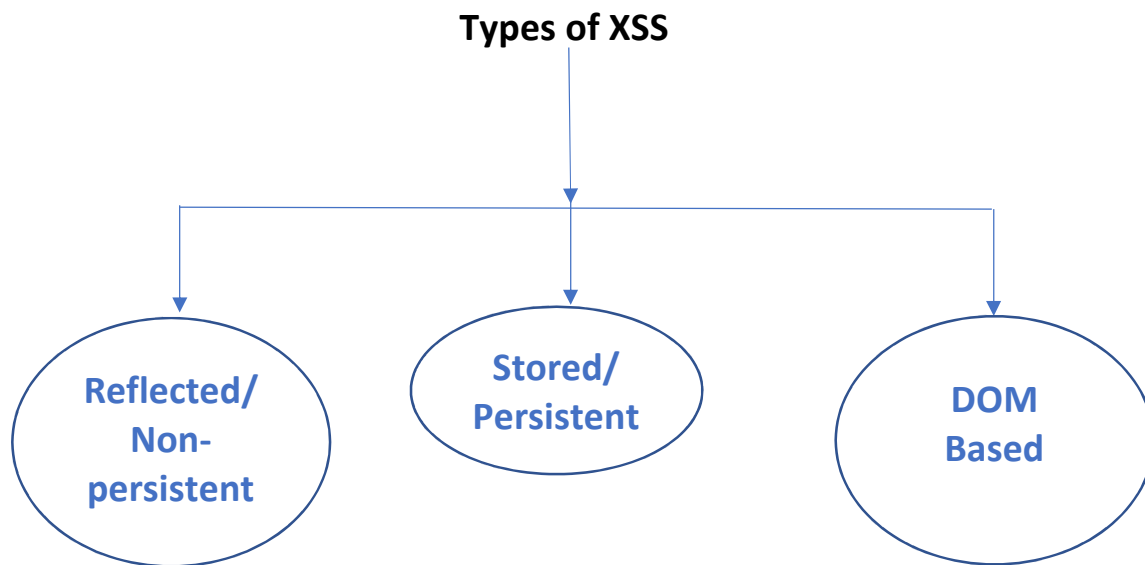


XSS Attacks

This article is all about learning XSS attacks and the ways to remediate them. It will assist the developers in understanding how actually the attackers look for the loopholes which they can exploit and get access of the Web application.

What is XSS?

Cross-Site Scripting is a type of injection problem in which malicious scripts (vb, js etc.) are injected into a trusted web site. XSS flaws occur whenever an application takes untrusted (typically user supplied) data and sends it invalidated or unencoded to a web browser. XSS allows attackers to execute script in the victim's browser and the malicious script can access any cookies, session tokens, or other sensitive information retained by our browser. Used with that site, they can even rewrite the content of the HTML page. It basically exploits the trust that a client browser has for the website. In XSS the user trusts that the content being displayed in their browser was intended by the website to be displayed.



1- Reflected or Non-persistent:

The reflected cross-site scripting is the most common type of the XSS flaw found in the web applications today.

- Reflected XSS is also called **Type-II XSS**.
- This type of XSS is reflected (executed in the browser) as soon as the injected code is processed by the web browser.


In this kind of flaw, the injected code is reflected off the web server, such as in a search result, an error message, or any other response that includes complete or partial input sent to the server as part of the request.

- The attacker needs to deliver the attack to the victims via another route, such as in an e-mail message, or on some other web server.
- The attacker sends the malicious link to the victim and when the victim is tricked into clicking on a malicious link or submitting a specially crafted form (through Social Engineering etc.), the injected code travels to the vulnerable web application's server, which reflects the attack back to the victim's browser.
- The browser then executes the malicious code as it came from a "trusted" server.

NOTE: One thing to note here is that the code injected is processed by the web server application and the response contains our injected code, and the injected code is not stored in the web application.

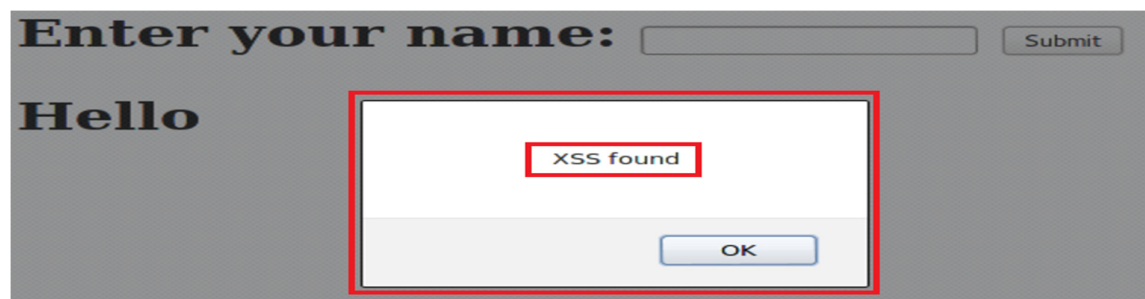
Example:

Without Reflected XSS:



A screenshot of a web form with a dark gray background. At the top, it says "Enter your name:" in a large, bold, black serif font. To the right of this text is a white text input field containing the word "Joy". The input field has a thin red border. To the right of the input field is a "Submit" button with a light gray background and a thin black border. Below the input field, the text "Hello Joy" is displayed in a large, bold, black serif font.

With Reflected XSS:



A screenshot of a web form with a dark gray background, similar to the one above. At the top, it says "Enter your name:" in a large, bold, black serif font. To the right of this text is an empty white text input field. To the right of the input field is a "Submit" button with a light gray background and a thin black border. Below the input field, the text "Hello" is displayed in a large, bold, black serif font. In the center of the form, there is a white rectangular dialog box with a thin red border. Inside the dialog box, the text "XSS found" is displayed in a small, black, sans-serif font. Below the text is an "OK" button with a light blue background and a thin black border.

2- Stored or Persistent:

The stored XSS vulnerability is a more devastating variant of a cross-site scripting flaw.

- Stored XSS is also called **Type-I XSS**.

In Stored kind of flaw,

- The attacker can inject malicious code into the vulnerable application and the injected code is permanently stored on the target servers (in a db, comment field, visitor log, etc.)
- Then permanently displayed on “normal” pages returned to other users during regular browsing.
- An attacker’s malicious script is rendered automatically, without the need to individually target victims or lure them to a third-party website.
- This is a dangerous flaw as any data received by the vulnerable web application (via email, system logs, etc.) that can be controlled by an attacker could become an injection vector.

Example:

- An attacker leaving a malicious script on a blog’s comment field of a vulnerable blogging web application.
- The malicious script will execute in the browsers of the users who will visit the blog later.

Without Reflected XSS:

Vulnerability: Stored Cross Site Scripting (XSS)

Name *

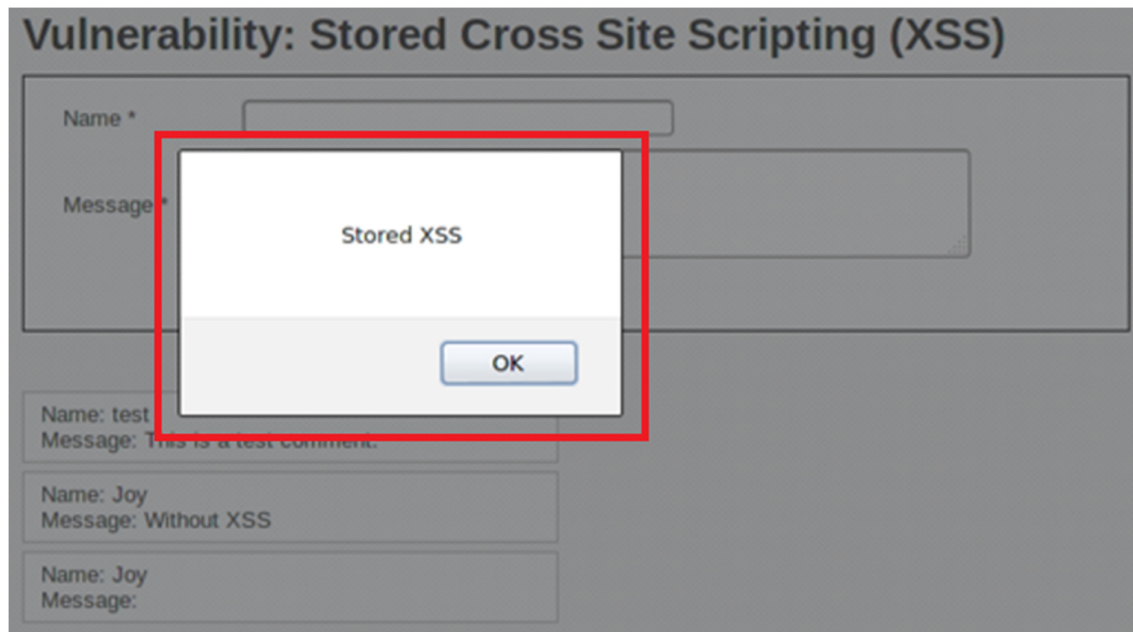
Message *

Sign Guestbook

Name: test
Message: This is a test comment.

Name: Joy
Message: Without XSS

With Reflected XSS:



3- DOM based:

DOM is a World Wide Web Consortium (W3C) specification, which defines the object model for representing XML and HTML structures.

- DOM Based XSS is also known as **type-0 XSS**.

It is an XSS flaw wherein the attack payload is executed as a result of modifying the DOM “environment” in the victim’s browser used by the original client-side script, so that the client-side code runs in an “unexpected” manner.

- DOM-based XSS is not a result of vulnerability within a server side script
- It is because of an improper handling of user supplied data in the client-side JavaScript.
- DOM-based XSS can be used to steal confidential information or hijack the user account.
- This type of vulnerability solely relies upon JavaScript and insecure use of dynamically obtained data from the DOM structure.

Modern web applications widely use Ajax technology to display and update important data without reloading the page.

Example:

- A live chat application which displays status of each contact in user's list.
- Users can use custom messages to display their status.
- The list of users and their statuses is constructed by a server-side script and passed to each chat member via a JSON object.

JSON object:

```
{"OnlineUsers": "4", "UserAndStatus": ["User1, Online", "User2, Online", "User3, Online", "User4, Busy"]}
```

JavaScript Code:

```
var http_request = new XMLHttpRequest();
var Contacts;
http_request.open("GET", url, true);
http_request.onreadystatechange = function ()
{
    if (http_request.readyState == 4)
    {
        if (http_request.status == 200) {
            Contacts = eval("(" + http_request.responseText + ")");
        }
        http_request = null;
    }
};
http_request.send(null);
```

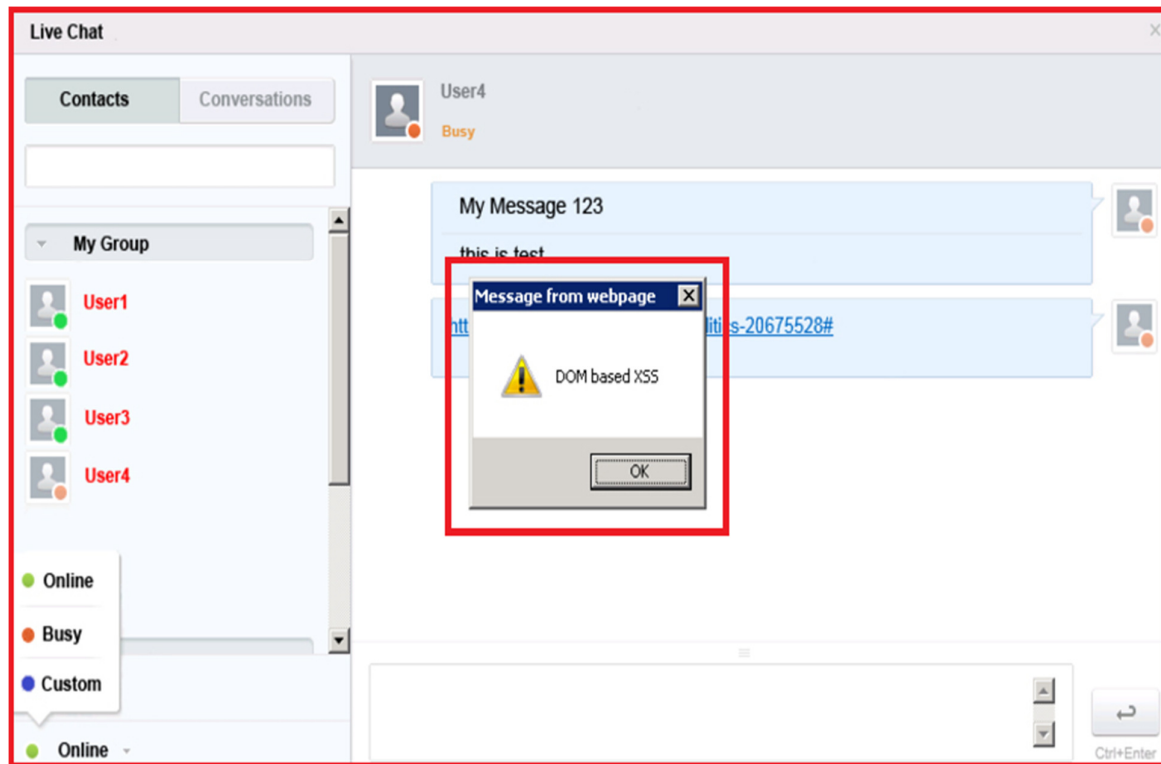
- The JavaScript **eval()** function is essential for JSON data.
- In case of improper input validation it is very easy to exploit.

Attacker's Approach and Malicious Input:

Attacker inputs the custom user status and injects the following string into JSON object:

```
Busy"}};alert("DOM based XSS");//
```

and execute **alert()** function in browsers of every chat member.



Remediation Techniques

There are basic rules that should be followed to protect application against XSS attacks,

A- Never insert untrusted input

1- directly in the script:

```
<script>UNTRUSTED INPUT</script>
```

This way it is impossible to sanitize data, because any input passed inside the script tag is treated as script by the browser.

2- directly in CSS:

```
<style>UNTRUSTED INPUT</style>
```

An attacker can use CSS to load and execute arbitrary script code in user's browser.

3- in a tag name:

```
<UNTRUSTED INPUT src="/images/...">
```

4- in an attribute name:

`<div UNTRUSTED INPUT ...=123>`

5- in an attribute value:

` (javascript:)`

6- inside HTML comment:

`<!-- UNTRUSTED INPUT -->`

Input in the above locations cannot be sanitized correctly and can be potentially used to perform cross-site scripting attacks.

7- Directly into HTML page.

8- Inside JavaScript event.

B- Perform sanitation of input data before inserting it into the page content

Any character that might be treated by the web browser as HTML content should be sanitized.

Character	Replacement
<	<
>	>
"	"
'	'
&	&
/	/

- Developers should use URL-encoding on input inserted into URL tags.
- When untrusted input is inserted into script or event all quotation symbols and backslashes should be escaped.
- All properly escaped data within JavaScript code should be included into quotes.
- When usage of untrusted input as strings or names within scripts or events is crucial for application design, this input should be sanitized and contain strictly letters and numbers.

C- Use native API and additional software whenever possible

- **ESAPI (The OWASP Enterprise Security API)** is a free, open source, web application security control library developed by OWASP, that makes it easier for programmers to write lower-risk applications.
- The ESAPI libraries are designed to make it easier for programmers to retrofit security into existing applications.
- The ESAPI libraries also serve as a solid foundation for new development.

Example:

- PHP has native API that can help protect application from XSS attacks.
- Developers can use **htmlspecialchars()** or **htmlentities()** functions to deal with untrusted input.
- **Microsoft web protection library** is also a great set of .NET assemblies that should be used when developing applications in .NET.

D- Always use preset character encoding of the displayed page

- Never rely on browser auto-select encoding functionality.
- An attacker might be able to bypass sanitation checks and perform successful XSS attacks if page encoding is not preset and user's browser is configured to use auto-select encoding.

E- Using Web Application Firewall (WAF)

- Web Application Firewall can be an efficient solution to prevent vulnerability exploitation while you are developing or waiting for a security patch.
- We will use an open source web application firewall **ModSecurity** developed by Trustwave.
- There are many rule sets for ModSecurity licensed under ASLv2 and widely distributed by security companies and organizations.
- These rule sets can be applied to cover all basic cases of vulnerabilities' exploitation and can be used on production servers.
- As a temporary solution to block a known XSS attack vector you can use the following universal ModSecurity rule that allows only digits and letters in the vulnerable parameter <PARAM>:

```
SecRule ARGS:<PARAM> !^[a-zA-Z0-9]+$ "block,phase:2,msg:'Possible XSS attack'"
```


Conclusion

With the advent of Web 2.0, Web Applications have significantly improved in functionality as well as presentation but at the same time the risk factor has also increased.

The ignorant implementation of these technologies impacts the functionality as well as the behaviour of the application which poses significant risk for the end user.

The risk becomes even graver when another flaw in a component is fused with it (in this case XSS + CSRF).

Thus, to be secure, the developers as well as the end users should take due precautions when dealing with web applications.

Key Notes

For further reading, below references could also be referred for better and in-depth reading purposes.

- 1- [https://www.owasp.org/index.php/Cross-site_Scripting_\(XSS\)](https://www.owasp.org/index.php/Cross-site_Scripting_(XSS))
- 2- <http://ha.ckers.org/xss.html>
- 3- <http://www.steve.org.uk/Security/XSS/>
- 4- [https://www.owasp.org/index.php/XSS_\(Cross_Site_Scripting\)_Prevention_Cheat_Sheet](https://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet)