Hunting for DOM-based XSS vulnerabilities: A complete guide

BY AYOUB · NOVEMBER 11, 2025

Traditional cross-site scripting (XSS) vulnerabilities were prevalent when server-side rendering (with languages like PHP, JSP, and ASP) was the norm. However, as applications become more complex and developers continue to shift application logic to the client-side, more complex client-side vulnerabilities are expected to arise.

In this article, we will cover what DOM-based cross-site scripting (XSS) vulnerabilities are, their potential impact, and how to identify and exploit them in modern applications effectively.

Let's dive in!

What is DOM-based cross-site scripting (XSS)

With traditional cross-site scripting (XSS) vulnerabilities, malicious input is sent to the server-side and rendered in the HTTP response without any additional input sanitization. With DOM-based cross-site scripting (XSS), malicious input is derived from a DOM source and purely evaluated in the browser through a DOM sink, meaning that the output is never visible in the HTTP response body.

The passing of unsanitized data (such as payloads) from a DOM source to a DOM sink enables the execution of arbitrary JavaScript code. Similar to a reflective XSS, an attacker can send a specially crafted link of the vulnerable page to the victim to execute client-side code and take control of the victim's session.

Before we move on to the testing phase, let's take a brief look at all DOM sources and sinks.

DOM sources

DOM sources are JavaScript properties that contain user-controllable data, which attackers can manipulate (typically through the URL). Below is a comprehensive list of all possible entry points for DOM-based XSS attacks:

DOM-based XSS: DOM sources explained

DOM sinks

DOM sinks are JavaScript functions that can execute or render user-controllable data. These are the places where DOM-based XSS vulnerabilities actually trigger when untrusted data flows from a source to a sink.

In the illustration below, you can find a few examples of DOM sinks:

DOM-based XSS: DOM sinks explained

Now that we understand the differences between a DOM sink and a source, we can move on to the testing phase, where you'll learn how to identify these vulnerability types.

Methodology

Unlike traditional XSS, where you inject your keyword and look for reflection and injection points, DOM-based XSS requires a different approach, whereby you inject your payload into a DOM source and intercept runtime DOM event handlers to find where and if your input is processed. There's also an alternative method for DOM-based XSS vulnerabilities, which involves thoroughly reviewing JavaScript code.

Let's take a closer look at both approaches.

Finding DOM-based XSS via static code analysis

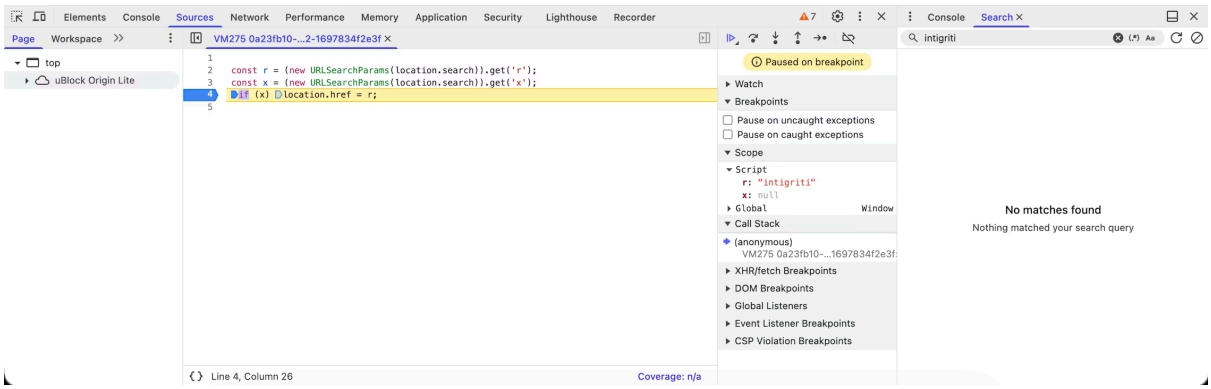
This approach requires a basic foundation of JavaScript. You'll need to manually review JS code, search for DOM sources, and work your way forward until you can find any references to a DOM source.

Additionally, you'll need to review every JavaScript file and code snippet individually, including those that are obfuscated and minified.

Finding DOM-based XSS via DOM runtime interception

This second approach involves actively intercepting DOM events emitted from event handlers in runtime and searching for places where your input may have been processed. You'll need to make use of your browser's developer console to search for DOM sinks and set up breakpoints where your input may be processed.

This method will allow you to intercept and analyse data as it moves from a source to a sink:



Intercepting DOM-based events via browser developer console

Use automated tools!

Both approaches are tedious and can form difficulties when testing at scale. Luckily for us, there are automated tooling available, such as Untrusted Types and DOM Invador that can help you easily spot DOM-based vulnerabilities, including DOM-based cross-site scripting (XSS).

DOM-based XSS exploitation

Now that we've established the core fundamentals of what DOM-based XSS vulnerabilities are, let's explore how you can exploit these in the wild.

Exploiting DOM-based XSS via innerHTML sink

As we've previously seen in the DOM sinks section of this article, the innerHTML sink is used to parse and render HTML tags, including malicious payloads.

You'll sometimes notice that innerHTML is preferred when concatenating server data with dynamic input from the client (such as the value of a URL query parameter). When this is the case, and no further validation is performed, we can practically inject any HTML tag with an event handler that would execute our arbitrary JavaScript code.

Take the following code snippet into consideration:

```
1  <!DOCTYPE html>
2  <html>
3      <head>
4          <title>Dashboard</title>
5      </head>
6      <body>
7          <div id="welcome"></div>
8          <script type="application/javascript">
9              function displayWelcomeMessage() {
10                 const urlParams = new URLSearchParams(location.search);
11                 const firstName = urlParams.get('firstName');
12
13                 if (firstName) {
14                     document.getElementById('welcome').innerHTML = 'Welcome, ' + firstName + '!';
15                 }
16             }
17
18             // Execute on page load
19             window.onload = displayWelcomeMessage;
20         </script>
21     </body>
22 </html>
```

DOM-based XSS via innerHTML DOM sink

On **line 10**, we can see that the vulnerable application reads data from the **firstName** query parameter and passes it to the DOM sink, specifically **innerHTML**. Since we fully control the **firstName** query parameter, we can essentially pass the following payload as its value and render any HTML tag, including XSS payloads:

```
<img src=x onerror=alert(/INTIGRITI/)>
```

The aforementioned case represents a basic example. In real-world scenarios, you'll encounter some form of validation. This can be a basic regex pattern filtering out malicious tags, a third-party package employed to sanitize input (such as DOMPurify) or server-side measures that HTML encode all data. In either case, there may be an opportunity to still evade any of the measures.

Tip!

Some developers use base64 or other encoding to format and display client-side data correctly. Always inspect the JavaScript code to identify if atob(), btoa(), or other decoding functions are used between the source and sink to avoid missing any potential DOM-based XSS cases.

Exploiting DOM-based XSS via dynamic eval functions

Although this is less common, you may occasionally encounter targets that execute client-side functions based on dynamic data, mostly coming from a user-controllable source (such as a query parameter). In instances like these, we're almost always a single step away from executing arbitrary JavaScript code.

Let's take a look at an example:

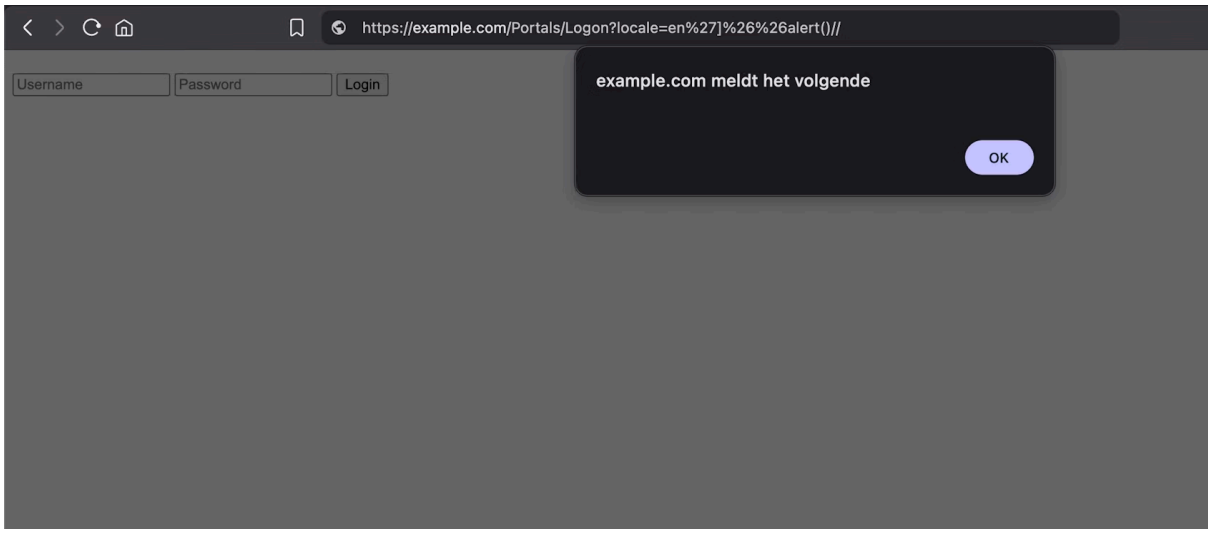
```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<html>
    <head>
        <title>Login</title>
    </head>
    <body>
        <h1 id="greeting"></h1>
        <form>
            <input
            type="text"
            placeholder="Username"
            />
            <input
            type="password"
            placeholder="Password"
            />
            <button type="submit">Login</button>
        </form>
        <script>
            // Legacy login form with multi-language support
            function setLanguage() {
                const urlParams = new URLSearchParams(window.location.search);
                const locale = urlParams.get('locale') || 'en';

                const loadLanguage = new Function('locale', `
                    const translations = {
                        en: 'Welcome',
                        es: 'Bienvenido',
                        fr: 'Bienvenue'
                    };
                    return translations['${locale}'] || 'Welcome';
                `);

                // Display translated message
                document.getElementById('greeting').textContent = loadLanguage(locale);
            }

            window.onload = setLanguage;
        </script>
    </body>
</html>

```

DOM-based XSS via Function DOM sink

This legacy login form seems to process the **locale** parameter and include it as an argument in the **Function()** sink. To execute arbitrary code, we must use the **locale** parameter to break out of the context and inject our own code:

```
en']&&alert()//
```

This payload would essentially allow us to call the alert() function, proving the arbitrary JavaScript code execution:



DOM-based XSS via Function DOM sink

Exploiting DOM-based XSS via client-side redirects

In-app redirects are commonly used to enhance the end user's app experience. Think of sign-in forms where you're redirected after a session token has been issued. However, when user-controllable data is immediately passed to a DOM sink without input validation, it may allow us, under certain conditions, to execute arbitrary JavaScript code, a possibility that not every developer is aware of.

Take the following example into consideration:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
        <title>Enterprise Portal - Sign In</title>
    </head>
    <body bgcolor="#f5f5f5">
        <center>
            <h2>Sign In</h2>
            <form id="loginForm" name="loginForm">
                <b>Username:</b> <input type="text" id="username" name="username"><br><br>
                <b>Password:</b> <input type="password" id="password" name="password"><br><br>
                <input type="submit" value="Sign In">
            </form>
            <p id="message" style="color: green;"></p>
        </center>

        <script type="text/javascript">
            function handleLogin(event) {
                event.preventDefault();

                const username = document.getElementById('username').value;
                const password = document.getElementById('password').value;

                // Simulate authentication
                if (username && password) {
                    const urlParams = new URLSearchParams(window.location.search);
                    const redirectURL = urlParams.get('redirectURL') || '/dashboard';

                    document.getElementById('message').textContent = 'Login successful! Redirecting...';

                    setTimeout(function() {
                        location.href = redirectURL;
                    }, 1000);
                } else {
                    document.getElementById('message').textContent = 'Invalid credentials provided!';
                }
            }

            document.getElementById('loginForm').addEventListener('submit', handleLogin);
        </script>
    </body>
</html>
```

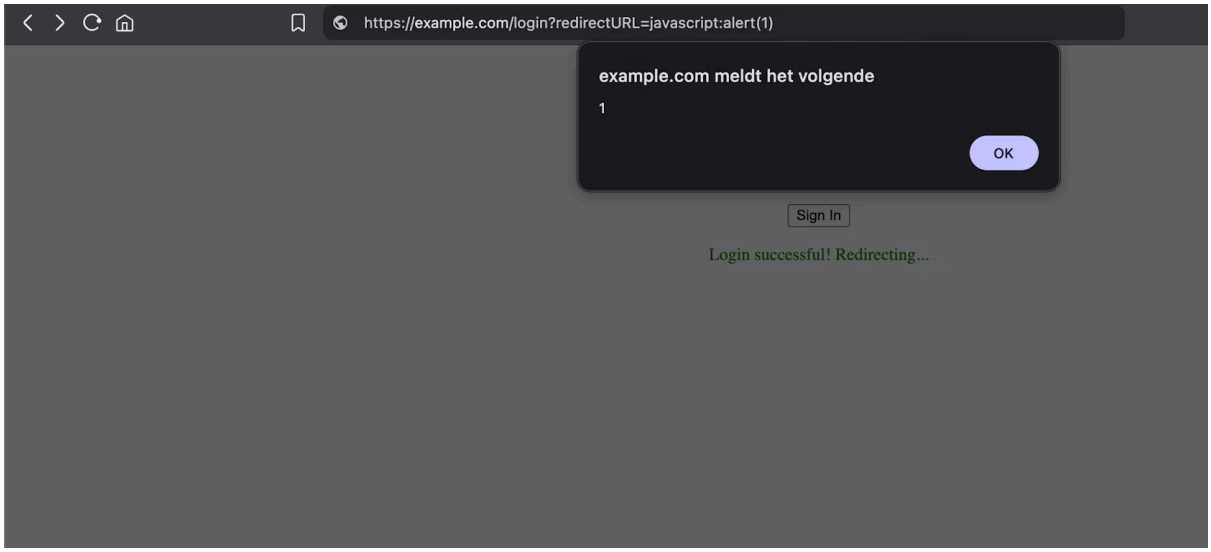
DOM-based XSS via Location DOM sink

Note how on **line 33** the value of the **redirectURL** parameter is passed to the location sink. Using our cheat sheet from before, we can determine that code execution is possible via the JavaScript protocol, at least if no Content-Security Policy (CSP) is enforced:

DOM-based XSS: DOM sinks explained

With this information, we can practically visit the following proof of concept URL to exploit this DOM-based cross-site scripting (XSS) vulnerability:

```
http://example.com/login?redirectURL=javascript:alert(1)
```



DOM-based XSS via Location DOM sink

Similar to our previous example, you may need to experiment with your payload to evade any filters. This can involve injecting null bytes, new line feed or carriage return characters (CR/LF), etc., depending on the filter.

Server-side or client-side redirect?

If the HTTP response returns a 3XX status code and the 'Location' HTTP response header, it is likely a server-side redirect, and (DOM-based) XSS will not be possible as browsers will follow the Location header and not render any HTML or execute any JavaScript code.

Otherwise, it is a client-side redirect, and DOM-based cross-site scripting may be possible under some conditions.

Exploiting DOM-based XSS via imported third-party packages

Modern web applications rely heavily on third-party JavaScript libraries and packages. These dependencies can sometimes introduce DOM-based XSS vulnerabilities when used without following best practices or when left outdated, creating an often-overlooked attack surface.

Let's take a look at a few examples.

DOM-based XSS via jQuery

jQuery is one of the most widely used JavaScript libraries. While it simplifies DOM manipulation, it also introduces several methods that can lead to DOM-based XSS when handling user-controllable input.

The primary culprit is jQuery's `html()` method, which behaves identically to `innerHTML` by parsing and rendering HTML content, including executable script tags and event handlers.

When developers pass unsanitized data from DOM sources (refer to our cheat sheet from before for examples) into methods such as `html()`, `append()`, or even jQuery selectors themselves, it may be possible to inject malicious payloads that execute in the victim's browser.

Client-side template injection in AngularJS/VueJS

In JavaScript frameworks like AngularJS and VueJS, client-side template injection (CSTI) can occur when a JavaScript templating engine processes user-controllable input without proper sanitization. Both frameworks use expression syntax to dynamically render data, and when user-controlled input reaches these expressions, arbitrary JavaScript code can be executed.

In AngularJS, for instance, we can use the constructor global property to call functions:

```
{{constructor.constructor('alert(1)')()}}
```

This template injection vulnerability is a type of DOM-based vulnerability that leads to DOM-based XSS. Make sure to be on the lookout for template injection vulnerabilities when you're testing targets that actively use AngularJS or VueJS.

Conclusion

DOM-based vulnerabilities, such as DOM XSS, often go unnoticed as they're hard to detect and test for at scale. In this article, we've explored various ways you can test and exploit these DOM-based vulnerabilities.

So, you've just learned something new about exploiting DOM-based XSS vulnerabilities... Right now, it's time to put your skills to the test! You can start by practicing on vulnerable labs and CTFs or... browse through our 70+ public bug bounty programs on Intigriti, and who knows, maybe earn a bounty on your next submission!

AUTHOR

Ayoub

Senior security content developer