it does on image 3



Your name is: YOU HAVE BEEN HTML INJECTED

SQL Injection Coming Soon.

Vulnerability 4: XML External Entities (XXE) Attack

How it works:

XML allows entities to be defined within a DOM structure, and be referenced later on in the structure. Using the SYSTEM command, the content of a file somewhere else on the machine can be used as the content of the XML entity. An XXE Vulnerability arises whenever an adversary can define and use their own XML entities to access resources stored on the server in normally inaccessible areas.

In the site:

In the site, there is a page called 'faux-definition.php' that has a mock functionality of obtaining a definition of a word by submitting an XML request that looks similar to the following:

```
<? xml version="1.0" encoding="UTF-8"?>
```

```
<define>
```

```
<word> myword </word>
```

</define>

The correct usage would be through a script like:

```
<script>
function subdata() {
    var xml = "<?xml version='1.0' encoding='UTF-8'?><define><word>word</word></define>"
    var req = new XMLHttpRequest();
    req.open("POST", "/tests/dumbpage.php", true);
    req.setRequestHeader("Content-Type", "text/xml");
    req.addEventListener("load", function() {
        console.log(req.responseText)
    });
    req.send(xml);
    event.preventDefault();
}
</script>
```

XML External Entities (XXE) Attack Image 1: XML request.

To get a result like:

```
Sending server: <?xml version='1.0'      randpage.html:10
encoding='UTF-8'?><root><thing>word</thing></root>

Server sent back: definition of word      randpage.html:14
```

XML External Entities (XXE) Attack Image 2: XML result.

But if an attacker wanted, they could use an external entity to attempt to access the server's htpasswd file by using the following script, found in the 'xxe-attack.html' file:

```
<script>
function subdata() {
    var xxe = "<!DOCTYPE foo [<!ENTITY xxe SYSTEM \"../../.htpasswd\" >]>"
    var xml = "<?xml version='1.0' encoding='UTF-8'?>"+xxe+"<root><thing>&xxe;</thing></root>"
    var req = new XMLHttpRequest();
    console.log("Sending server: "+xml);
    req.open("POST", "/tests/dumbpage.php", true);
    req.setRequestHeader("Content-Type", "text/xml");
    req.addEventListener("load", function() {
        console.log("Server sent back: "+req.responseText);
    });
    req.send(xml);
    event.preventDefault();
}
</script>
```

XML External Entities (XXE) Attack Image 3: XML attack script.

And obtain the following result in their browser after executing:

Server sent back: user1:password1 user2:password2 user3:password3 is not a word

XML External Entities (XXE) Attack Image 4: Browser result.

By defining the entity xxe using SYSTEM ../../.htpasswd and submitting it to the server, when the server parses the XML, it creates the entity xxe and places the contents of the .htpasswd file within the entity, even though the file is located outside of the public directory of the site. The site then displays an error message that reveals the content of the file to the attacker.

Defenses:

Coming soon

Vulnerability 5: Broken Access Controls

How it Works:

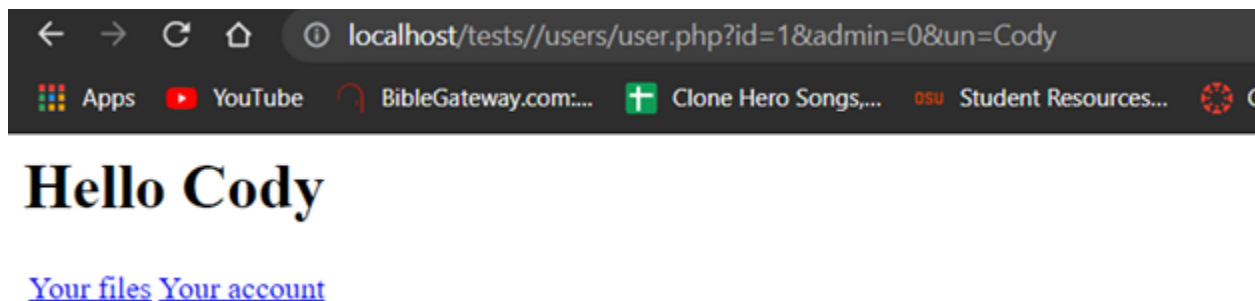
Every business needs to ensure that only trusted individuals can access its sensitive information, and to ensure that personal information remains private. This is done through access control. Sometimes, though, users who shouldn't have access to certain information or privileges may be able to obtain those privileges or imitate a different user who does have those privileges.

In the site:

The site contains two examples of broken access control. The first and most easily exploitable is that a user's credentials are submitted to the /user.php page through a query in the URL. So an attacker can easily change the query from something like

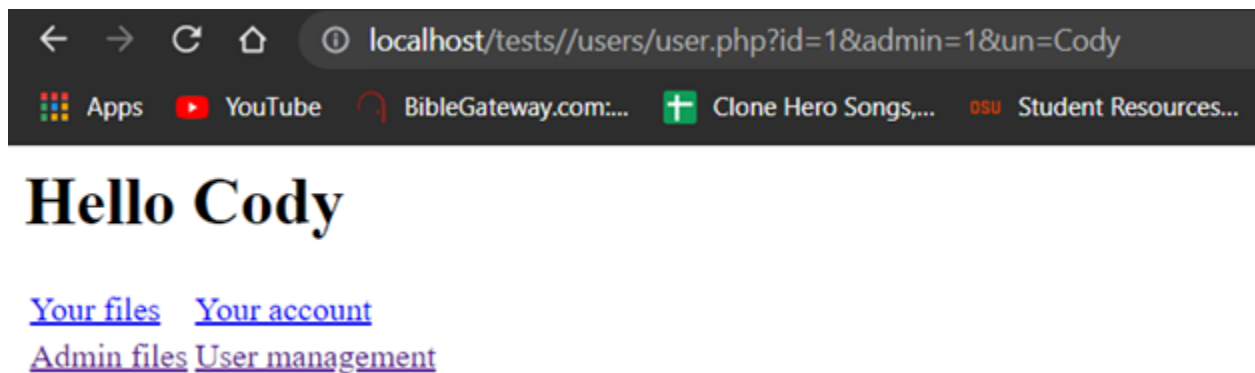
Team Members:
Cody Medaris
Ray Franklin
Emanuel Ramirez

/user.php?id=123&admin=0 to /users?id=123&admin=1 to gain access to administrative privileges, or to /user.php?id=456&admin=0 to gain access to another person's information. An example of what a regular user's page should look like:



Broken Access Controls Image 1: Regular user page.

Notice that the query parameter 'admin' in the URL is 0. If I go to the address bar and change it to 1, I get access to admin pages:

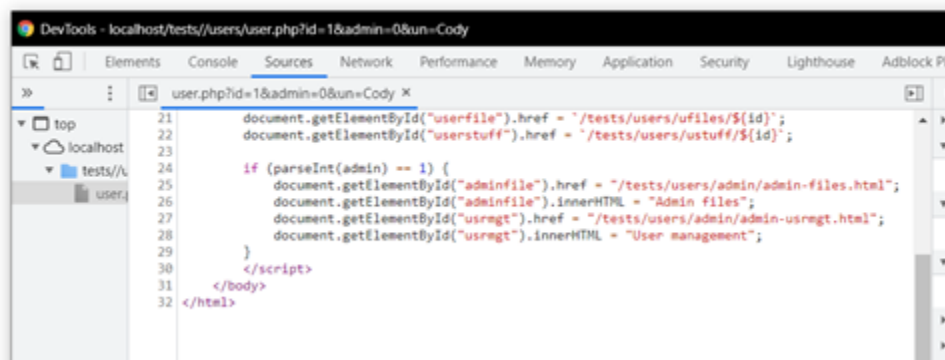


Broken Access Controls Image 1: Admin page.

The other issue is that the links for pages containing functionality meant for admins are given away by the javascript code used to add links to those pages to the admin's home page. Anyone visiting /user.php could open their browser's dev tools and see those links:

Hello Cody

[Your files](#) [Your account](#)



Broken Access Controls Image 3: Admin links.

Here I see that /tests/users/admin/admin-files.html and /admin-usrmgt.html would be linked on the admin page, so I simply type the URL localhost/tests/users/admin/admin-usrmgt.html into the address bar and mess around with user info.

Defenses:

Coming soon

Vulnerability 6: Security Misconfiguration

How it works:

Security misconfigurations can happen in a variety of ways. Maybe you password protected a directory but you used the easily guessed username admin and a password of, well, password. Or maybe you didn't disable directory indexing and so the whole world can see all the files on your server. Not to mention the improper handling of untrusted input that can lead to Injection attacks, XXE attacks, and XSS attacks.

In the Site:

When you think about it, all the vulnerabilities in the site can be considered the results of security misconfigurations. Rather than repeat what has already been mentioned in those vulnerabilities' write-ups, I wanted to focus on the ways that directories and files can be left

exposed to anyone who wants to see them. If we go to the security misconfiguration page of the site, we see something like:

Index of /tests/sec-mis

<u>Name</u>	<u>Last modified</u>	<u>Size</u>	<u>Description</u>
 Parent Directory		-	
 home-page.html	2021-07-15 21:25	76	
 scripts/	2021-07-15 21:21	-	
 secretstuff/	2021-07-15 21:23	-	

Security Misconfiguration Image 1: Exposed directories.

Instead of displaying the home-page.html file like intended, the user gets to see the files and directories in the sec-mis folder, including ones that are meant to be secret. This gives an attacker a glimpse into the structure of the site, and if the “secretstuff” directory is unprotected (and it is), they could get access to sensitive information:

Index of /tests/sec-mis/secretstuff

<u>Name</u>	<u>Last modified</u>	<u>Size</u>	<u>Description</u>
 Parent Directory		-	
 secret.txt	2021-07-08 01:47	18	
 userlog.txt	2021-07-15 21:23	36	

Security Misconfiguration Image 2: Unprotected files.

Defenses:

Coming Soon

Vulnerability 7: Cross Site Scripting (XSS)

How it Works:

Cross site scripting is a common vulnerability where the attacker injects script based code into a website. The browser interprets the code and executes the attacker's script. There are two basic styles of XSS and one lesser common version.

- **Stored XSS Attacks:** This occurs when the script is stored on the server and the script is run each time the data is retrieved from that location..
- **Blind Cross-Site Scripting:** Similar to a stored attack, but the stored information is only triggered from the backend application, usually when accessed by an administrator.
- **Reflected XSS Attacks:** This is an attack that is non-persistent. It is not stored in the server, but rather it is sent to the server then executed once the code returns to the client. Often the attacker will use a malicious link via email, the goal is to trick the user into clicking this link, which will send the script to the server and execute when reflected back to the user.

In the Site:

There are two pages for cross-site scripting examples, one vulnerable and one secured. The vulnerable page contains a search field and the user can enter any text they like into it. Here we see the site and it's displayed hint.

Please search below.

You searched for:

Hint, try searching for: `<script>alert('XSS attack successful!')</script>`

This website was made for educational purposes.

See below for further reading-

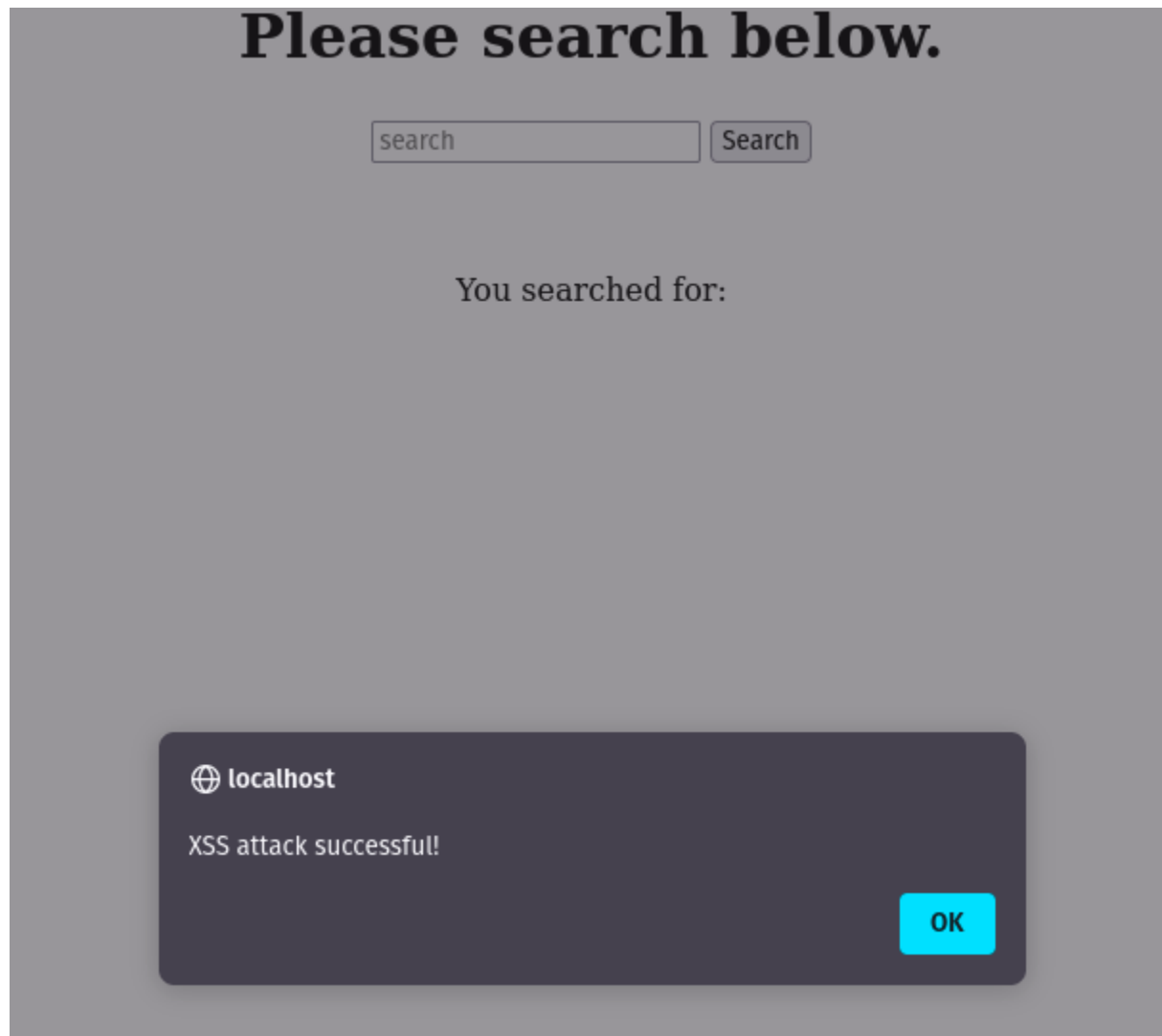
- OWASP: <https://owasp.org/www-community/attacks/xss/>
- Wikipedia: https://en.wikipedia.org/wiki/Cross-site_scripting

XSS Image 1: Vulnerable search field.

When the user enters a request in the field, the browser reads the request and interprets it accordingly. Usually this would be simple text, but in our case we're going to use the browsers built-in ability to interpret script commands. Here's the input we will use.

```
<script>alert('XSS attack successful!')</script>
```

The browser sees the script tag and executes the code within. Here's the exploited attack succeeding:



XSS Image 2: Successful attack.

Instead of displaying the text that we searched for, when we pressed the search button, the code was executed by the browser. Now that we know the browser will run the script we enter, it could be rewritten to contain malicious code that steals user data like session cookies and would allow the attacker to login as that user. There are a lot of other creative and damaging ways an attacker could successfully craft an attack to capitalize on this script vulnerability.

Defenses:

Detection: Any user input from a web application will be subject to XSS attacks unless proper actions are taken to secure the input. This means the vast majority of inputs will be vulnerable by default, and preventative measures will need to be taken. You can perform some simple tests by sending one of the many filter evasion methods found on OWASP's website. It will provide a large list of possible scripts to try and see if your site is vulnerable. Effectively, if the script successfully runs, your site is not secure.

Solution: The best method to prevent XSS is by sanitizing inputs. This is performed by utilizing various functions built into the coding languages themselves. Using these security encoding libraries will help filter out any special characters that allow the attacker to execute scripts. If you must allow some level of special characters in your inputs, it can be done such that you disallow all of them by default and add in only the ones you need to allow. This does open your site to a potential attack, but the likelihood decreases significantly by using this method.

Here's the secured version where the Javascript does not get executed:

Please search below.

You searched for: `<script>alert('XSS attack successful!')</script>`

Attack didn't work? Check out [OWASP](#) it has a list of all common filter evasion methods.

This website was made for educational purposes.

See below for further reading-

- OWASP: <https://owasp.org/www-community/attacks/xss/>
- Wikipedia: https://en.wikipedia.org/wiki/Cross-site_scripting

XSS Image 3: Secure site.

The script was entered as previously done, but this time the browser intercepted and changed the request to turn it into benign text. The Javascript alert box was not triggered. This was accomplished by sanitizing the user input before the browser could interpret it.

Further prevention: To further prevent threats, the developer should have the mindset that all user data is not to be trusted. If you assume every input from a user is malicious, you may seem a bit paranoid, but the reality is that you will be one of the few who are building secure sites. This does mean every bit of information input into the application will need to be carefully reviewed and controlled. Through completely managing the user inputs before performing any actions with the information you can build a secure application.

References for sources used:

1. "OWASP Top Ten," OWASP. [Online]. Available:

<https://owasp.org/www-project-top-ten/>. [Accessed: 19-Jul-2021].

Team Members:
Cody Medaris
Ray Franklin
Emanuel Ramirez

2. *Cross Site Scripting Prevention Cheat Sheet*. Cross Site Scripting Prevention - OWASP Cheat Sheet Series. (n.d.).
https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html#Why_Can_I_Just_HTML_Entity_Encode_Untrusted_Data.3F.
3. DrapsTV. (2015, January 22). *XSS Tutorial #1 - What is Cross Site Scripting?* YouTube.
https://www.youtube.com/watch?v=M_nIcKTxGk.
4. Wikimedia Foundation. (2021, July 10). *Cross-site scripting*. Wikipedia.
https://en.wikipedia.org/wiki/Cross-site_scripting.
5. *XSS Filter Evasion Cheat Sheet*. OWASP. (n.d.).
<https://owasp.org/www-community/xss-filter-evasion-cheatsheet>.