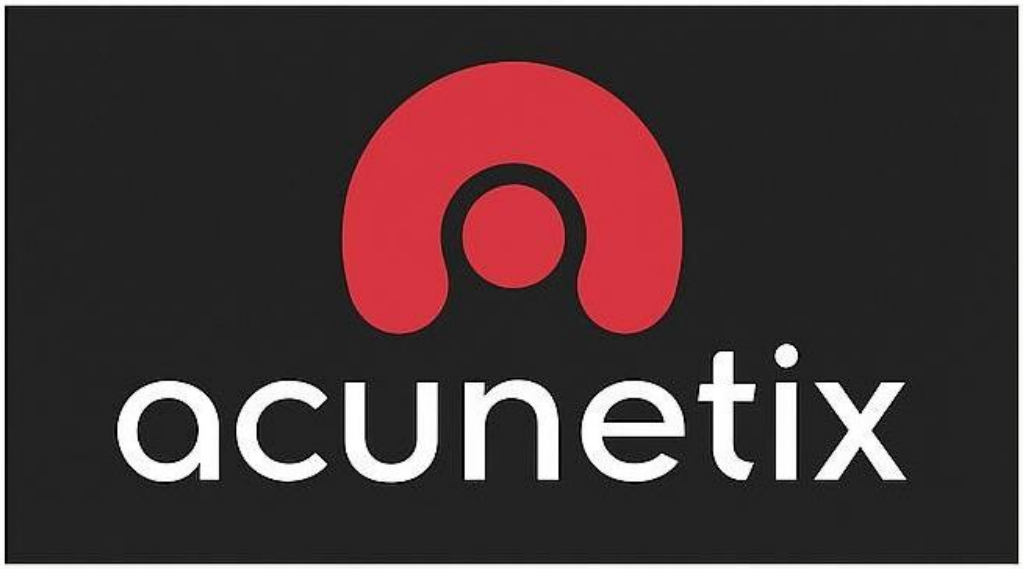
VAPT REPORT

ON

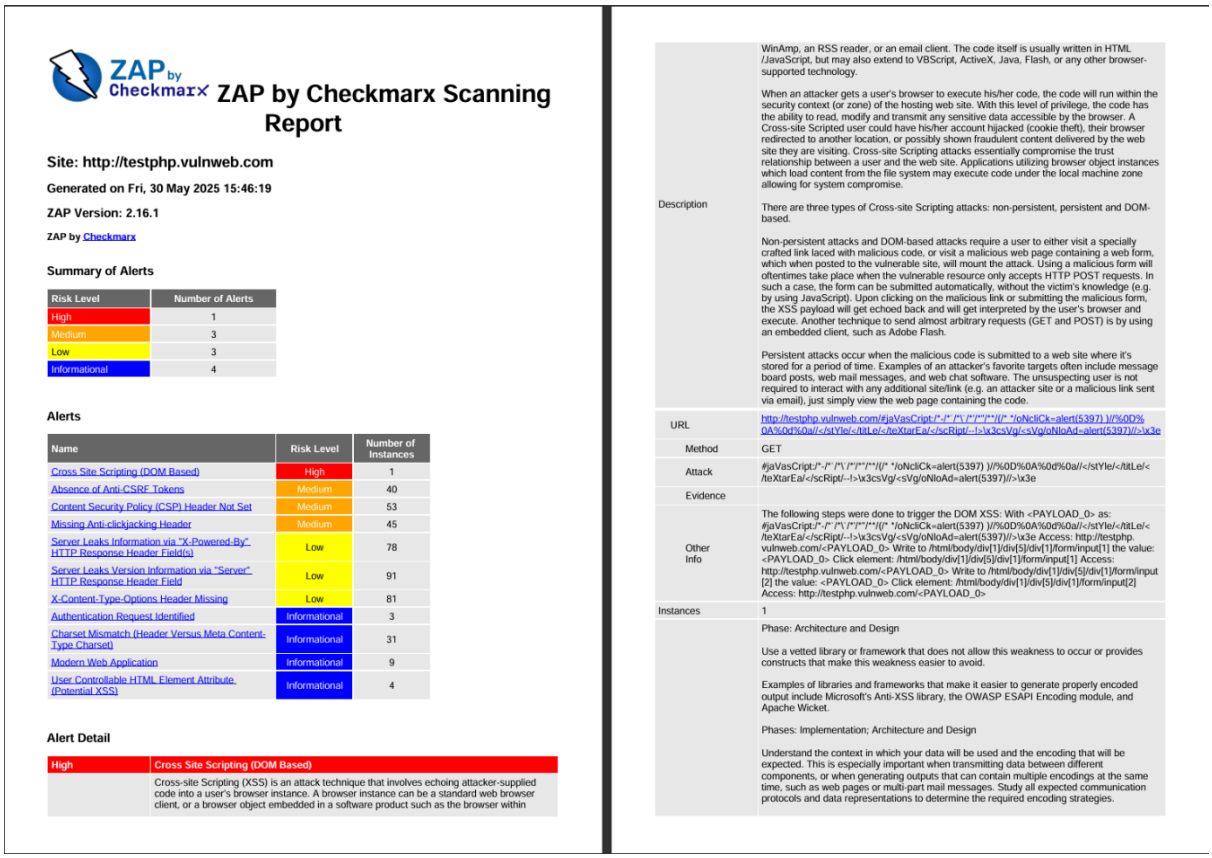


Reported by

Shayan Chakraborty

Web Application Vulnerability Assessment Report

- **Target Website:** http://testphp.vulnweb.com
- **Vulnerability Identified:** Reflected Cross Site Scripting (XSS)
- **Severity:** High
- **Payload Used:** <?foo="><x foo='?><script>javascript:alert(1)</script>'>">



ZAP by Checkmarx ZAP by Checkmarx Scanning Report

Site: http://testphp.vulnweb.com

Generated on Fri, 30 May 2025 15:46:19

ZAP Version: 2.16.1

ZAP by Checkmarx

Summary of Alerts

Risk Level	Number of Alerts
High	1
Medium	3
Low	3
Informational	4

Alerts

Name	Risk Level	Number of Instances
Cross Site Scripting (DOM Based)	High	1
Absence of Anti-CSRF Tokens	Medium	40
Content Security Policy (CSP) Header Not Set	Medium	53
Missing Anti-clickjacking Header	Medium	45
Server Leaks Information via "X-Powered-By" HTTP Response Header Field(s)	Low	78
Server Leaks Version Information via "Server" HTTP Response Header Field	Low	91
X-Content-Type-Options Header Missing	Low	81
Authentication Request Identified	Informational	3
Charset Mismatch (Header Versus Meta Content-Type Charset)	Informational	31
Modern Web Application	Informational	9
User Controllable HTML Element Attribute (Potential XSS)	Informational	4

Alert Detail

High	Cross Site Scripting (DOM Based)
	Cross-site Scripting (XSS) is an attack technique that involves echoing attacker-supplied code into a user's browser instance. A browser instance can be a standard web browser client, or a browser object embedded in a software product such as the browser within

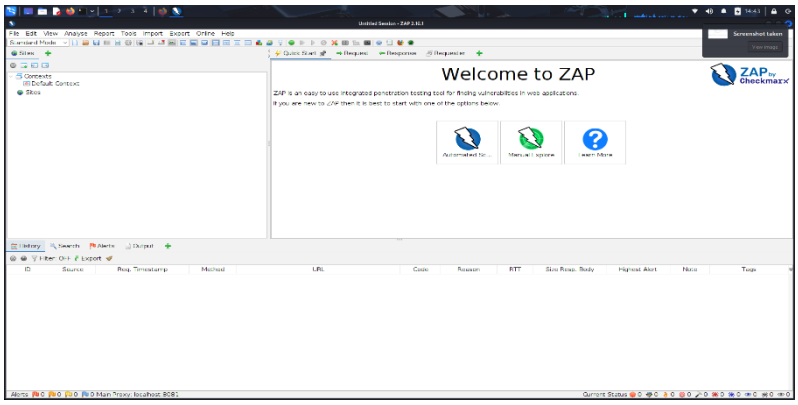
Description	WinAmp, an RSS reader, or an email client. The code itself is usually written in HTML /JavaScript, but may also extend to VBScript, ActiveX, Java, Flash, or any other browser-supported technology. When an attacker gets a user's browser to execute his/her code, the code will run within the security context (or zone) of the hosting web site. With this level of privilege, the code has the ability to read, modify and transmit any sensitive data accessible by the browser. A Cross-site Scripted user could have his/her account hijacked (cookie theft), their browser redirected to another location, or possibly shown fraudulent content delivered by the web site they are visiting. Cross-site Scripting attacks essentially compromise the trust relationship between a user and the web site. Applications utilizing browser object instances which load content from the file system may execute code under the local machine zone allowing for system compromise. There are three types of Cross-site Scripting attacks: non-persistent, persistent and DOM-based. Non-persistent attacks and DOM-based attacks require a user to either visit a specially crafted link laced with malicious code, or visit a malicious web page containing a web form, which when posted to the vulnerable site, will mount the attack. Using a malicious form will oftentimes take place when the vulnerable resource only accepts HTTP POST requests. In such a case, the form can be submitted automatically, without the victim's knowledge (e.g. by using JavaScript). Upon clicking on the malicious link or submitting the malicious form, the XSS payload will get echoed back and will get interpreted by the user's browser and execute. Another technique to send almost arbitrary requests (GET and POST) is by using an embedded client, such as Adobe Flash. Persistent attacks occur when the malicious code is submitted to a web site where it's stored for a period of time. Examples of an attacker's favorite targets often include message board posts, web mail messages, and web chat software. The unsuspecting user is not required to interact with any additional site/link (e.g. an attacker site or a malicious link sent via email), just simply view the web page containing the code.
URL	http://testphp.vulnweb.com/#jaVasCript:/*-/* /*\ /*'/*"/**/(/* */oNcliCk=alert(5397))//%0D%0A%0d%0a//</stYle/</titLe/</teXtarEa/</scRipt/--!>\x3csVg/<sVg/oNloAd=alert(5397)//>\x3e
Method	GET
Attack	#jaVasCript:/*-/* /*\ /*'/*"/**/(/* */oNcliCk=alert(5397))//%0D%0A%0d%0a//</stYle/</titLe/</teXtarEa/</scRipt/--!>\x3csVg/<sVg/oNloAd=alert(5397)//>\x3e
Evidence	
Other Info	The following steps were done to trigger the DOM XSS: With <PAYLOAD_0> as: #jaVasCript:/*-/* /*\ /*'/*"/**/(/* */oNcliCk=alert(5397))//%0D%0A%0d%0a//</stYle/</titLe/</teXtarEa/</scRipt/--!>\x3csVg/<sVg/oNloAd=alert(5397)//>\x3e Access: http://testphp.vulnweb.com/<PAYLOAD_0> Write to /html/body/div[1]/div[5]/div[1]/form/input[1] the value: <PAYLOAD_0> Click element: /html/body/div[1]/div[5]/div[1]/form/input[1] Access: http://testphp.vulnweb.com/<PAYLOAD_0> Write to /html/body/div[1]/div[5]/div[1]/form/input[2] the value: <PAYLOAD_0> Click element: /html/body/div[1]/div[5]/div[1]/form/input[2] Access: http://testphp.vulnweb.com/<PAYLOAD_0>
Instances	1
	Phase: Architecture and Design Use a vetted library or framework that does not allow this weakness to occur or provides constructs that make this weakness easier to avoid. Examples of libraries and frameworks that make it easier to generate properly encoded output include Microsoft's Anti-XSS library, the OWASP ESAPI Encoding module, and Apache Wicket. Phases: Implementation; Architecture and Design Understand the context in which your data will be used and the encoding that will be expected. This is especially important when transmitting data between different components, or when generating outputs that can contain multiple encodings at the same time, such as web pages or multi-part mail messages. Study all expected communication protocols and data representations to determine the required encoding strategies.

Tools Used:

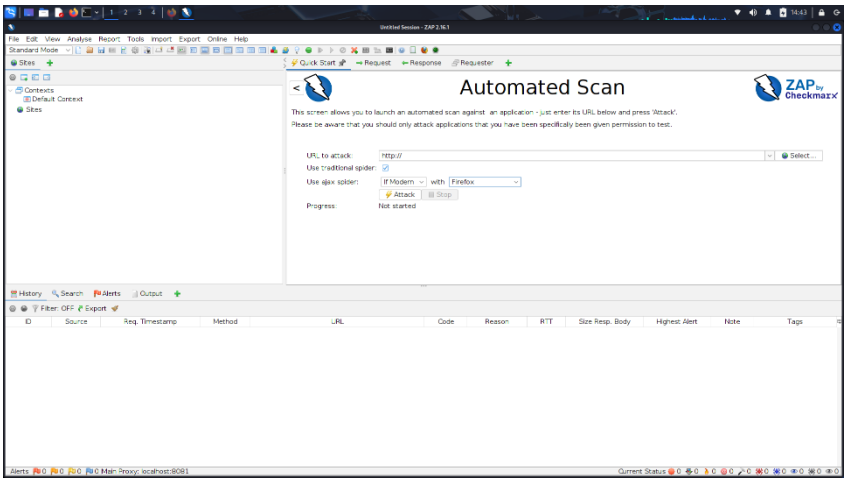
- OWASP ZAP
- Burp Suite
- Firefox browser (for manual validation)
- Kali Linux (penetration testing environment)

Step 1: Initial Recon with OWASP ZAP

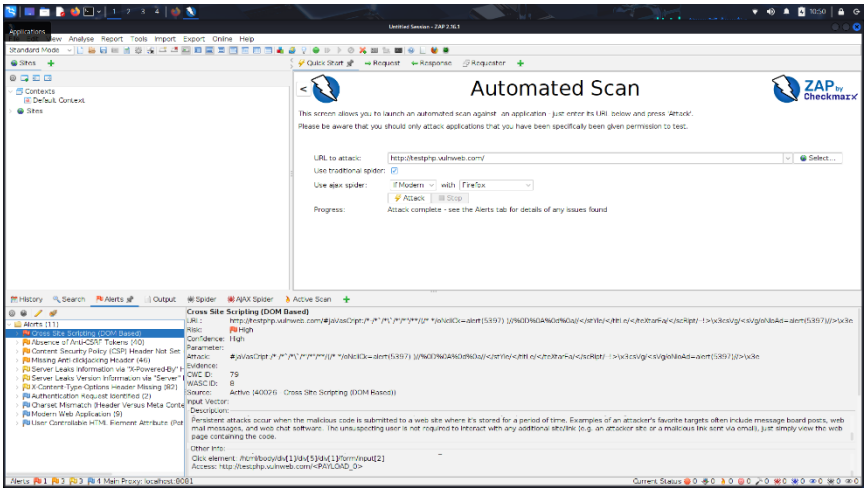
- Open **OWASP ZAP** and start a scan for the target website: `http://testphp.vulnweb.com`



- Enable both traditional and Ajax spiders for full discovery.



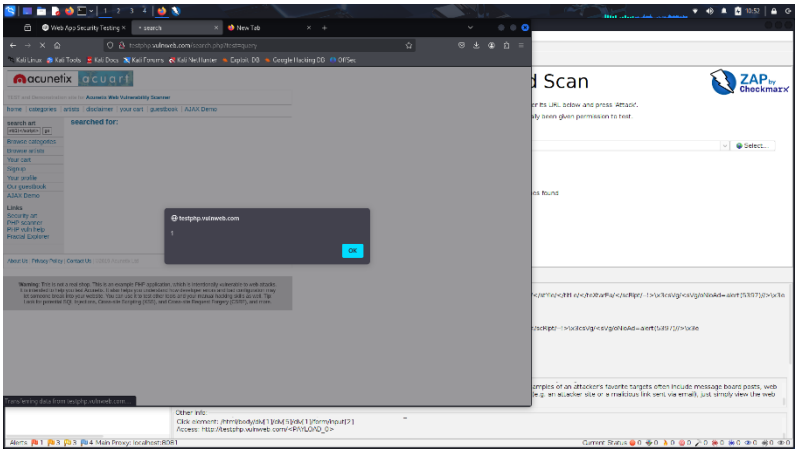
- Let ZAP crawl the site and detect initial vulnerabilities.



🎯 Step 2: Locate Search Input for XSS Testing

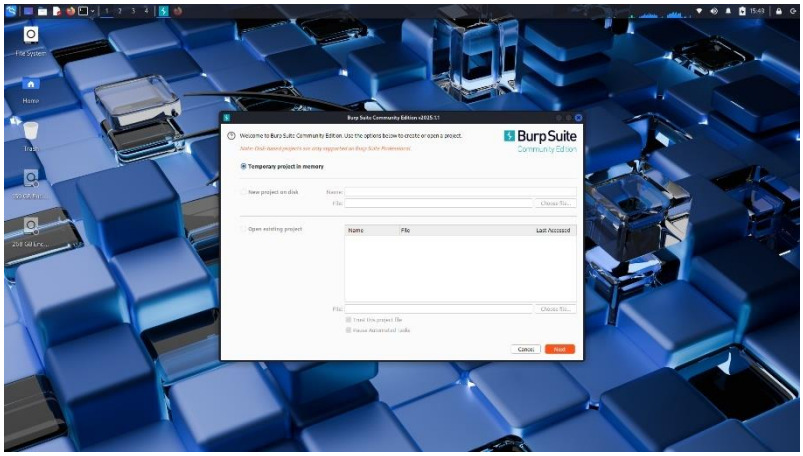
- Open the target page: http://testphp.vulnweb.com/search.php
- Test with simple inputs like `shayan` or `<script>alert(1)</script>` to detect reflected behaviour

- Confirm basic payload is reflected in the response.

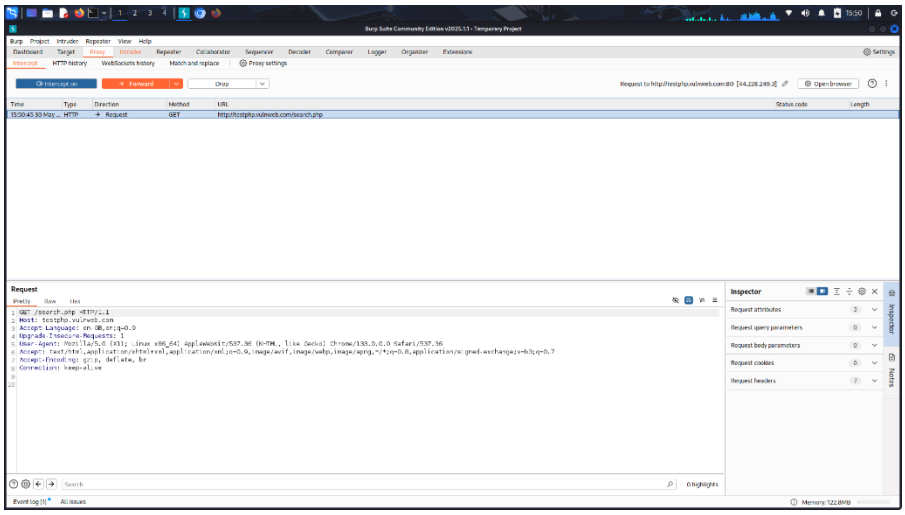


Step 3: Intercept the Request with Burp Suite

- Open **Burp Suite**, enable the proxy, and intercept the search request.

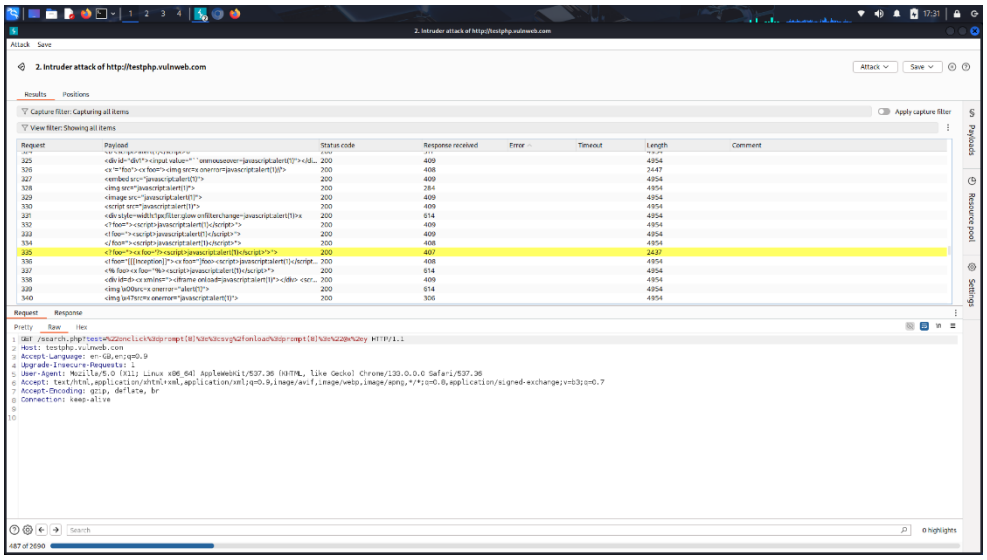


- Forward the captured request to **Intruder** for automated fuzzing.



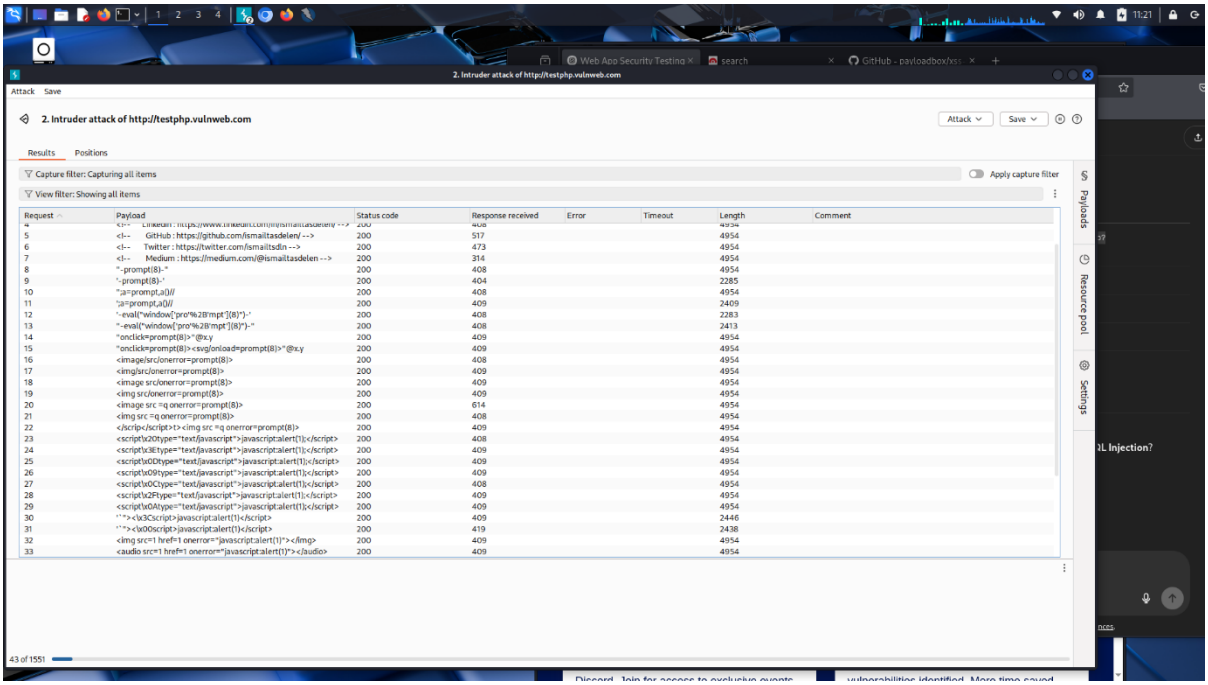
🎯 Step 4: Configure Burp Intruder Attack

- Set the attack position on the search query parameter.
- Load a list of XSS payloads or create a custom list.
- Add your discovered payload: `<?foo="><x foo='?><script>javascript:alert(1)</script>'>">`
- Start the attack and monitor status codes and response lengths.



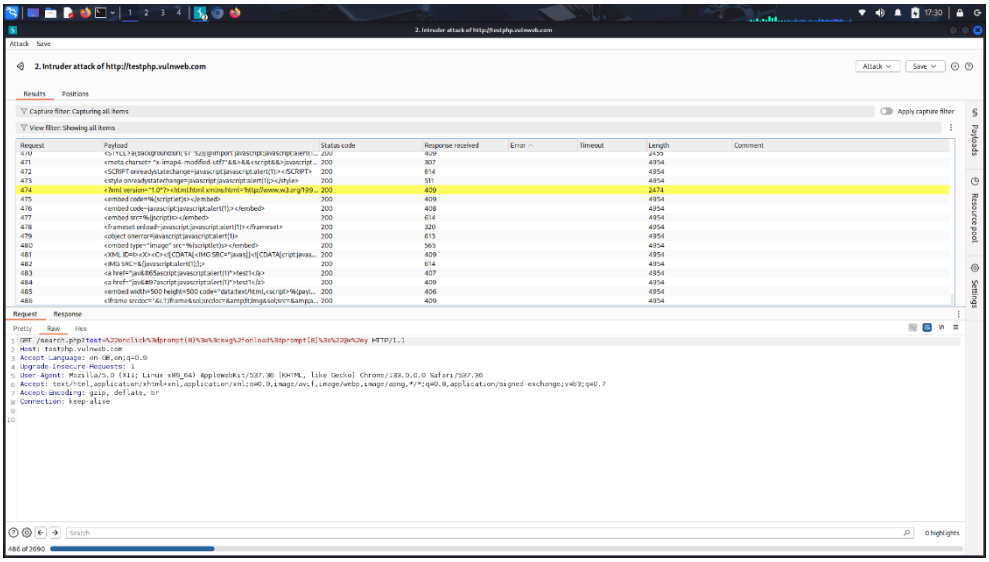
🕵️‍♂️ Step 5: Identify Successful Payloads

- Sort the results by response length or status code.
- Look for anomalies such as different lengths (indicating injected scripts).



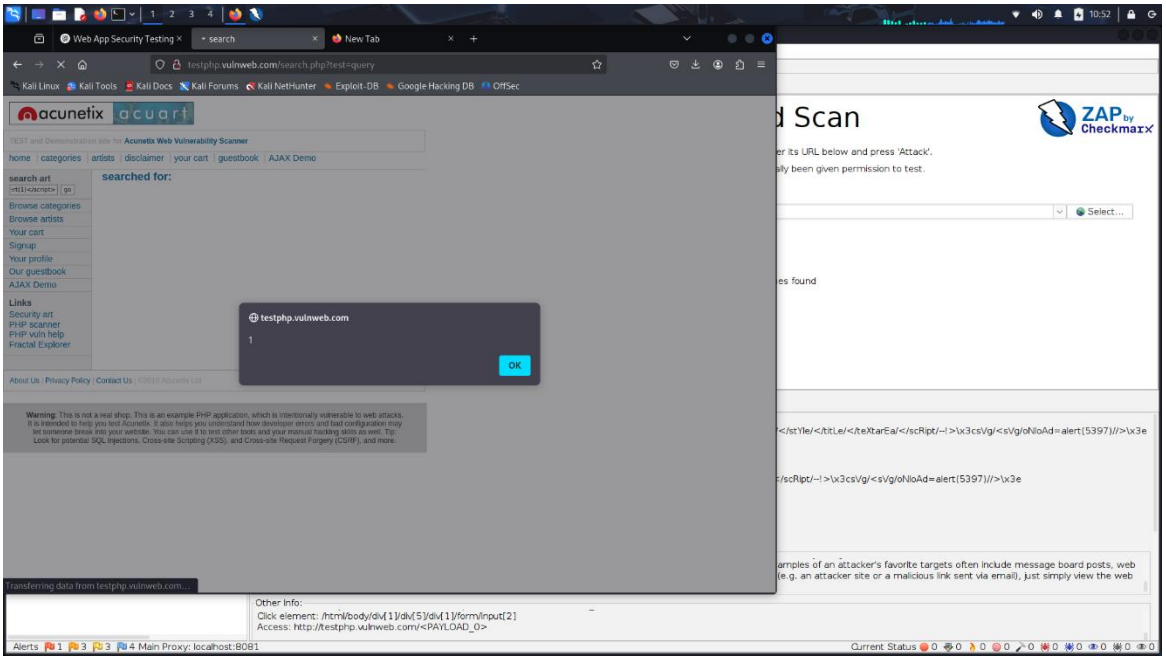
✅ Step 6: Validate Payload in Browser

- Copy the full URL with the payload.



- Open it in Firefox.

- Confirm a pop-up with alert `1` appears — proving XSS execution.



🔍 Summary

- **Vulnerable URL:** `http://testphp.vulnweb.com/search.php`
- **Vulnerable Parameter:** `test` (search input)
- **Confirmed Payload:** `<?foo="><x foo='?><script>javascript:alert(1)</script>'>">`
- **Confirmed Vulnerability:** Reflected XSS

☠️ Vulnerability Details

- **Type:** Reflected XSS

- **URL:** http://testphp.vulnweb.com/search.php

- **Payload Used:** `<?foo="><x foo='?><script>javascript:alert(1)</script>'>">`

- **Proof of Concept (PoC):**
  Browser displays an alert box with message “1” after payload injection.

💥 Impact:

- Session hijacking
- Credential theft
- Phishing
- Account takeover
- Website defacement

🛡️ Mitigation

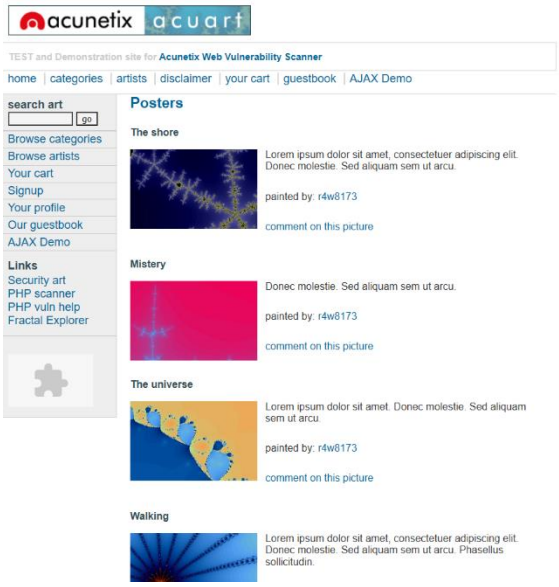
- Escape HTML output
- Input validation and sanitization
- Use Content Security Policy (CSP)
- Avoid using innerHTML, eval, and similar functions
- Use security frameworks and libraries with built-in protection
- Enable strict parameter handling
- Use HTTPS

💉 SQL Injection Assessment – Step-by-Step Guide

- **Target Website:** http://testphp.vulnweb.com
- **Vulnerability Identified:** SQL Injection (GET-based)
- **Severity:** High
- **Tools Used:**
  - Browser (Firefox)
  - Burp Suite
  - SQLMap
  - Kali Linux Terminal

🔍 Step 1: Identify Input Parameters

- Visit the site and navigate to a page with URL parameters, such as: **http://testphp.vulnweb.com/listproducts.php?cat=1**
- Check for parameter manipulation possibility (cat=1 is the injection point).
- Try entering: **' OR '1'='1**
- If the page behaves differently or displays all products, it may be vulnerable.



🛠 Step 2: Use SQLMap to Automate Exploitation

- Launch a terminal in Kali Linux.
- Run the following command to test the parameter:

sqlmap -u "http://testphp.vulnweb.com/listproducts.php?cat=1" --dbs

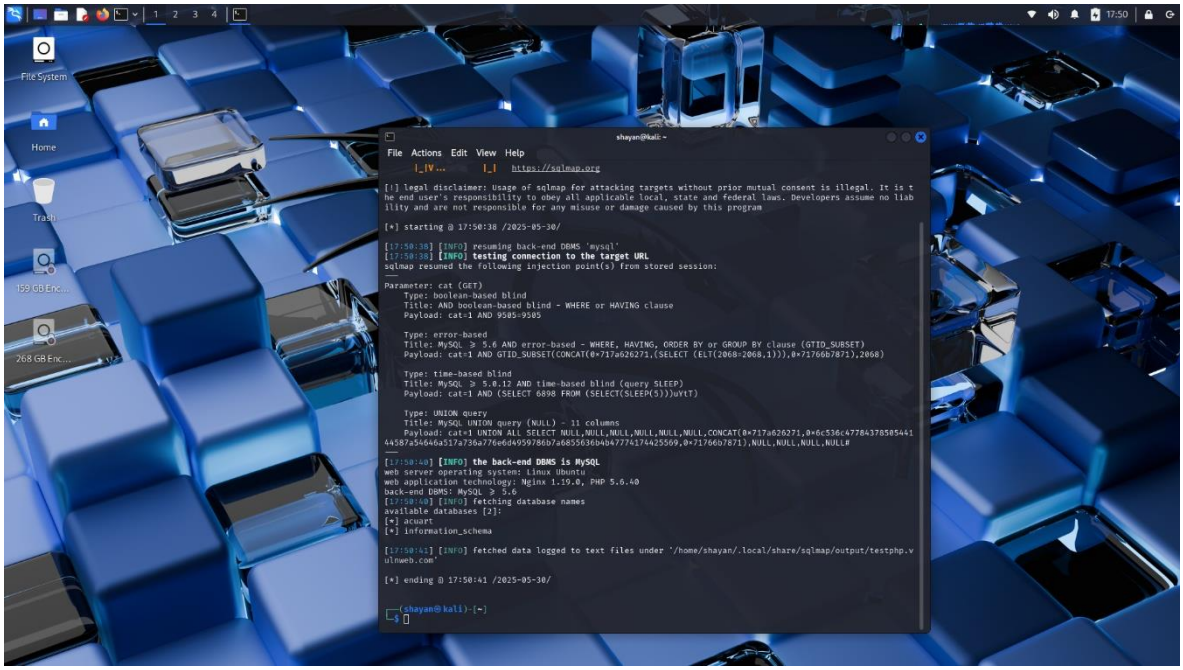
- SQLMap will probe the URL and try extracting database names.

🏢 Step 3: List Tables from Identified Database

- After discovering the database (e.g., acuart), run:

sqlmap -u "http://testphp.vulnweb.com/listproducts.php?cat=1" -D acuart --tables

- SQLMap will enumerate available tables within acuart.

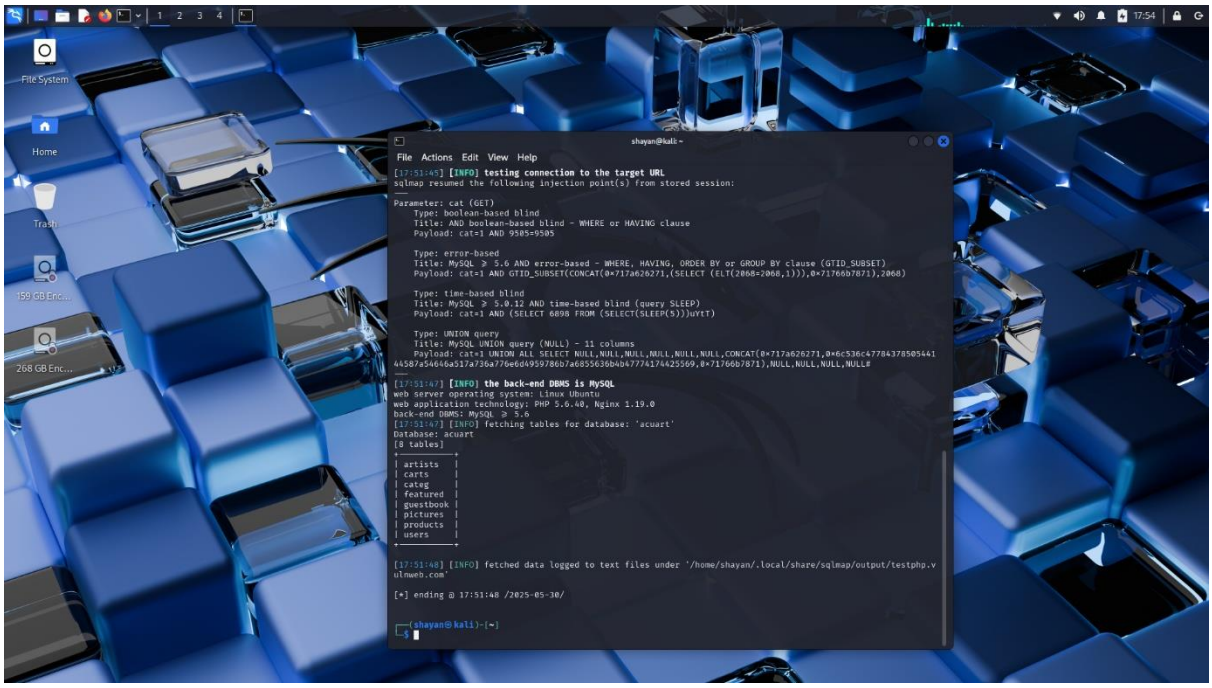


📋 Step 4: Dump Data from a Table

- Choose an interesting table (e.g., users) and dump its data:

sqlmap -u "http://testphp.vulnweb.com/listproducts.php?cat=1" -D acuart -T users --dump

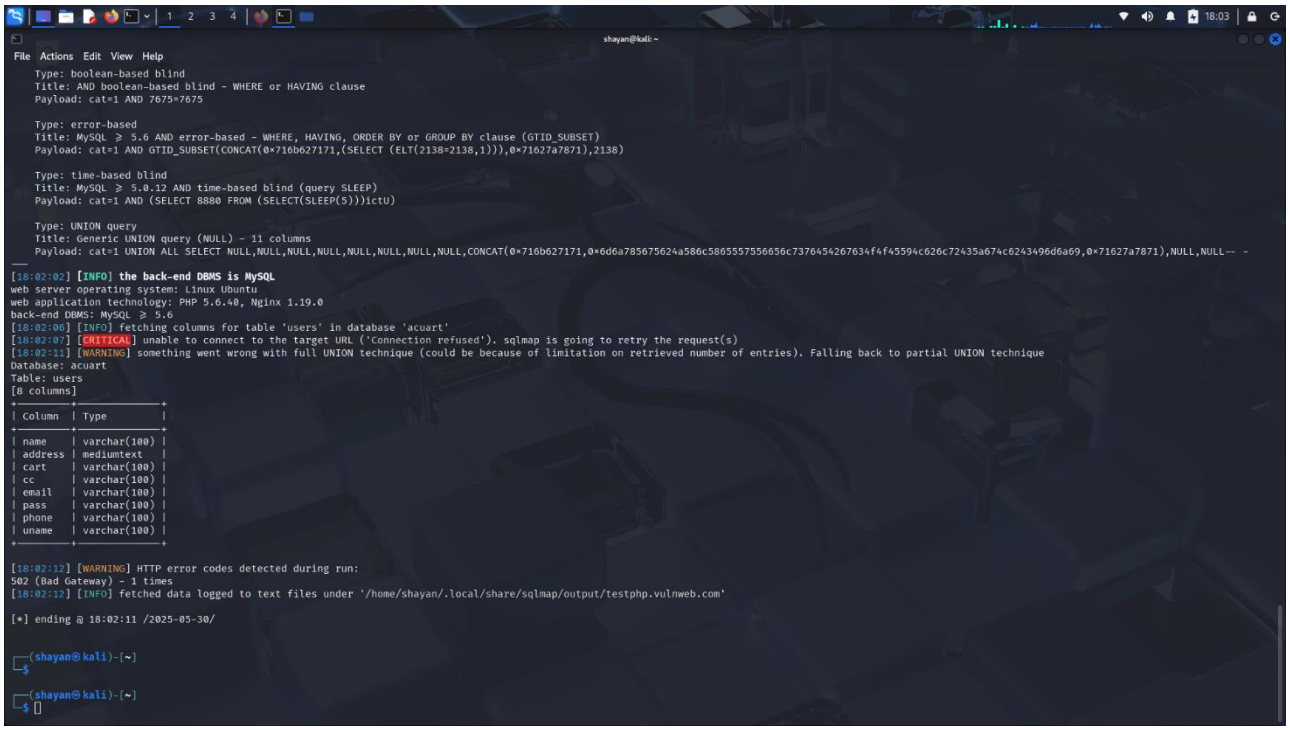
- SQLMap will attempt to extract usernames, passwords, emails, etc.



🔓 Step 5: Review Extracted Data

- Analyze the dumped output from SQLMap.
- Look for credentials, PII, or sensitive business information.

- Confirm that the injection allowed full database compromise.



🔍 Summary

- **Vulnerable URL:** http://testphp.vulnweb.com/listproducts.php?cat=‘1’
- **Injection Point:** cat parameter
- **Test Payload:** ' OR '1'='1
- **Tool Used for Exploitation:** SQLMap
- **Outcome:** Full database enumeration and data extraction

🛡️ SQL Injection Mitigation Recommendations

- Use **prepared statements** (parameterized queries)
- Validate and sanitize all user input
- Employ a **Web Application Firewall (WAF)**
- Apply **least privilege** principles on DB access
- Regularly test and monitor for injection vulnerabilities