Sri Lanka Institute of Information Technology

Bug Bounty Report

Reflected XSS Vulnerability, Absence of Anti - CSRF Token and Clickjacking.

Target URL: https://grantadvisor.org

Student Name – Y.M.L.K Bandara

Student ID – IT23620834

IE2062 - Web Security

B.Sc. (Hons) in information Technology Specializing in Cyber Security

Table of Contents

Executive Summary

This report summarizes the results of a manual web security assessment of https://grantadvisor.org/. The assessment focused on user-facing web functionality (search, forms, account/profile actions) and used non-destructive techniques (browser testing, Burp Suite, OWASP ZAP). Three confirmed vulnerabilities were discovered and validated with safe proof-of-concepts:

- Reflected Cross-Site Scripting (XSS) - input from the search (or other reflected parameter) is rendered without proper encoding, allowing attacker-supplied JavaScript to execute in a user's browser (OWASP A07). Severity: High.
- Missing CSRF Protection - several state-changing endpoints accept cross-origin POSTs without verifying an anti-CSRF token or adequate origin checks, enabling an attacker to induce authenticated users to perform unwanted actions (OWASP A08). Severity: High.
- Clickjacking (Missing Frame Protection) - the site does not send X-Frame-Options or a CSP frame-ancestors directive, allowing the site to be framed and user interactions to be hijacked via overlays. Severity: Medium.

Impact: together these issues allow attackers to steal sessions, perform actions on behalf of users, or trick users into authorizing sensitive operations - potentially leading to account compromise, data disclosure, or fraudulent transactions. No destructive testing was performed and any sensitive output in PoCs has been redacted in this report.

Methodology

The assessment followed a disciplined, repeatable process to discover, validate and document web application vulnerabilities while maintaining ethical constraints and minimizing impact. Testing was manual-led and supported by industry tools where appropriate; automated scans were used only to assist enumeration and reduce human error. The workflow is broken into four phases: Reconnaissance, Scanning, Exploitation, and Verification. Each phase describes concrete steps, tools used, expected outputs, and safety controls.

1. Reconnaissance — understand the target

Goal: Build a model of the application surface (hosts, endpoints, technologies) so testing focuses on likely attack vectors.

Steps

- Passive enumeration of domain and assets to avoid noisy traffic.
- Fingerprint technologies (web server, frameworks, WAF) to select appropriate payloads.
- Identify likely application entry points (forms, query parameters, file endpoints).

Tools & why

- Amass - discover subdomains and related services. Useful when scope includes subdomains.
- WhatWeb - lightweight tech- fingerprint to identify server software, frameworks, and plugins.
- wafw00f - detect presence of a web application firewall so payloads and testing cadence can be adjusted.

2. Scanning & Mapping — enumerate inputs and attack surface

Goal: Create an accurate site map of endpoints, parameters and input vectors for focused manual testing.

Steps

- Map web application pages and capture all parameters (GET/POST).
- Run lightweight, non-invasive scans to detect misconfigurations and visible weaknesses.
- Record all endpoints that accept user input (search, forms, file uploads, profile endpoints).

Tools & why

- Nmap - verify open ports and services (where in-scope), useful for environment awareness.
- OWASP ZAP (spider + passive scan) - crawl site to enumerate pages and discover inputs; passive scan helps highlight obvious misconfigurations without active exploitation.
- Burp Suite (Proxy) - intercept requests while browsing to capture parameters, cookies, and headers for later manipulation. Burp's site map is essential for manual testing.

3. Exploitation — targeted manual testing and PoC construction

Goal: Manually test prioritized input points for meaningful vulnerabilities and produce minimal, non-destructive proofs-of-concept that demonstrate impact.

Approach

- Prefer manual, context-aware testing over blind automation. Use automated tools to confirm findings when helpful, but craft PoCs manually so results are reproducible and explainable.

Test cases & techniques

The vulnerabilities identified during reconnaissance and scanning were further tested to confirm exploitability using controlled Proof-of-Concepts (PoCs) against test accounts and a local demo environment.

- **Reflected XSS:** Injected payloads into the search input and observed them reflected and executed in the browser.
- **CSRF:** Built forged requests from a locally hosted PoC form (generated in Burp) that performed profile updates while a victim session was active; attacker values were reflected in the URL and, in a follow-up test, persisted to the victim profile.
- **Clickjacking:** Verified the site can be framed (no X-Frame-Options / frame-ancestors); created a labelled demo page embedding the site in an <iframe> to demonstrate click-redirection risks.

Tools & why

- Burp Suite (Repeater / Intruder) - craft and replay requests, iterate payloads, and enumerate numeric IDs.
- Browser DevTools - inspect DOM, debug JS, and confirm execution of payloads and console output.
- sqlmap (optional, controlled) - for confirmation only if manual evidence indicates blind/complex SQLi; used carefully to avoid destructive actions.

4. Verification & Documentation

Goal: Confirm findings, eliminate false positives, and produce a clear, reproducible report (with remediation steps).

Steps

- Reproduce each finding multiple times (different browsers / clean sessions) to ensure reliability.
- Capture evidence: screenshots, raw HTTP requests/responses (Burp export), and exact payloads.
- Map each vulnerability to its OWASP category and provide impact analysis (confidentiality, integrity, availability).
- Provide prioritized remediation guidance (concrete code/config examples: parameterized queries, output encoding, CSRF tokens, X-Frame-Options/CSP, authorization checks).

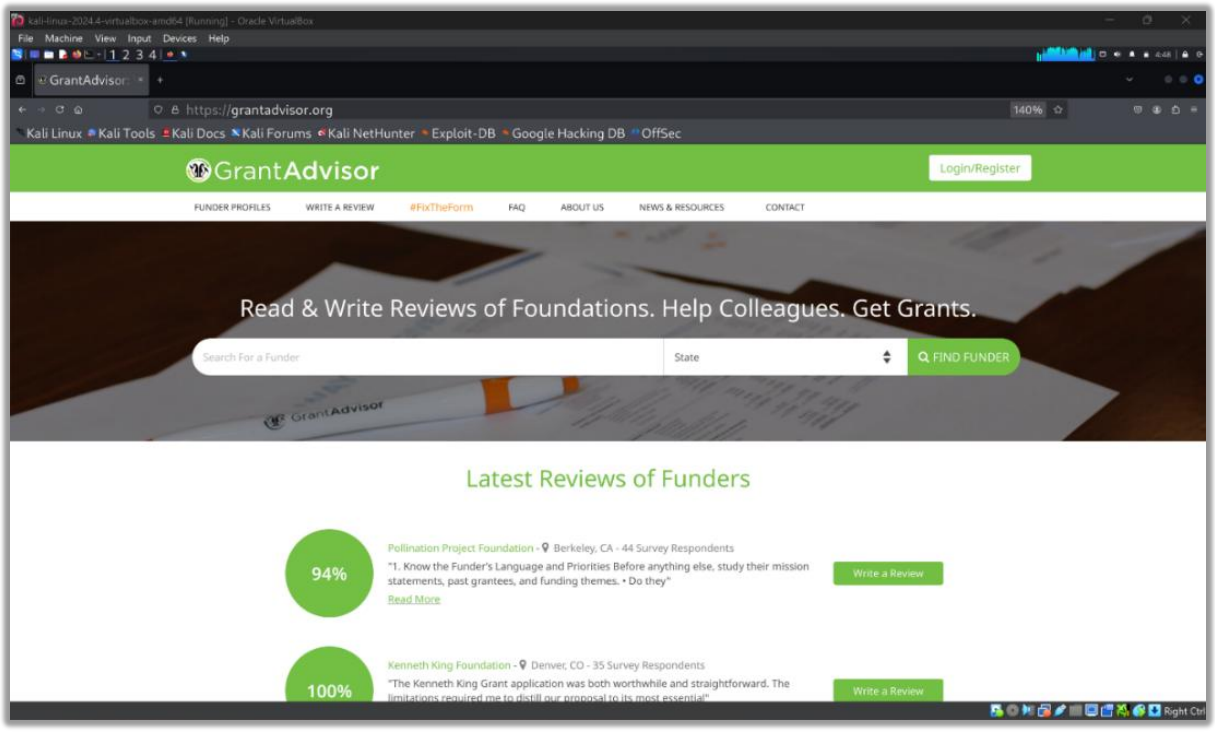
Deliverables

- Executive summary, methodology, detailed findings (for each vuln: description, discovery steps, PoC, impact, remediation), and an appendix with raw evidence and test timestamps.

Why this order

- Recon → Scan → Exploit → Verify gives a logical escalation: first understand the target, then safely find inputs, then validate and demonstrate issues, and finally produce actionable remediation.

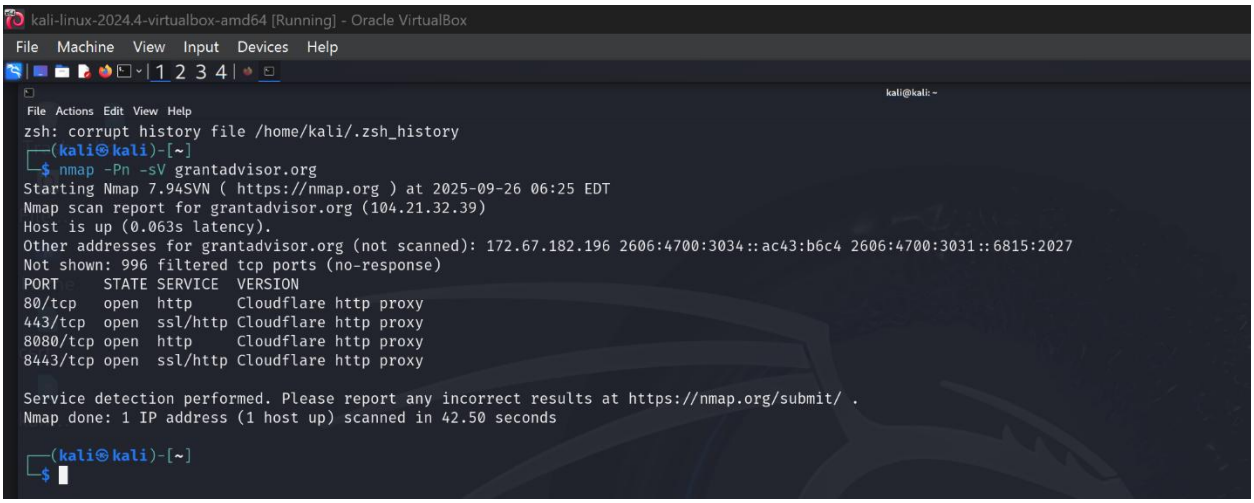
Target URL: https://grantadvisor.org (via https://www.openbugbounty.org/bugbounty-list/)



Reconnaissance: Gather information about the target.

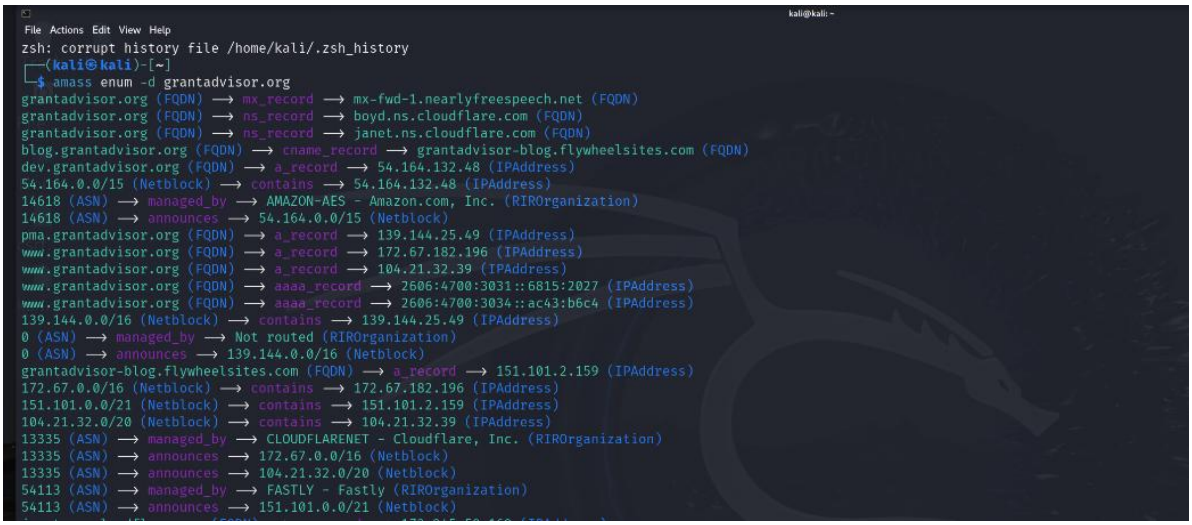
Nmap – Used to find hosts, open ports, and running services on a network.

I ran *nmap -Pn -sV grantadvisor.org* command to perform a service/version scan without host discovery; Nmap resolved the domain to 104.21.32.39, found ports **80, 443, 8080, 8443** are open.



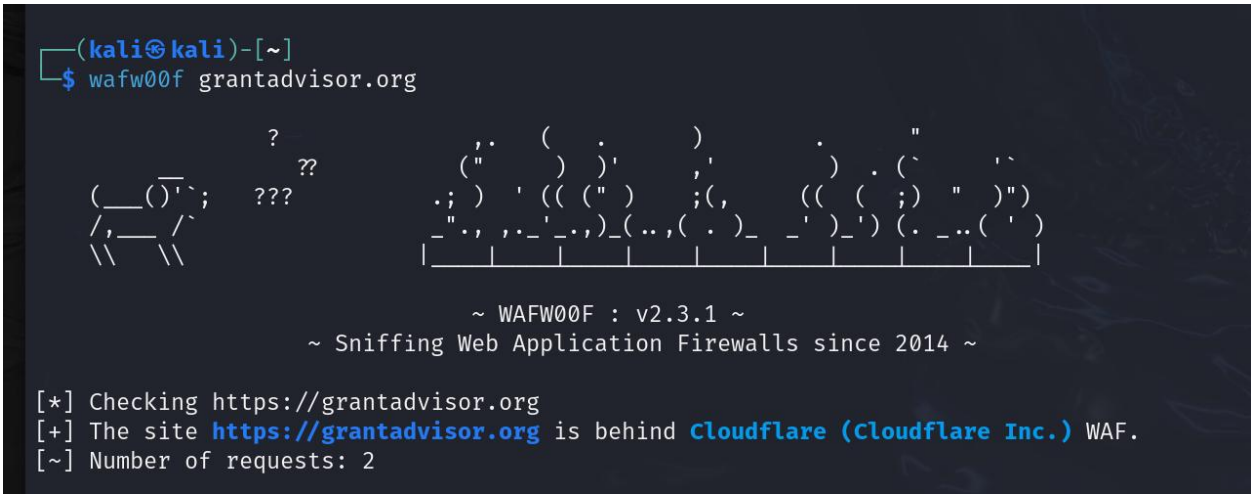
Amass – Used to find subdomains and map the network footprint of a domain.

I ran *amass enum -d grantadvisor.org* command to discover subdomains, DNS records, IP addresses, netblocks, and ASN information for mapping the domain's attack surface.



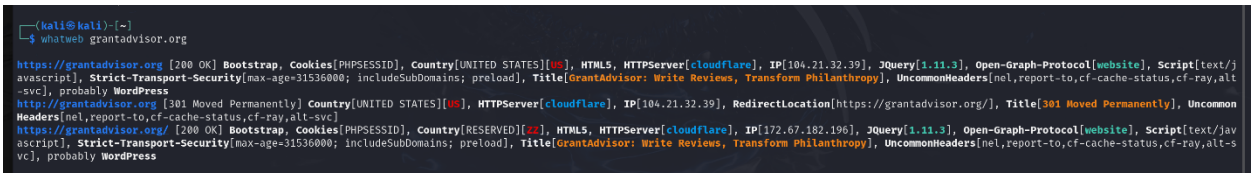
WAFW00F - detects and identifies a target's Web Application Firewall (WAF).

I ran *wafw00f grantadvisor.org* command and found out that it is protected by **Cloudflare's WAF** (2 requests were used).



Whatweb – to identifies technologies used on a website.

I ran *whatweb grantadvisor.org* command to fingerprint the website and identify the technologies it uses.



Vulnerability 1 - Reflected Cross-Site Scripting (XSS)

Reflected XSS in search functionality of https://grantadvisor.org/

OWASP Category: A03 – Injection.

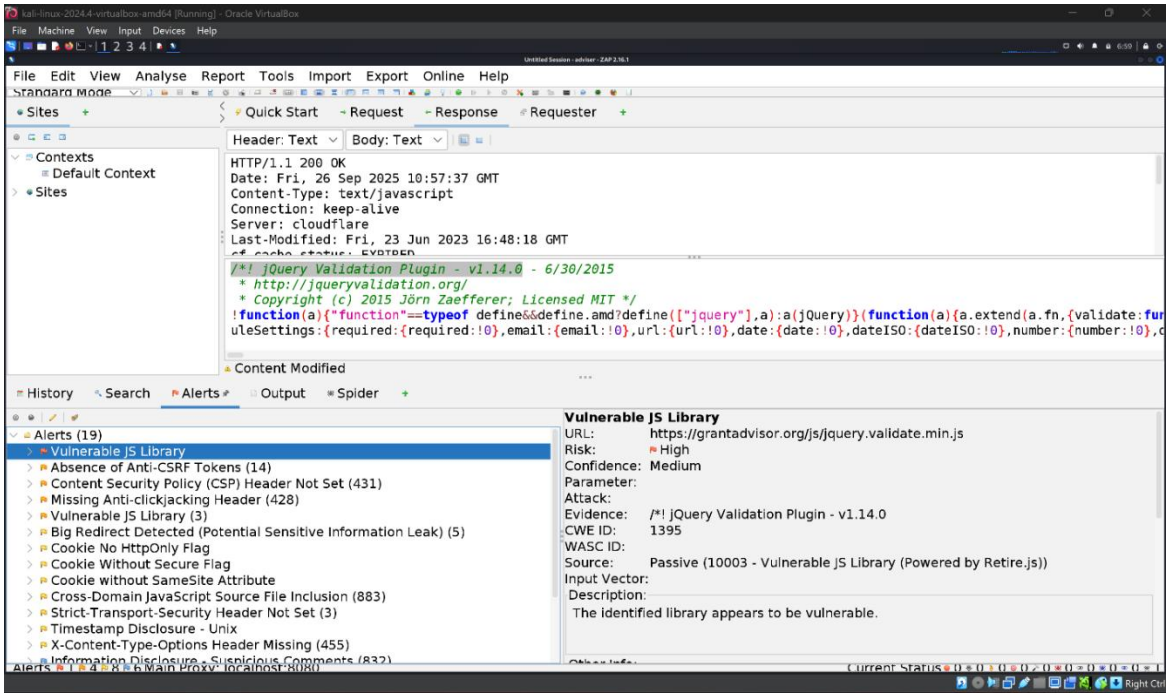
Vulnerability Description

XSS is a vulnerability that allows attackers to inject malicious scripts into web pages viewed by other users. Reflected XSS occurs when user input (like URL parameters or form fields) is immediately included in the page response without proper validation, causing the script to execute in the victim's browser.

Severity Rating: High (attacker-controlled scripts can execute in victims' browsers)

Vulnerability Discovery

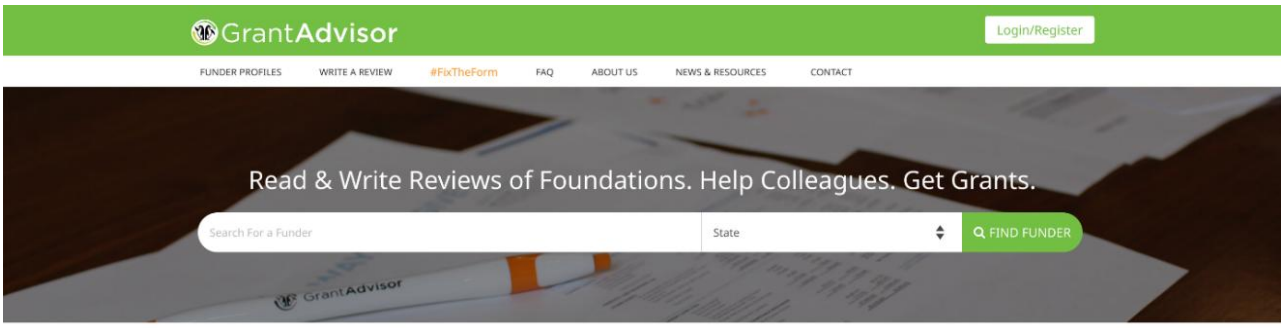
Ran an initial information-gathering pass using OWASP ZAP (site spidering and passive scan). ZAP flagged a client-side dependency as a vulnerable JavaScript library.



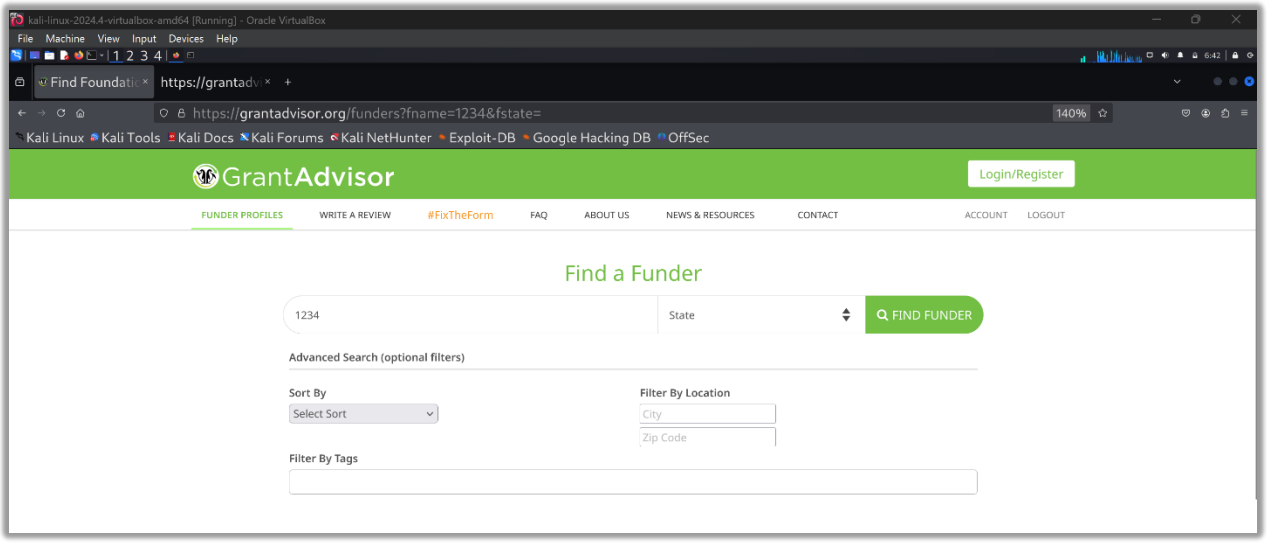
So, I suspected that this site might be vulnerable to reflected or DOM XSS when ZAP flags a vulnerable client JS library.

Proof of Concept (PoC)

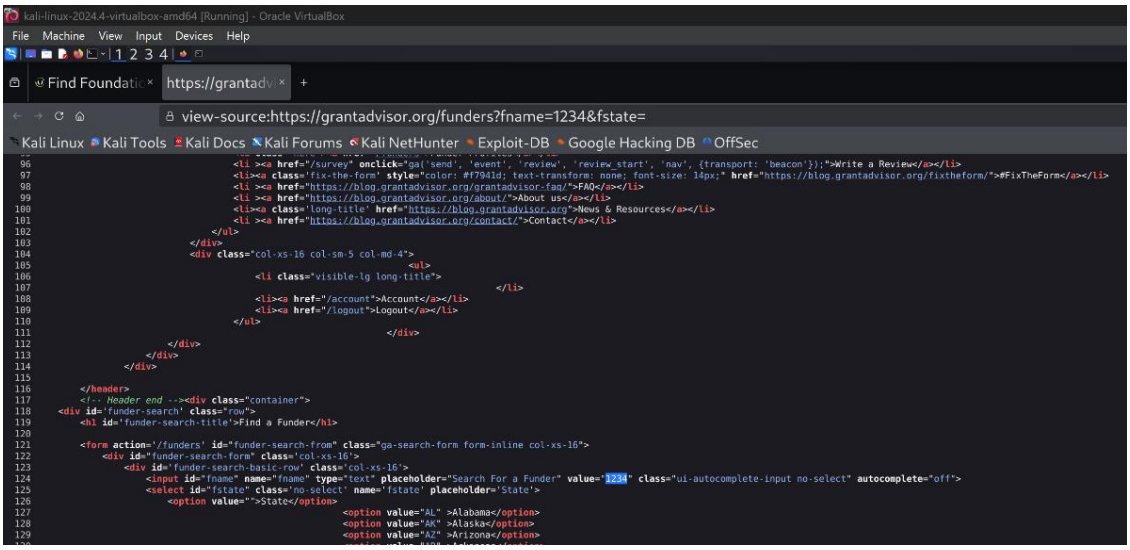
01. I browsed the site manually and identified the search input as a likely reflection point.



02. I entered '1234' into the search box to observe how the application returns and renders user input.



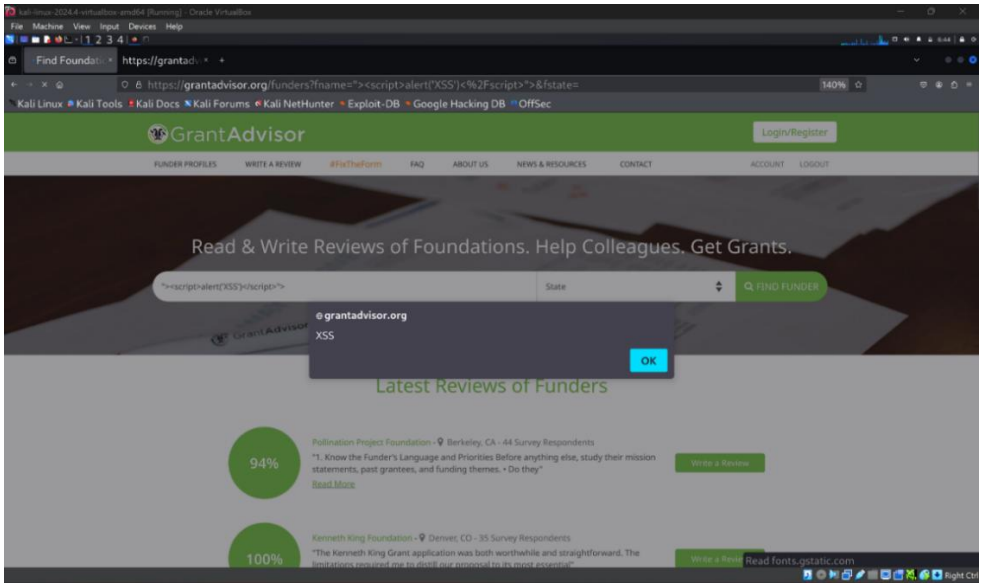
03. I viewed the HTML/response and observed the submitted value reflected back without filtering or encoding.



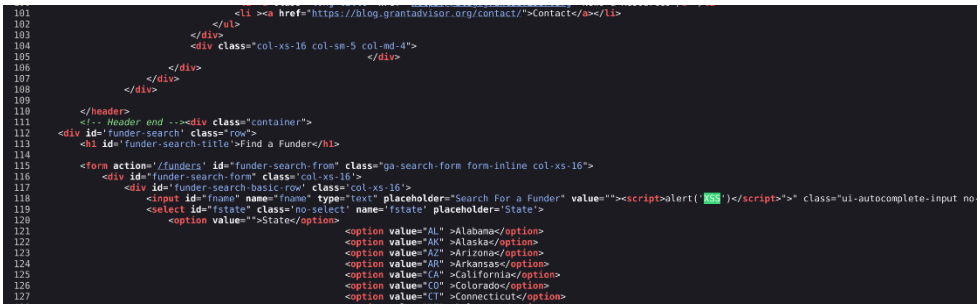
Exploitation

I entered the payload *"><script>alert('XSS')</script>"*> into the search bar to test for execution. The browser displayed an alert dialog, confirming the XSS was executed.

- The payload (*"><script>alert('XSS')</script>">)* breaks out of the HTML context where your input is placed**.** The initial " and > close whatever attribute or tag context the application placed your input into (for example value="...here..." or inside a tag). By closing that context the payload makes the browser treat what follows as normal HTML rather than literal text.



Why this is reflected XSS:



Since the payload appears in the page source, I confirmed this is reflected XSS.

Reflected XSS: The server echoes input into the HTML. You can see it in the raw page source. The attack runs immediately when the page loads.

DOM-based XSS: The server does not include input in the HTML. Instead, a client-side script reads your input from the URL, hash, or other sources and injects it into the page. In that case, you won't see it in view-source, only in the live DOM after JavaScript runs.

In a real attack scenario, an attacker can craft a URL that includes malicious script and trick a user into clicking it (for example, via email or a forum post). When the user opens the link, the injected script runs in their browser context and can perform actions such as reading visible page content or manipulating the page. As a result, the user's session or sensitive information may be exposed or abused without their knowledge.

Impact of Reflected XSS

Reflected XSS is a client-side vulnerability where malicious scripts execute in a user's browser when they interact with a crafted link. While the attack doesn't persist on the server, it can affect any user who visits the malicious URL.

Impact:

- Session hijacking: Attackers can steal cookies or session tokens to impersonate users.
- Unauthorized actions: Malicious scripts can perform actions on behalf of the victim.
- Phishing attacks: Fake forms or messages can trick users into revealing sensitive information.
- Malware distribution / redirects: Users can be redirected to malicious websites or prompted to download malware.
- Website defacement: Attackers can temporarily modify the content seen by users.
- Targeted attacks: Any user who clicks the crafted link is vulnerable.

Impact specific to GrantAdvisor.org:

- **Loss of reviewer anonymity:** Attackers could exploit reflected XSS to steal session tokens or cookies, revealing the identities of nonprofit reviewers who post honest or critical feedback about foundations. This violates one of GrantAdvisor's core promises - anonymity - and could expose users to professional retaliation, loss of partnerships, or exclusion from future funding opportunities.
- **Review manipulation and misinformation**: Malicious scripts could modify how reviews are displayed in real time, allowing attackers or biased actors to insert false information or alter existing ratings. This could mislead organizations that rely on these reviews when making funding or partnership decisions.
- **Account compromise and impersonation**: If attackers use XSS to capture login details or session cookies, they could hijack staff or reviewer accounts, delete legitimate reviews, or publish fabricated feedback. This undermines the integrity of the entire platform.
- **Reputation and trust damage**: A single visible XSS attack could cause widespread loss of confidence among users and partner foundations. Once users perceive the site as insecure, they may avoid sharing feedback, threatening the platform's role as a trusted transparency tool in the nonprofit funding sector.

Technical Solution to Fix Reflected XSS:

1. **Input Validation / Sanitization:**
    - Validate all user inputs on the server side to allow only expected characters and formats.
    - Reject or properly handle unexpected or dangerous characters.
2. **Context-Aware Output Encoding:**
    - Encode user input before including it in HTML, JavaScript, or URL contexts.
3. **Use Security Libraries / Framework Features:**
    - Modern web frameworks often have built-in XSS protection mechanisms; ensure they are enabled.
4. **Content Security Policy (CSP):**
    - Implement a strong CSP to restrict which scripts can execute in the browser.
5. **Keep Dependencies Updated:**
    - Update any vulnerable JavaScript libraries flagged by tools like ZAP to their latest secure versions.

Vulnerability 2 -Cross-Site Request Forgery (CSRF) - Absence of Anti-CSRF Token

OWASP Category: A08 - Cross-Site Request Forgery (CSRF)

Vulnerability Description

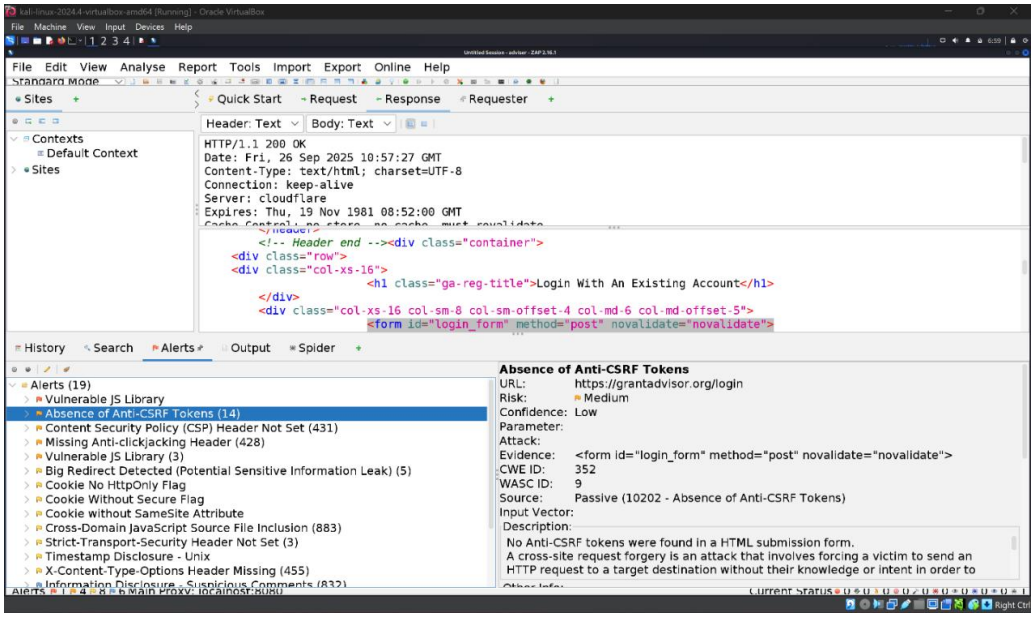
Cross-Site Request Forgery (CSRF) is an attack that tricks an authenticated user's browser into sending malicious requests to a web application without their knowledge. The absence of an anti-CSRF token means the application does not implement unique, unpredictable tokens in requests to verify a user's intent. Without this protection, an attacker can craft malicious links or forms that, when visited by a logged-in user, execute unintended actions such as changing account details or performing transactions on their behalf.

Severity: High - attacker can cause an authenticated user's browser to perform state-changing actions without their consent.

Target URL: https://grantadvisor.org/ - account/profile update & registration endpoints

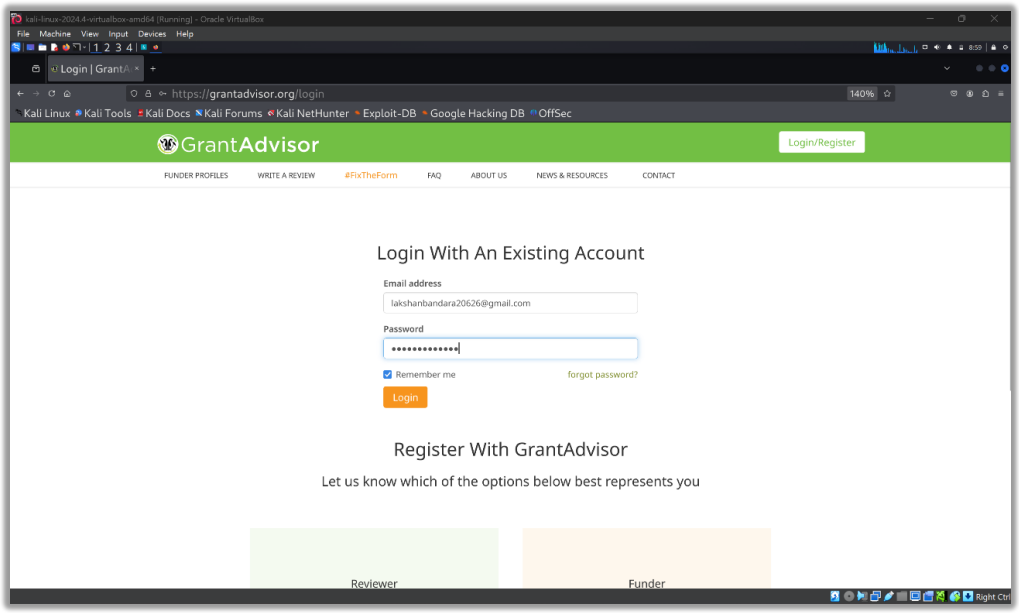
Vulnerability Discovery

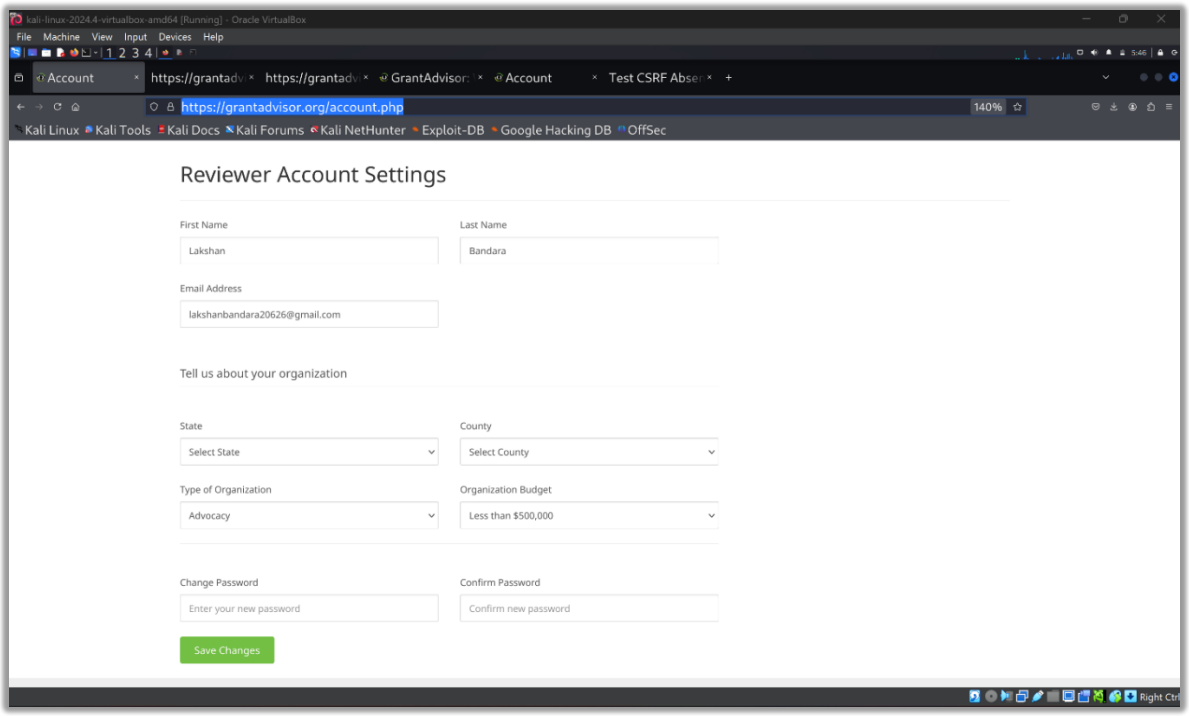
I ran OWASP ZAP's scan - ZAP flagged a missing/absent CSRF token for relevant state-changing endpoints.



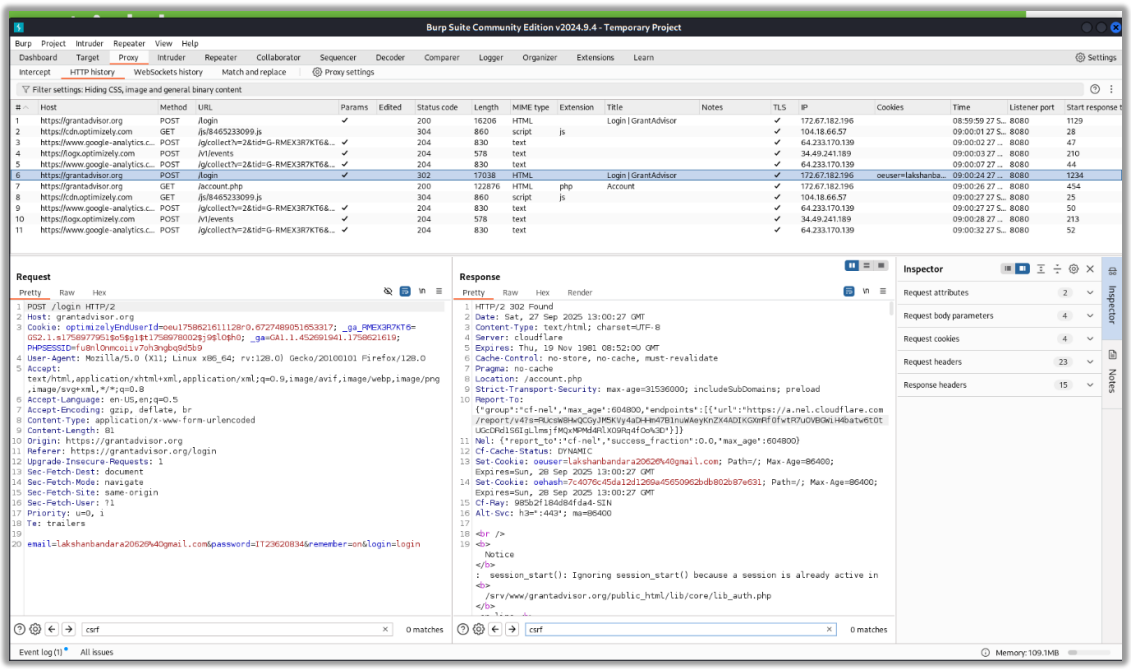
Proof of Concept (PoC)

1. I registered and authenticated on https://grantadvisor.org/ to get a valid session for testing. Then I logged into the site.





2. Then I captured the login/session request with Burp Suite. I searched the captured requests and page source for any CSRF/anti-CSRF token and confirmed it was absent.



How CSRF tokens usually appear (POST Request)

POST /account/update HTTP/1.1

Host: grantadvisor.org

User-Agent: ...

Content-Type: application/x-www-form-urlencoded

Content-Length: 89

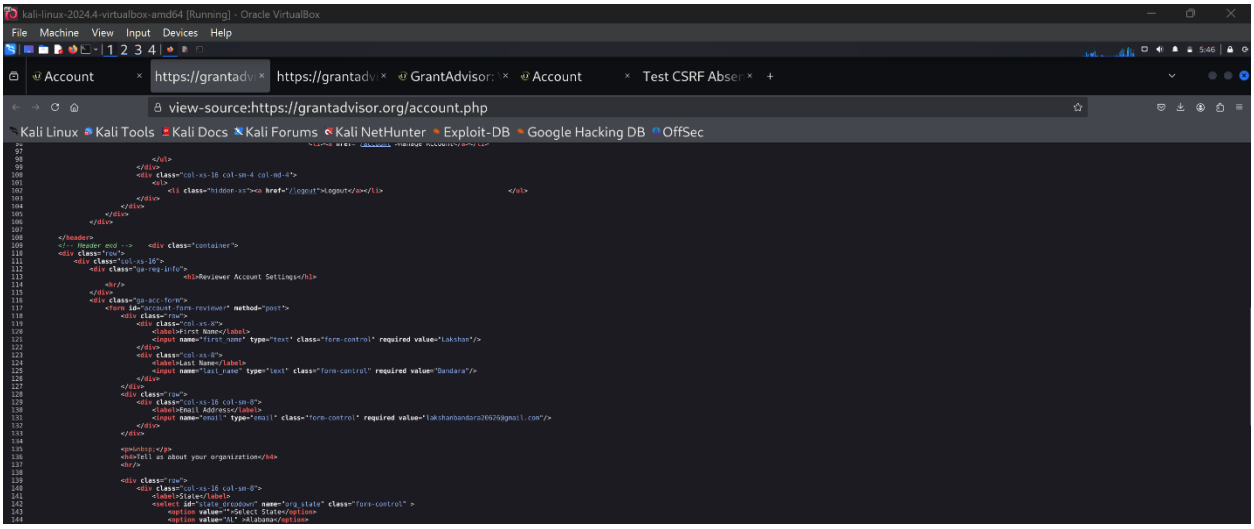
Cookie: session=abcd1234...

csrf_token=b7f9a2c4-3a1e-4d6b-9f2c-1a2b3c4d5e6f&email=user%40example.com

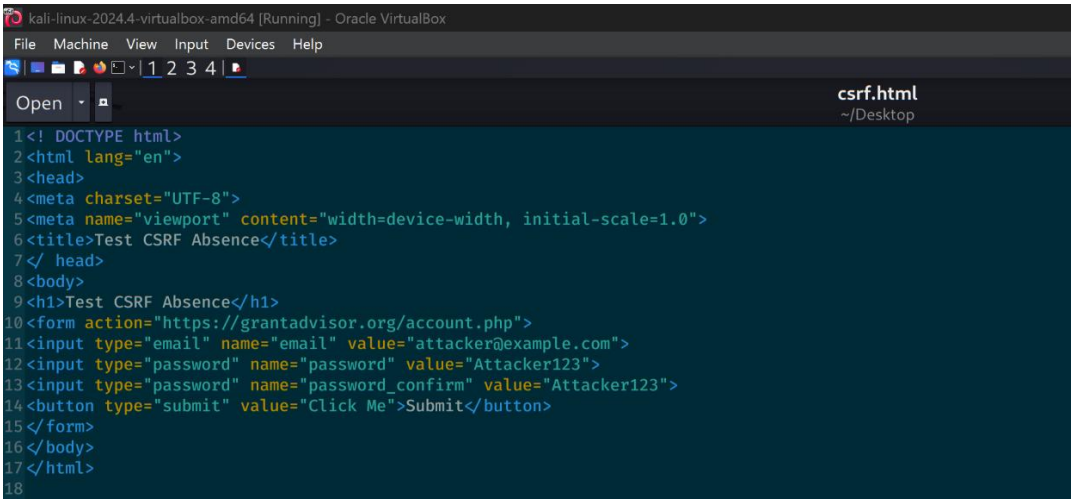
How CSRF tokens usually appear (HTML Form)

```
<form method="POST" action="/account/update">
  <input type="hidden" name="csrf_token" value="b7f9a2c4-3a1e-4d6b-9f2c-1a2b3c4d5e6f">
  <input type="text" name="email" value="user@example.com">
  <button type="submit">Save</button>
</form>
```

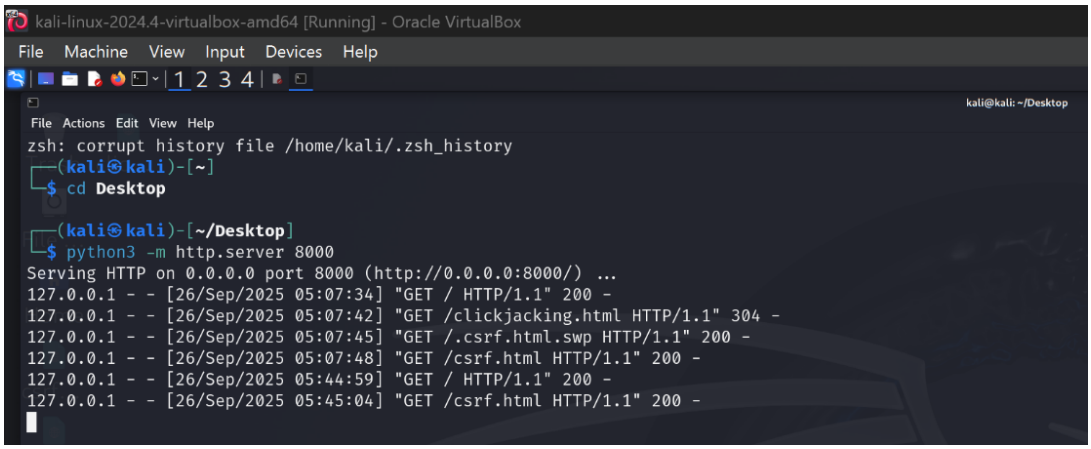
3. I inspected the profile page source to identify form field names (email, password, confirm password) used by the site.



4. I built an HTML page using those field names, that submits the site's profile update endpoint with attacker-controlled credentials.

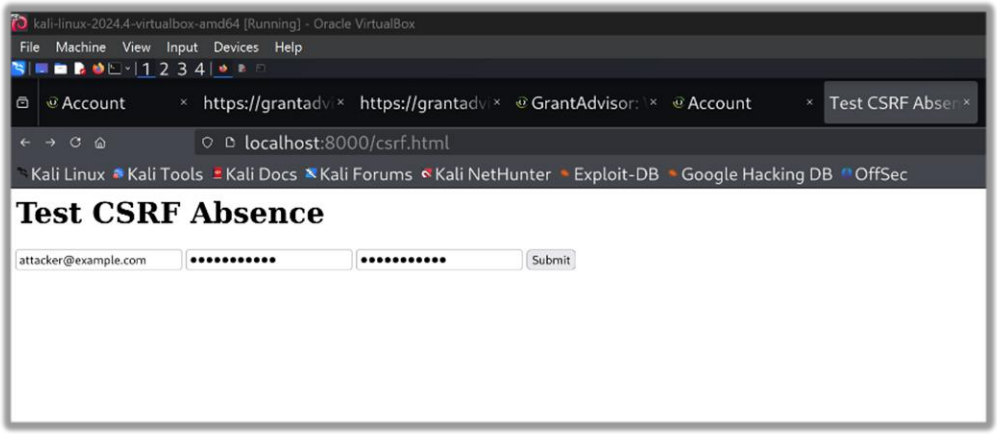


5. I hosted the HTML on a local server using *python3 -m http.server 8000* to simulate an external origin.

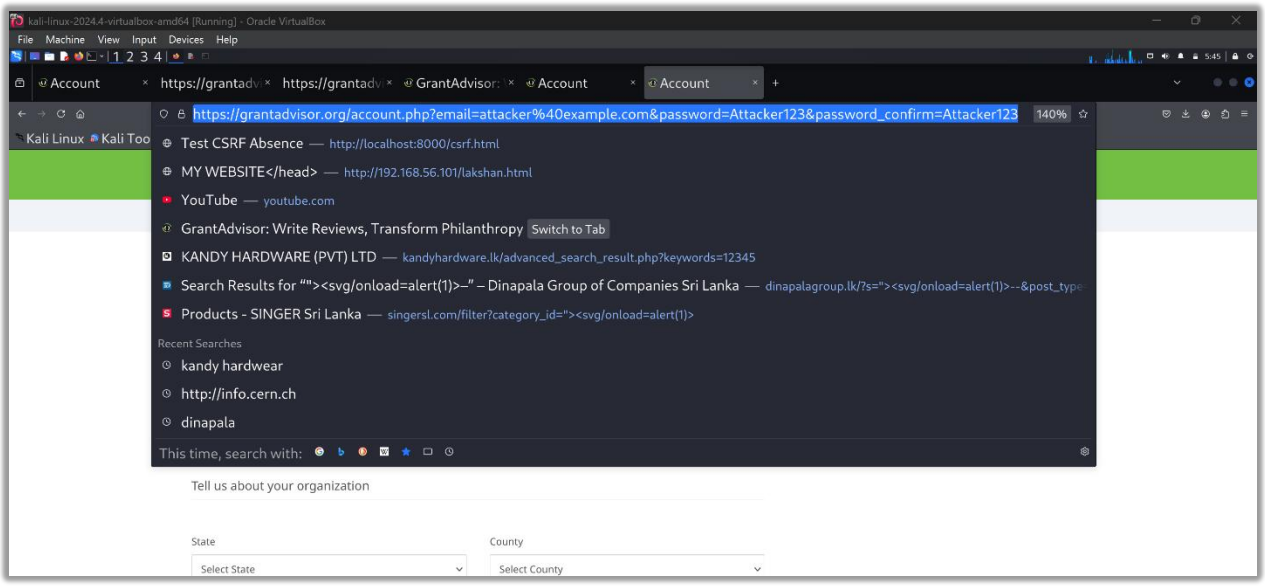


Note - Although I used the same browser, http://localhost:8000 is a different origin than https://grantadvisor.org, so hosting the attacker page locally correctly simulates an external origin for CSRF testing.

6. I opened the attacker page using http://localhost:8000/csrf.html while logged in as the victim and submitted the form.

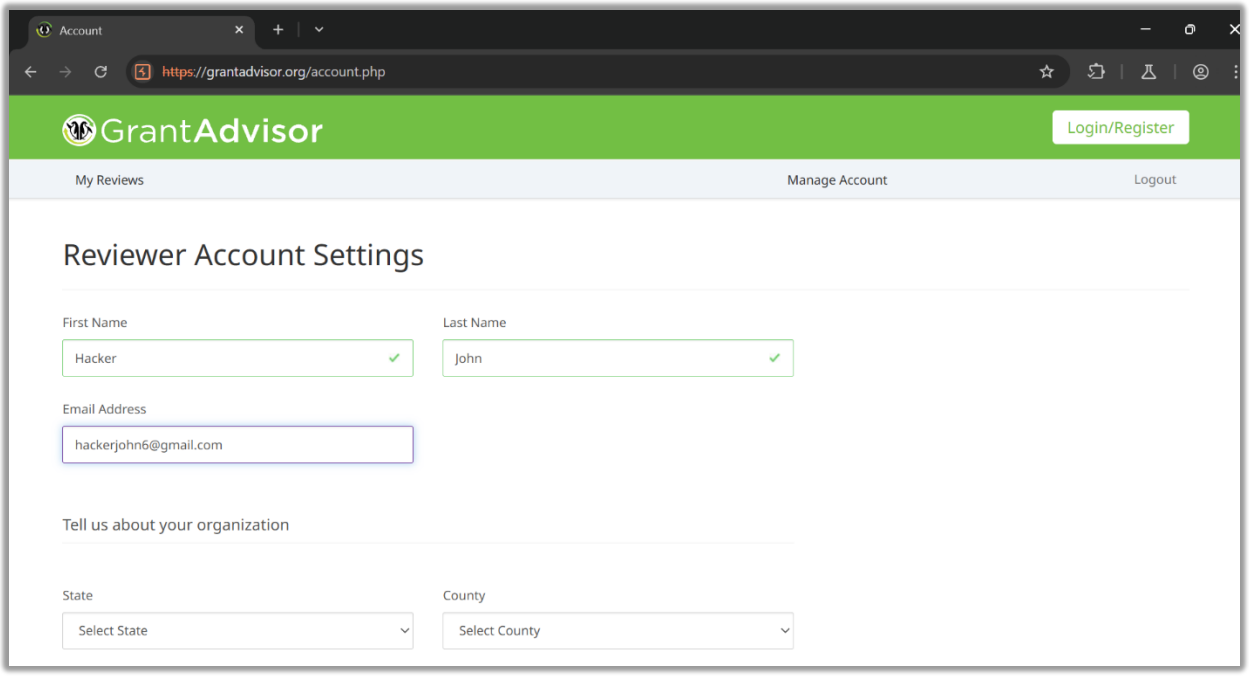


7. The action succeeded despite originating from another origin; it directed me to the victim's logged page and the URL contained the attacker credentials and the victim session was affected.

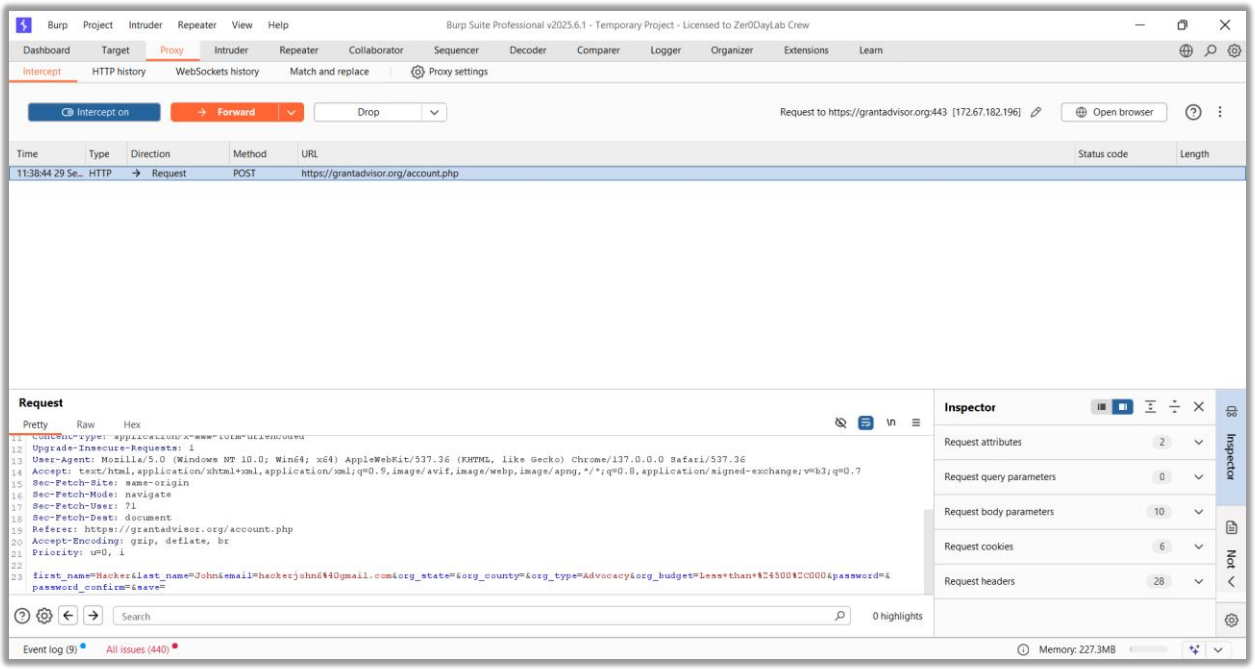


Exploitation

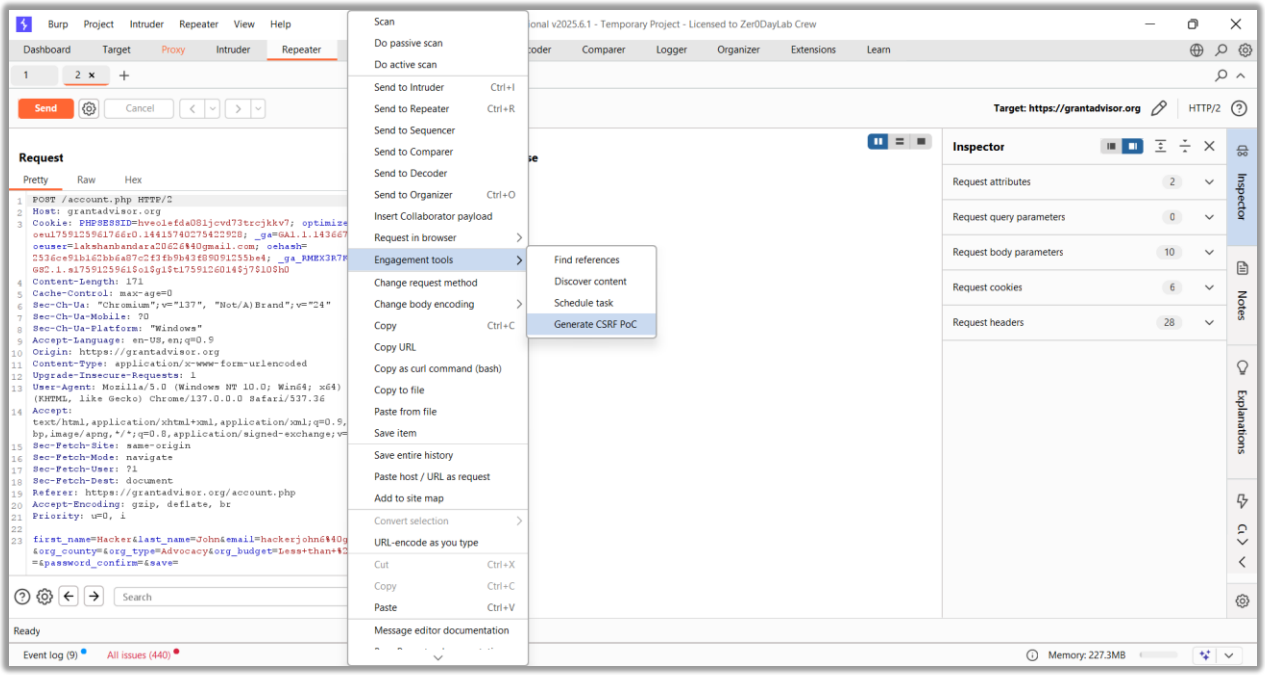
1. First, I updated my own account profile on https://grantadvisor.org/ and saved the change.



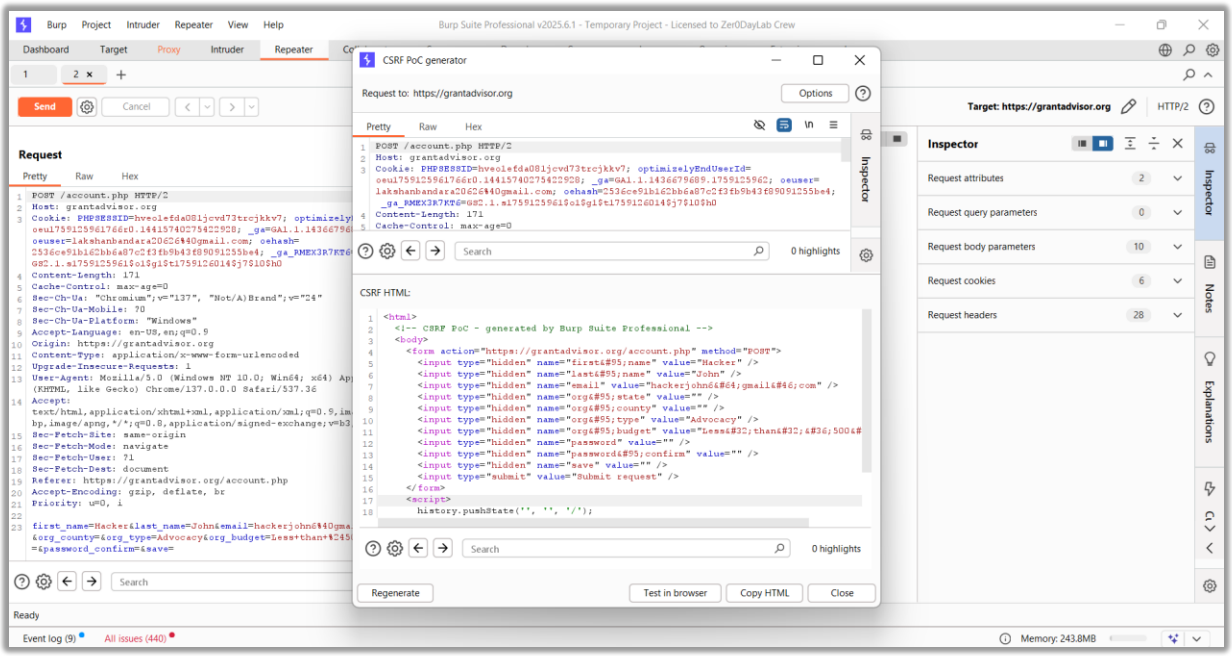
2. I captured the profile-update request in Burp Suite while authenticated.

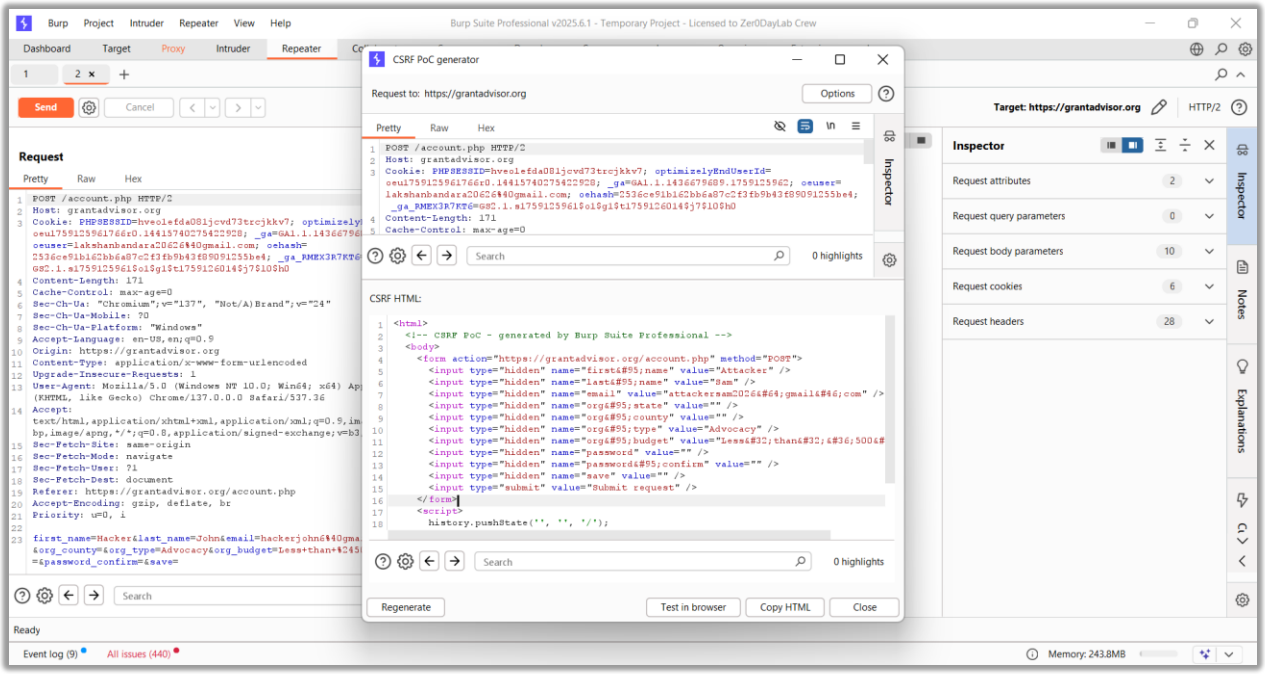


3. After I sent the captured request to Burp Repeater, using Burp's engagement tools I generated a CSRF PoC HTML form from the captured request.



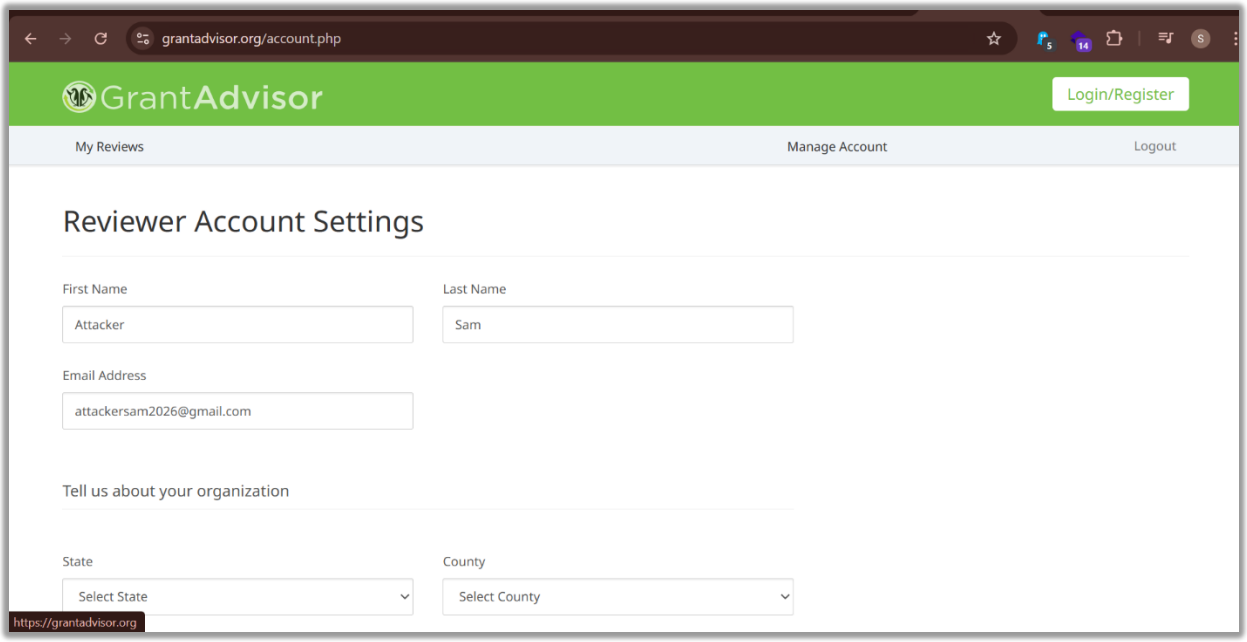
4. I edited the generated PoC form to include the attacker user details I wanted to apply to the victim.





5. I opened the PoC HTML in a browser (simulating the attacker page) and submitted the form while the victim was logged in.

6. The site processed the cross-origin request and redirected to the victim's profile — the profile fields were now changed to the attacker's details.



Why is this CSRF?

The site accepted a profile update request that was sent from another website (the attacker's HTML form). Because the victim was already logged in, their session cookies were automatically included with the request, so the server thought it was a valid action from the victim. Since there was no CSRF token or origin check, the attacker's request was processed as if it came directly from the victim.

In a real attack scenario, an attacker can host a page that silently submits a form or request to the target site and lure an authenticated user to visit it. Because the browser automatically includes the user's session cookies, the site processes the request as coming from the legitimate user. As a result, the attacker can cause the user's account to be changed or actions to be performed on their behalf without their consent.

Impacts of the CSRF

- Account takeover / credential change: Attacker can change victim's email/password and take over accounts.
- Unauthorized actions: Modify profile data, change settings, or perform other state-changing actions as the victim.
- Privilege abuse: If admin workflows are vulnerable, attackers could enact higher-impact changes.

- Phishing / persistent access: Combine with social engineering to lock users out or redirect them.
- Wider attack surface: Any logged-in user who visits the attacker page is affected.

CSRF – Impact (Specific to GrantAdvisor.org)

- **Unauthorized review manipulation:** Attackers could trick logged-in users into unknowingly submitting or editing reviews. This means foundations could suppress negative feedback or create artificial positive reputations by exploiting unsuspecting users.
- **Persistent account takeover:** A CSRF attack that changes a user's email or password gives attackers permanent access to that account. They could monitor which foundations a nonprofit is researching, gaining insider intelligence that could affect grant competitions or negotiations.
- **Exposure of reviewer identity:** By silently altering account settings, an attacker could remove anonymity protections or make user details public. This would violate the platform's commitment to safe and confidential participation for nonprofit staff.
- **Data integrity and large-scale impact:** A coordinated CSRF campaign sent through phishing emails or embedded links could compromise many accounts at once, corrupting the review database with fake content and reducing the platform's credibility as a source of genuine, community-driven insight.

Technical remediation

1. Implement server-side CSRF tokens (unique per session/form) and validate them on all state-changing requests.
2. Validate Origin/Referer headers for sensitive endpoints and reject mismatches.
3. Set cookies with SameSite=Lax or Strict where appropriate to reduce cross-site cookie sending.
4. Enable framework CSRF middleware or double-submit cookie patterns if available.
5. Require re-authentication for critical changes (current password or MFA).
6. Use POST for state changes and reject sensitive GET requests.

Vulnerability 3 — Clickjacking (Missing Anti-Clickjacking Header)

OWASP Category: A05 — Security Misconfiguration

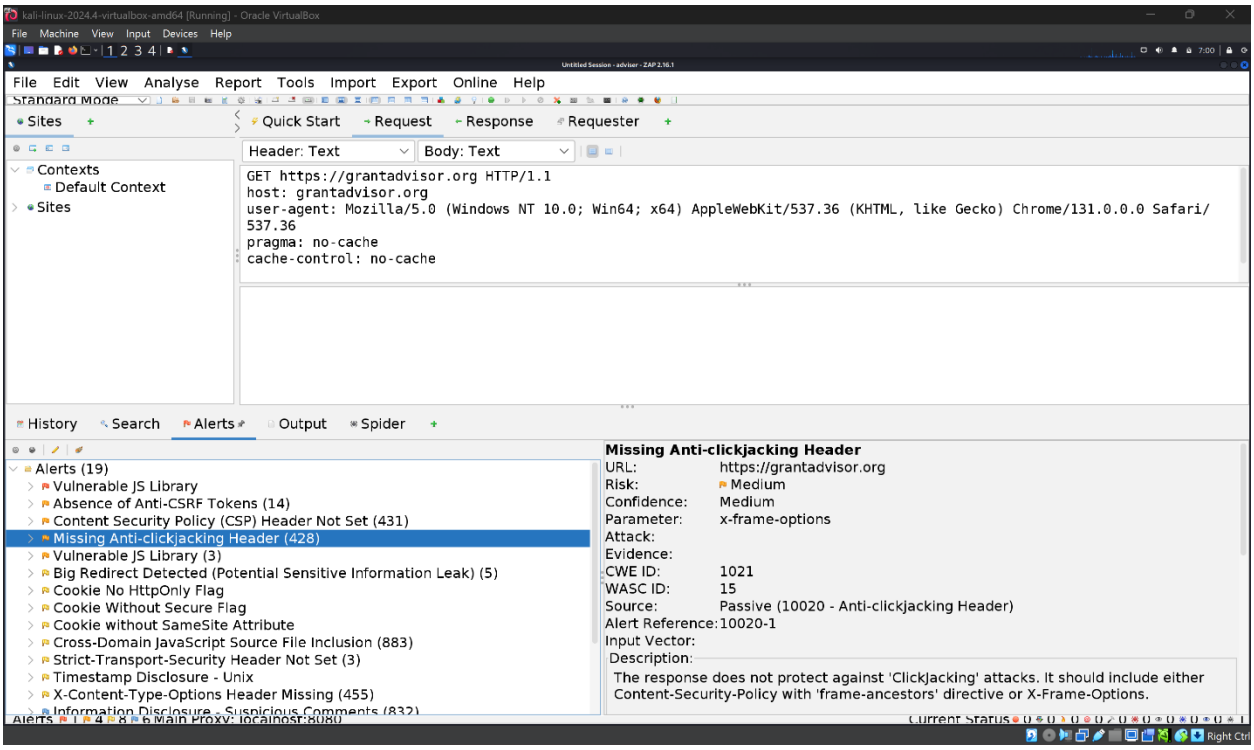
Vulnerability Description

Clickjacking is an attack that tricks a user into clicking on something different from what they perceive, potentially leading to unauthorized actions or information disclosure. The **absence of anti-clickjacking headers** (such as X-Frame-Options or Content-Security-Policy: frame-ancestors) allows the application to be embedded within an attacker-controlled iframe. This enables attackers to overlay deceptive elements and hijack user clicks, performing unintended actions within the application.

Severity: Medium - attacker-controlled UI overlay can trick authenticated users into performing actions.

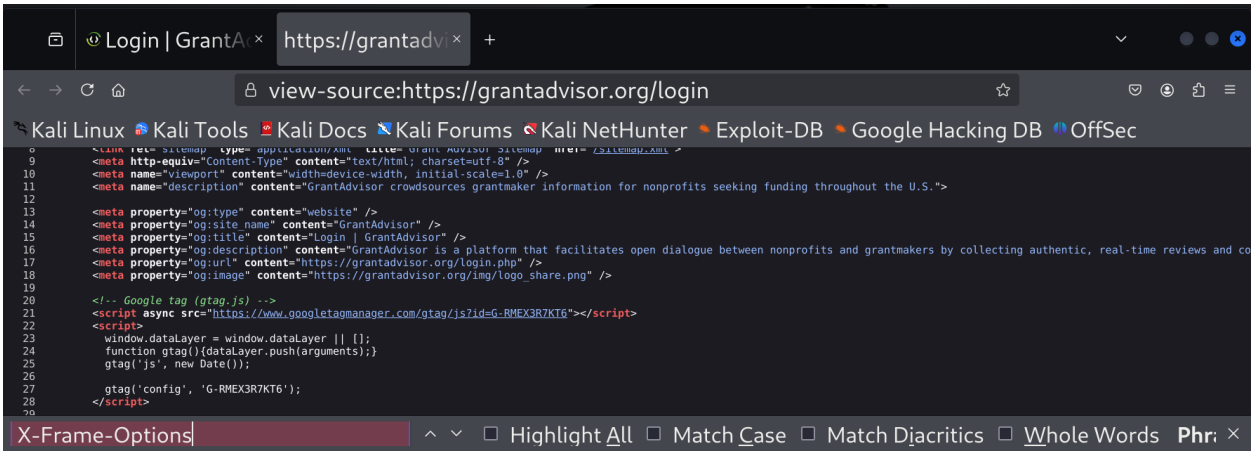
Vulnerability Discovery

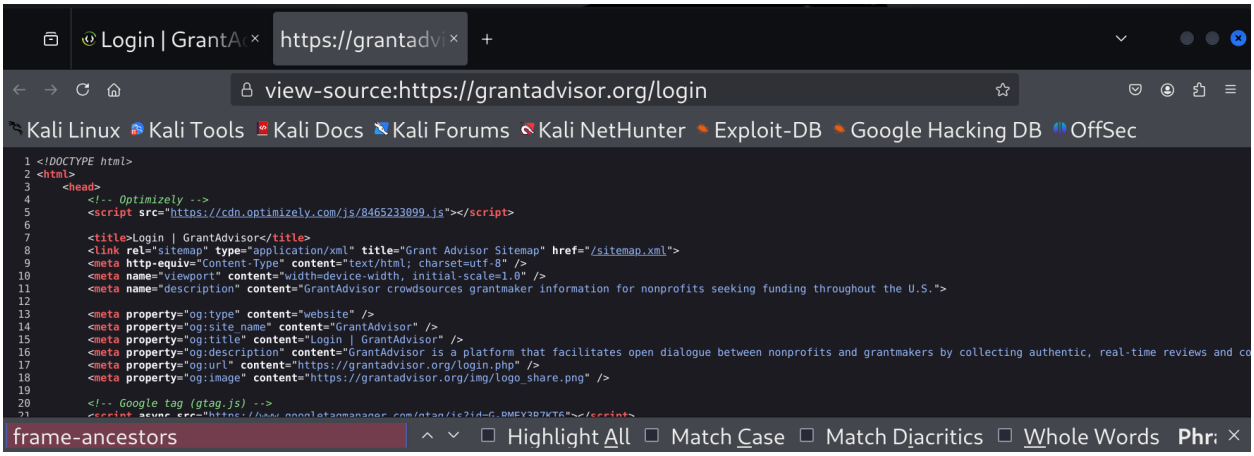
I ran OWASP ZAP against the site — ZAP flagged a missing anti-clickjacking header.



Proof of Concept (PoC)

1. I inspected the site source for frame protection — I searched the HTML for any X-Frame-Options headers or Content-Security-Policy: frame-ancestors and found none.

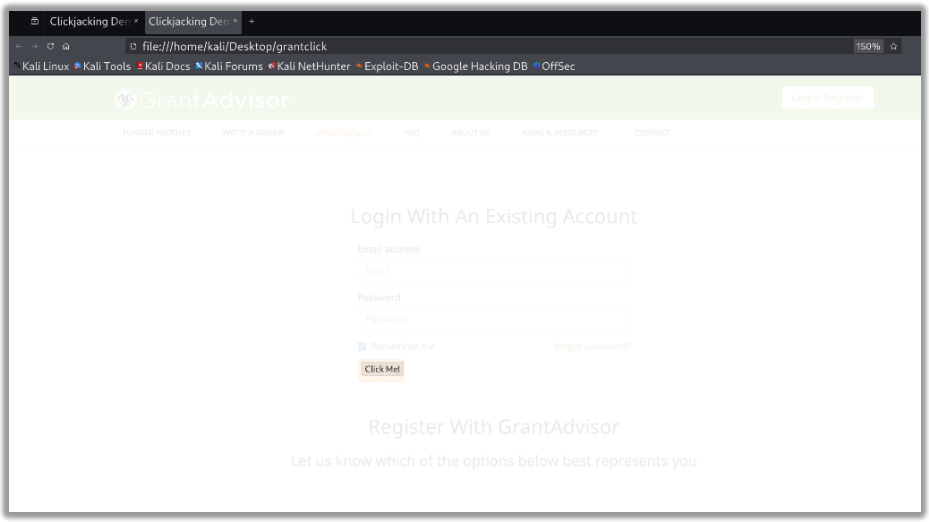




2. Next, I created an attacker HTML page with an <iframe> — I embedded https://grantadvisor.org/ inside an <iframe> and set low opacity so the framed site is visible but a button overlays it.
    - I added a visible “Click me” button over the iframe — the button is positioned above the iframe so a user thinks they are clicking the button while actually interacting with the framed site beneath.

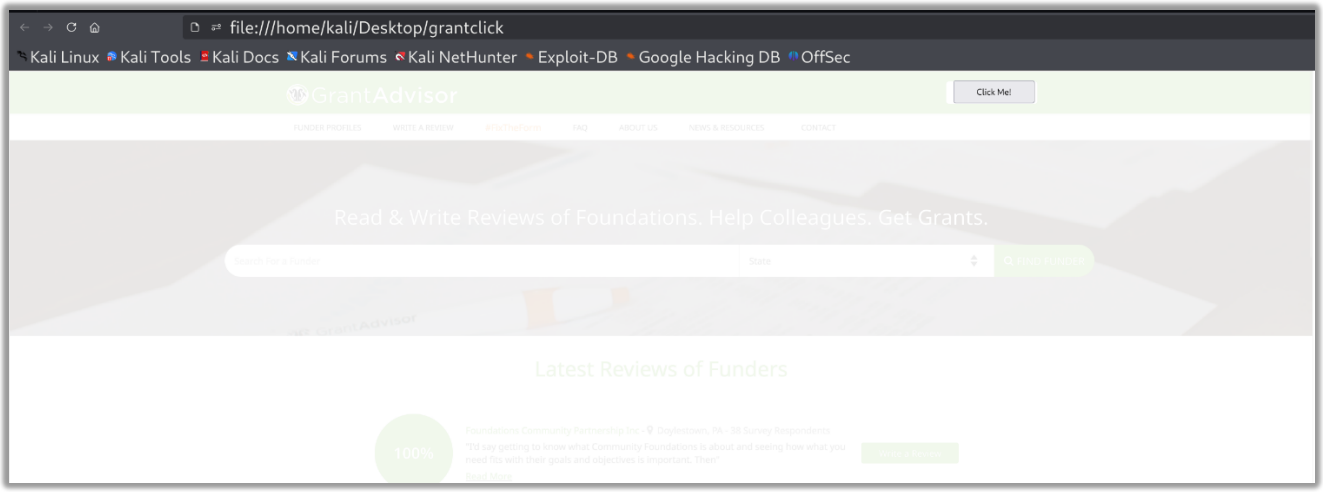
```html
<!DOCTYPE html>
<html>
<head>
  <title>Clickjacking Demo</title>
  <style>
    iframe {
      width: 2000px;
      height: 1000px;
      opacity: 0.1; /* partially transparent */
      position: absolute;
      top: 0;
      left: 0;
      z-index: 2;
    }
    button {
      position: absolute;
      top: 435px;
      left: 800px;
      z-index: 2;
    }
  </style>
</head>
<body>

  <button>Click Me!</button>

  <iframe src="https://grantadvisor.org/login"></iframe>

</body>
</html>
```

3. After opening the locally hosted page I created, the site loaded inside the faded iframe and my visible ‘Click me’ button sat on top. When I clicked the button, the click was delivered to the framed page and triggered the site’s UI.

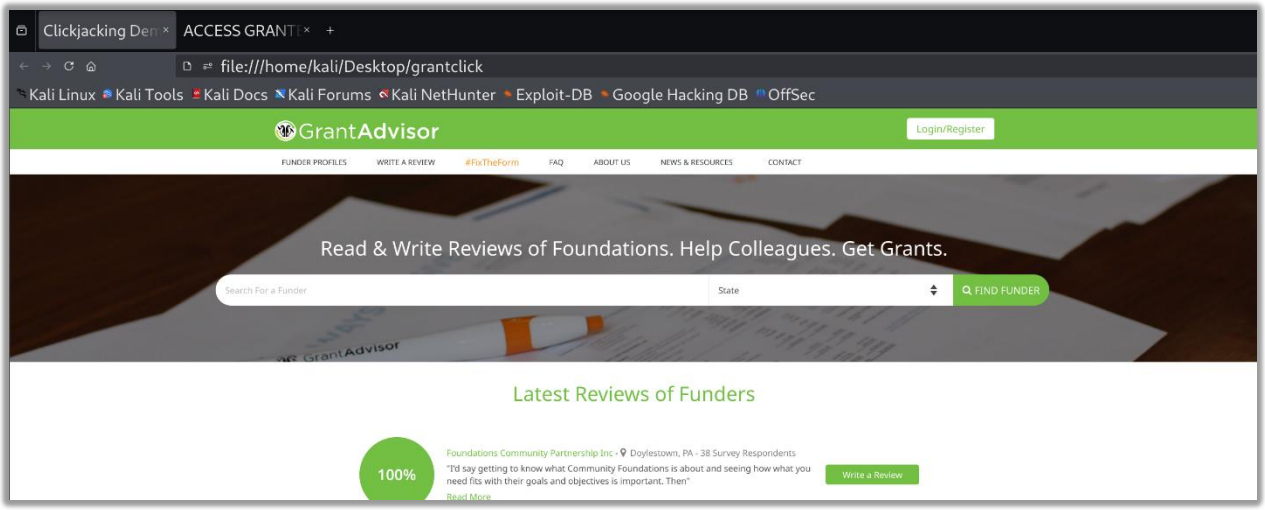


Exploitation

1. Then I positioned the iframe to display the target website's sign-in page with the sign-in button visible.



2. After that, I created a transparent overlay button positioned exactly over the legitimate sign-in button on the iframe:

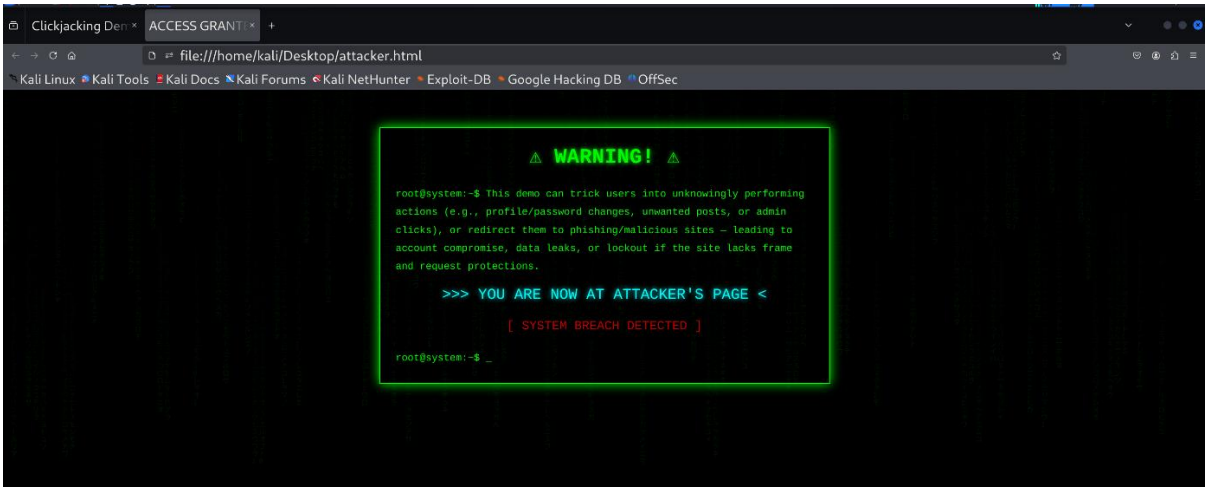


3. Added malicious redirect functionality to the transparent button:
    - Used onclick event handler with window.open() function.
    - Configured to open a malicious site (attacker-controlled page) in a new tab when clicked.

grantclick

```html
<!DOCTYPE html>
<html>
<head>
  <title>Clickjacking Demo</title>
  <style>
    iframe {
      width: 2000px;
      height: 1000px;
      opacity: 10;
      position: absolute;
      top: 0;
      left: 0;
      z-index: 2;
    }
    button {
      position: absolute;
      top: 15px;
      left: 1450px;
      z-index: 3;
      background-color: transparent;
      border: none;
      cursor: pointer;
      color: transparent;
      width: 125px;
      height: 35px;

    }
  </style>
</head>
<body>

  <button onclick="window.open('attacker.html', '_blank')">  Click Me!  </button>
  <iframe src="https://grantadvisor.org"></iframe>

</body>
</html>
```

4. Gets redirected to attacker's page while believing they clicked the legitimate button.



Note - In a real attack scenario**,** the visible element (for example, a button) can be carefully aligned on top of a sensitive control within the target application (such as account settings, a financial transaction, or an authorization button). As a result, when the user believes they are clicking the visible element, the click is actually passed through to the hidden application and performs an unintended action on their behalf.

This is clickjacking because the site can be framed by another origin and user clicks on apparent UI elements are actually sent to the framed site—there is no X-Frame-Options or Content-Security-Policy: frame-ancestors preventing framing.

Impacts of Clickjacking.

An attacker can trick authenticated users into performing unintended actions by overlaying UI elements on a framed version of the site.

- Unauthorized actions: Users can be coerced into clicking buttons that perform state-changing operations (e.g., submit forms, click “confirm” or “delete”).
- Phishing & credential theft vectors: Overlay UI can prompt users for sensitive input believing it’s safe.
- UX manipulation / fraud: Misleading UI can cause users to perform monetary or administrative actions.
- Targeted attacks on high-privilege users: If admins are tricked, impact can be severe (configuration changes, content removal).
- Erodes user trust: Users may be redirected or see altered interactions leading to reputational damage.

Clickjacking – Impact (Specific to GrantAdvisor.org)

- **Forced and misleading interactions:** If GrantAdvisor pages can be framed, attackers could overlay invisible buttons that trick users into submitting fake 5-star ratings or “liking” certain foundations. This would inflate reputations and distort public perception of grantmakers.
- **Unintended review deletion:** Users could unknowingly trigger hidden “delete” or “report” buttons while interacting with what appears to be a harmless webpage, leading to the quiet removal of critical feedback that other nonprofits rely on.

- **Unauthorized account or privacy changes:** Clickjacking could be used to make users change notification settings, privacy options, or even credentials without realizing it — creating opportunities for further account compromise.
- **Rapid spread through trusted networks:** Because the nonprofit sector relies heavily on professional trust networks and shared resources, a single malicious clickjacking page could quickly reach many users, causing widespread manipulation before administrators detect it.

Technical remediation

- Set X-Frame-Options: DENY or SAMEORIGIN to block framing.
- Use CSP frame-ancestors to control allowed frames.
- Add extra checks (re-auth/confirm) for sensitive actions.
- Design UI so key actions can’t be hidden/overlaid.
- Only allow trusted domains to embed if needed.
- Verify with ZAP/browser after fixes.

Conclusions and Reflections

What I Learned

- I learned how to use security tools like OWASP ZAP and Burp Suite not just for scanning but also for confirming vulnerabilities and creating exploits.
- I now have a clearer understanding of vulnerabilities such as Reflected XSS, CSRF, and Clickjacking, including how they differ and why they are dangerous.
- I learned how to craft proof-of-concept exploits (payloads and HTML forms) to safely demonstrate real attacks.
- This process showed me how theoretical knowledge connects to real-world scenarios, where even small security gaps can lead to serious risks.
- I improved my ability to document findings step by step, turning technical evidence into a clear narrative.

Challenges Faced

- Technical issues: My Kali Linux virtual machine frequently lagged, which made scanning and intercepting requests slower and sometimes frustrating.
- Finding a target: I relied on OpenBugBounty to identify a vulnerable site, but the site was down when I first began, delaying my progress.
- Manual effort: Locating real vulnerabilities, especially XSS, required a lot of manual testing and patience — many attempts led to nothing useful.
- Inconsistent results: Sometimes the target site reflected inputs without saving them, forcing me to repeat tests and carefully analyze server responses.
- Documentation challenge: Explaining each step in a simple and clear way was not always easy, especially when balancing technical details with readability.