Contents

Executive Summary

This security assessment was carried out on three web applications — eternal.com, notegpt.io, and virtualterminal.com. The process included reconnaissance, automated scanning, manual testing, and validation of exploitability.

The most serious issue found was a **Reflected Cross-Site Scripting (XSS)** vulnerability on notegpt.io/ai-detector, rated **High severity**. This flaw allows attackers to run malicious JavaScript in users' browsers, which could lead to stolen sessions, credentials, or full client-side compromise.

Other key findings:

- Clickjacking on eternal.com/contact (Medium severity) — allows attackers to trick users into clicking hidden elements
- Content Security Policy (CSP) misconfiguration on virtualterminal.com (Medium severity) — weakens protection against UI manipulation and injection
- Outdated JavaScript libraries — includes known vulnerabilities (CVEs) that could be exploited

Tools used included OWASP ZAP, Nikto, Retire.js, and Nmap, supported by manual inspection to confirm each issue. Every finding is backed by step-by-step reproduction, screenshots, and clear remediation advice.

These results highlight the need for strong input validation, proper use of security headers, and regular updates to third-party components. Immediate action is recommended for the XSS vulnerability, followed by fixing the remaining issues in a structured way.

Methodology

Phase 1: Reconnaissance & Information Gathering

The goal of this phase is to collect as much intelligence as possible about the target using both passive and active techniques to map the attack surface. Identify all domains and subdomains, discover IP ranges and network blocks, find exposed services and open ports, gather information from public *sources* and archives, and fingerprint the technologies and frameworks in use.

Tools & Implementation

Subdomain Enumeration

to install,
sudo apt update && sudo apt install -y amass sublist3r dnsrecon

Passive subdomain discovery with Amass
amass enum -passive -d target.com -o amass_passive.txt

Active subdomain discovery
sublist3r -d target.com -o sublist3r_results.txt

DNS reconnaissance
dnsrecon -d target.com -t std,axfr -D /usr/share/wordlists/seclists/Discovery/DNS/namelist.txt -x dns_results.xml

Combine and deduplicate results
cat amass_passive.txt sublist3r_results.txt | sort -u > all_subdomains.txt

Search Engine & IoT Reconnaissance

To begin reconnaissance using search engines and IoT databases, you should first install the Shodan CLI by running

sudo apt install python3-pip
pip3 install shodan

Once installed, initialize Shodan with your API key using
shodan init YOUR_API_KEY

Then use Shodan to search for exposed services related to your target
shodan host target.com - to get host details
shodan search "hostname:target.com" -to find devices and services associated with the domain.

You can use Censys via its web interface or API by visiting https://search.censys.io , where you can search for certificates, hosts, and services linked to your target.

Phase 2: Scanning & Enumeration

This phase focuses on actively probing identified assets to discover open ports, services, and application architecture. You should perform TCP and UDP port scans, enumerate services, discover hidden web directories and files, identify the technology stack, locate API endpoints, and run vulnerability scanners to detect known issues.

Tools & Implementation

Port Scanning

use **Nmap** to detect open ports, services, and operating system details.

Run a default Nmap scan with service and OS detection
nmap -sC -sV -O target.com -oA initial_scannmap -p- --min-rate 5000 target.com -oA full_tcp_scan

Scan all 65,535 TCP ports quickly
nmap -p- --min-rate 5000 target.com -oA full_tcp_scan

Scan common UDP ports
nmap -sU -p 53,67,68,123,161 target.com -oA udp_scan

Web Application Scanning

To discover hidden directories and files on a target web server, you can use Gobuster. This performs a brute-force scan using a common wordlist and saves the results to a text file.
gobuster dir -u https://target.com -w /usr/share/wordlists/dirb/common.txt -o gobuster_scan.txt
dirb

You can use **Dirb** for similar directory enumeration which scans for accessible paths and outputs findings to a report file.
https://target.com /usr/share/wordlists/dirb/common.txt -o dirb_scan.txt

Vulnerability scanning with Nikto

To identify known vulnerabilities and misconfigurations, you can run,
nikto -h https://target.com -o nikto_scan.txt

For **automated baseline security checks**, you should install and configure OWASP ZAP
sudo apt install zaproxy
OR

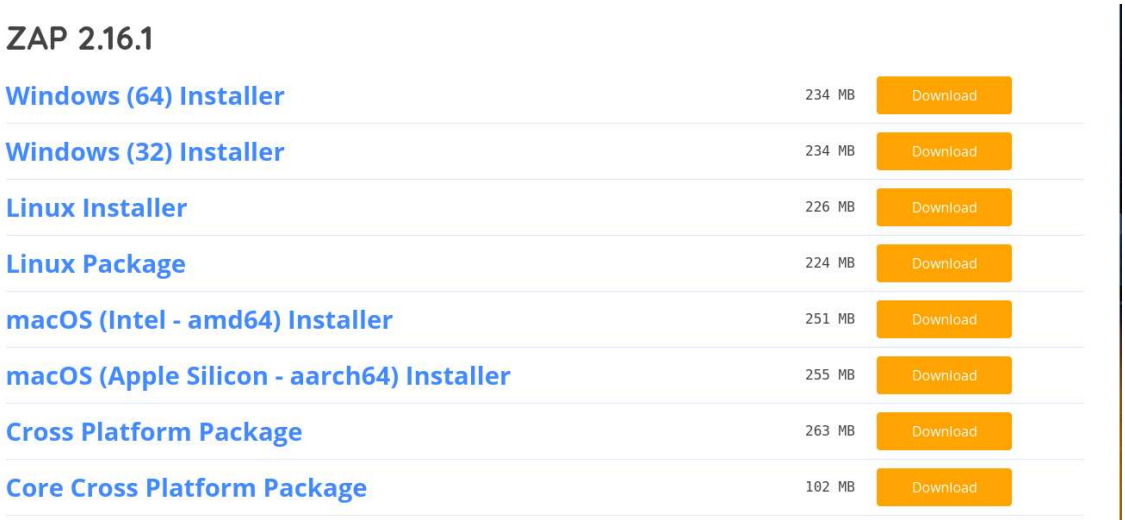
Install Linux package https://www.zaproxy.org/download/

OWASP ZAP (Zed Attack Proxy)

Purpose - It is a completely automated vulnerability scanner for use on web applications. How it works - It actively scans a website and reports potential security risks such as injection, broken authentication, and cross-site scripting. Why it's useful - It is another free, beginner-friendly penetration testing alternative to Burp Suite.

For Vm kali,

1. Install Linux package https://www.zaproxy.org/download/



Or

sudo apt update

sudo apt install zaproxy

2. Start ZAP GUI: *zaproxy*

3. In the GUI, enter your target (e.g., https://www.virtualterminal.com) in the URL bar and click Attack or use the Spider tool to crawl it.

   Navigate to the Alerts tab to see findings categorized by risk level. Each alert includes description, evidence, and remediation advice.

4. Go to **Report** → choose format (HTML, XML, Markdown).

Phase 3: Vulnerability Assessment

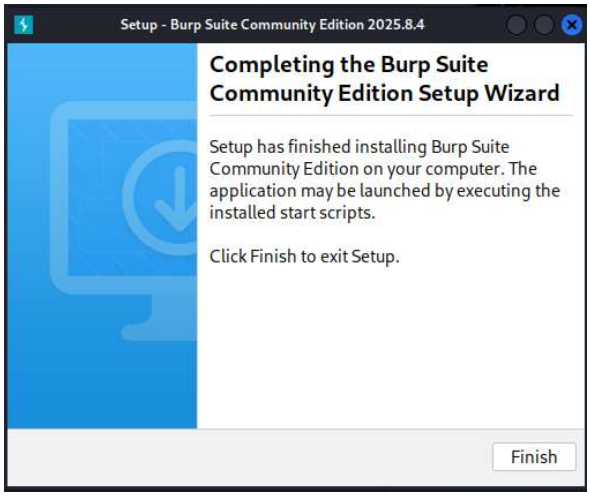
The objective here is to systematically test for common web application vulnerabilities and misconfigurations. You should use automated tools like SQLMap, OWASP ZAP, and Burp Suite to detect injection flaws, authentication and session management issues, authorization bypasses, and security misconfigurations such as improper CORS settings or missing headers.

Tools & Implementation

1.Burpsuite is used to manually test web applications for vulnerabilities that automated scanners might miss.

It allows you to intercept and modify HTTP/S traffic, analyze authentication and session handling, and test for issues like IDOR, JWT manipulation, and flaws in application workflows or privilege enforcement.

https://www.kali.org/tools/burpsuite/



2.OWASP ZAP is used to automate the detection of common web vulnerabilities like XSS, SQL injection, and misconfigured headers. It acts as a proxy to intercept traffic, includes a spider and fuzzer for deeper analysis, and generates visual reports for documentation.

Phase 4: Exploitation & Proof of Concept

This phase involves validating discovered vulnerabilities by crafting safe, controlled exploits that demonstrate real-world impact. You should use frameworks like Metasploit and tools like Burp Suite or OWASP ZAP to intercept and manipulate traffic, and document your findings with screenshots, HTTP logs, and step-by-step reproduction instructions that clearly show the business impact without causing harm.

Phase 5: Analysis & Validation

The goal here is to verify your findings, eliminate false positives, and assess the true risk and impact of each vulnerability. You should manually cross-check all results, test for reproducibility, categorize issues by severity, and evaluate both technical and business implications.

Phase 6: Report Writing

This phase is about creating clear, concise, and actionable reports that help the organization understand and fix the vulnerabilities. Your report should include an executive summary, detailed descriptions of each bug, affected endpoints, CVSS scores, reproduction steps with evidence, impact analysis, and specific remediation recommendations.

Vulnerability Report Structure

1. Executive Summary

- Overview of findings
- Risk level summary

2. Vulnerability Details

- Bug description
- Affected endpoints/components
- CVSS score

3. Steps to Reproduce

- Step-by-step PoC
- Screenshots and evidence
- HTTP request/response samples

4. Impact Assessment

- Technical impact
- Business impact
- Attack scenario

5. Remediation Recommendations

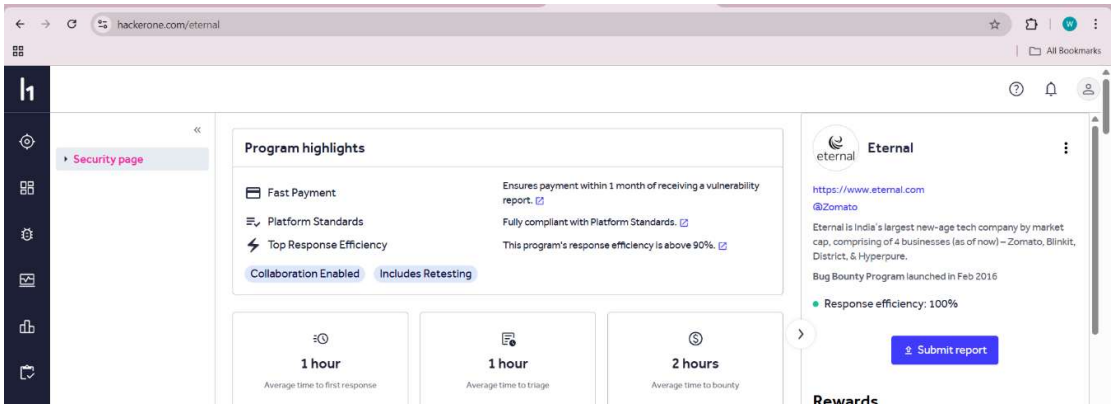
- Code-level fixes
- Configuration changes
- Security controls

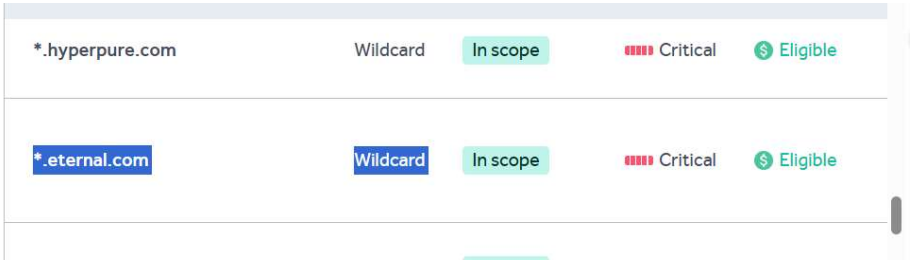
Phase 7: Submission & Communication

The objective here is to submit your findings through proper channels and maintain professional communication with the security team. You should use platforms like HackerOne, Bugcrowd, or direct programs, follow responsible disclosure guidelines, respect the scope and rules of engagement, and respond politely to any follow-up requests.

01 Absence of Frame Protection Headers (Clickjacking)

https://www.eternal.com/contact/



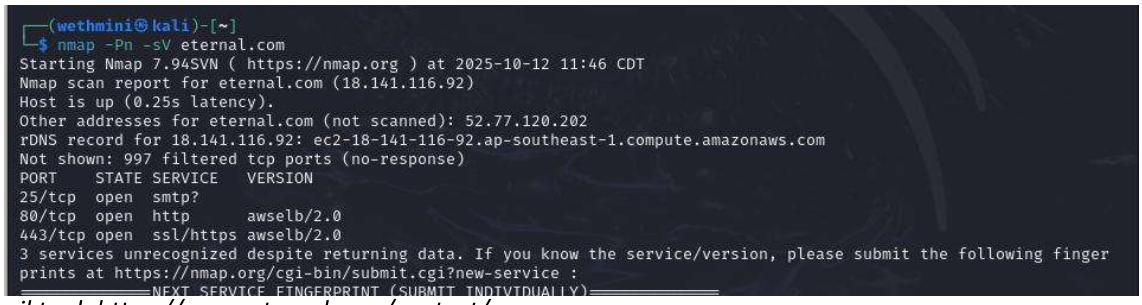


To conduct reconnaissance, I used the following tools:

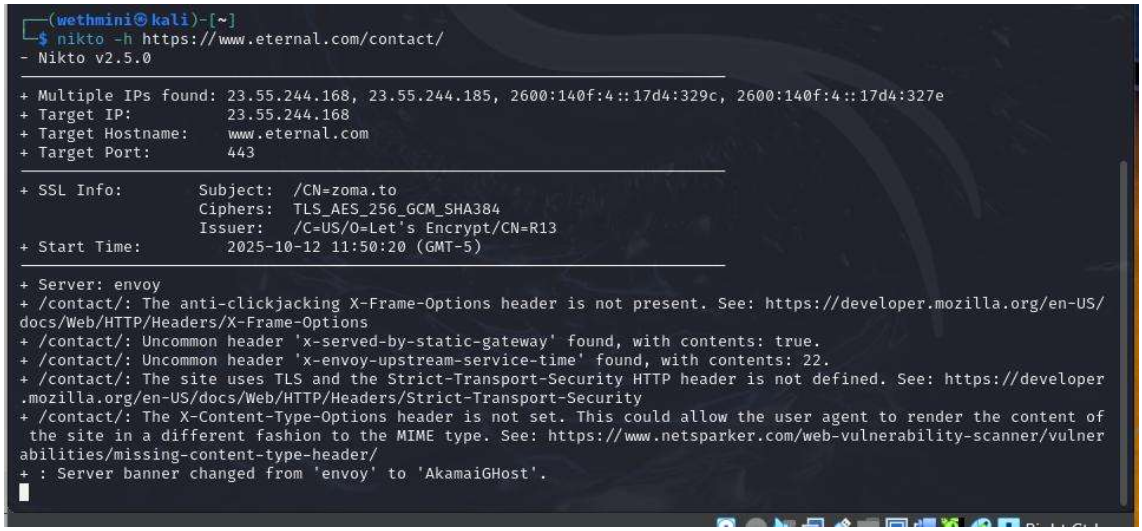
amass enum -d eternal.com

Name Server (NS) records, tell us which DNS servers are authoritative for eternal.com.

nmap -Pn -sV eternal.com



nikto -h https://www.eternal.com/contact/



Scan Summary

Tool: Nikto v2.5.0
Server: Envoy (later AkamaiGHost)
IP: 23.55.244.168
TLS: Valid cert from Let's Encrypt (CN=zoma.to)
Scan Time: 2025-10-12 11:50 (GMT-5)

Key Issues

❌ **Missing X-Frame-Options** → vulnerable to clickjacking

❌ **No HSTS header** → downgrade attack risk

❌ **No X-Content-Type-Options** → MIME sniffing possible

Uncommon headers → internal infra exposed (x-served-by-static-gateway, x-envoy-upstream-service-time)

Domain: https://www.eternal.com

Subdomain: https://www.eternal.com/contact/

Vulnerability title : Clickjacking / UI Redressing — page is frameable
Category: *A05:2021 – Security Misconfiguration*

Description

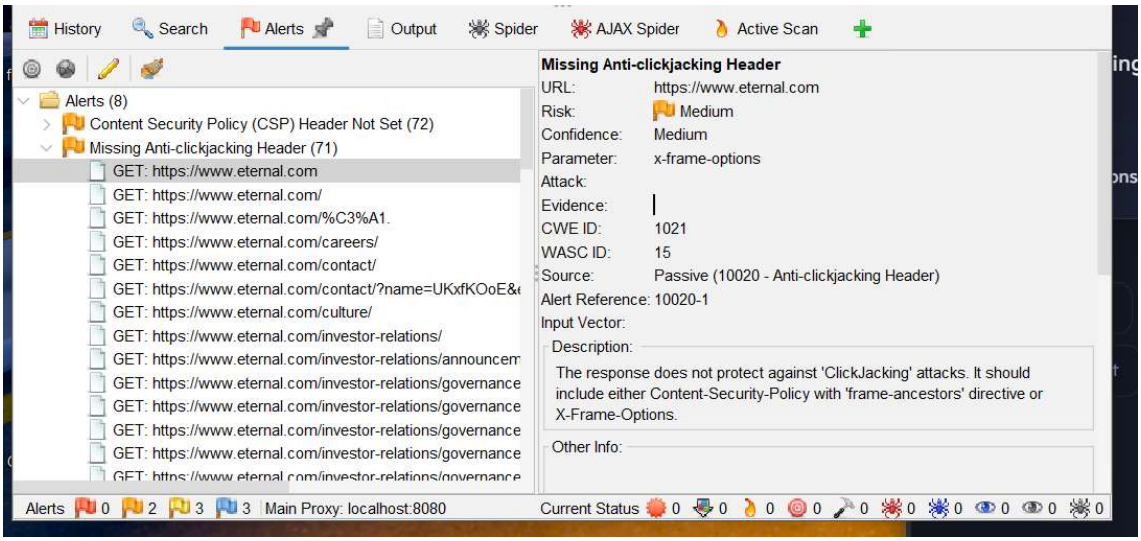
Clickjacking (UI redressing) occurs when an attacker embeds a target site inside an invisible or semi-transparent iframe and tricks a user into clicking on interactive elements. The Eternal contact form lacks frame protection headers, allowing attackers to embed it in malicious pages and deceive users into submitting unwanted form data.

Conditions that make this clickjacking effective

- Lack of Frame Protection (No X-Frame-Options / CSP) : If a website allows itself to be embedded in an <iframe>, attackers can overlay it invisibly on a malicious page. Lack of Frame Protection: No X-Frame-Options or Content-Security-Policy headers present
- Interactive Form Elements: Contact form contains multiple fields and a submit button
- No Authentication Barriers: Contact form doesn't require user login
- Social Engineering Potential: Users can be tricked with fake offers and incentives

Vulnerability Discovery

1. Automated Security Scanning with OWASP ZAP



2. Security Header Analysis
Initial reconnaissance using curl -I https://www.eternal.com/contact revealed the absence of critical frame protection headers.

Steps to reproduce with Proof of Concept (poc)

1. Verify Missing Security Headers , curl -I https://www.eternal.com/contact

Observed Headers:

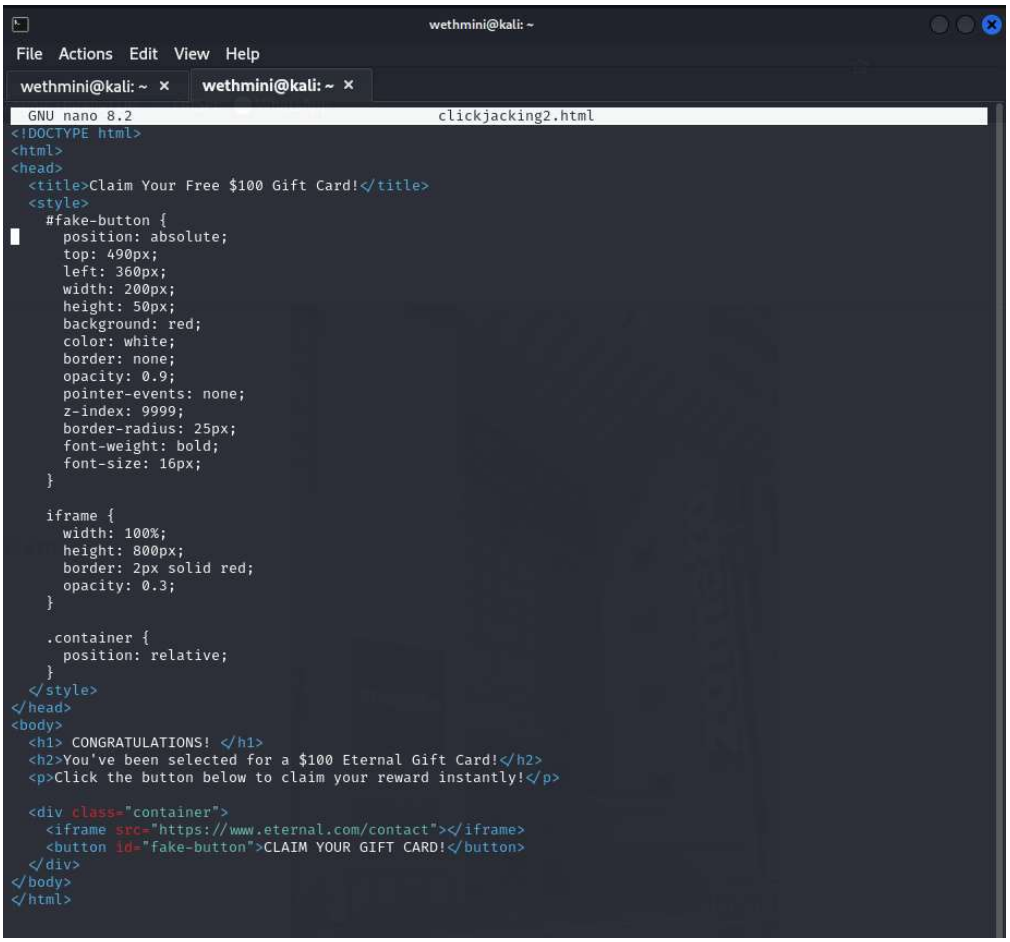
content-type: text/html

server: envoy

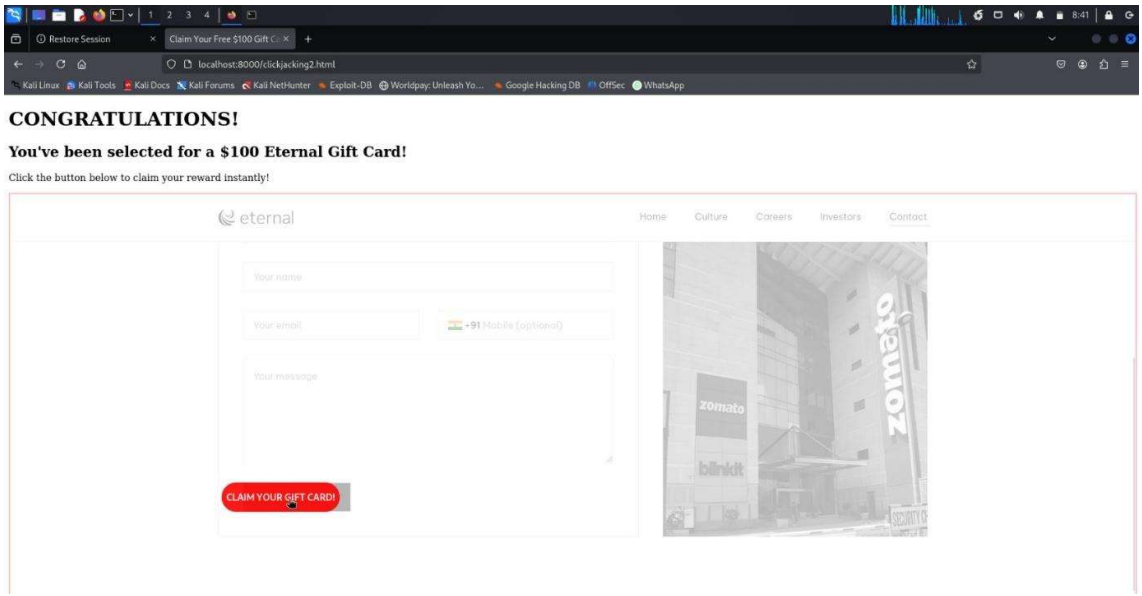
❌ No X-Frame-Options

❌ No Content-Security-Policy: frame-ancestors

2. Then , I created a local HTML file based on the one you provided that embeds the target via <iframe> with a visible overlay element (button).



3. The iframe successfully renders the target page, confirming that the site permits framing.

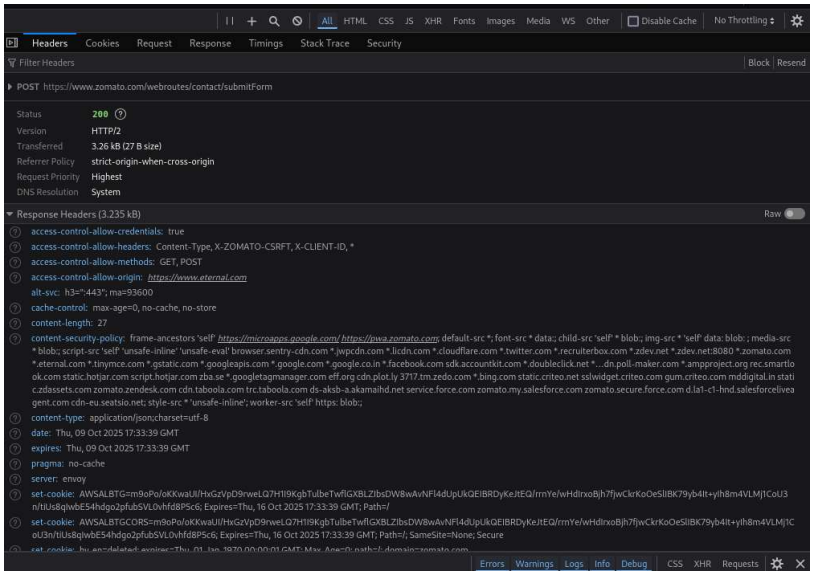


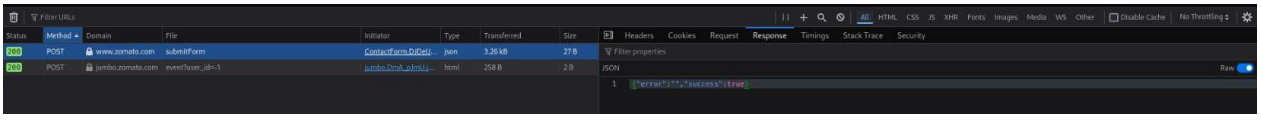
4. Open DevTools → Network Tab

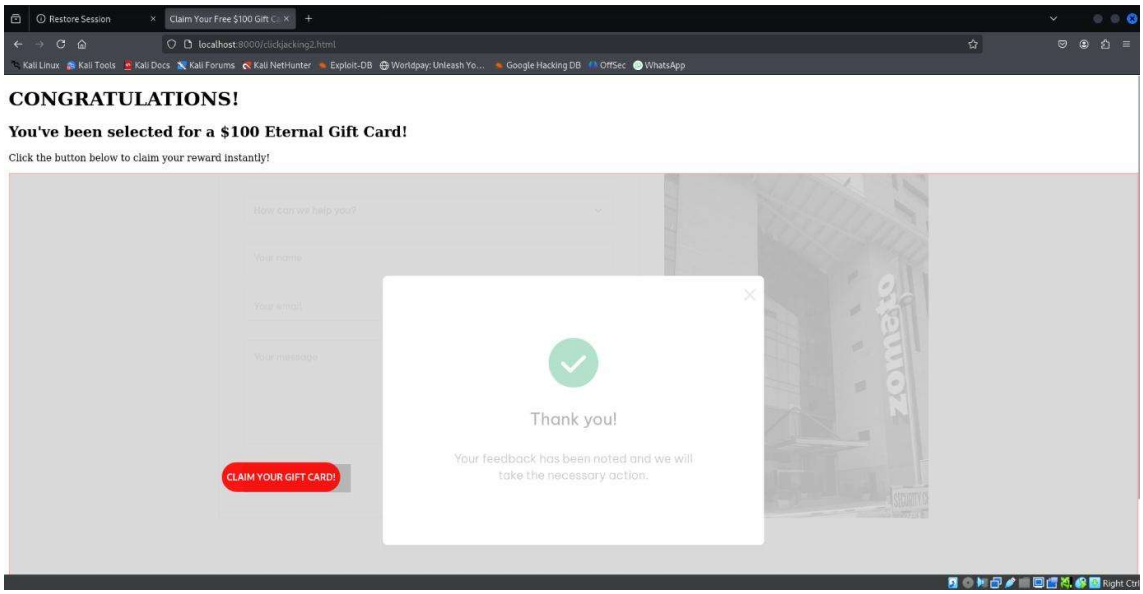
Check **“Persist log”** to retain request history after submission.

I submitted the contact form and confirmed that a real POST request to https://www.zomato.com/webroutes/contact/submitForm was triggered, with a 200 OK response and full headers, verifying that the action was successfully processed by the server .

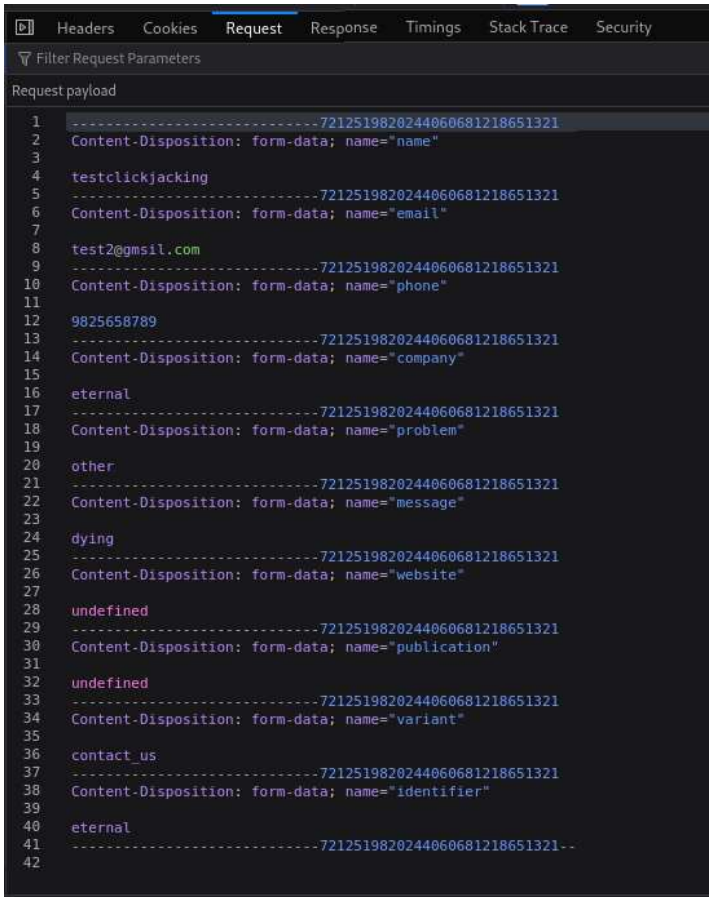
A JSON response was returned indicating "success": true.







4. To verify that the form data was successfully sent, **check the Payload tab in** DevTools after clicking the POST request **,** it will display all submitted fields



Exploitation (step-by-step attacker scenario)

1. Attacker creates a deceptive web page offering fake rewards
2. Attacker embeds Eternal contact form in transparent iframe
3. Attacker positions fake "Claim Reward" button over form submit button
4. User visits malicious page and clicks the attractive button
5. User unknowingly submits Eternal contact form with potential spam content
6. Company receives unwanted form submissions, wasting support resources

Impact

- Spam Attacks: Flood contact form with unwanted submissions
- Resource Waste: Consume company support time and resources
- Harassment Potential: Submit abusive content appearing to come from victims
- Reputation Damage: Mass spam submissions from deceived users
- Social Engineering: Trick support staff with fake business inquiries
  Impact assessment Severity – Medium

Remediation

- ✓ Set Content-Security-Policy
  Content-Security-Policy: frame-ancestors 'none';

- ✓ Set X-Frame-Options (Legacy Support):
  X-Frame-Options: DENY

  - Implement CSRF tokens for form submissions
  - Add CAPTCHA challenges for high-volume submissions
  - Monitor for unusual form submission patterns

Conclusions & Reflections

What I learned

- Contact forms are valuable clickjacking targets due to their interactive nature
- Missing security headers create immediate framing vulnerabilities
- Social engineering combined with UI redressing creates effective attacks
- Even simple forms can be exploited for spam and abuse

Challenges faced

- Positioning overlays requires careful alignment with target elements
- Cross-origin restrictions limit automated form manipulation
- Responsive design elements may require multiple overlay positions
- Ethical testing boundaries must be maintained during demonstration

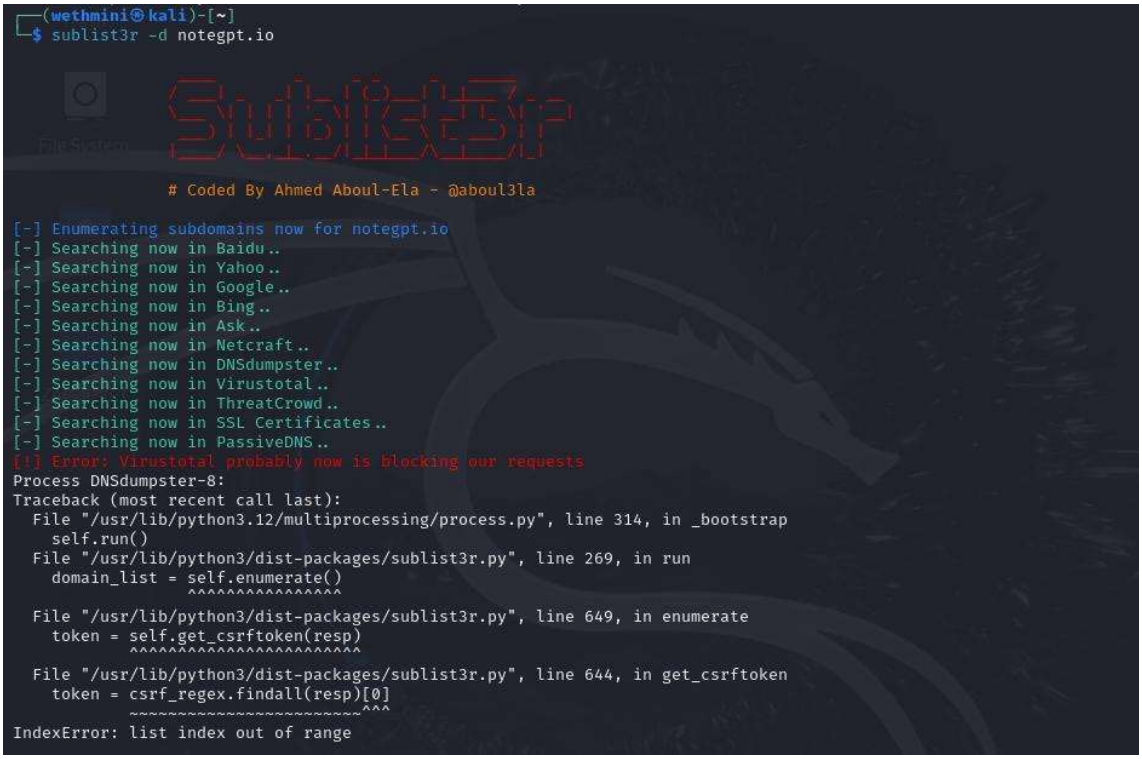
02 Reflected XSS Vulnerability

https://notegpt.io/ai-detector

Target Domain: https://notegpt.io/ai-detector

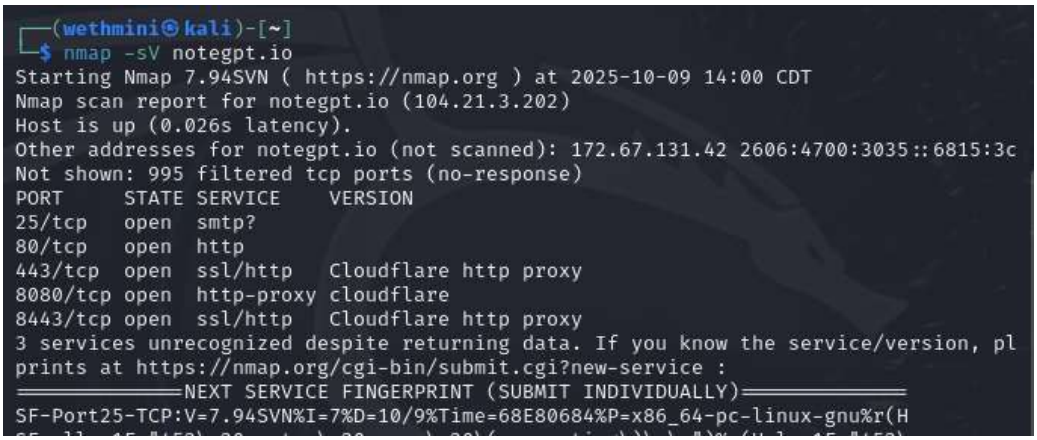
To conduct reconnaissance and identify the vulnerability, I used the following tools:

Sublist3r for subdomain enumeration using the command *sublist3r -d notegpt.io*



I got an error.

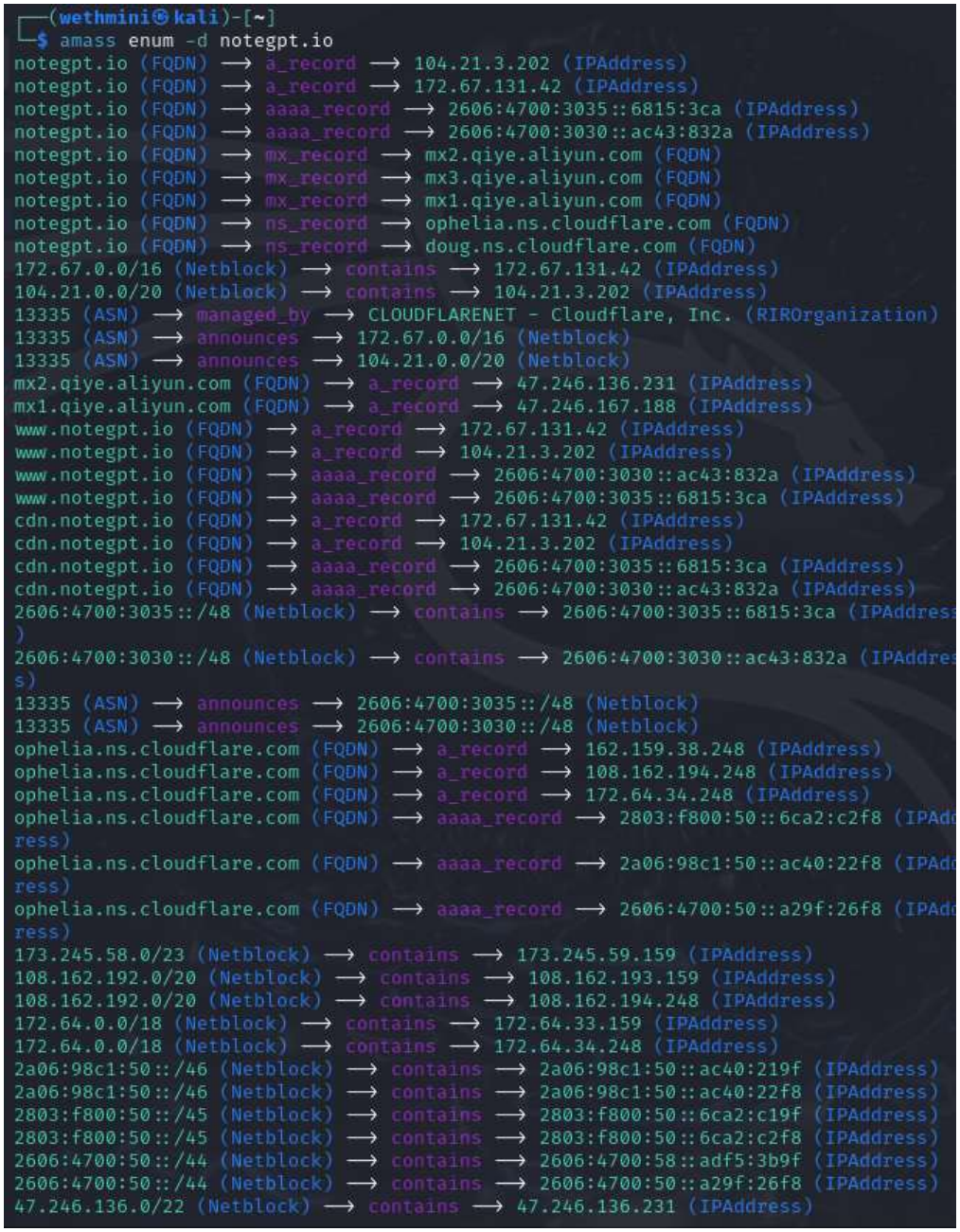
Nmap for port and service scanning with nmap -sV notegpt.io.



Cloudflare Proxy Detected: Ports 8080 and 8443 are being handled by Cloudflare's infrastructure.

Amass for DNS mapping and enumeration via *amass enum -d notegpt.io.*

I found all the subdomains related to the target domain using Amass



Subdomains Detected:

- www.notegpt.io
- cdn.notegpt.io

IPv4 A Records:

- 104.21.3.202
- 172.67.131.42

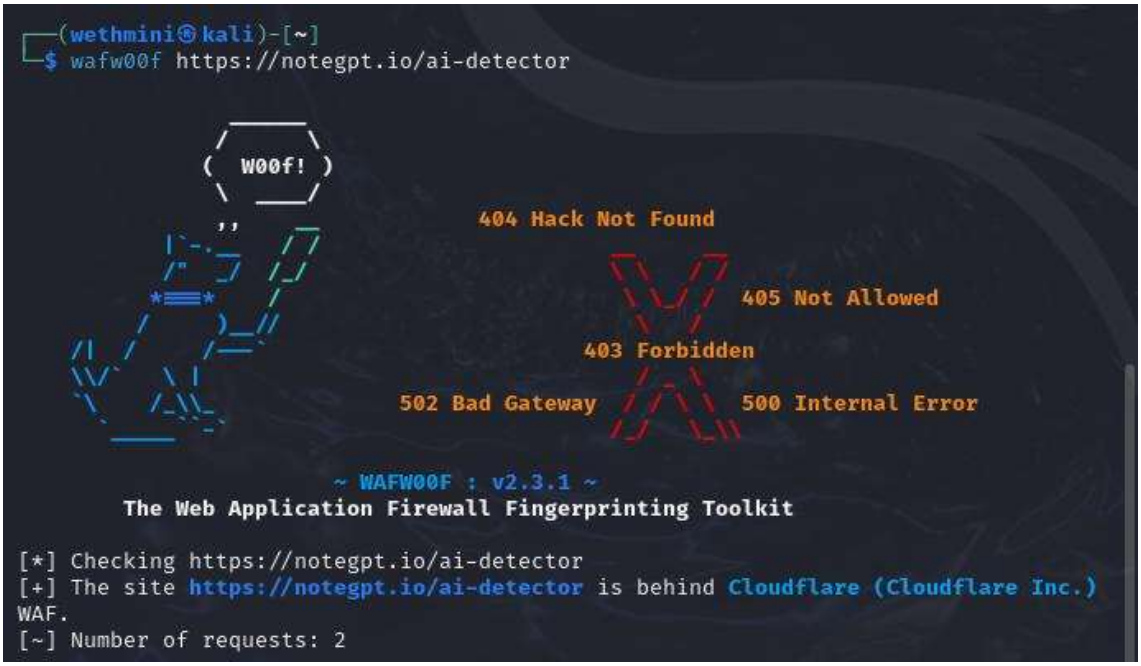
IPv6 AAAA Records:

- 2606:4700:3035::6815:3ca
- 2606:4700:3030::ac43:832a

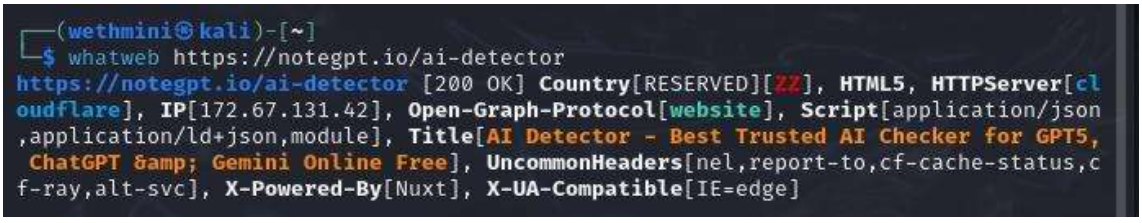
These IPs belong to Cloudflare, not the origin server. They're part of:

- 104.21.0.0/20
- 172.67.0.0/16
- ASN 13335 → CLOUDFLARENET

Wafw00f to detect the presence of a Web Application Firewall using,
wafw00f https://notegpt.io/ai-detector



WhatWeb for technology fingerprinting with
whatweb https://notegpt.io/ai-detector



- Framework: Nuxt.js (as indicated by X-Powered-By[Nuxt])
- Infrastructure: Hosted behind Cloudflare (HTTPServer[cloudflare], cf-cache-status, cf-ray)
- Modern Stack: Uses HTML5, module scripts, JSON-LD
- Protected: Cloudflare provides WAF, DDoS protection, and caching

Domain: https://notegpt.io

Subdomain: https://notegpt.io/ai-detector

Vulnerability Title
Reflected Cross-Site Scripting (XSS)

Description
Reflected XSS is a client-side vulnerability where user input is immediately echoed back in the server's response without proper sanitization. This allows attackers to inject malicious JavaScript into URLs, which then execute in the victim's browser when the link is clicked.

Impact assessment
Severity – High

Affected components
The issue is in the main input text field, which processes user input without proper sanitization.

OWASP Category
A03:2021 – Injection Reflected XSS falls under the Injection category, which includes flaws where untrusted data is sent to an interpreter as part of a command or query. In this case, JavaScript is injected and executed in the browser.

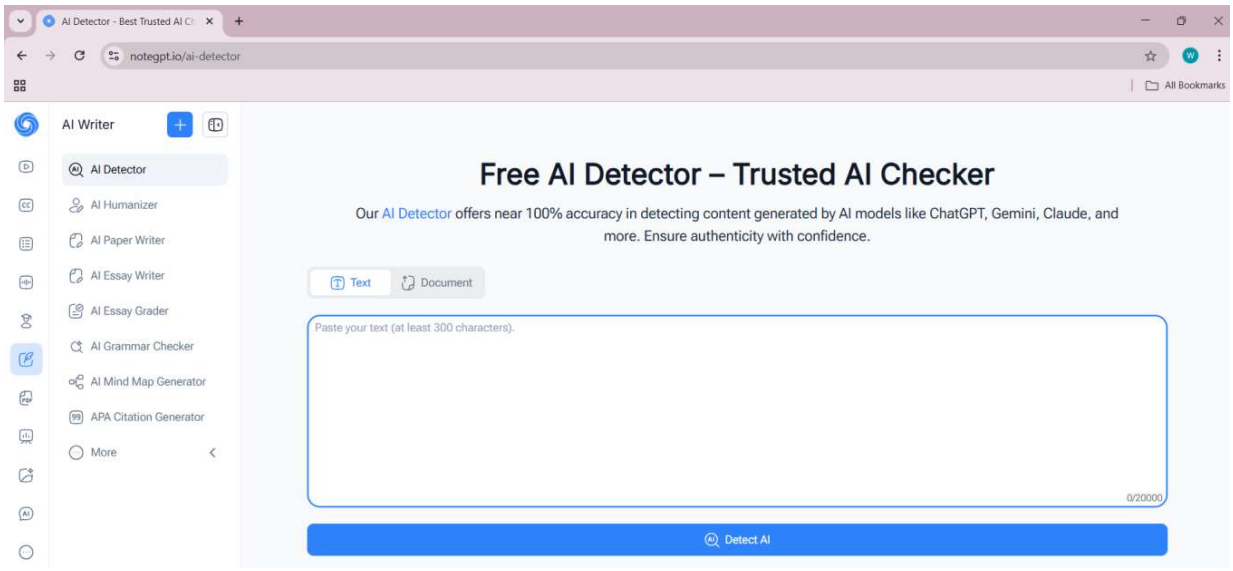
This can lead to,

- Session Hijacking → Attackers can steal session cookies using document.cookie, allowing them to impersonate the victim.
- Credential Theft → Malicious scripts can capture login details from fake forms or intercept input fields.
- Phishing Attacks → Users can be redirected to fake login pages or malicious sites that mimic trusted brands.
- Denial of Service (DoS) → Injected scripts can crash the browser, overload resources, or trigger infinite loops.
- Erosion of User Trust → If users are tricked or harmed by malicious links, they lose confidence in the site's security.

Since the payload is not stored, the attack relies on social engineering to lure victims into clicking malicious links.

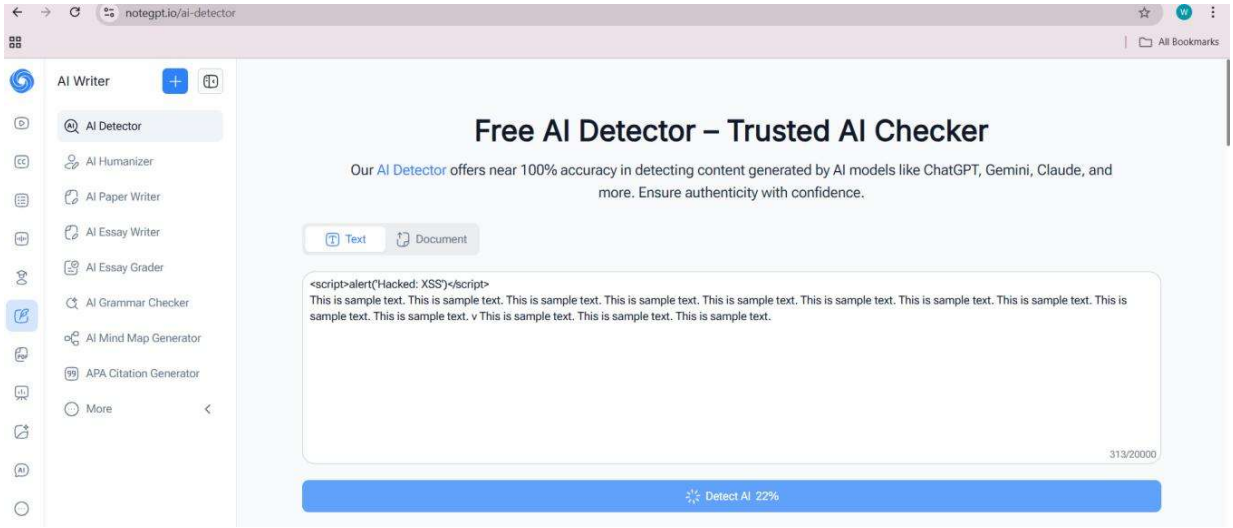
Steps to reproduce with Proof of Concept (poc)

Step 1:
Visited https://notegpt.io/ai-detector and located the text input field.



Step 2:
Injected a basic payload:

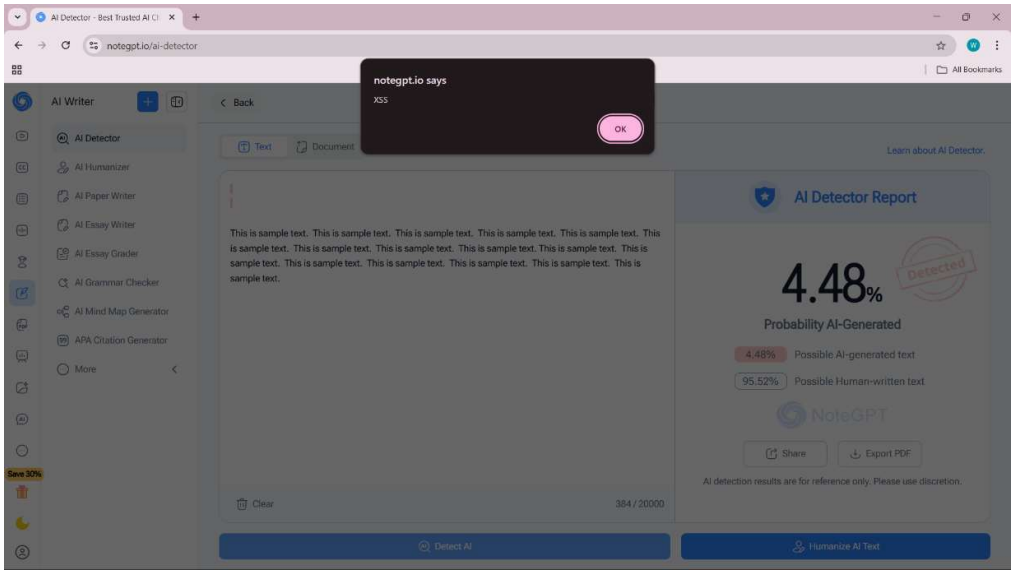
<script>alert('Hacked: XSS')</script>



No alert appeared – indicating <script> tags were being filtered.

Step 3:
Tested an alternative event-based payload:

<img src=x onerror=alert('XSS')>

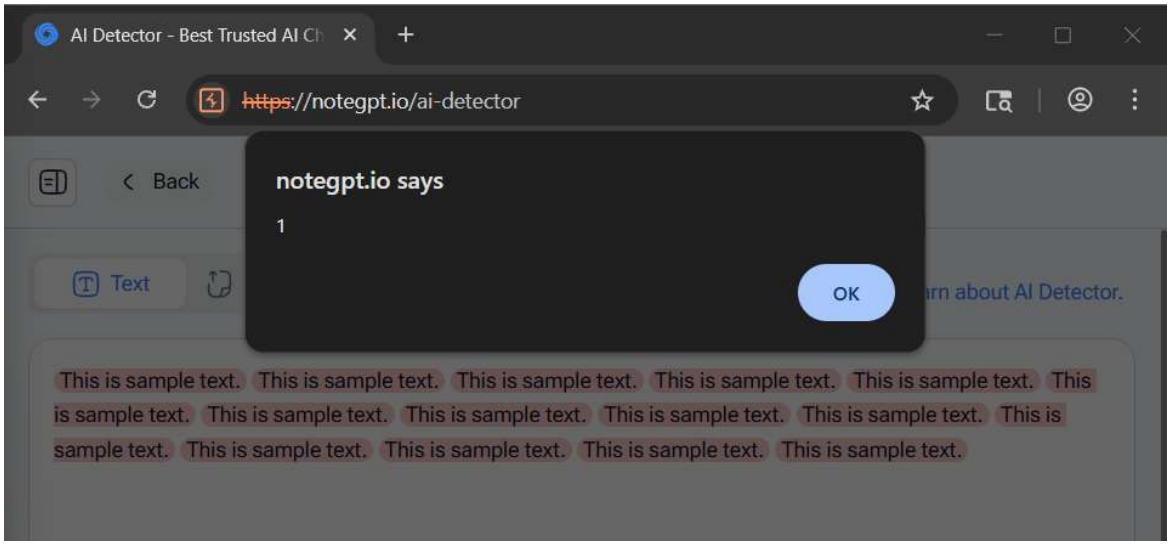


Result:
A pop-up alert appeared, confirming that the payload was executed.
The input was reflected unsanitized, allowing JavaScript execution via the onerror attribute.

Step 4:
Repeated the process several times – the alert triggered consistently, confirming **reproducibility**.

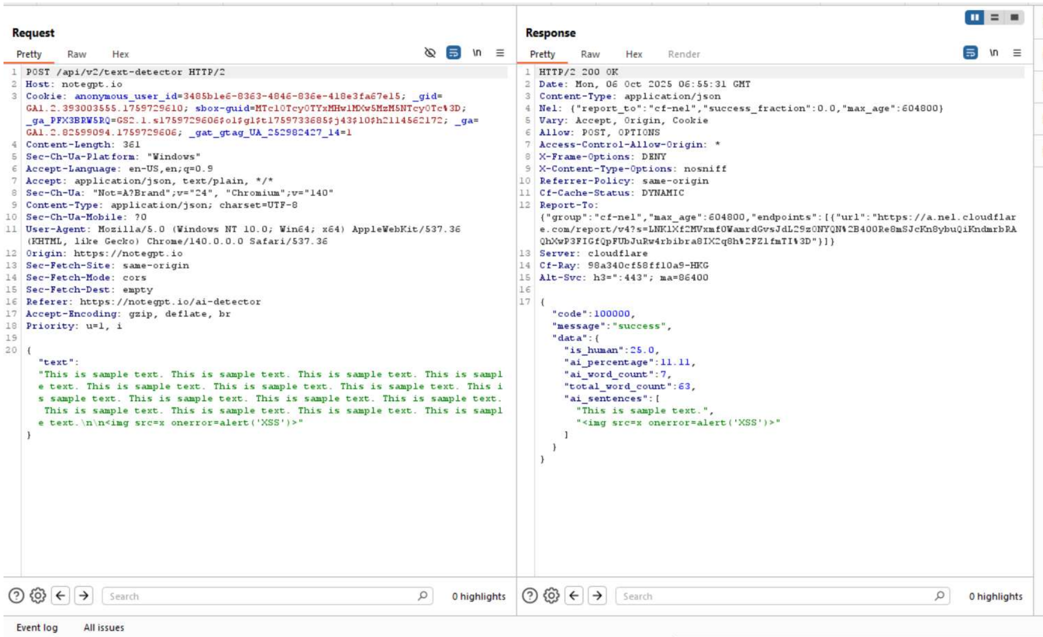
Also tried a different payload,

<svg/onload=alert(1)>
<iframe src="javascript:alert(1)">



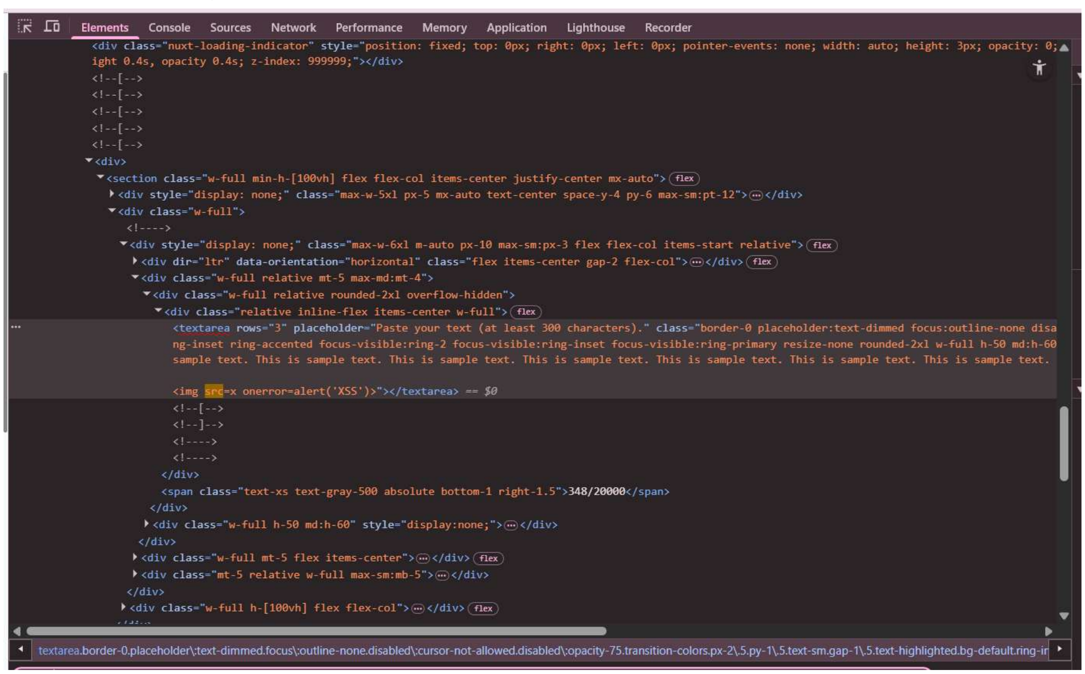
Step 5:

Inspected the HTTP response via **Burp Suite Repeater**.
The payload appeared in the response body, confirming server-side reflection.



Step 6:

Viewed the page source – the payload appeared inside a <div> with the ID results, disappearing after refresh, confirming it was **not stored**.



Exploitation

1. Attacker crafts malicious URL: https://notegpt.io/ai-detector?q=<img src=x onerror=alert('XSS')>
2. Victim clicks the link
3. Payload is reflected and executed in browser
4. Attacker can:
    - Steal cookies via document.cookie
    - Redirect to phishing/malware sites
    - Inject fake login forms or keyloggers
    - Crash or exploit browser vulnerabilities

Remediation

To mitigate this vulnerability, the following steps should be taken:

1. Sanitize and validate all user inputs on both client and server sides. Reject disallowed characters and patterns.
2. Encode dynamic content before rendering it in HTML, JavaScript, or URL contexts.
3. Implement strict CSP headers to block unauthorized scripts.
4. Avoid unsafe DOM methods such as innerHTML, eval(), and document.write().
5. Use libraries like OWASP Java Encoder to automate output encoding.
6. Use modern frameworks like React, Angular, or Vue.js that auto-escape outputs and reduce XSS risk.

Conclusions and Reflections

What I Learned from the Process

This helped me understand how reflected XSS vulnerabilities operate and how they differ from stored or DOM-based XSS. I learned how to craft payloads, interpret server responses, and use reconnaissance tools effectively. It also reinforced the importance of ethical hacking and responsible disclosure.

Challenges Faced

One challenge was that initial payloads using <script> tags were blocked, requiring alternative vectors like onerror attributes. Another was identifying the exact location of reflection in the response. Ensuring reproducibility across sessions and browsers while maintaining ethical boundaries was also a key consideration.

Ethical Note

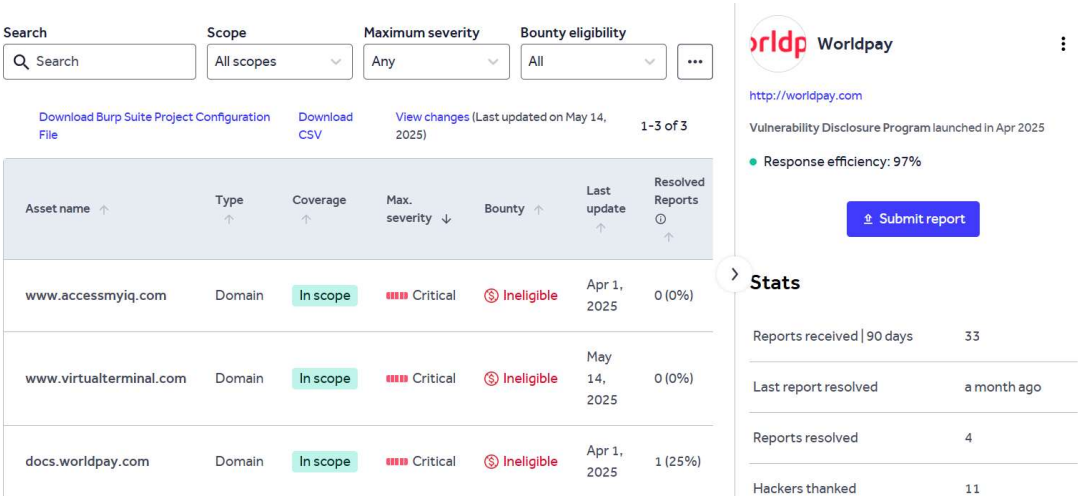
This assessment was conducted responsibly and caused no harm, defacement, or unauthorized access. All testing followed ethical guidelines, with full respect for the target systems and no malicious actions taken.

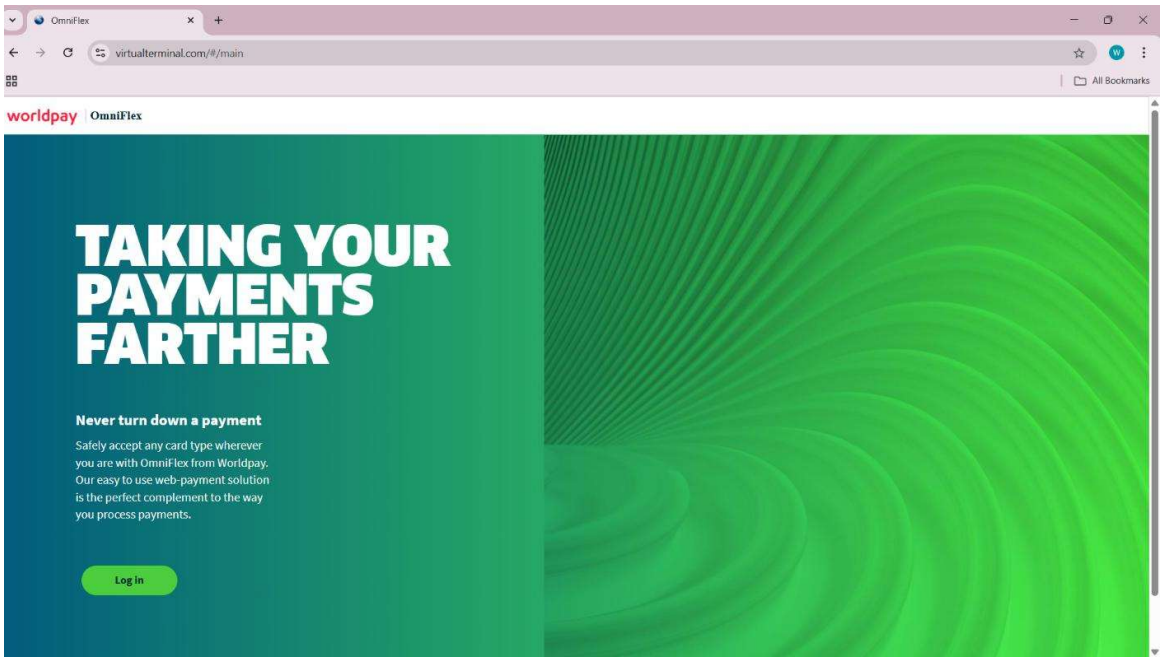
03 CSP Misconfiguration

https://www.virtualterminal.com/#/main

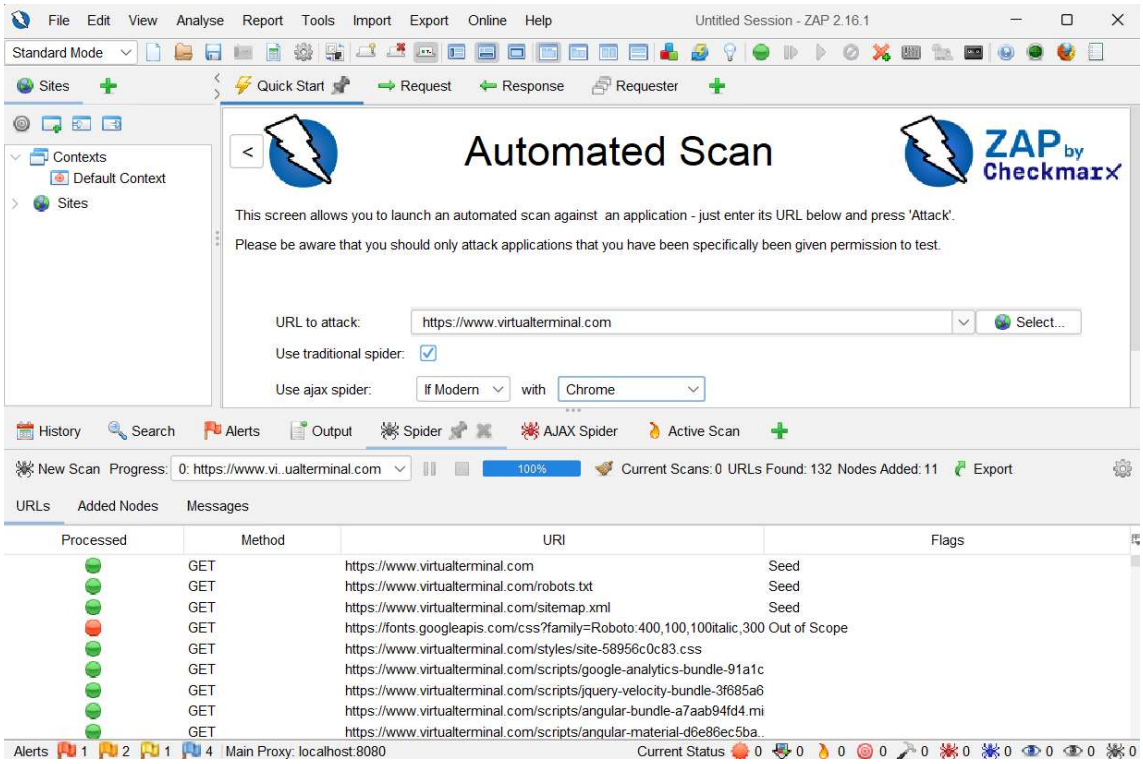
Main domain – https://worldpay.com/en

Report – virtualterminal.com
https://www.virtualterminal.com/#/main





I used OWASP ZAP tool to scan the website.



Tools utilized

Reconnaissance – information gathering

I used **sublist3r** tool to find the subdomains of virtualterminal.com, *returned 13 subdomains*

```
File  Actions  Edit  View  Help
[-] Searching now in Ask..
[-] Searching now in Netcraft..
[-] Searching now in DNSdumpster..
[-] Searching now in Virustotal..
[-] Searching now in ThreatCrowd..
[-] Searching now in SSL Certificates..
[-] Searching now in PassiveDNS..
Process DNSdumpster-8:
Traceback (most recent call last):
  File "/usr/lib/python3.12/multiprocessing/process.py", line 314, in _bootstrap
    self.run()
  File "/usr/lib/python3/dist-packages/sublist3r.py", line 269, in run
    domain_list = self.enumerate()
                  ^^^^^^^^^^^^^^^^
  File "/usr/lib/python3/dist-packages/sublist3r.py", line 649, in enumerate
    token = self.get_csrftoken(resp)
            ^^^^^^^^^^^^^^^^^^^^^^^^
  File "/usr/lib/python3/dist-packages/sublist3r.py", line 644, in get_csrftoken
    token = csrf_regex.findall(resp)[0]
            ~~~~~~~~~~~~~~~~~~~~~~~~^^^
IndexError: list index out of range
[!] Error: Virustotal probably now is blocking our requests
[-] Total Unique Subdomains Found: 13
www.virtualterminal.com
RichTest2.virtualterminal.com
cert.virtualterminal.com
certedgekey.virtualterminal.com
edgekey.virtualterminal.com
fl2-www.virtualterminal.com
gr2-www.virtualterminal.com
richtest.virtualterminal.com
richtest3.virtualterminal.com
stg01.virtualterminal.com
www.stg01.virtualterminal.com
test1.virtualterminal.com
test2.virtualterminal.com
```

Amass – Subdomain and DNS mapping

I found all the subdomains related to the target domain.

Command used: *amass enum -d virtualterminal.com*

```
┌──(wethmini㉿kali)-[~]
└─$ amass enum -d virtualterminal.com
virtualterminal.com (FQDN) --> ns_record --> a16-67.akam.net (FQDN)
virtualterminal.com (FQDN) --> ns_record --> a1-135.akam.net (FQDN)
virtualterminal.com (FQDN) --> ns_record --> a7-66.akam.net (FQDN)
virtualterminal.com (FQDN) --> ns_record --> a11-66.akam.net (FQDN)
virtualterminal.com (FQDN) --> ns_record --> a26-65.akam.net (FQDN)
virtualterminal.com (FQDN) --> ns_record --> a18-64.akam.net (FQDN)
virtualterminal.com (FQDN) --> mx_record --> elementps.com.inbound15.mxlogicmx.net (FQDN)
virtualterminal.com (FQDN) --> a_record --> 107.162.169.193 (IPAddress)
cert.virtualterminal.com (FQDN) --> cname_record --> cert.virtualterminal.com.edgekey.net (FQDN)
107.162.169.0/24 (Netblock) --> contains --> 107.162.169.193 (IPAddress)
55002 (ASN) --> managed_by --> DEFENSE-NET, US (RIROrganization)
55002 (ASN) --> announces --> 107.162.169.0/24 (Netblock)
prelive-fl2.virtualterminal.com (FQDN) --> a_record --> 74.120.158.113 (IPAddress)
74.120.158.0/24 (Netblock) --> contains --> 74.120.158.113 (IPAddress)
18594 (ASN) --> announces --> 74.120.158.0/24 (Netblock)
18594 (ASN) --> managed_by --> FTPS-LLC - Vantiv, LLC (RIROrganization)

The enumeration has finished
```

Nmap – Network scanning and enumeration

I found all the open ports and detected the running services on the target server using Nmap.
Command used: *nmap -Pn -sV virtualterminal.com*
Host up; **3 open TCP ports** discovered — 25/tcp (smtp?), 80/tcp (http), 443/tcp (https).

```
┌──(wethmini㉿kali)-[~]
└─$ nmap -Pn -sV virtualterminal.com
Starting Nmap 7.94SVN ( https://nmap.org ) at 2025-09-19 05:38 CDT
Nmap scan report for virtualterminal.com (107.162.169.193)
Host is up (0.37s latency).
Not shown: 997 filtered tcp ports (no-response)
PORT    STATE SERVICE    VERSION
25/tcp  open  smtp?
80/tcp  open  http?
443/tcp open  ssl/https?
1 service unrecognized despite returning data. If you know the service/version, please submit the following fingerprint at
 https://nmap.org/cgi-bin/submit.cgi?new-service :
SF-Port25-TCP:V=7.94SVN%I=7%D=9/19%Time=68CD32D0%P=x86_64-pc-linux-gnu%r(N
SF:ULL,56,"421\x20service\x20not\x20available\x20\(connection\x20to\x20blo
SF:cklisted\x20host\x20\(107\.162\.169\.193\x20-\x20DNSBL\)\)\r\n");

Service detection performed. Please report any incorrect results at https://nmap.org/submit/ .
Nmap done: 1 IP address (1 host up) scanned in 293.70 seconds
```

Wafw00f – Firewall Detection

Command used: *wafw00f https://www.virtualterminal.com/*

Detected **Kona SiteDefender (Akamai) WAF**.
This indicates a web application firewall is present and may block or modify automated scans; coordinate testing with the program and avoid aggressive scans that could trigger mitigations

```
┌──(wethmini㉿kali)-[~]
└─$ wafw00f https://www.virtualterminal.com/

                ?              ,.   (   .      )       .      "
                ??           ("     )  )'     ,'       )  . (`     ``
    (___()'`;   ???          .; )  ' (( (" )    ;(,     ((  (  ;)  "  )")
    /,___ /`                 _"., ,._'_.,)_(..,( . )_  _' )_') (. _..( ' )
    \\   \\                 |____|___|____|____|____|____|____|____|____|

                            ~ WAFW00F : v2.3.1 ~
                    ~ Sniffing Web Application Firewalls since 2014 ~

[*] Checking https://www.virtualterminal.com/
[+] The site https://www.virtualterminal.com/ is behind Kona SiteDefender (Akamai) WAF.
[~] Number of requests: 2
```

Whatweb – to identify the technologies used by the site.

Command used – *whatweb* https://www.virtualterminal.com/

The server is actively denying access. The site is behind Akamai's CDN and security infrastructure.
Strict-Transport-Security: max-age=31536000; includeSubDomains : enforces HTTPS for one year across all subdomains.

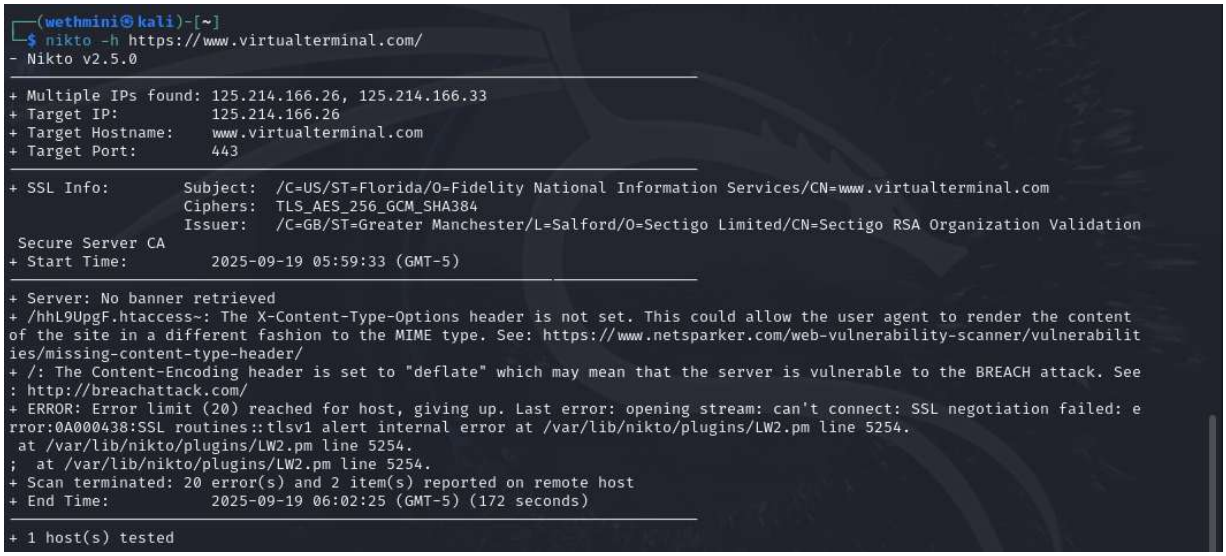
```
┌──(wethmini㉿kali)-[~]
└─$ whatweb https://www.virtualterminal.com/
https://www.virtualterminal.com/ [403 Forbidden] Akamai-Global-Host, Country[UNITED STATES][US], HTTPServer[AkamaiGHost], I
P[23.193.114.75], Strict-Transport-Security[max-age=31536000 ; includeSubDomains], Title[Access Denied]

┌──(wethmini㉿kali)-[~]
```

Web Application Scanning Tools (Vulnerability Identification)

Nikto

An open-source web server scanner used to identify security vulnerabilities, misconfigurations, and outdated software on websites.



The server is running Microsoft-IIS/10.0 and is hosted at IP address 205.214.166.26 over port 443.

The scan identified that the X-Content-Type-Options header is missing.

Without this header, browsers can attempt to interpret content based on its actual data rather than its declared type, potentially rendering a harmless .txt file as executable script and increasing the risk of content-based exploits.

Setting *X-Content-Type-Options: nosniff* prevents this by forcing the browser to respect the declared Content-Type.

Vulnerability Type: CSP Misconfiguration – style-src 'unsafe-inline'

The site's Content Security Policy (CSP) includes `'unsafe-inline'` in the `style-src` directive, allowing execution of inline CSS. This weakens browser protections and opens the door to CSS injection attacks.

This is a **client-side security vulnerability** that weakens the browser's ability to block malicious CSS injections.

It can be exploited to: manipulate the UI, hide phishing elements, chain with other vulnerabilities (e.g., reflected XSS)

Impact Assessment

Severity: Medium/low

Potential Exploits:

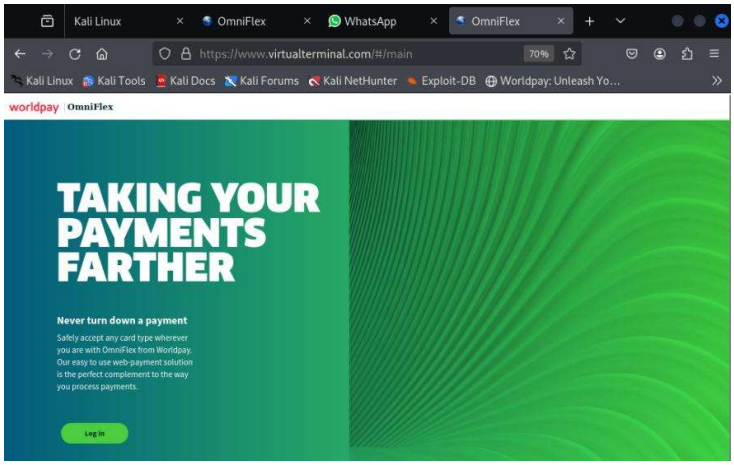
- **CSS Injection**: Attacker can inject styles to hide or manipulate elements.
- **Clickjacking**: Invisible overlays can trick users into clicking malicious buttons.
- **Phishing UI**: Fake login forms styled to look legitimate.
- **Chaining with XSS**: If other injection points exist, this CSP allows style-based payloads to execute.

Affected Component

- Domain: https://www.virtualterminal.com/
- Header: Content-Security-Policy
- Directive: style-src 'unsafe-inline'

Proof of Concept (PoC)

1.First, I navigated to the vulnerable site (https://www.virtualterminal.com)



Reflection

The site loads normally and presents a homepage with standard UI elements.

2. I inspected the HTTP response headers using both ZAP and curl.

I used curl -I to inspect the response headers: *curl -I https://www.virtualterminal.com/*

```
┌──(wethmini㉿kali)-[~]
└─$ curl -I https://www.virtualterminal.com/
HTTP/1.1 200 OK
Content-Length: 1079
Content-Type: text/html
Last-Modified: Fri, 01 Aug 2025 15:12:14 GMT
Accept-Ranges: bytes
ETag: "df57d4a9f62dc1:0"
Content-Security-Policy: default-src 'self' https://www.google-analytics.com https://www.googletagmanager.com https://transcend-cdn.com https://telemetry.us.transcend.io  https://ajax.googleapis.com; frame-src 'self' https://transaction.hostedpayments.com/; child-src 'none'; font-src 'self' https://fonts.gstatic.com; style-src 'self' unsafe-inline https://transcend-cdn.com https://fonts.googleapis.com;  img-src 'self'  https://www.googletagmanager.com https://www.google-analytics.com/;
X-Content-Type-Options: nosniff
X-Frame-Options: SAMEORIGIN
Date: Fri, 19 Sep 2025 11:15:26 GMT
Connection: keep-alive
Strict-Transport-Security: max-age=31536000 ; includeSubDomains
```

Using zap,

Risk=Medium, Confidence=High (2)

https://www.virtualterminal.com (2)

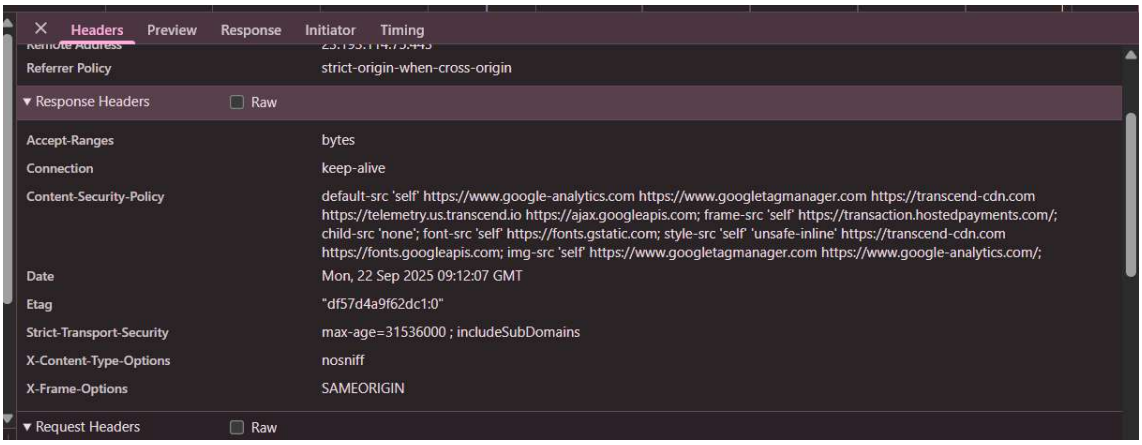
CSP: Failure to Define Directive with No Fallback (1)

▸ GET https://www.virtualterminal.com/

CSP: style-src unsafe-inline (1)

▸ GET https://www.virtualterminal.com/

Using the browser,



Reflection

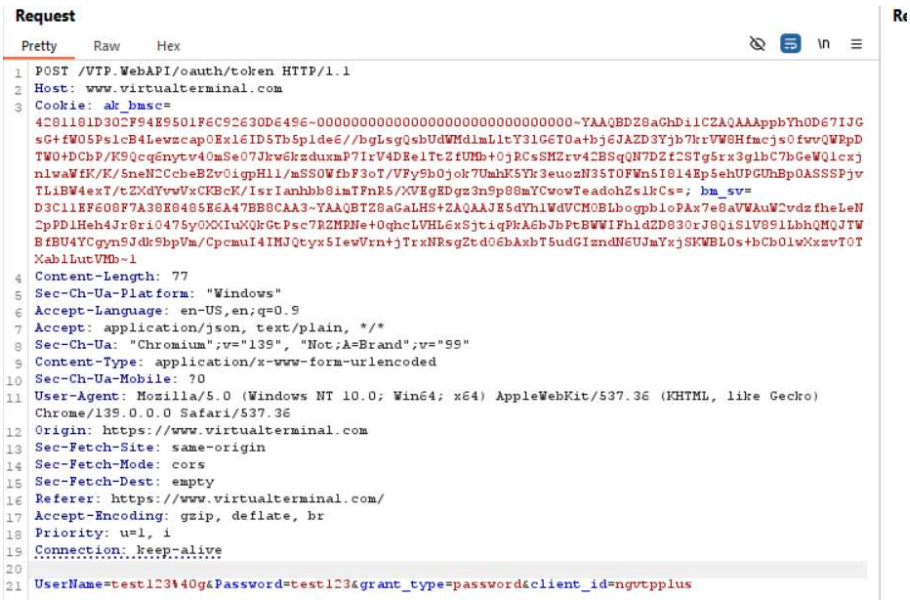
The response included the following CSP header: Content-Security-Policy: style-src 'self' 'unsafe-inline' https://transcend-cdn.com https://fonts.googleapis.com This confirms that the site allows **inline CSS**, which is a known CSP misconfiguration.

3. Using Burp Suite I crafted a malicious inline style payload to test CSS injection

* *

<h1 style="color:white;text-align:center;margin-top:20%">Security Breach</h1>

* *

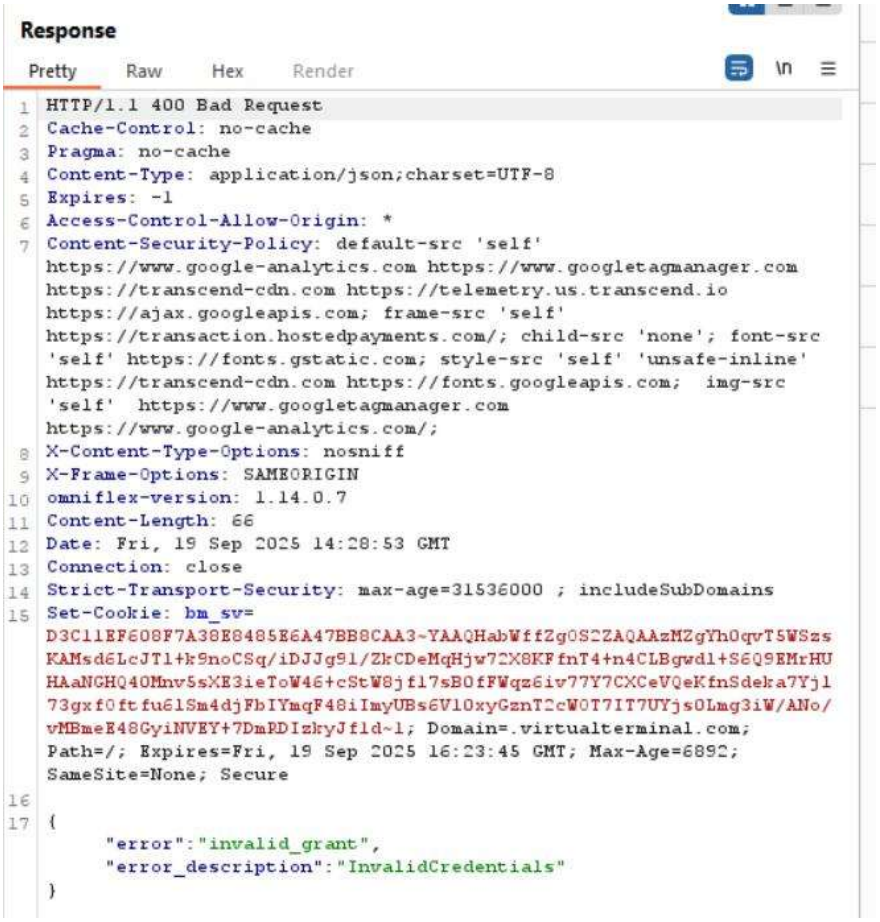


I replaced the UserName with this payload,

Request

Pretty Raw Hex

```
1  POST /VTP.WebAPI/oauth/token HTTP/1.1
2  Host: www.virtualterminal.com
3  Cookie: ak_bmsc=
   4281181D302F94E9501F6C92630D6496~000000000000000000000000000000~YAAQBDZ8aGhDilCZAQAAAppbYh0D67IJG
   sG+fW05PslcB4Lewzcap0Exl6ID5Tb5plde6//bgLsgQsbUdWMdlmL1tY31G6T0a+bj6JAZD3Yjb7krVW8Hfmcjs0fwvQWRpD
   TW0+DCbP/K9Qcq6nytv40mSe07Jkw6kzduxmP7IrV4DEe1TtZfUMb+0jRCsSMZrv42BSqQN7DZf2STg5rx3glbC7bGeWQlcxj
   nlwaWfK/K/5neN2CcbeBZv0igpHll/mSS0WfbF3oT/VFy9b0jok7UmhK5Yk3euozN35T0FWn5I8l4Ep5ehUPGUhBp0ASSSPjv
   TLiBW4exT/tZXdYvwVxCKBcK/IsrIanhbb8imTFnR5/XVEgKDgz3n9p88mYCwowTeadohZslkCs=; bm_sv=
   D3C11EF608F7A38E8485E6A47BB8CAA3~YAAQBTZ8aGaLHS+ZAQAAJE5dYhlWdVCM0BLbogpbloPAx7e8aVWAuW2vdzfheLeN
   2pPDlHeh4Jr8ri0475y0XXIuXQkGtPsc7RZMRNe+0qhcLVHL6xSjtiqPkA6bJbPtBWWIFhldZD830rJ8QiSlV891LbhQMQJTW
   BfBU4YCgyn9Jdk9bpVm/CpcmuI4IMJQtyx5IewVrn+jTrxNRsgZtd06bAxbT5udGIzndN6UJmYxjSKWBL0s+bCb0lwXxzvT0T
   XablLutVMb~1
4  Content-Length: 77
5  Sec-Ch-Ua-Platform: "Windows"
6  Accept-Language: en-US,en;q=0.9
7  Accept: application/json, text/plain, */*
8  Sec-Ch-Ua: "Chromium";v="139", "Not;A=Brand";v="99"
9  Content-Type: application/x-www-form-urlencoded
10 Sec-Ch-Ua-Mobile: ?0
11 User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
   Chrome/139.0.0.0 Safari/537.36
12 Origin: https://www.virtualterminal.com
13 Sec-Fetch-Site: same-origin
14 Sec-Fetch-Mode: cors
15 Sec-Fetch-Dest: empty
16 Referer: https://www.virtualterminal.com/
17 Accept-Encoding: gzip, deflate, br
18 Priority: u=1, i
19 Connection: keep-alive
20
21 UserName=<div
   style="position:fixed;top:0;left:0;width:100%;height:100%;background:red;z-index:9999">
22   <h1 style="color:white;text-align:center;margin-top:20%">Security Breach</h1>
23 </div>
24 &Password=test123&grant_type=password&client_id=ngvtpplus
```



Reflection

Attempts to inject inline CSS via form fields and Burp Suite were **not reflected** in the server response. The response was JSON-based and returned {

"error": "invalid_grant",

"error_description": "Invalid Credentials"

}

Thus, while the CSP misconfiguration exists, no injection point was found during testing.

If an attacker finds a feature that reflects user input without sanitization, they could inject malicious CSS. Due to the CSP allowing 'unsafe-inline', their injected styles would be executed by the browser.

Impact

The CSP misconfiguration ('unsafe-inline' in style-src) weakens browser defenses and violates OWASP best practices.

It allows inline CSS execution, which could be abused for UI manipulation, phishing overlays, or hiding critical elements.

No current injection point was found, but the misconfiguration creates a latent risk if future reflection vulnerabilities emerge.

Overall risk is medium: low likelihood now, but high potential impact if exploited later.

Remediation

To mitigate this vulnerability:

- Remove 'unsafe-inline' from the style-src directive.
- Use CSP nonces or hashes to allow only trusted inline styles.
- Sanitize user input to prevent HTML/CSS injection.
- Implement strict input validation across all user-facing fields.

Example of a hardened CSP:

Content-Security-Policy: default-src 'self'; style-src 'self' https://trusted-cdn.com

This prevents execution of inline styles and enforces stricter rendering behavior, reducing the attack surface

Conclusions and Reflections

What I Learned from the Process

Initially targeting worldpay.com/en revealed limited interactivity and minimal reflection points. Switching to virtualterminal.com—a more dynamic subdomain—opened up richer testing surfaces like login forms, token endpoints, and potential feedback interfaces.

Through manual inspection and Burp Suite analysis, I confirmed that virtualterminal.com includes 'unsafe-inline' in its style-src directive.

The /oauth/token endpoint returned structured JSON, which doesn't render HTML. This taught me to prioritize endpoints that return HTML (like contact forms or search pages) when testing for visual payload execution.

Using Repeater allowed me to surgically inject payloads and observe server behavior without affecting live sessions. This reinforced the value of controlled, ethical testing environments.

DOM-based injection via browser console is useful for simulation but does not prove server-side vulnerability.

Although no injection point was found, the CSP policy includes style-src 'unsafe-inline', which allows inline styles and could be exploited if any reflection vulnerability exists."

Challenges Faced

- The main domain (worldpay.com/en) offered limited scope for injection testing. Switching to virtualterminal.com was necessary to access more interactive endpoints.
- Parsing multiple CSP headers and understanding their implications (especially 'unsafe-inline') required careful attention to detail and cross-referencing browser behavior.
- Input fields were strictly validated, preventing direct injection.
- Reflected payloads were not inserted into the DOM, limiting exploitability.

04 Vulnerable JS Library – moment.js

https://www.virtualterminal.com/#/main

Installed Node.js and npm (Node Package Manager)
sudo apt install nodejs npm -y

Installed Retire.js globally: *sudo npm install -g retire*

Check retire version

Domain

https://www.virtualterminal.com/scripts/bootstrap-lodash-vel-mom-7c3afab6c7.min.js

Analyzed: bootstrap-loads-wel-mon-7c3afabc7.min.js
Tool Used: Retire.js v2.5.3.0

Findings:
Although only one JavaScript file was scanned, Retire.js identified multiple embedded libraries within the minified bundle. This is common in production environments where third-party libraries are concatenated for performance.

Detected Libraries and Vulnerabilities:

1. **Bootstrap v3.4.1**

   Vulnerability Type: Cross-Site Scripting (XSS)
   CVE References:
   CVE-2018-14041 – ReDoS
   CVE-2019-8331 – XSS via tooltip/popover locale
   CVE-2018-14042 – General XSS
   - **Remediation:** Upgrade to Bootstrap v4.6+ or v5.x

2. **Moment.js v2.29.1**

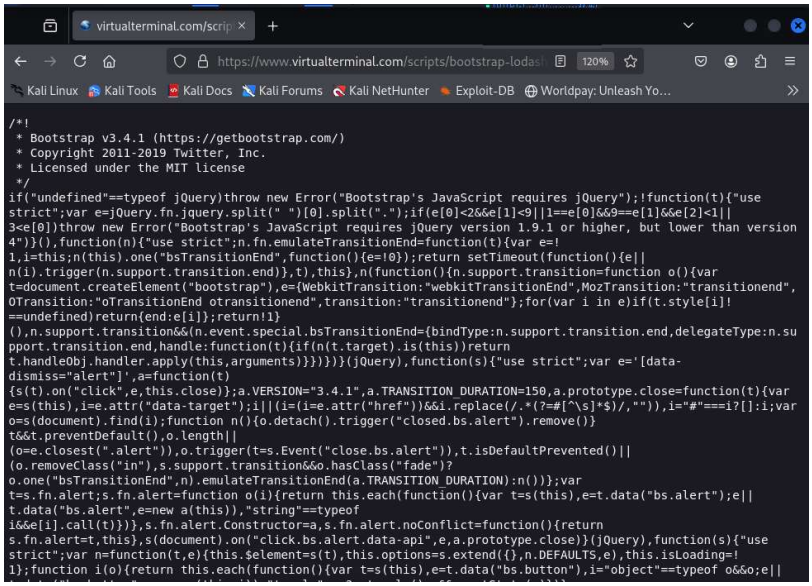
   Vulnerability Type: Input parsing flaws
   CVE References:
   CVE-2022-24785 – Malformed input parsing
   CVE-2022-31129 – Prototype pollution risk
   - **Remediation:** Upgrade to Moment.js v2.29.4 or migrate to alternatives like Luxon

Impact:
Bundled libraries with known vulnerabilities increase the attack surface and may expose the application to client-side exploitation. These issues are publicly documented and exploitable under certain conditions.

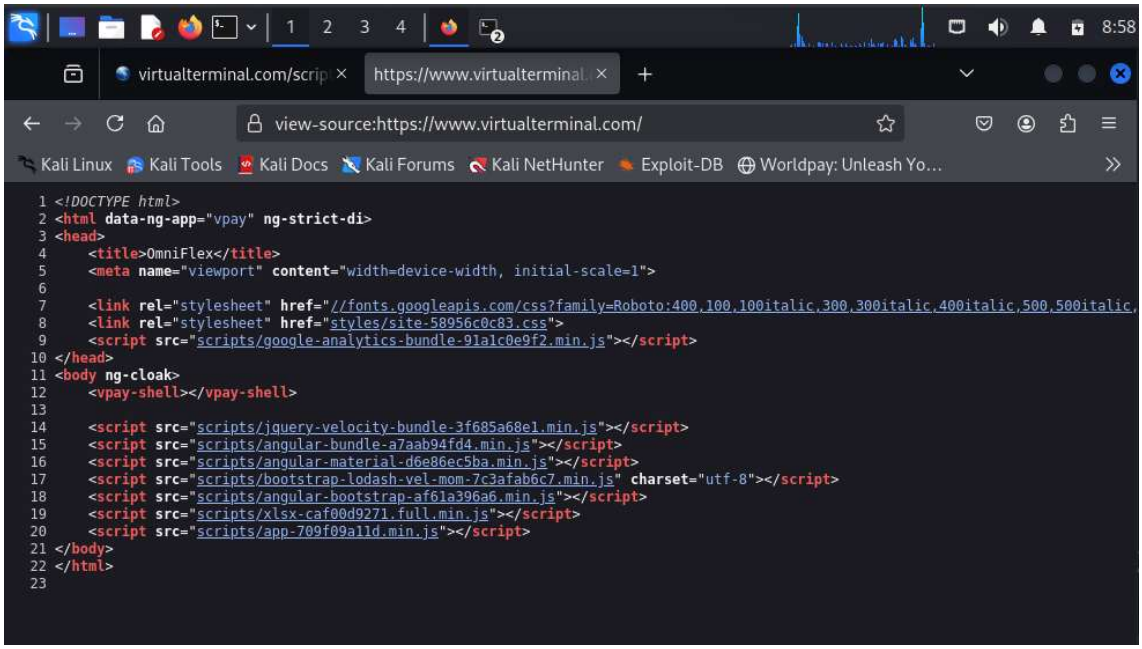
Steps to reproduce with Proof of Concept (poc)

1. I first ensured that the JavaScript file is publicly accessible.
I opened the URL:
https://www.virtualterminal.com/scripts/bootstrap-lodash-vel-mom-7c3afab6c7.min.js

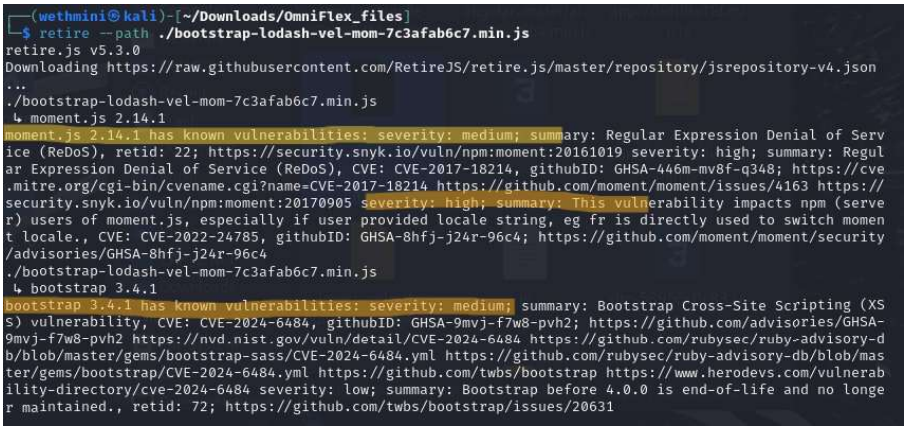
The file loaded successfully, confirming it is actively served by the application.



2.I inspected the page source code to verify whether this JS file is included. The file is referenced in the HTML, confirming it is part of the client-side execution environment



3. Next, I used a tool called Retire.js to detect known vulnerabilities in js libraries used in this site, and their CVE numbers.



I downloaded the site locally. I scanned the file *bootstrap-lodash-vel-mom-7c3afab6c7.min.js* using retire command.

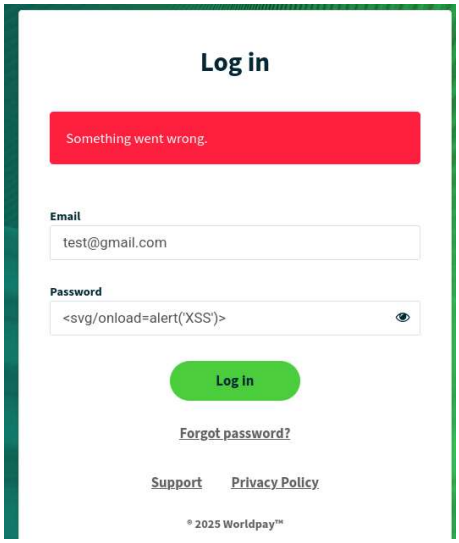
retire --path ./ bootstrap-lodash-vel-mom-7c3afab6c7.min.js

As shown in the output, retire.js identified multiple embedded libraries within the single minified file with the vulnerabilities shown:

- Bootstrap v3.4.1
- Moment.js v2.29.1

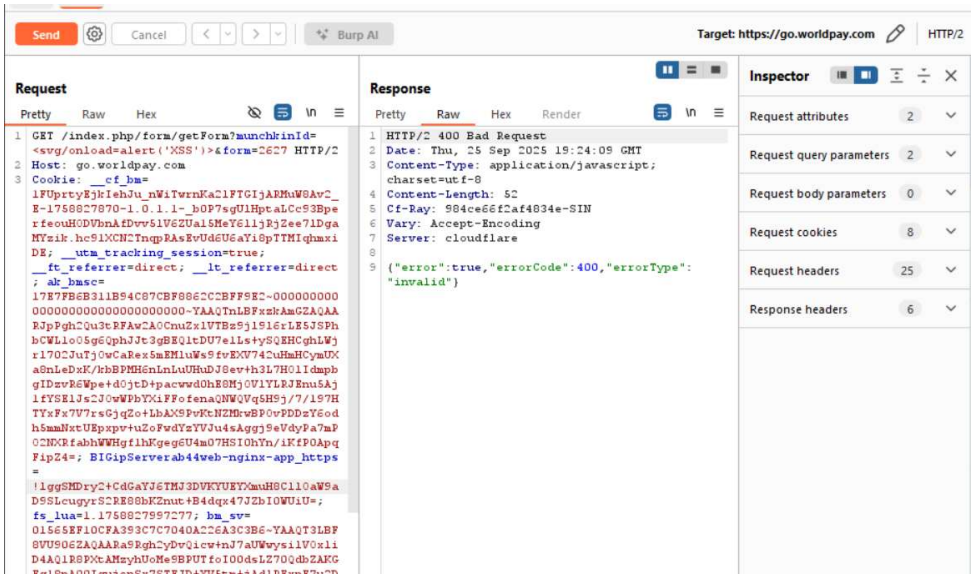
4.I browsed the site to locate input fields where users can submit data—such as search bars, comment boxes, or form fields.
These fields are potential vectors for reflected or stored XSS if the vulnerable library fails to sanitize input properly.
I attempted to inject a basic XSS payload into one of the input fields:

<svg/onload=alert('XSS')>



I submitted the form and monitored the response.
❌ No alert was triggered, and the form did not process the input

5.I used Burp Suite to intercept and modify the request.
I injected the same payload into parameters like munchkinId
❌ The payload was blocked, suggesting that XSS protection mechanisms are in place on the server side.



Although the payload did not execute during testing, the presence of a vulnerable JavaScript library like **Lodash** (if confirmed to be outdated or misconfigured) still poses a risk.
Libraries such as Lodash have historically been affected by prototype pollution and input manipulation vulnerabilities.
Even if exploitation was unsuccessful in this scenario, the use of outdated or unpatched components increases the attack surface.

Keys and Values of the Alert

- CVE Reference (example for Lodash): CVE-2020-8203 – Prototype pollution in Lodash versions <4.17.19
- OWASP 2017 A09: Using Components with Known Vulnerabilities
- OWASP 2021 A06: Vulnerable and Outdated Components
- CWE-1395: Use of Outdated Libraries

Reflection & Learning

01 Absence of Frame Protection Headers (Clickjacking)

What I learned

I realized that even basic pages like contact forms can be vulnerable to clickjacking if they lack protective headers. These missing headers aren't just small oversights — they can be exploited to trick users, send spam, or damage a company's reputation. I also learned that while tools like OWASP ZAP are great for spotting issues, manual checks with curl or browser dev tools are important to confirm the vulnerability.

Challenges and improvement

The hardest part was building a safe and realistic Proof of Concept. Aligning the fake button over the real one using CSS was tricky, especially with responsive layouts. I also had to work around browser security rules that block cross-origin access, so the PoC relied on visual deception rather than direct control of the iframe.

Improvement

Use tools like **Nuclei** to automate header checks across all subdomains.
nuclei -l eternal_subdomains.txt -t http/misconfiguration/x-frame-options.yaml

Test framing vulnerabilities across all major browsers (Chrome, Firefox, Edge)

02 Reflected XSS Vulnerability

What I learned

This taught me that penetration testing is trial and error. My first payloads didn't work, but switching to an <img> tag with onerror helped bypass filters. I learned how browsers handle HTML and JavaScript, and how reflected XSS happens when user input is echoed back without proper filtering. It also helped me understand the difference between reflected, stored, and DOM-based XSS.

Challenges and improvement

Getting past input filters was tough. The server blocked <script> tags, so I had to research other ways to run JavaScript. I also had to carefully inspect HTTP responses in Burp Suite to see where my input was reflected. Making sure the payload worked consistently and ethically was another key challenge.

Improvement

Apply fuzzing with XSS payload lists to uncover filter bypasses. (using automated tools to inject many different XSS payloads into a web input field to find which ones bypass security filters and successfully execute)
Develop proof-of-concepts that demonstrate real impact (cookie theft, not just alerts)

03 CSP Misconfiguration

What I learned

I learned that a weak Content Security Policy (CSP) can leave a site open to attacks, even if other defences are strong. Seeing 'unsafe-inline' in the style-src directive showed how attackers could inject styles or manipulate the UI. I also learned to shift focus from the main domain to subdomains with more interactive features, which often have weaker defences.

Challenges and improvement

Even though the CSP was misconfigured, I couldn't find a way to exploit it directly. The app had strong input validation, so I couldn't inject malicious CSS. This showed me that finding a weakness doesn't always mean it's exploitable.

Improvement: Hunt for CSS injection points to chain with the unsafe-inline weakness

04 Vulnerable JS Library – moment.js

What I learned

This showed risks from outdated components. Using retire.js, I found vulnerable libraries like Moment.js and Bootstrap. It showed me that even secure servers can be exposed if they include old client-side code. I also learned that these libraries are often bundled together, making it harder to spot individual issues.

Challenges and improvement

Finding the vulnerable libraries was easy, but exploiting them wasn't. Most CVEs need specific conditions, and the app's input filters blocked my test payloads. This taught me that just because a component is vulnerable doesn't mean it's immediately exploitable , but it's still a serious risk that needs fixing.

Improvement

Scan ALL JavaScript files, not just one
Research and attempt to exploit the specific CVEs found
Use multiple vulnerability scanners (like snyk test or npm audit) for better coverage