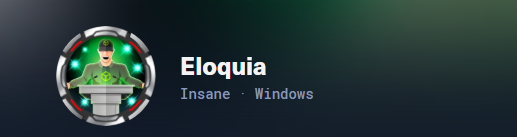
****

[**https://app.hackthebox.com/machines/Eloquia?tab=play\_machine**](https://app.hackthebox.com/machines/Eloquia?tab=play_machine)

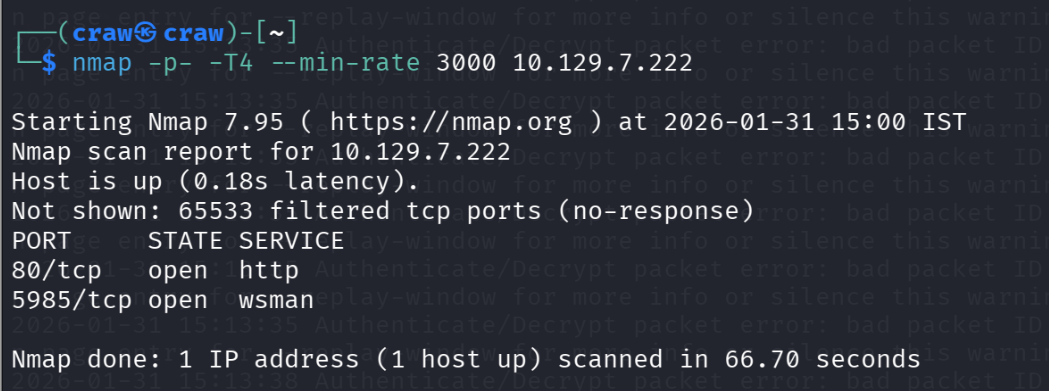
**RECONNAISSANCE AND ENUMERATION**

**Step-1:**

**Nmap Scanning**

**Scan Results for 10.129.7.222**

* **Operating System:** Likely **Windows** (indicated by port 5985).
* **Open Ports:**
  + **80 (HTTP):** A web server is running.
  + **5985 (WSMAN):** Windows Remote Management is active.
* **Status:** The scan checked all 65,535 ports. Only these two were open; the rest (65,533) were filtered/blocked by a firewall.



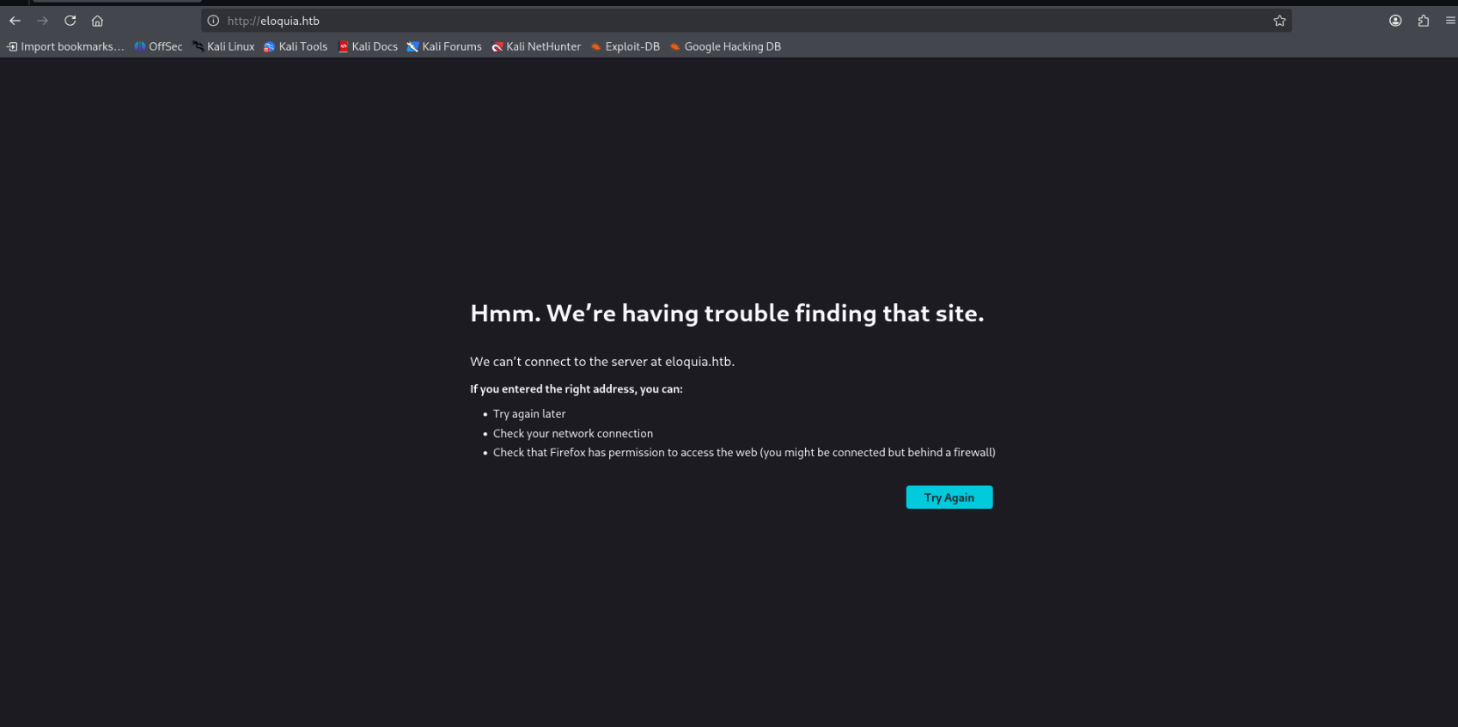
**Step-2:**

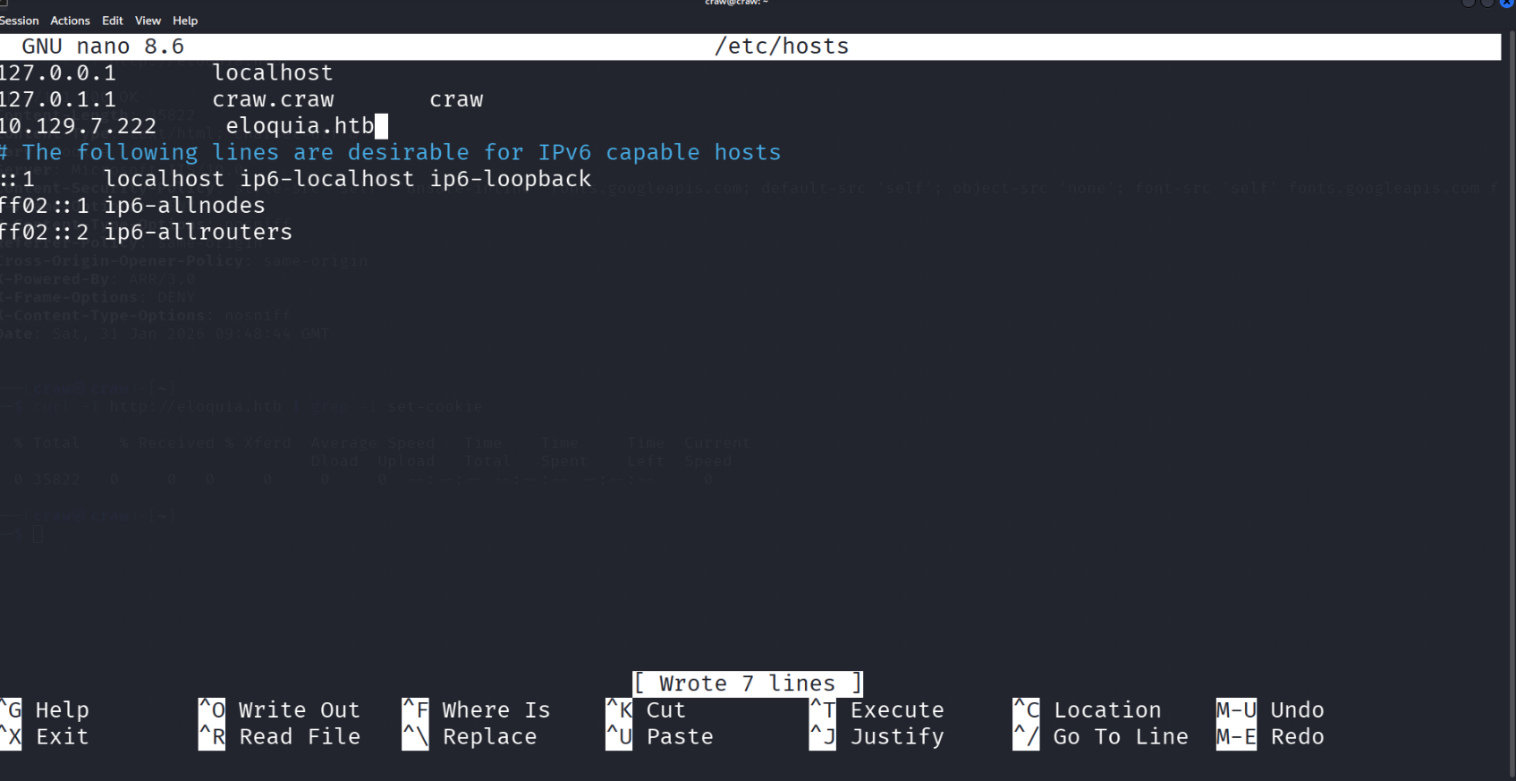
**Configure Local DNS Resolution**

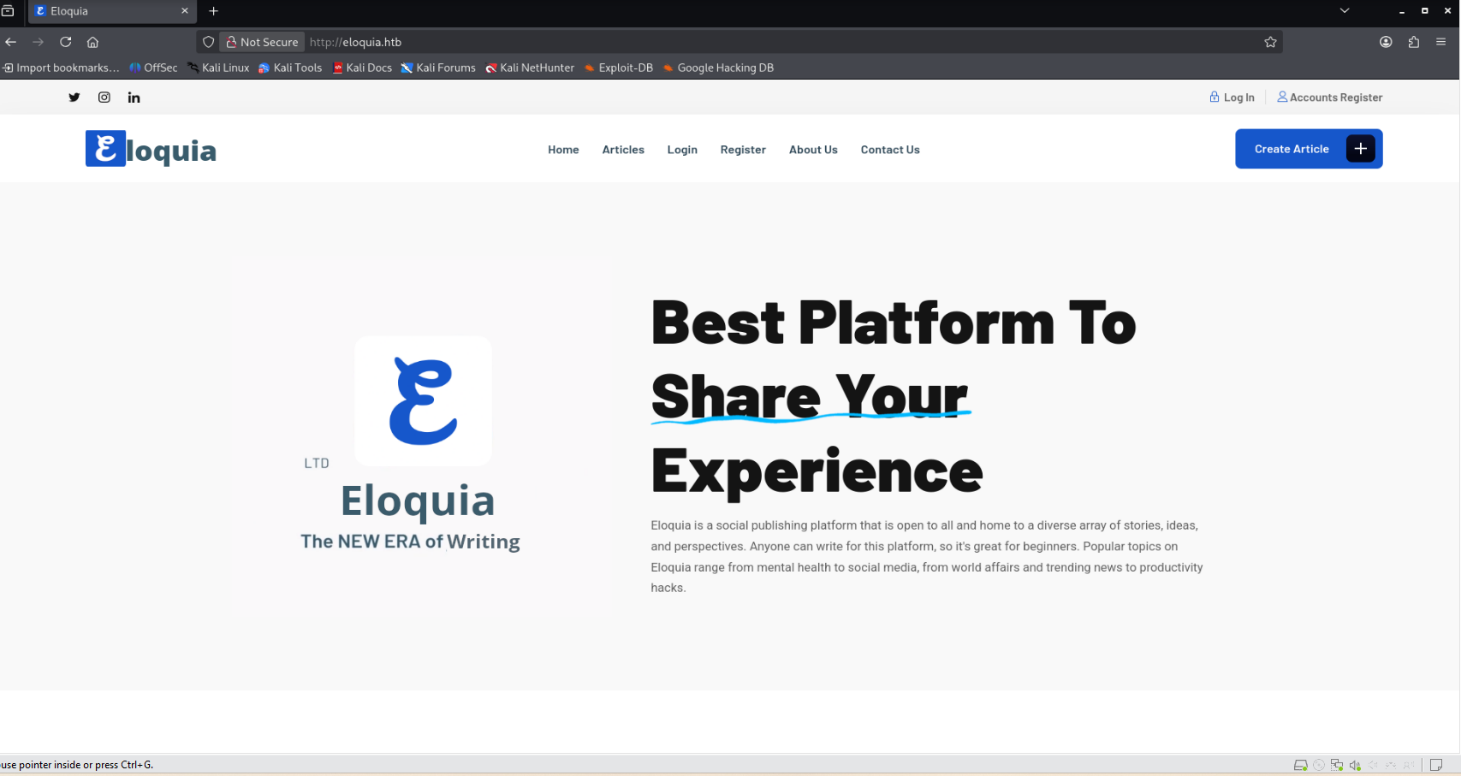
* **The Problem:** Your computer does not know the IP address for eloquia.htb because it is a private/internal domain, not a public one.
* **The Fix:** You must manually map the target IP address to the domain name in your **hosts file**.

1. Open your **/etc/hosts** file as an administrator (root).
2. Add the following line to the bottom of the file: 10.129.7.222 eloquia.htb
3. Save the file and refresh the page.

Once you do this, the browser will load the website, and you can continue your testing.



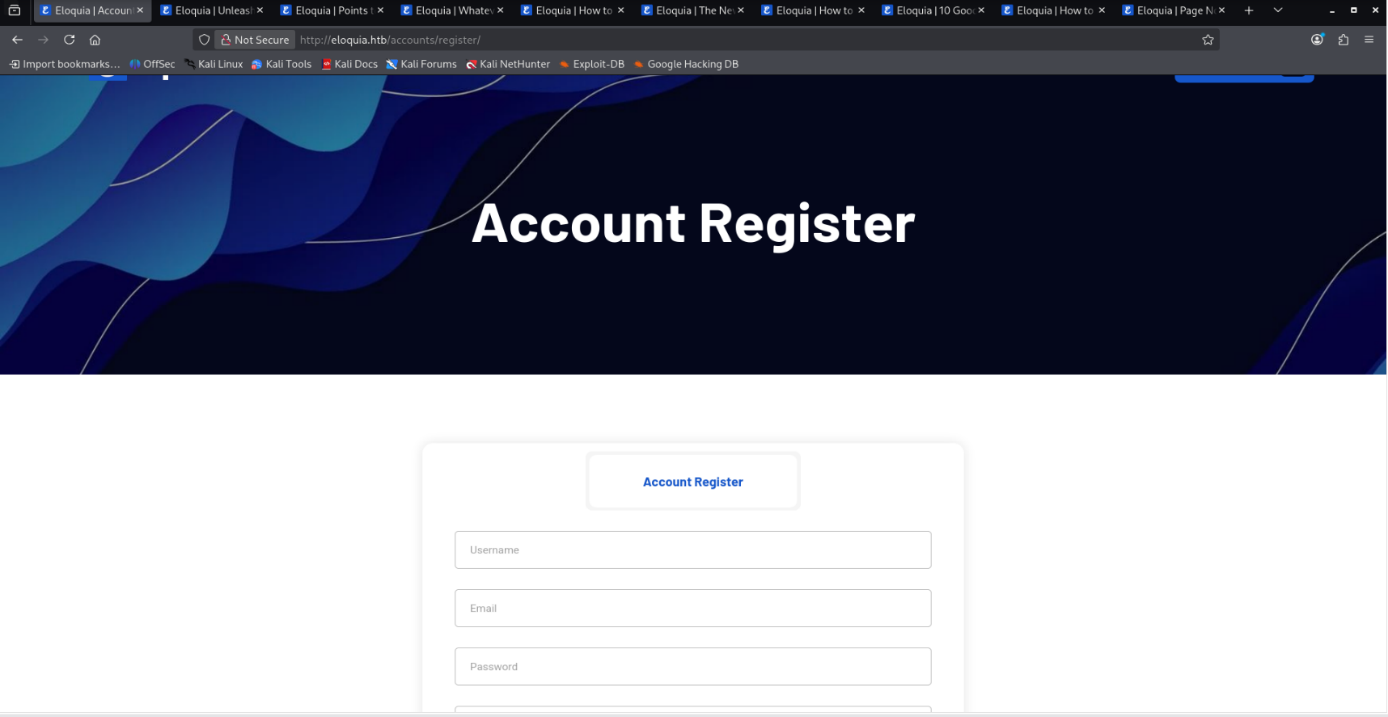


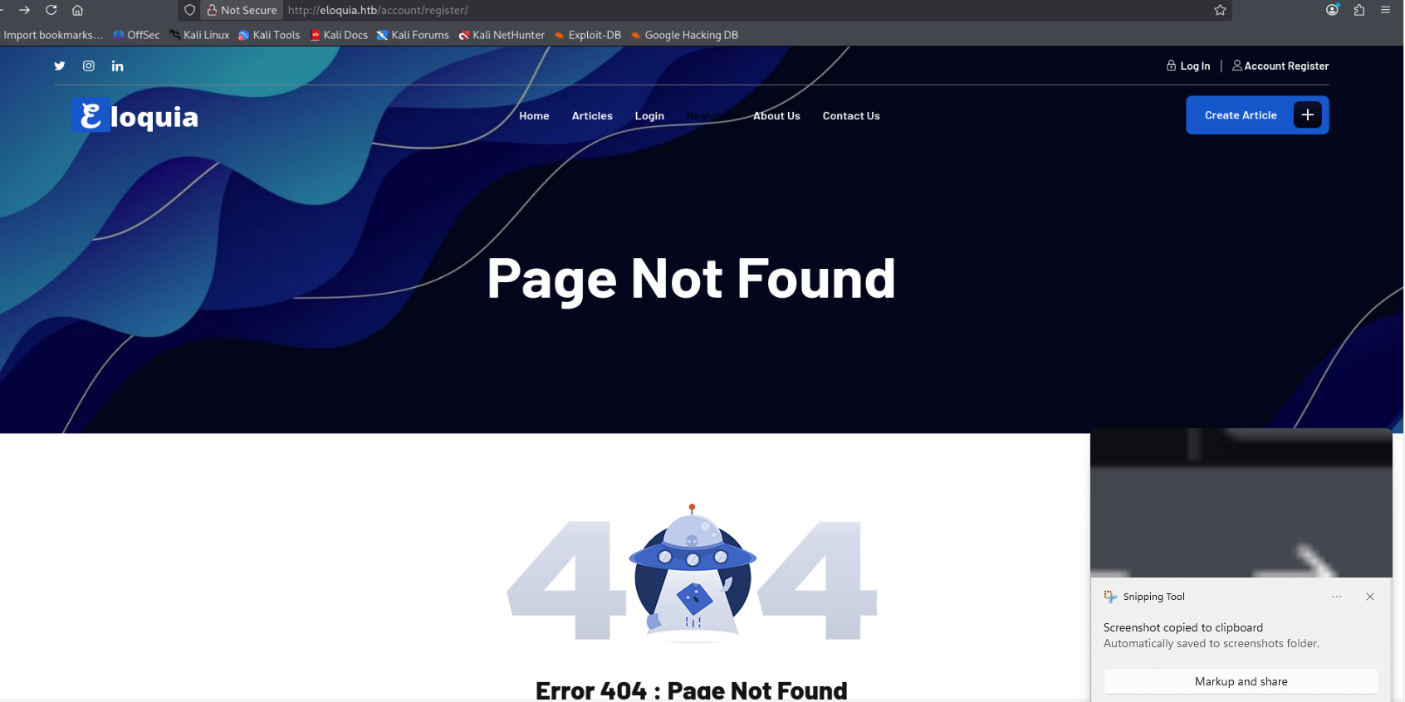


**Step 3:**

**Registration Endpoint Discovery**

* **Enumeration:** We systematically checked every accessible page to map the application structure and identify potential input vectors.
* **Findings:**
  + **Valid Endpoint:** We discovered a functional registration form at /accounts/register/ (plural).
  + **Dead Endpoint:** A similar path at /account/register/ (singular) does not exist and returns a custom **404 Page Not Found** error.
* **Conclusion:** The application allows public registration, providing a clear path to create a user and access authenticated areas.

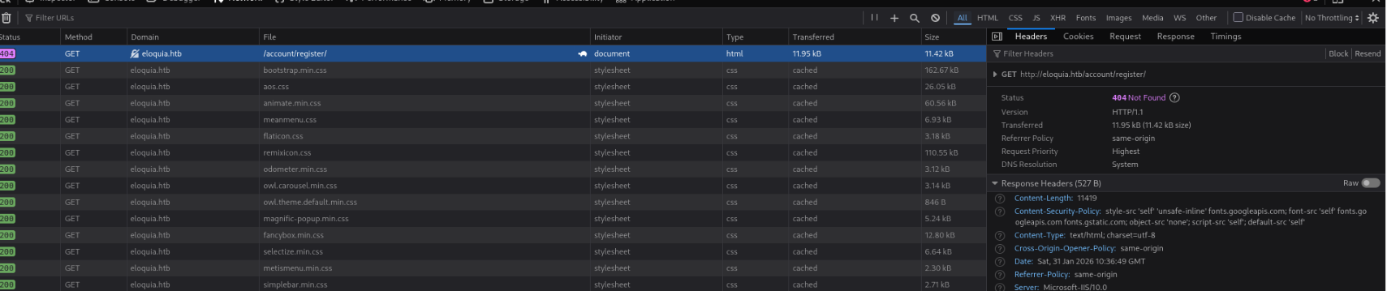


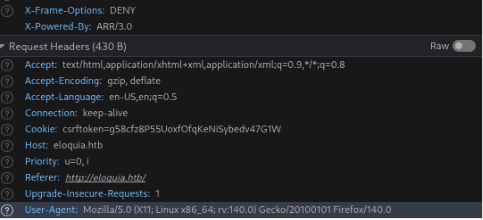


**Step 4:**

**Network Traffic & Header Analysis**

* **Error Validation:** We analyzed the browser's network logs and confirmed that the singular path /account/register/ is invalid, triggering a **404 Not Found** response.
* **Security Token Discovery:** Inspecting the request headers revealed a csrftoken cookie being sent to the server. This is a critical finding, as it proves the application uses Cross-Site Request Forgery protection, which we must handle during exploitation.
* **Architecture Confirmation:** The headers re-confirmed the presence of **ARR/3.0**, verifying the reverse proxy setup.

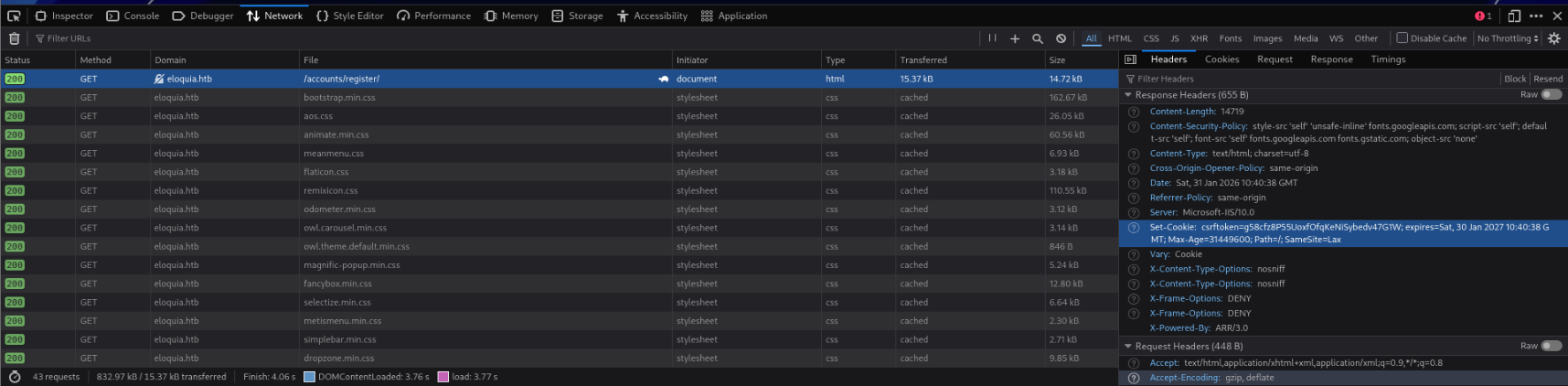




**Step 5:**

**Registration Page Verification**

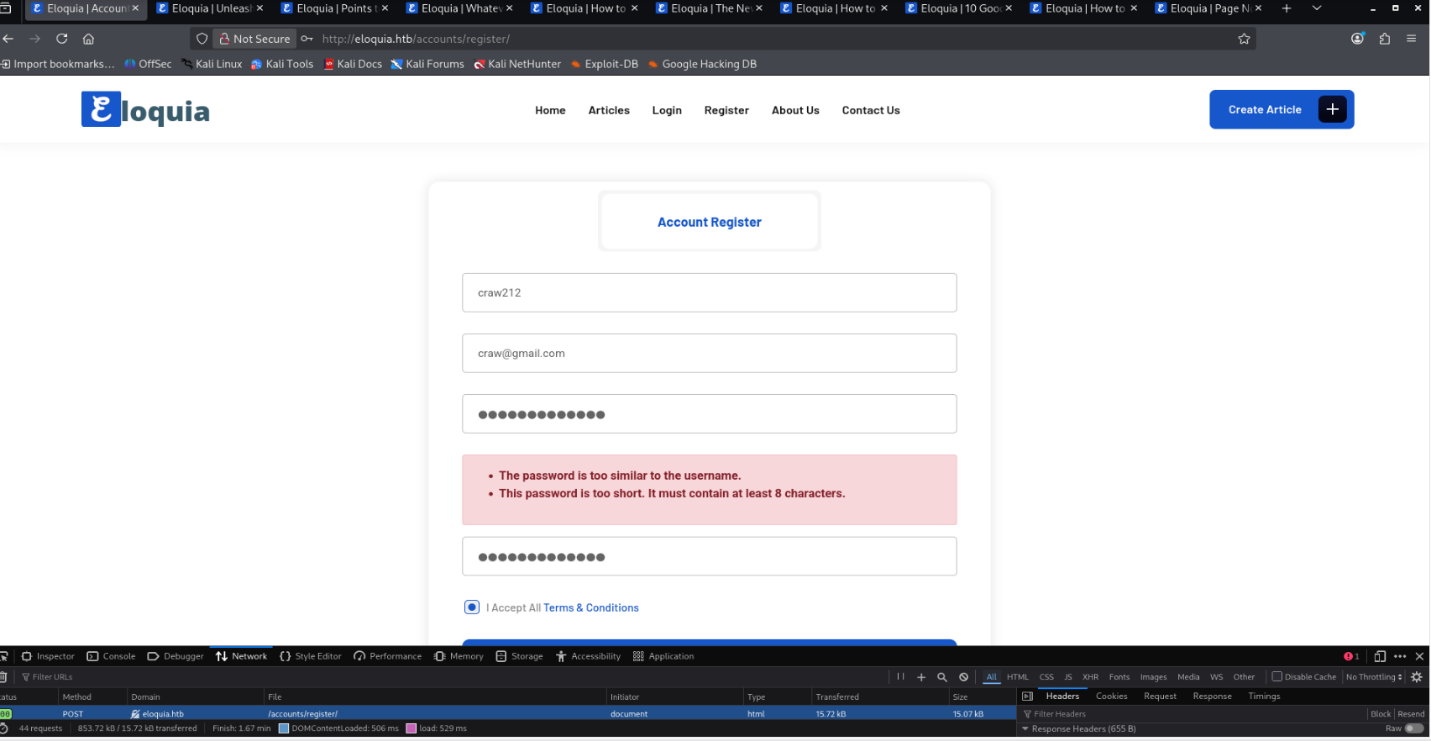
* **Endpoint Confirmation:** We navigated to /accounts/register/ and received a **200 OK** status, confirming this is the correct page for user creation.
* **Security Mechanism:** The response headers included a Set-Cookie directive assigning a csrftoken. This confirms the application enforces Cross-Site Request Forgery protection, which means any future automated exploitation scripts must capture and include this token to work.
* **Infrastructure Check:** The X-Powered-By: ARR/3.0 header persisted, reinforcing that we are interacting with the backend through a Microsoft reverse proxy.

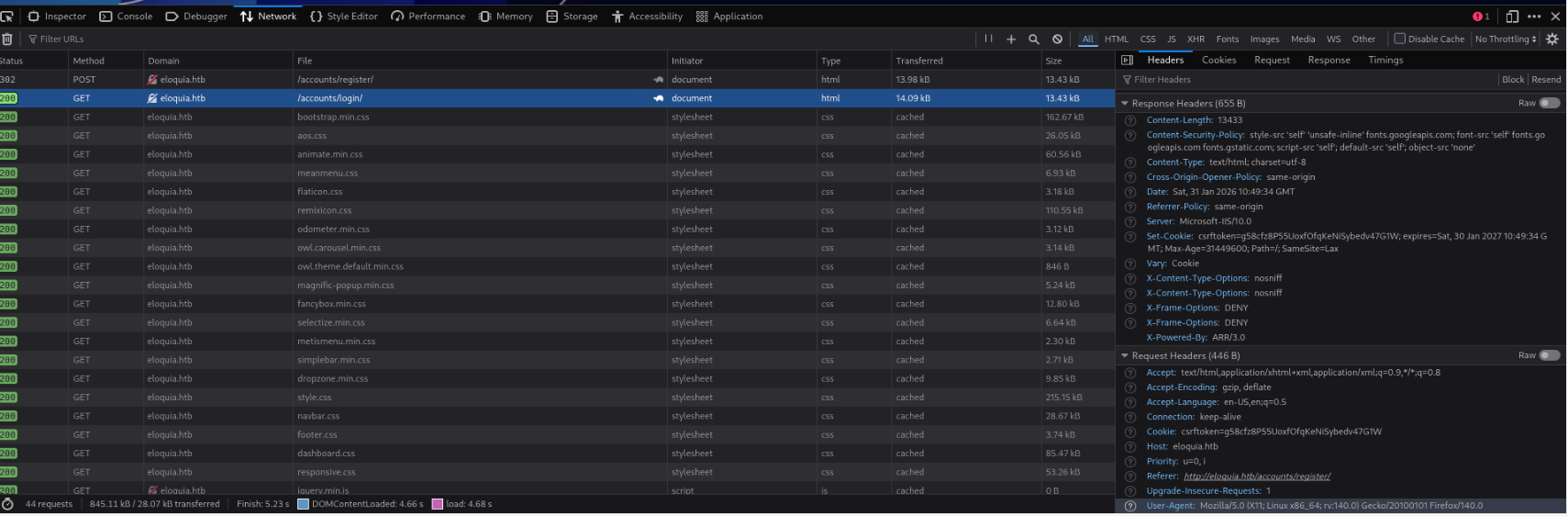


**Step 6:**

**User Registration & Password Policy Analysis**

* **Policy Enumeration:** During the account creation process, we identified strict password constraints. The application rejected weak credentials, explicitly stating that passwords must be at least 8 characters long and cannot be similar to the username.
* **Successful Execution:** After meeting these complexity requirements, the registration request returned a **302 Found** status code.
* **Workflow Validation:** This redirect automatically transitioned the session to /accounts/login/, confirming that the user account was successfully created and is ready for authentication.

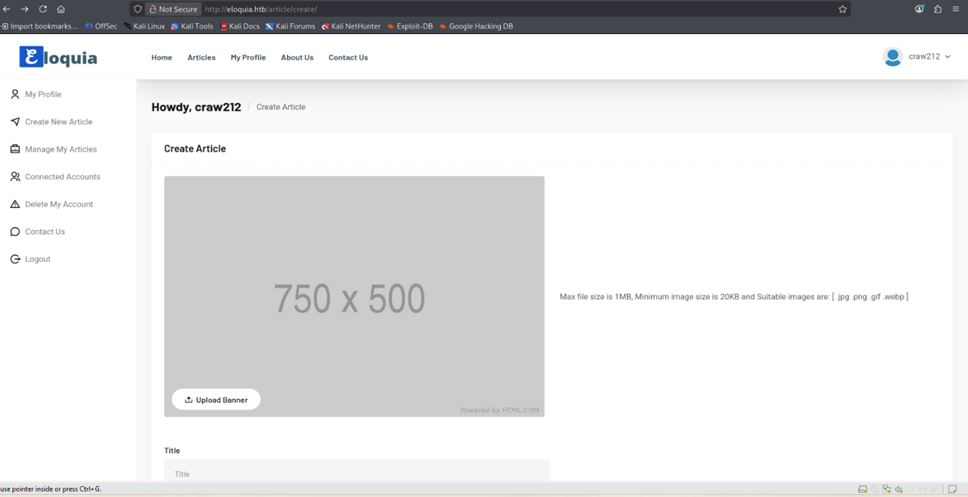


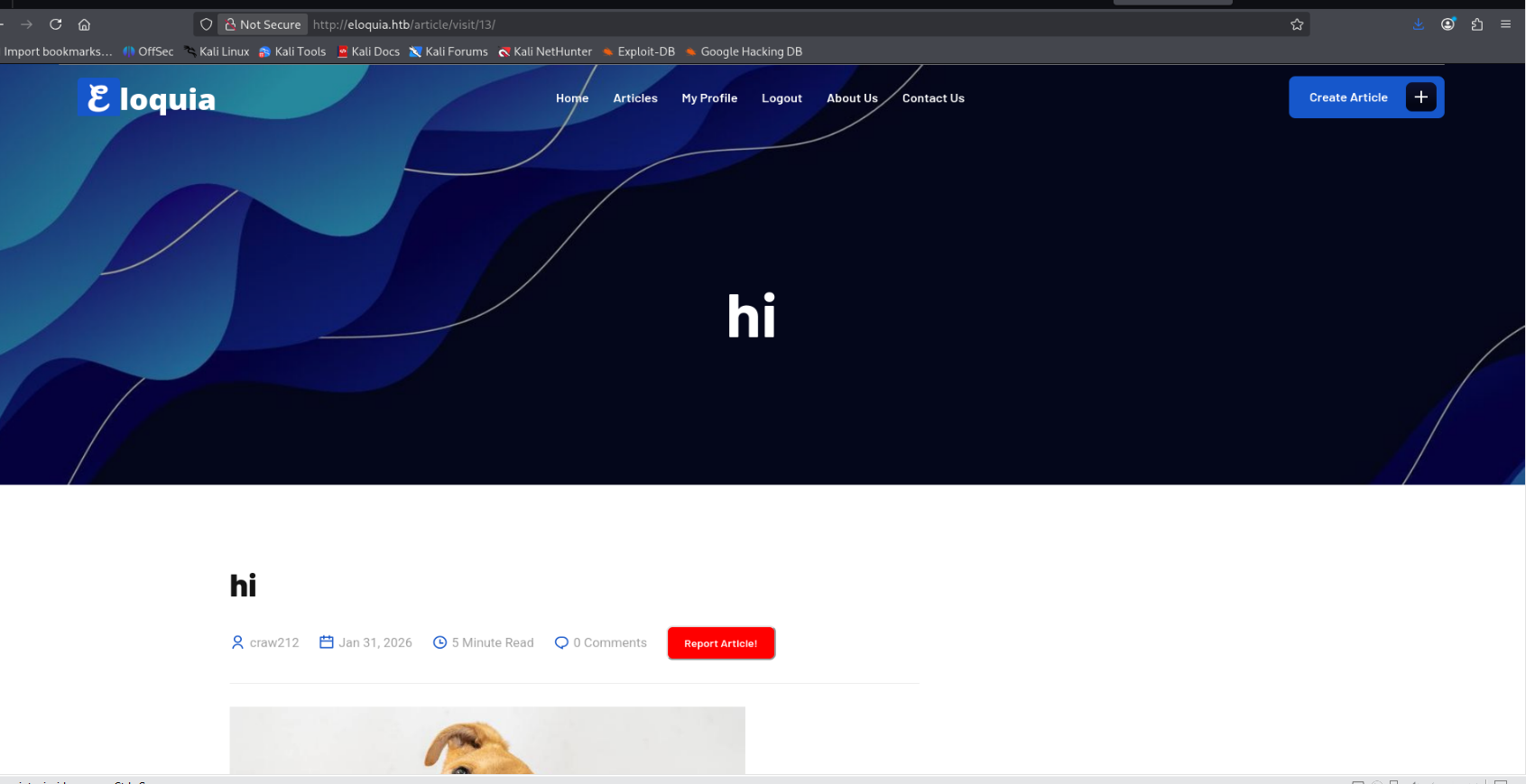


**Step 7:**

**Authenticated Attack Surface Analysis**

* **Feature Discovery:** Upon logging in, we accessed the "Create Article" dashboard, which exposes a file upload interface for article banners.
* **Constraint Enumeration:** The application displays specific validation rules for uploads: files must be between **20KB and 1MB** and restricted to image formats (.jpg, .png, .gif, .webp).
* **Functionality Test:** We successfully uploaded a compliant image and verified that the server processes and renders the file on the published article page, confirming this as a viable vector for file-based attacks.

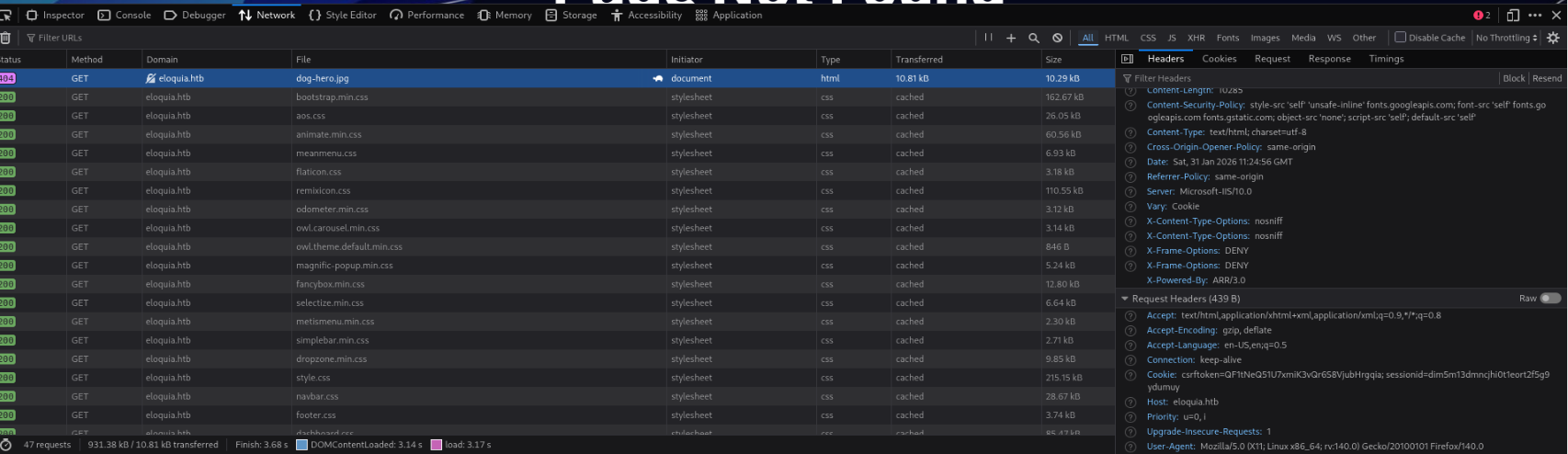


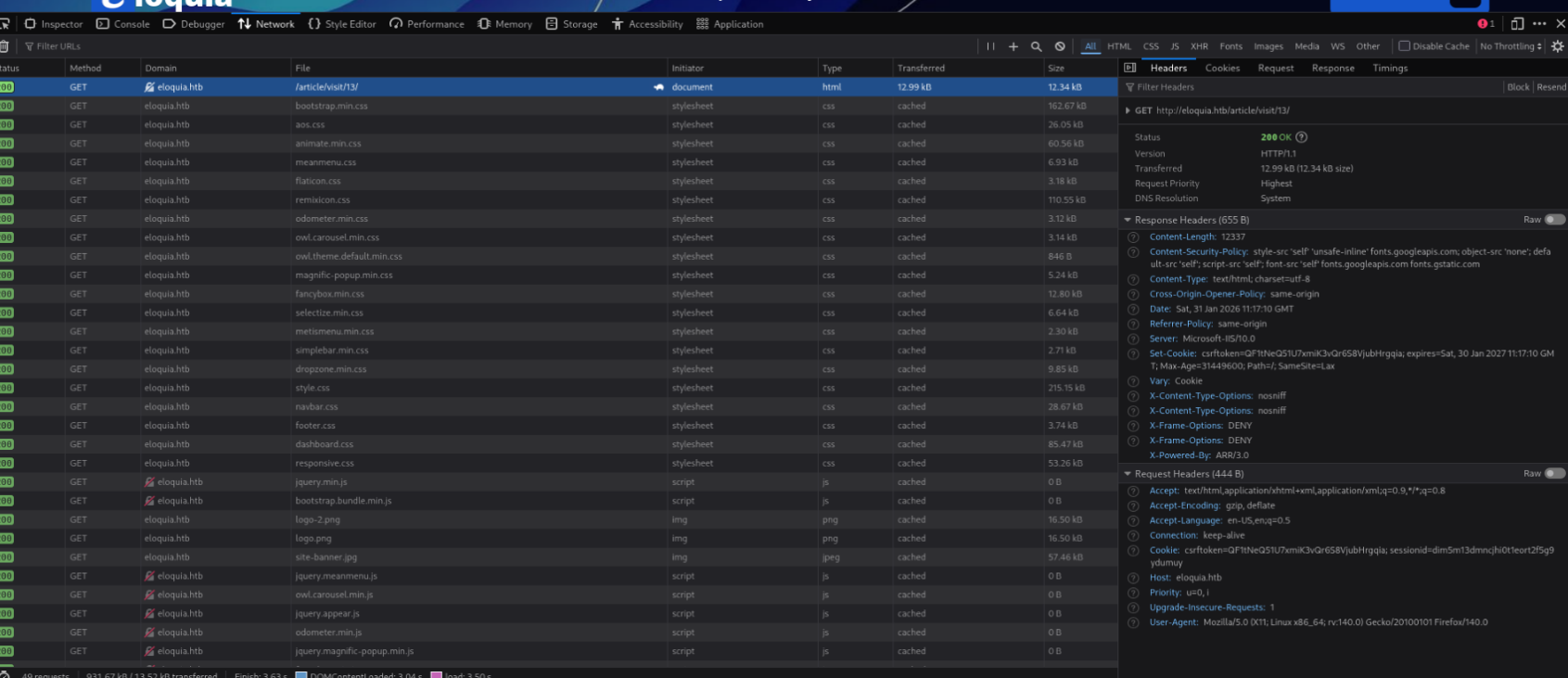


**Step 8:**

**File Path Analysis**

* **Direct Access Attempt:** We attempted to access the uploaded image directly at the web server's root (http://eloquia.htb/Dog-hero.jpg) to check for loose file permissions, but the server returned a **404 Not Found** error.
* **Contextual Verification:** We then navigated to the created article page at /article/visit/13/, which returned a **200 OK** status.
* **Conclusion:** This indicates that while the upload was successful, the application does not store user-generated content in the web root. We must inspect the article's source code to identify the actual upload directory (e.g., /media/ or /static/) to plan our shell execution path.

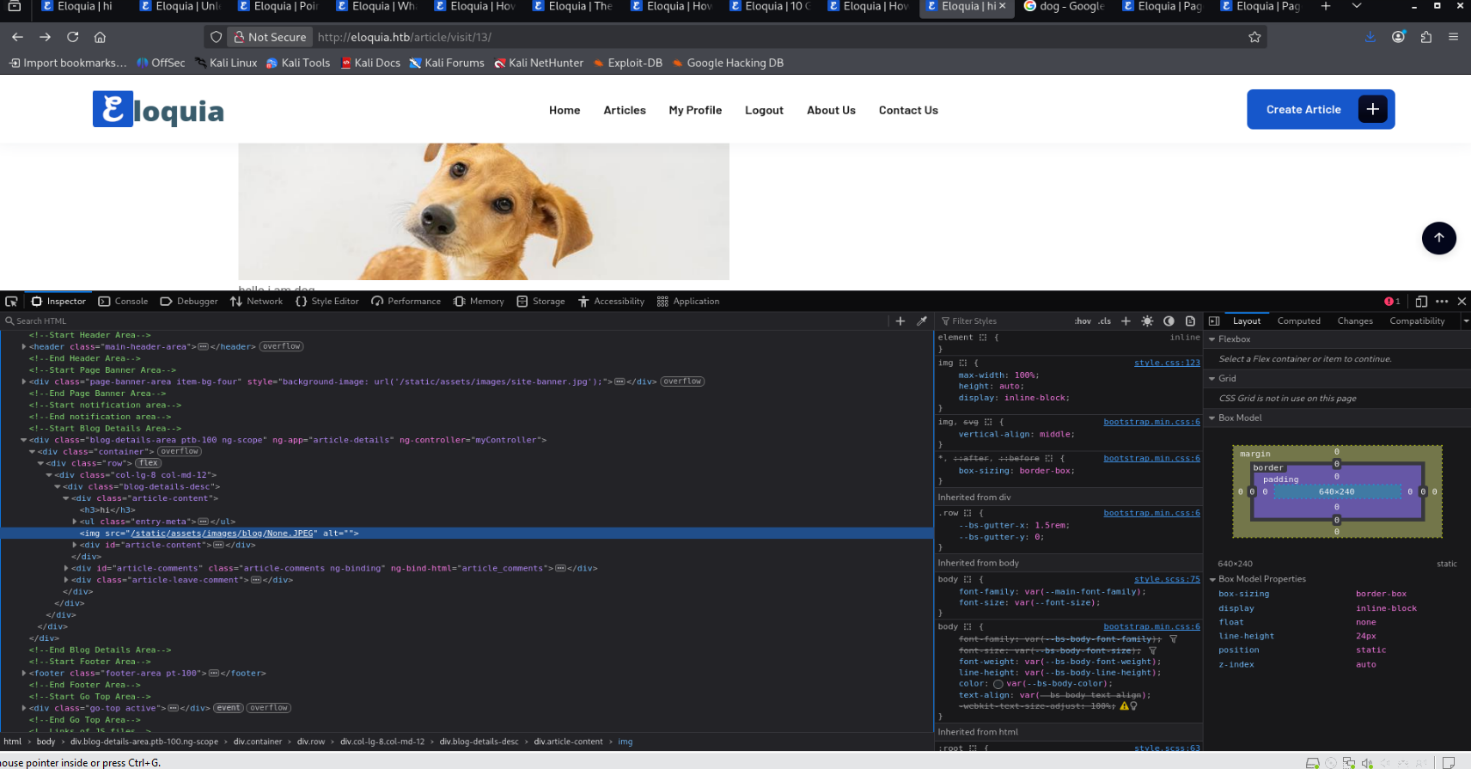




**Step 9:**

**Backend Behavior Analysis (The "None.JPEG" Anomaly)**

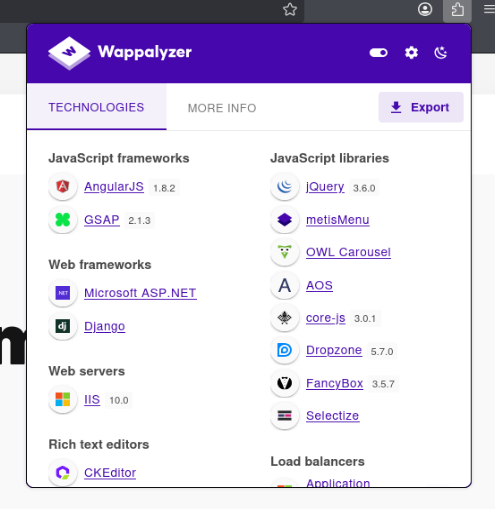
* **Path Verification:** We inspected the HTML source to find the exact location of our uploaded file. The code revealed the path: /static/assets/images/blog/None.JPEG.
* **Renaming Anomaly:** The server automatically renamed our uploaded image to **"None.JPEG"**.
* **Technology Fingerprint:** This is a significant finding. In Python (specifically frameworks like Django), if a filename variable is empty (None) and gets converted to a string, it becomes the word "None." This strongly confirms the backend is running **Python**.



**Step 10:**

**Technology Stack Confirmation**

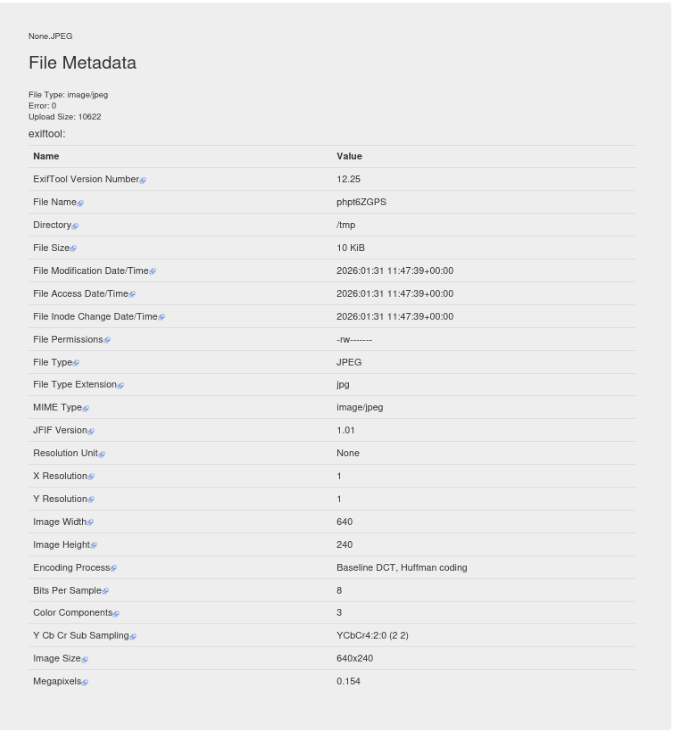
* **Verification:** To confirm our suspicion about the backend, we ran **Wappalyzer**. It explicitly detected **Django** as the web framework.
* **Correlation:** This definitively proves that the "None.JPEG" error was caused by a Python backend, validating our theory that the application logic is running on Python behind the IIS reverse proxy.
* **Component Identification:** The scan also identified **Dropzone 5.7.0**, which is the JavaScript library responsible for the file upload interface we previously tested.



**Step 11:**

**Step: Vulnerability Identification (CVE-2021-22204)**

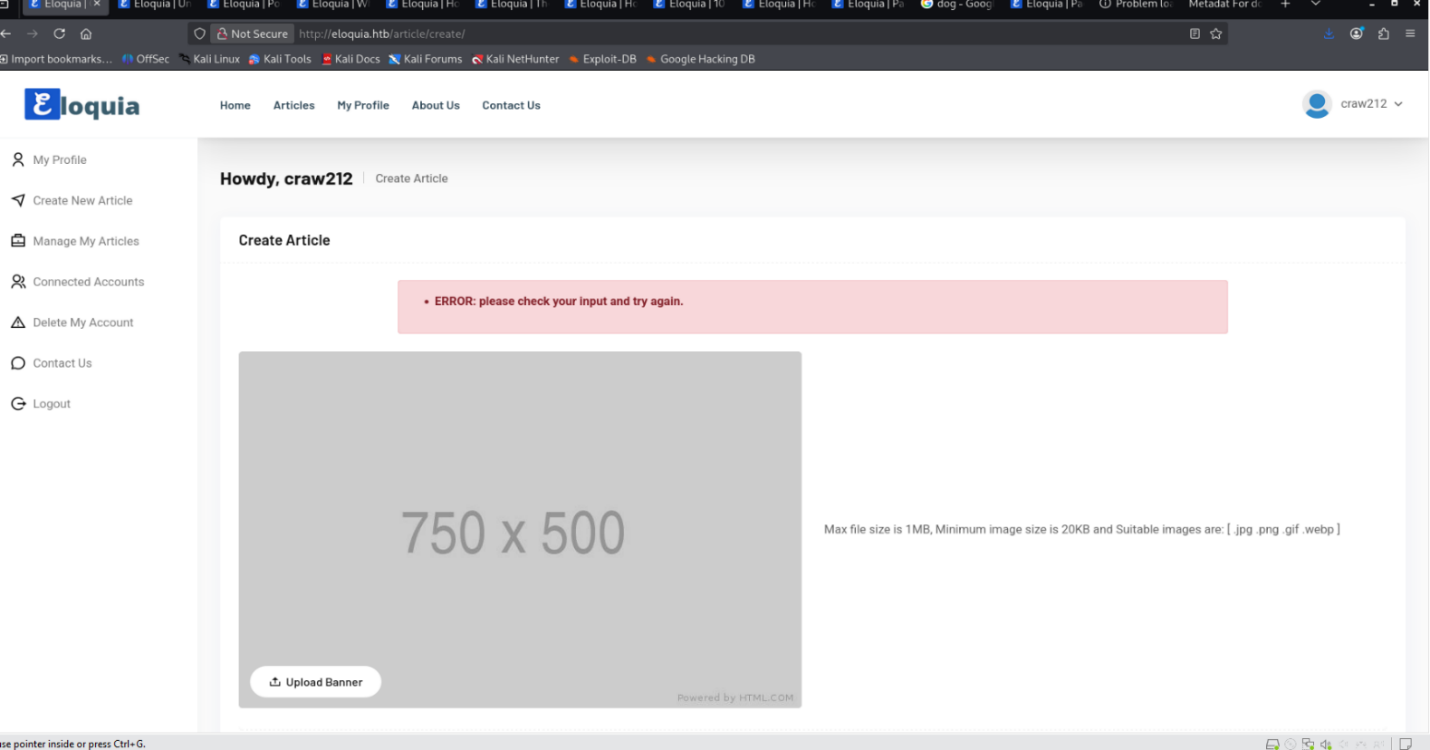
* **Metadata Extraction:** When we uploaded the image, the server processed it and displayed detailed metadata, including file permissions, MIME type, and image dimensions.
* **Critical Finding:** The output explicitly lists the **ExifTool Version Number** as **12.25**.
* **Vulnerability Context:** This specific version (12.25) is known to be vulnerable to **CVE-2021-22204**. This is a critical remote code execution (RCE) vulnerability where improperly sanitized DjVu metadata can trigger arbitrary command execution during the parsing process. This confirms our primary exploitation vector.

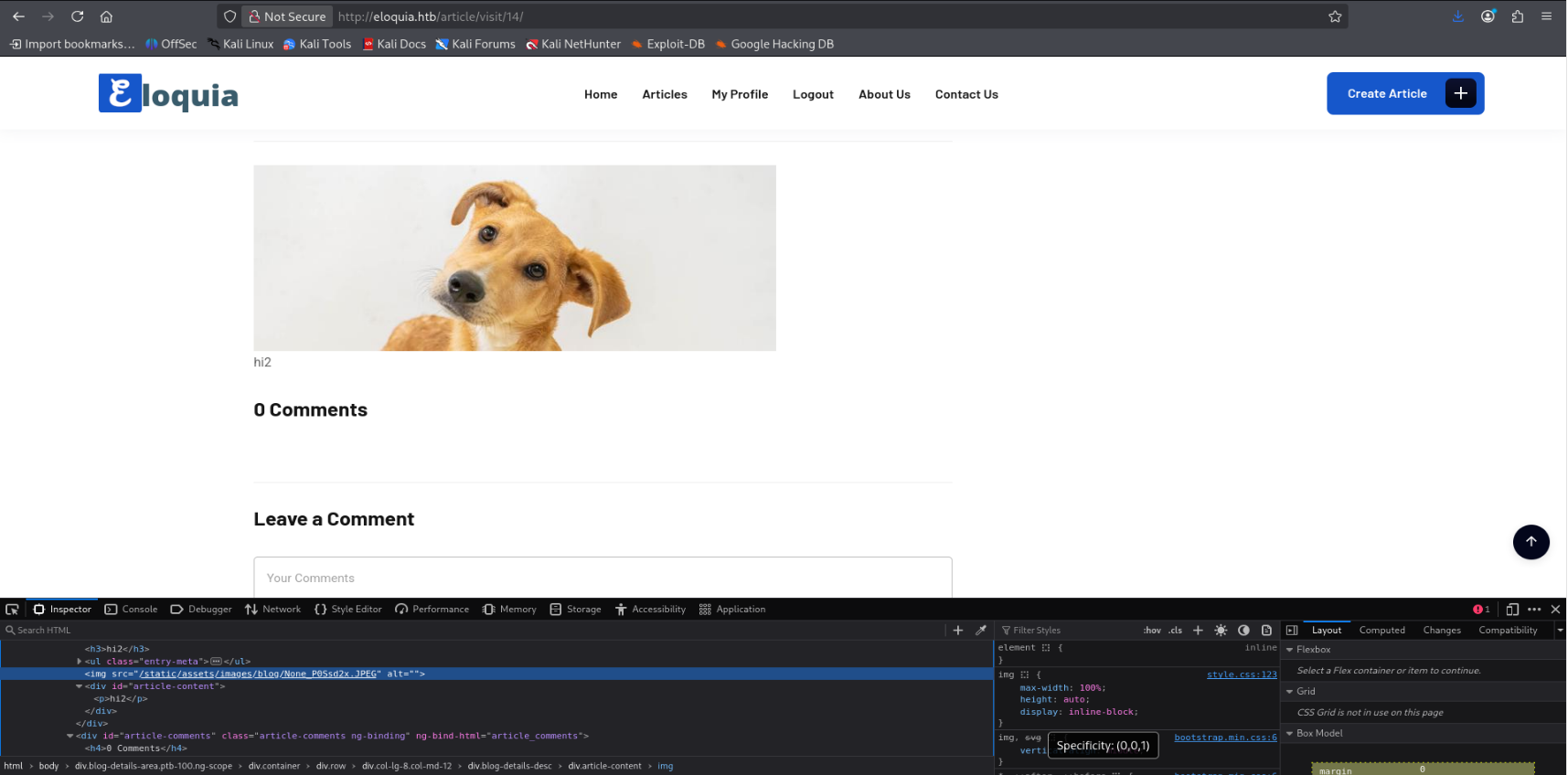


**Step 12:**

**Input Validation & Logic Analysis**

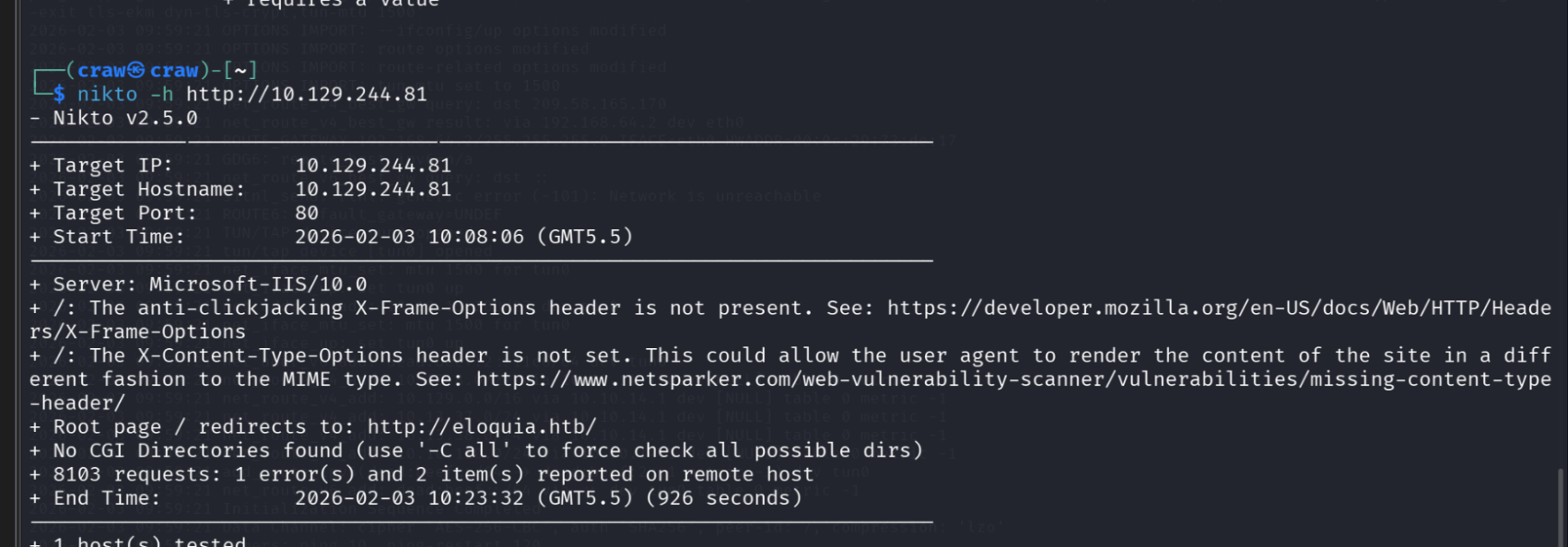
* **Constraint Verification:** We attempted to upload none.jpg, but the server rejected it with an error. This confirms the application strictly enforces the **20KB minimum file size** rule displayed on the UI, as our payload was too small.
* **Successful Bypass:** We uploaded a larger file, dog-hero.jpg (approx. 30KB), which passed the validation checks and was successfully stored on the server.
* **Backend Behavior Confirmation:** The server renamed the valid file to None\_P0Ssd2x.JPEG. This random suffix is a standard **Django** feature used to prevent filename collisions. Combined with the "None" prefix, this definitively proves the backend logic is mishandling the filename variable (converting a Python None object to a string) before saving it.





**Step 13:**

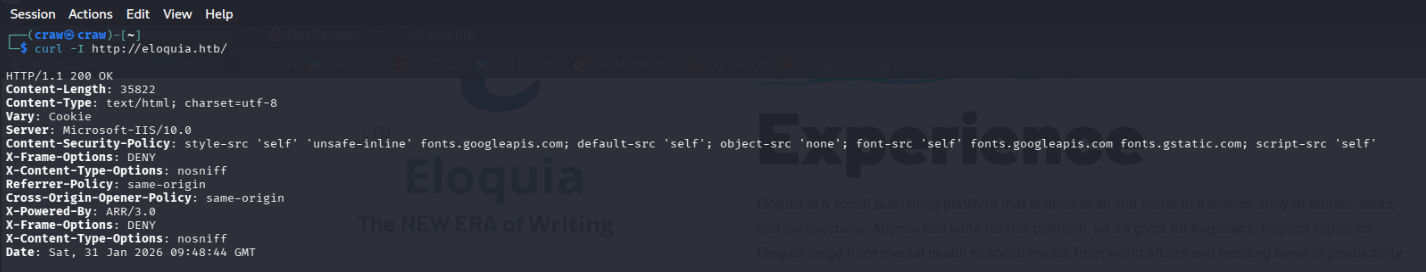
**Automated Server Vulnerable Scanning**

* **Action:** We performed a **Nikto** scan against the target IP to check for common server misconfigurations and outdated software.
* **Result:** The scan re-confirmed the server is running **Microsoft-IIS/10.0** and validated the global redirect rule pointing to http://eloquia.htb/.
* **Analysis:** While Nikto flagged missing security headers (like X-Frame-Options), it reported "No CGI Directories found," suggesting that standard IIS exploits are unlikely to work. This reinforces that our primary path to exploitation lies within the custom **Django** application logic we identified earlier.

**Step 14:**

**Domain Configuration & Fingerprinting**

* **Server Verification:** We verified the fix using curl, which returned a **200 OK** status. The response headers revealed critical information about the target:
  + **Server:** Microsoft-IIS/10.0
  + **Component:** ARR/3.0 (Application Request Routing) – *This indicates the server is using a reverse proxy, which is a key area to investigate for vulnerabilities.*



**Step 15:**

**Session Cookie Analysis**

* **Action:** We isolated the HTTP headers to check for session management using the command curl -I http://eloquia.htb | grep -i set-cookie.
* **Result:** The command returned no output. This confirms that the web server does not issue a Set-Cookie header (and thus no session ID) when a user simply visits the landing page.

