**Document common web vulnerabilities ie. XSS**

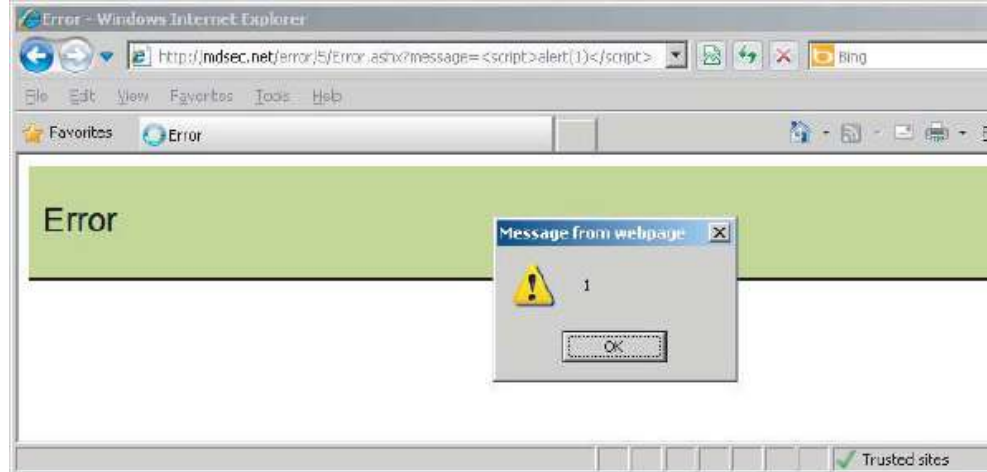
This document highlights the shift in focus from server-side to client-side attacks in web application security. While traditional attacks, like SQL injection, targeted servers and indirectly affected users, modern threats primarily focus on exploiting vulnerabilities in the user's browser. **Cross-Site Scripting (XSS)** is a leading example of such client-side attacks, where malicious scripts are executed in the victim's browser, leading to session hijacking, unauthorized actions, and data theft. As web applications evolve, client-side vulnerabilities have become more common and easier to exploit, making user-targeted attacks an increasing security concern.

**Varieties of XSS:**

XSS vulnerabilities come in various forms and may be divided into three varieties: **reflected, stored, and DOM-based**. Although these have several features in common, they also have important differences in how they can be identified and exploited. We will examine each variety of XSS in turn.

**Reflected XSS**:

Occurs when user input is immediately reflected back in the server's response without proper validation or sanitization. A common example is when a web application displays error messages dynamically based on a user-supplied parameter. If the input is directly embedded in the HTML of the response, malicious JavaScript can be injected and executed in the victim's browser.



For instance, a crafted URL like http://mdsec.net/error/5/Error.ashx?message=<script>alert(1)</script> inserts and executes a script in place of the error message. This lack of input sanitization makes the application vulnerable to reflected XSS. Performing this simple test serves verifies two important things. First, the contents of the message parameter can be replaced with arbitrary data that gets returned to the browser. Second, whatever processing the server-side application is performing on this data (if any), it is insufficient to prevent us from supplying JavaScript code that is executed when the page is displayed in the browser.

**Stored Cross-Site Scripting (XSS):**

Stored XSS Vulnerabilities occur when malicious user input is stored on a web application (typically in a database) and then displayed to other users without proper sanitization. This type of XSS is particularly dangerous in applications that allow user interactions, such as forums, comment sections, or auctions. For example, an attacker could embed a malicious script in a question or comment, which would then be executed when another user, such as a buyer or seller, views the content. This could lead to various outcomes, such as unauthorized actions like bidding or closing auctions.

Stored XSS typically involves two interactions: first, the attacker submits the malicious input, and second, the victim unknowingly triggers the script when they view the compromised page. Since the malicious code is stored on the server and affects multiple users, stored XSS can have a broader impact than other XSS types. This is sometimes referred to as "**second-order XSS**" because of the two-step nature of the attack, although it doesn’t necessarily involve a cross-site element.

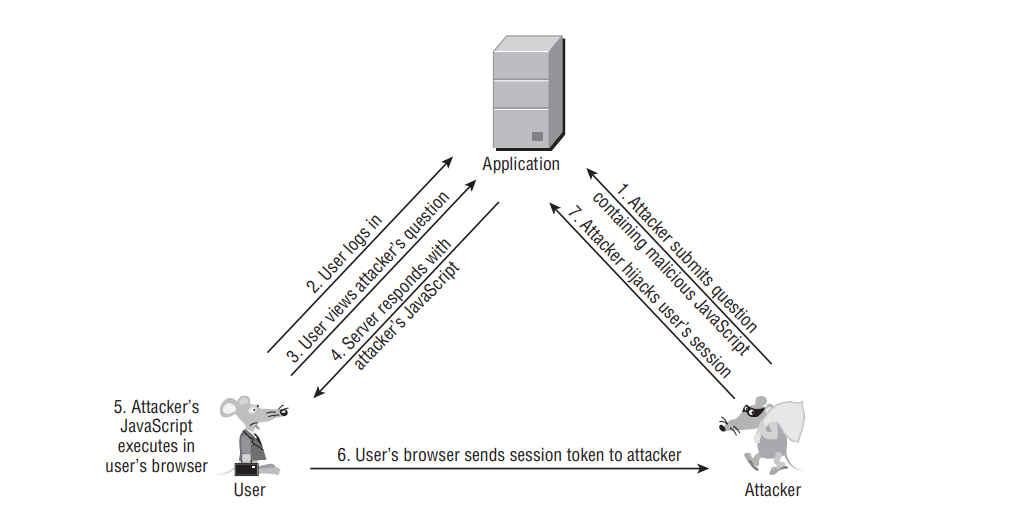


Figure: Steps involved in stored XSS attack

**DOM-Based Cross-Site Scripting (XSS):**

DOM- Based XSS vulnerabilities differ from reflected and stored XSS in that they occur entirely within the client-side environment. In DOM-based XSS, malicious scripts are executed by manipulating the **Document Object Model (DOM)** without the server including the attacker's script in its response.

Here’s how it works: the attacker crafts a URL with embedded JavaScript, and when a user clicks the URL, their browser loads the page. The server’s response does not directly contain the malicious script. However, client-side JavaScript on the page processes parts of the URL (e.g., query parameters) and uses that data to update the page's content dynamically. If this processing does not sanitize or validate the data, it may lead to the execution of the attacker’s script within the victim's browser.

For example, if a script on the page extracts a parameter from the URL and uses document.write() to insert it into the HTML, an attacker could inject JavaScript into the URL parameter. When the page loads, this unsanitized input is written into the DOM, executing the attacker's code. This type of XSS occurs purely on the client side, making it a **DOM-based vulnerability**.

A sample vulnerable URL might look like this: http://example.com/error?message=<script>alert(‘xss’)</script>, where the message parameter is processed by client-side code and executed without server-side involvement.

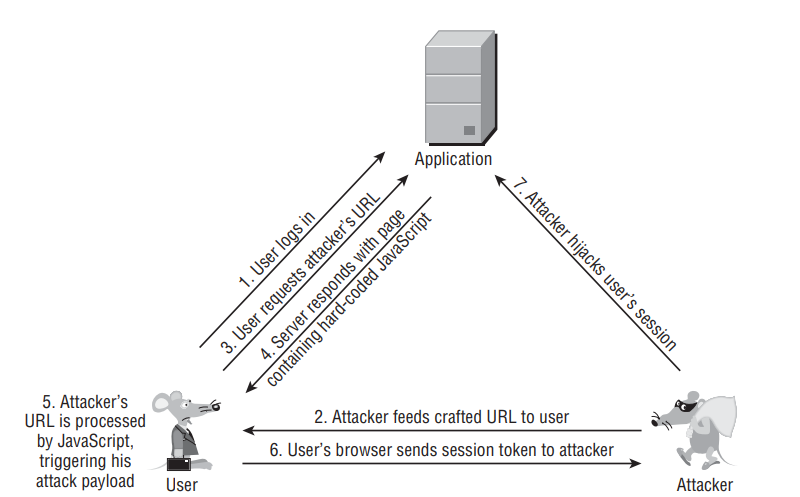


Figure: Steps involved in the DOM based XSS attack

**Preventing XSS Attacks**

Cross-Site Scripting (XSS) vulnerabilities are prevalent across many web applications due to the difficulty of identifying all potential points where user-controlled data can interfere with an application’s behavior. Although conceptually straightforward to prevent, the sheer number of ways user data can be mishandled makes XSS a common vulnerability, even in high-security environments.

There are different types of XSS vulnerabilities: **Reflected**, **Stored**, and **DOM-based XSS**. Each has slightly different defenses due to their root causes. This section explores how to prevent these vulnerabilities by focusing on input validation, output sanitization, and eliminating dangerous insertion points.

**Preventing Reflected and Stored XSS**

**Understanding the Root Cause**

Reflected and stored XSS vulnerabilities arise when user-controllable data is injected into application responses without proper validation or sanitization. User-supplied data can break the structure of HTML pages, enabling an attacker to execute malicious scripts. Whether the data comes directly from a request (as in reflected XSS) or is stored for future use (as in stored XSS), the principle remains the same—data is handled unsafely.

**General Defense Approach**

To prevent these vulnerabilities, developers should follow a threefold strategy:

1. **Validate input** - Ensure that data received by the application is within the expected format.
2. **Validate output** - Sanitize all data before inserting it into HTML responses.
3. **Eliminate dangerous insertion points** - Avoid inserting user data into locations where it can be misused.

**Input Validation**

Input validation ensures that only properly formatted data is accepted by the application. While it may not completely prevent XSS attacks, it adds an important layer of defense. Developers should consider these input validation techniques:

* **Limit input length**: Ensure that user input does not exceed the expected length.
* **Restrict characters**: Only allow characters that are needed for the specific field. For instance, a username field might only allow alphanumeric characters.
* **Regular expression matching**: Use regular expressions to enforce strict formatting on inputs like email addresses or account numbers.

**Output Validation (HTML Encoding)**

Output validation, or HTML encoding, is a critical defense. When displaying user data in an HTML page, certain characters should be replaced with their corresponding HTML entities to prevent them from being interpreted as part of the document’s structure. For example:

* & becomes &amp;
* < becomes &lt;
* > becomes &gt;
* " becomes &quot;
* ' becomes &apos;

By encoding special characters, browsers treat user input as content rather than code, preventing attackers from breaking out of intended structures. Additionally, any input that gets inserted into a tag attribute should be carefully encoded to avoid bypass attacks (e.g., using the javascript: pseudo-protocol in URLs).

In programming environments like **ASP.NET**, developers can use built-in APIs such as Server.HTMLEncode() to automatically encode malicious characters. However, in languages like **Java**, there are no built-in methods, so developers may need to create custom encoding functions.

**Escaping JavaScript Contexts**

When user input is inserted into JavaScript contexts, special care should be taken to escape characters like backslashes and quotes that could lead to script injection. Failing to do so might allow attackers to manipulate the content and execute arbitrary code.

For example:

<img src="javascript&#58;alert(document.cookie)">

<img src="image.gif" onload="alert(&apos;xss&apos;)">

In the above examples, even if some characters are encoded, attackers can still execute scripts through bypasses like the onload attribute. Therefore, avoid inserting user input into sensitive contexts like event handlers or JavaScript code whenever possible.

**Avoid Dangerous Insertion Points**

Some locations in an application are inherently dangerous to place user data. These include:

* **Inside <script> tags**: Avoid placing user-controlled data directly into JavaScript code.
* **Tag attributes**: Embedding user data into attributes like href can allow attackers to use techniques like the javascript: protocol to execute scripts.

If inserting user input in these contexts is unavoidable, use techniques like rigorous escaping and encoding to mitigate the risk. However, eliminating these dangerous insertion points altogether is the best practice.

**Preventing Character Set Manipulation**

Character set manipulation is another way attackers might bypass XSS filters. If an attacker can change the encoding used by the application (e.g., by manipulating a request parameter that specifies character encoding), they might inject malicious data in an unexpected form. To prevent this, always specify the character encoding in response headers (e.g., Content-Type: text/html; charset=ISO-8859-1) and ensure that it cannot be altered.

**Allowing Limited HTML**

Some applications need to allow user-submitted HTML content, such as in blog comments or message boards. In this case, output sanitization measures like HTML encoding would break the application by converting HTML tags into plain text. To handle this safely:

* Implement a **whitelist** of allowed HTML tags and attributes.
* Use tools like **OWASP AntiSamy** to validate user-supplied HTML.
* If possible, use a **custom intermediate markup language** to let users format their content without directly using HTML.

For example, an application might allow <b> and <i> tags but ignore their attributes to prevent script injection. A secure implementation would prevent users from injecting event handlers like onclick or embedding JavaScript through styles or URLs.

**Preventing DOM-Based XSS**

DOM-based XSS occurs when client-side JavaScript processes untrusted data from the DOM (e.g., URL parameters) and inserts it into the HTML without proper validation or sanitization. Unlike reflected or stored XSS, DOM-based XSS doesn’t involve the server sending a malicious script—it happens entirely on the client side.

**Input Validation for DOM-based XSS**

To prevent DOM-based XSS, client-side scripts should validate data before using it. For instance, if a script is processing data from the URL, it should check that the data only contains expected characters. A simple regular expression can be used for this purpose:

var a = document.URL;

a = a.substring(a.indexOf("message=") + 8, a.length);

a = unescape(a);

var regex = /^([A-Za-z0-9+\s])\*$/;

if (regex.test(a)) {

document.write(a);

}

In this example, the script ensures that only alphanumeric characters and whitespace are allowed in the message parameter.

**Output Validation for DOM-based XSS**

Similar to reflected XSS, DOM-based XSS can be prevented by HTML encoding data before inserting it into the page. For example:

function sanitize(str) {

var div = document.createElement('div');

div.appendChild(document.createTextNode(str));

return div.innerHTML;

}

This function safely encodes user input by creating a DOM element and appending text to it, ensuring the browser handles it as content rather than code.

**Avoid Using Client-Side Scripts for DOM Insertion**

Whenever possible, avoid using client-side scripts to insert data into the page based on the URL or other untrusted sources. This reduces the risk of DOM-based XSS. If such behavior is necessary, combine rigorous input validation with output encoding to minimize the risk.

**Conclusion**

Preventing XSS vulnerabilities requires a multi-layered approach, incorporating both input validation and output sanitization. Developers should avoid dangerous insertion points, use encoding to safely handle user data, and apply extra caution when dealing with client-side scripts. By following these practices, applications can significantly reduce the risk of XSS attacks, ensuring a safer experience for users.