Multi-Stage XSS Attack Simulation — Major Project

Author: **Sneha Chiliveru**

Course: **Cyber Security**
Date: **10-10-2025**

Abstract:

This project demonstrates reflected, stored, and DOM-based Cross-Site Scripting (XSS) in a fully isolated lab environment. It focuses on safe, benign proof-of-concepts (POCs), detection and forensics, and layered mitigations (CSP, output encoding, cookie hardening). All testing is performed on host-only/internal VMs under the student's control with synthetic tokens — no real secrets are accessed or exfiltrated.

Table of Contents:

---

1. Safety & Ethics Statement

- All testing was performed on isolated VMs inside a host-only/internal network under my control.

- No public or production websites were tested.
- No real cookies, credentials, or personal data were accessed or exfiltrated. Synthetic tokens (POC-UIDs) were used where demonstration of exfiltration mechanics was required.
- Snapshots were taken before active testing and restored as needed.

---

2. Project Goals & Learning Outcomes:

You will be able to:

- Explain differences between reflected, stored, and DOM XSS.
- Build safe, benign POCs that visibly demonstrate impact without exfiltrating real secrets.
- Implement logging and forensic trails that correlate evidence across browser and server artifacts.
- Deploy multiple mitigation layers and validate their effectiveness.

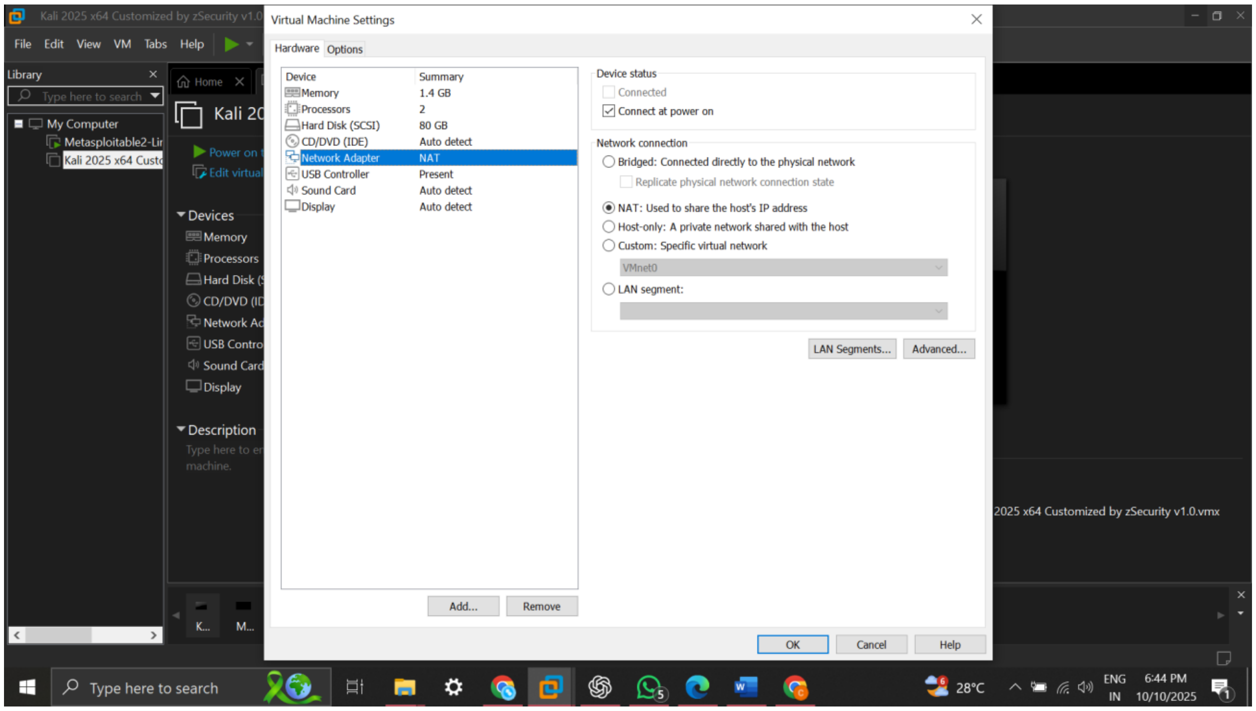
---

3. Lab Environment & Network Design:

Host OS: Kali Linux running virtualization software (VMware).

VMs (NAT)

- VM1 — DVWA: Linux (Kali), Apache2, PHP, MySQL. DVWA security set to *Low*. Network: NAT. Snapshot before testing.



Virtual Machine Settings

Hardware | Options

Device	Summary
Memory	1.4 GB
Processors	2
Hard Disk (SCSI)	80 GB
CD/DVD (IDE)	Auto detect
Network Adapter	NAT
USB Controller	Present
Sound Card	Auto detect
Display	Auto detect

Device status
- Connected
- [✓] Connect at power on

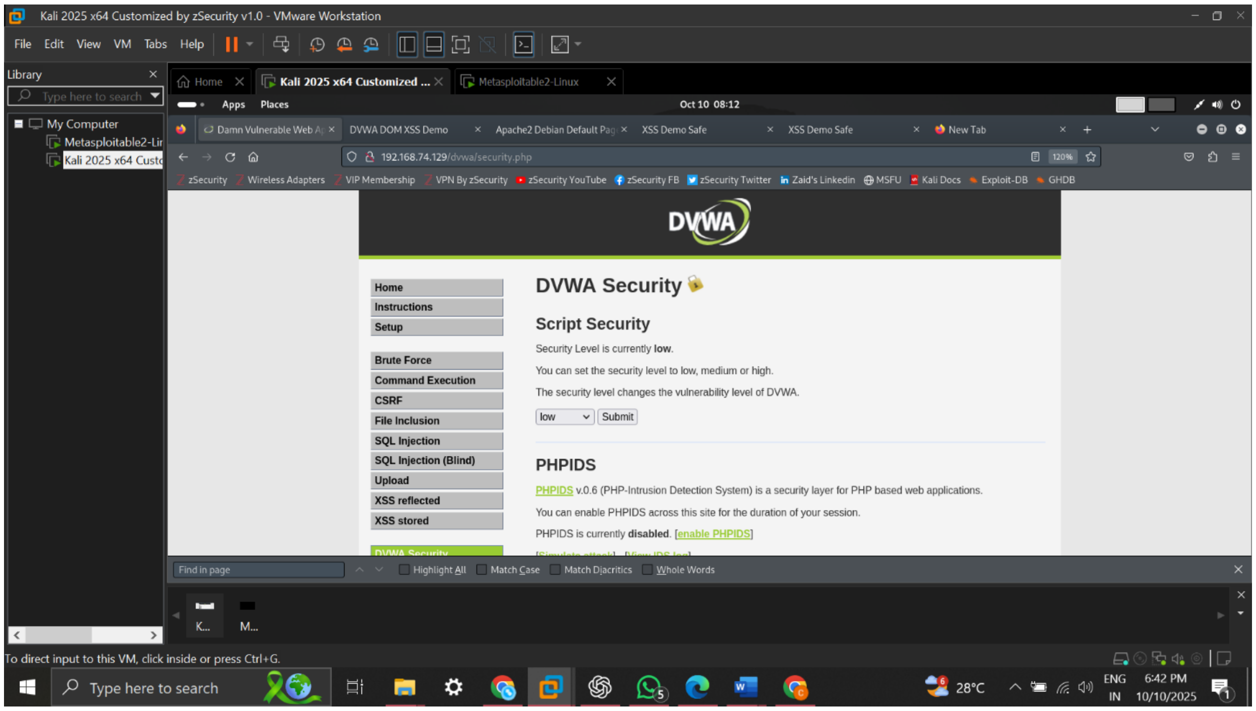
Network connection
- Bridged: Connected directly to the physical network
  - Replicate physical network connection state
- [●] NAT: Used to share the host's IP address
- Host-only: A private network shared with the host
- Custom: Specific virtual network
  - VMnet0
- LAN segment:

- VM2 — Attacker (Kali): Browser with DevTools.
- VM3 — Local Receiver (safe_logger): Minimal Python HTTP server to accept synthetic POSTs for demonstration only. Host-only networking.

<u>4. Step-by-step Lab Setup:</u>

4.1 Virtual machine provisioning:

1. Create 3 VMs (DVWA, Kali, Logger) on VMware.
2. Set each VM to use Host-only network adapter only (NAT) or use an internal network that only your host participates in.
3. Install guest OS and updates offline or via a controlled environment.
4. For DVWA: install Apache, PHP, MySQL, and DVWA (clone release), create database, set permissions, set security level to *low*.



DVWA Security

Script Security

Security Level is currently **low**.

You can set the security level to low, medium or high.

The security level changes the vulnerability level of DVWA.

low Submit

PHPIDS

PHPIDS v.0.6 (PHP-Intrusion Detection System) is a security layer for PHP based web applications.

You can enable PHPIDS across this site for the duration of your session.

PHPIDS is currently **disabled**. [enable PHPIDS]

5. For Kali: ensure browser and Burp Suite availability.
6. For Logger: place safe_logger.py . Run with python3 safe_logger.py on port 8080.

4.2 DVWA configuration notes

- Set DVWA/security/config.inc.php to low.
- Start Apache with verbose logging enabled (custom log format in Apache conf to include Referer and User-Agent).

---

5. POCs — Reflected, Stored, DOM:

5.1 Reflected XSS

Vulnerable page: DVWA vulnerabilities/xss_r (query parameter).

Benign payload (paste in query parameter):

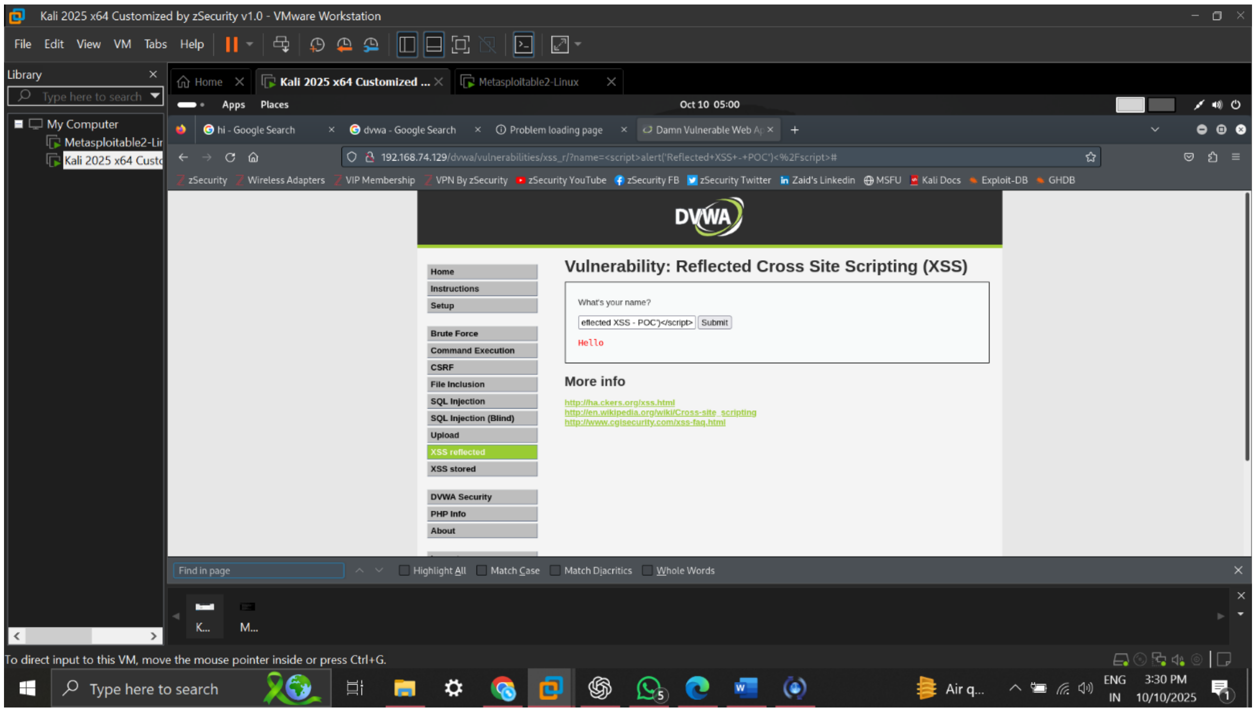
<script>alert('Reflected XSS - POC-UID-10001')</script>

Steps to demonstrate:

1. Log in to DVWA.

2. In attacker VM (or browser), navigate to vulnerable search page and submit the payload as the query parameter.
3. Observe the alert popup. Capture screenshot of alert and the URL showing the payload.
4. Record browser console (DevTools) extract and timestamp.

Evidences:



192.168.74.129/dvwa/vulnerabilities/xss_r/?name=<script>alert('Reflected+XSS+-+POC')<%2Fscript>#

Vulnerability: Reflected Cross Site Scripting (XSS)

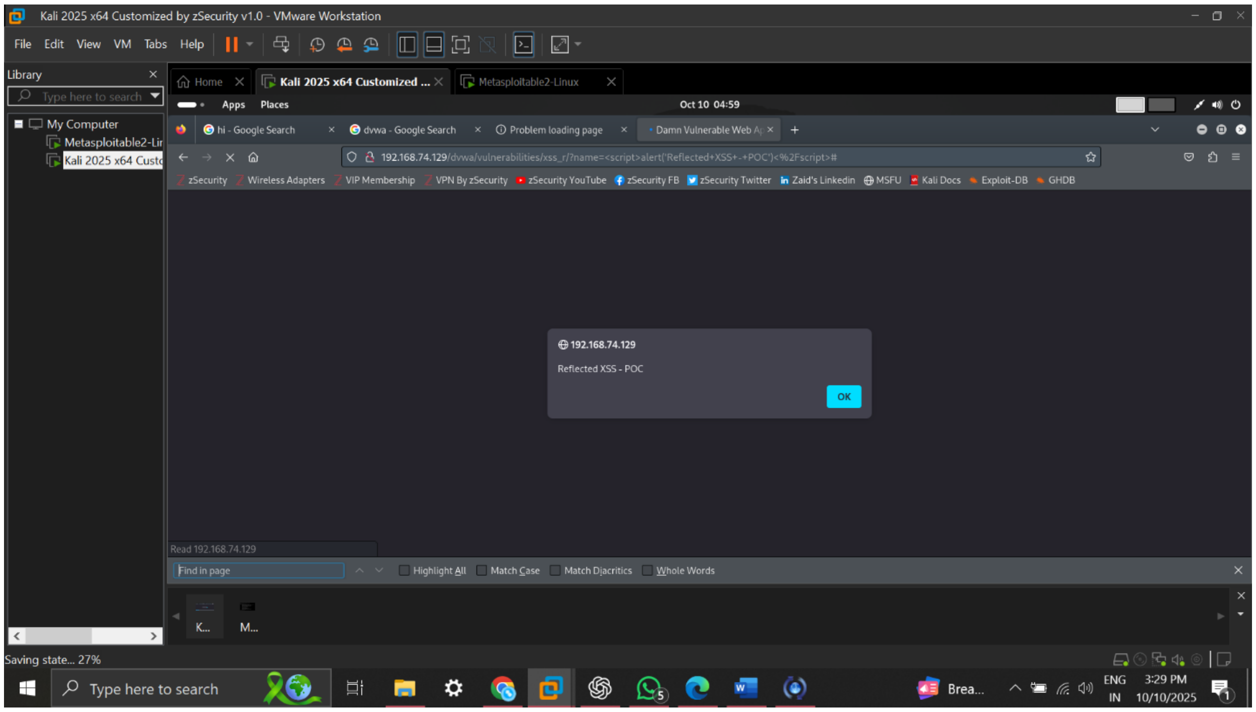
What's your name?

eflected XSS - POC')</script> Submit

Hello

More info

http://ha.ckers.org/xss.html
http://en.wikipedia.org/wiki/Cross-site_scripting
http://www.cgisecurity.com/xss-faq.html

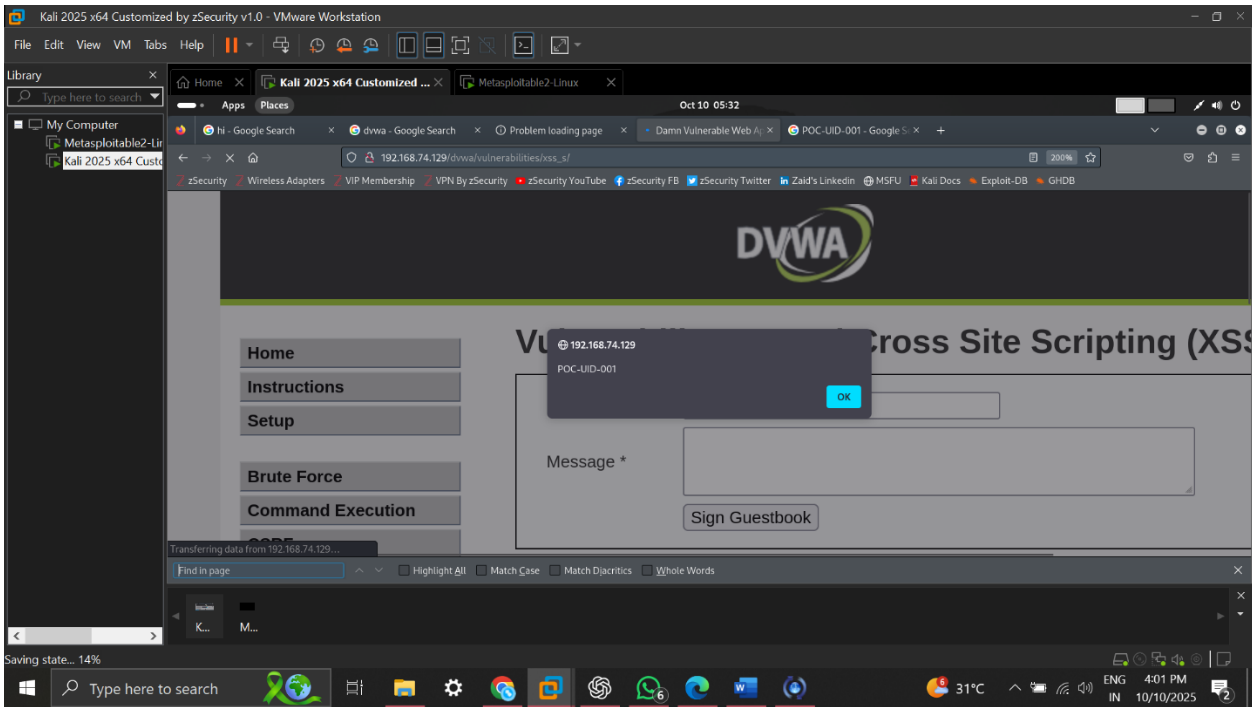


192.168.74.129/dvwa/vulnerabilities/xss_r/?name=<script>alert('Reflected+XSS+-+POC')<%2Fscript>#

192.168.74.129

Reflected XSS - POC

OK


Kali 2025 x64 Customized by zSecurity v1.0 - VMware Workstation
192.168.74.129/dvwa/vulnerabilities/xss_s/
DVWA
Home
Instructions
Setup
Brute Force
Command Execution
Vulnerability: Stored Cross Site Scripting (XSS)
Message *
Sign Guestbook
OK

192.168.74.129

POC-UID-001

192.168.56.101 - - [10/Oct/2025:12:34:56 +0530] "GET /dvwa/vulnerabilities/xss_r/?name=<script>alert('Reflected XSS - POC-UID-10001')</script> HTTP/1.1" 200 452 "-" "Mozilla/5.0 (...)"

5.2 Stored XSS

Vulnerable page: DVWA vulnerabilities/xss_s (message board).

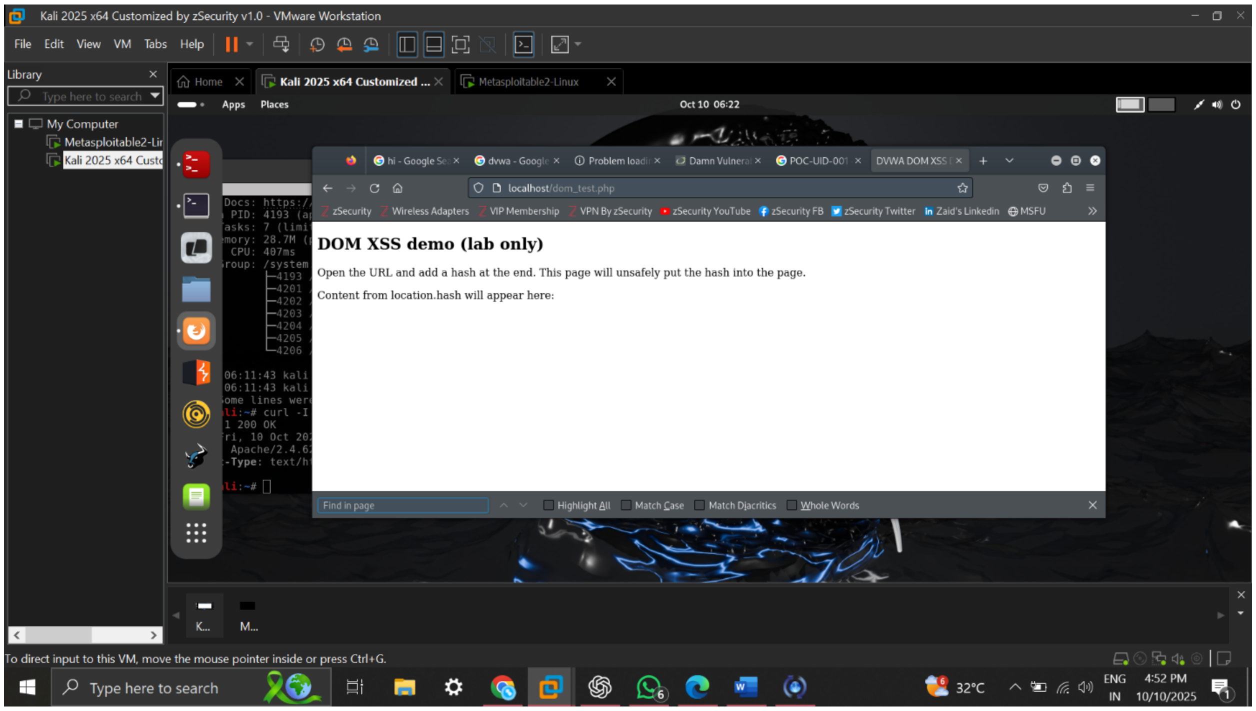
Benign payload:

<script>document.body.insertAdjacentHTML('afterbegin','<div id="poc">Stored XSS POC - POC-UID-20002</div>')</script>

Steps:

1. While logged in, post the above payload as a comment.
2. Visit the page that renders stored comments.
3. Observe DOM changed content (presence of the #poc div).

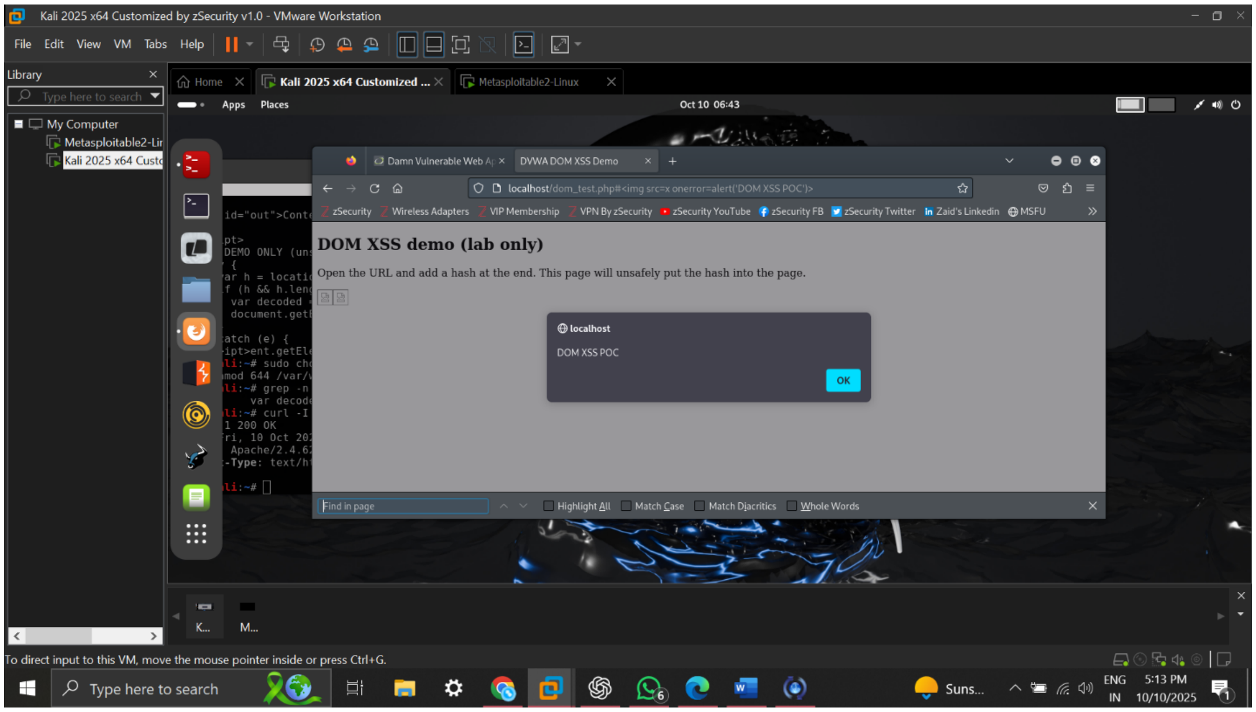
   Evidences:


```
localhost/dom_test.php
```

DOM XSS demo (lab only)

Open the URL and add a hash at the end. This page will unsafely put the hash into the page.

Content from location.hash will appear here:


```
localhost/dom_test.php#<img src=x onerror=alert('DOM XSS POC')>
```

DOM XSS demo (lab only)

Open the URL and add a hash at the end. This page will unsafely put the hash into the page.

localhost

DOM XSS POC

OK

5.3 DOM-based XSS:

Vulnerable page: DVWA page that writes location.hash or location.search into DOM unsafely (client-side JS).

Benign payload (put after # in URL):

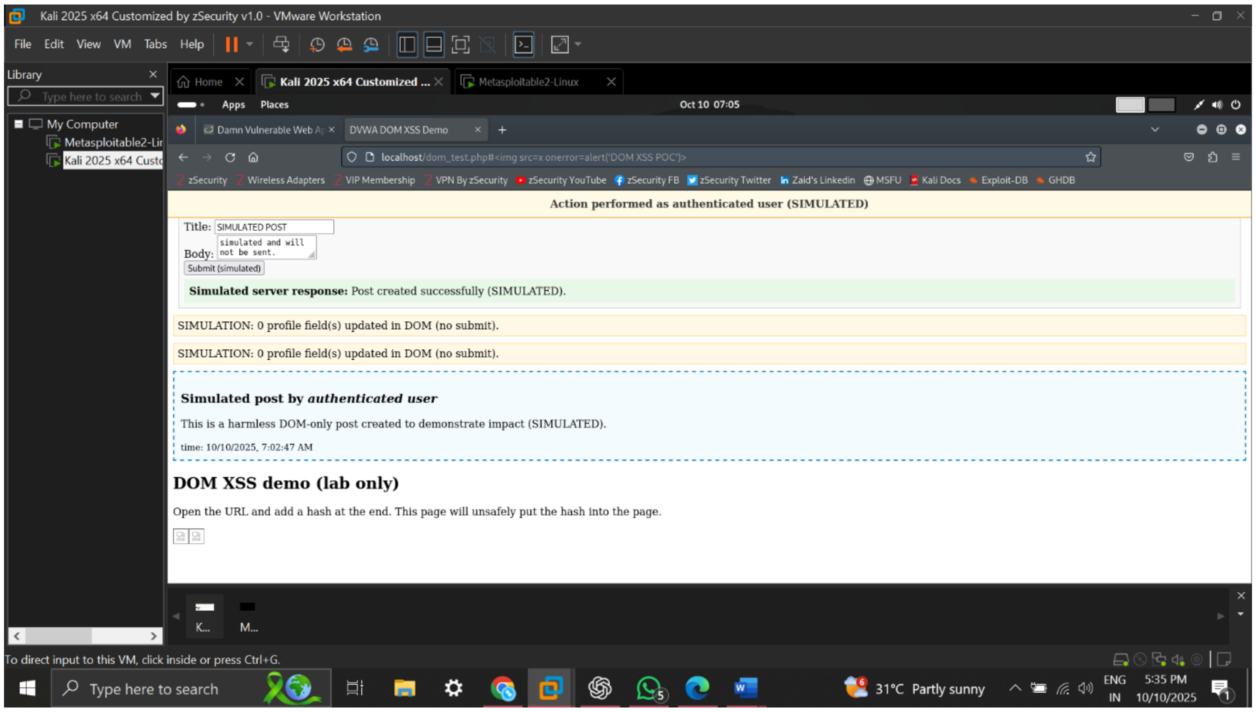
#<img onerror="alert('DOM XSS POC - POC-UID-30003')">

Steps:

1. Navigate to the vulnerable page and append the above hash to the URL.
2. Reload. Observe alert popup or DOM modification.
3. Screenshot alert and DevTools showing where the hash was used.

Evidence to include:

- **Screenshot of alert.**
- **Screenshot of DevTools showing the vulnerable code that echoes location.hash.**


Kali 2025 x64 Customized by zSecurity v1.0 - VMware Workstation
File Edit View VM Tabs Help
Library
My Computer
Metasploitable2-Linux
Kali 2025 x64 Customized
Home
Kali 2025 x64 Customized ...
Metasploitable2-Linux
Apps Places
Oct 10 07:05
Damn Vulnerable Web Ap
DVWA DOM XSS Demo
zSecurity Wireless Adapters VIP Membership VPN By zSecurity zSecurity YouTube zSecurity FB zSecurity Twitter Zaid's Linkedin MSFU Kali Docs Exploit-DB GHDB
To direct input to this VM, click inside or press Ctrl+G.
Type here to search
31°C Partly sunny
ENG IN 5:35 PM 10/10/2025

localhost/dom_test.php#<img src=x onerror=alert('DOM XSS POC')>

Action performed as authenticated user (SIMULATED)

Title: SIMULATED POST

Body: simulated and will not be sent.

Submit (simulated)

Simulated server response: Post created successfully (SIMULATED).

SIMULATION: 0 profile field(s) updated in DOM (no submit).

SIMULATION: 0 profile field(s) updated in DOM (no submit).

Simulated post by *authenticated user*

This is a harmless DOM-only post created to demonstrate impact (SIMULATED).

time: 10/10/2025, 7:02:47 AM

DOM XSS demo (lab only)

Open the URL and add a hash at the end. This page will unsafely put the hash into the page.

---

6. Simulated Escalation (safe) — Demonstrate impact without stealing secrets:

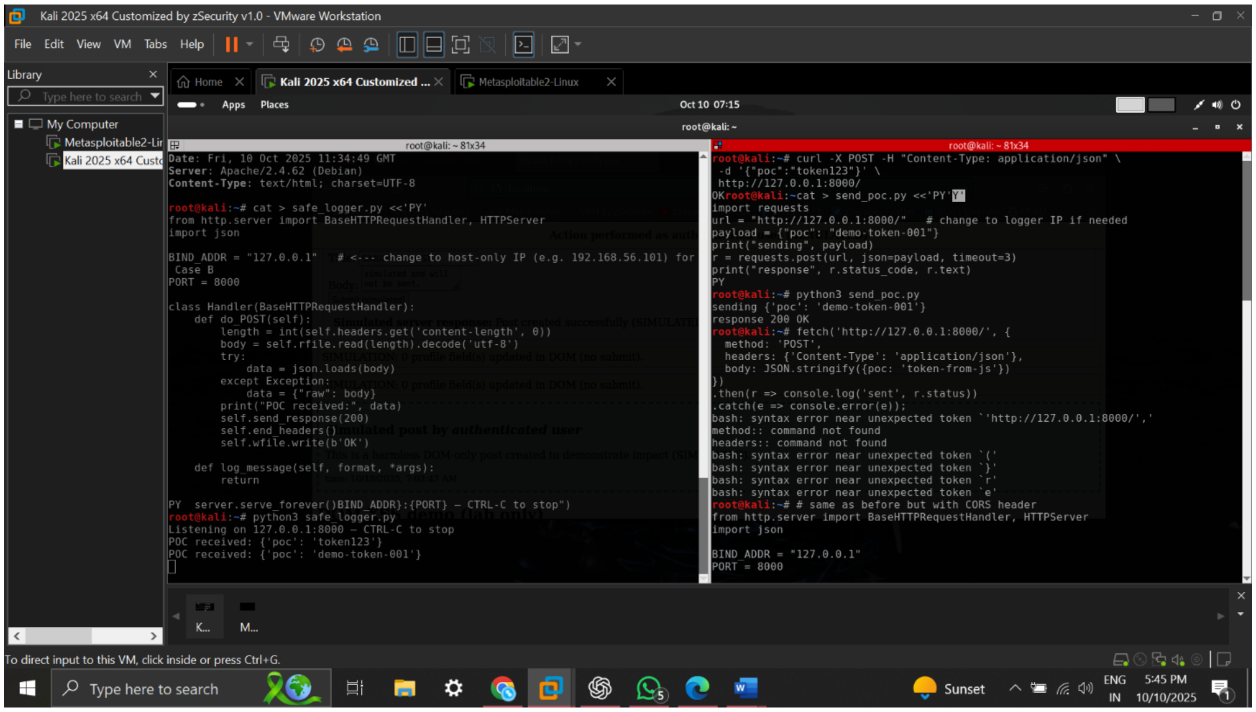
Goal: show that JS executing in the context of an authenticated user can perform actions using that user's session — but do not read or transmit real cookies. Instead, simulate actions by performing DOM changes or safe form submissions that do not reveal credentials.

Example simulated action (via injected stored XSS payload):

Simulation: use an XHR to submit a synthetic token to the local logger (VM3) — *only* send {"poc":"POC-UID-40005"} and never send cookies or credentials.

Explanation: In a real-world exploit an attacker could create requests that the browser will send with the victim's session cookies. In this lab, we do not capture or reuse real cookies; this section demonstrates the *mechanics* and possible impact in a safe, observable way.

Evidence:



```
Date: Fri, 10 Oct 2025 11:34:49 GMT
Server: Apache/2.4.62 (Debian)
Content-Type: text/html; charset=UTF-8

root@kali:~# cat > safe_logger.py <<'PY'
from http.server import BaseHTTPRequestHandler, HTTPServer
import json

BIND_ADDR = "127.0.0.1"  # <--- change to host-only IP (e.g. 192.168.56.101) for
 Case B
PORT = 8000

class Handler(BaseHTTPRequestHandler):
    def do_POST(self):
        length = int(self.headers.get('content-length', 0))
        body = self.rfile.read(length).decode('utf-8')
        try:
            data = json.loads(body)
        except Exception:
            data = {"raw": body}
        print("POC received:", data)
        self.send_response(200)
        self.end_headers()
        self.wfile.write(b'OK')

    def log_message(self, format, *args):
        return

PY  server.serve_forever()BIND_ADDR}:{PORT} — CTRL-C to stop")
root@kali:~# python3 safe_logger.py
Listening on 127.0.0.1:8000 — CTRL-C to stop
POC received: {'poc': 'token123'}
POC received: {'poc': 'demo-token-001'}
```

```
root@kali:~# curl -X POST -H "Content-Type: application/json" \
 -d '{"poc":"token123"}' \
 http://127.0.0.1:8000/
OKroot@kali:~cat > send_poc.py <<'PY'Y'
import requests
url = "http://127.0.0.1:8000/"   # change to logger IP if needed
payload = {"poc": "demo-token-001"}
print("sending", payload)
r = requests.post(url, json=payload, timeout=3)
print("response", r.status_code, r.text)
PY
root@kali:~# python3 send_poc.py
sending {'poc': 'demo-token-001'}
response 200 OK
root@kali:~# fetch('http://127.0.0.1:8000/', {
  method: 'POST',
  headers: {'Content-Type': 'application/json'},
  body: JSON.stringify({poc: 'token-from-js'})
})
.then(r => console.log('sent', r.status))
.catch(e => console.error(e));
bash: syntax error near unexpected token `'http://127.0.0.1:8000/','
method:: command not found
headers:: command not found
bash: syntax error near unexpected token `('
bash: syntax error near unexpected token `}'
bash: syntax error near unexpected token `r'
bash: syntax error near unexpected token `e'
root@kali:~# # same as before but with CORS header
from http.server import BaseHTTPRequestHandler, HTTPServer
import json

BIND_ADDR = "127.0.0.1"
PORT = 8000
```

---

7. Safe Local Receiver & Example Logs:

safe_logger.py

Run on VM3 and accept JSON POSTs to http://<logger-ip>:8000/.

Example safe POST from POC (only synthetic token):

POST / HTTP/1.1

Host: 192.168.56.103:8000

Content-Type: application/json

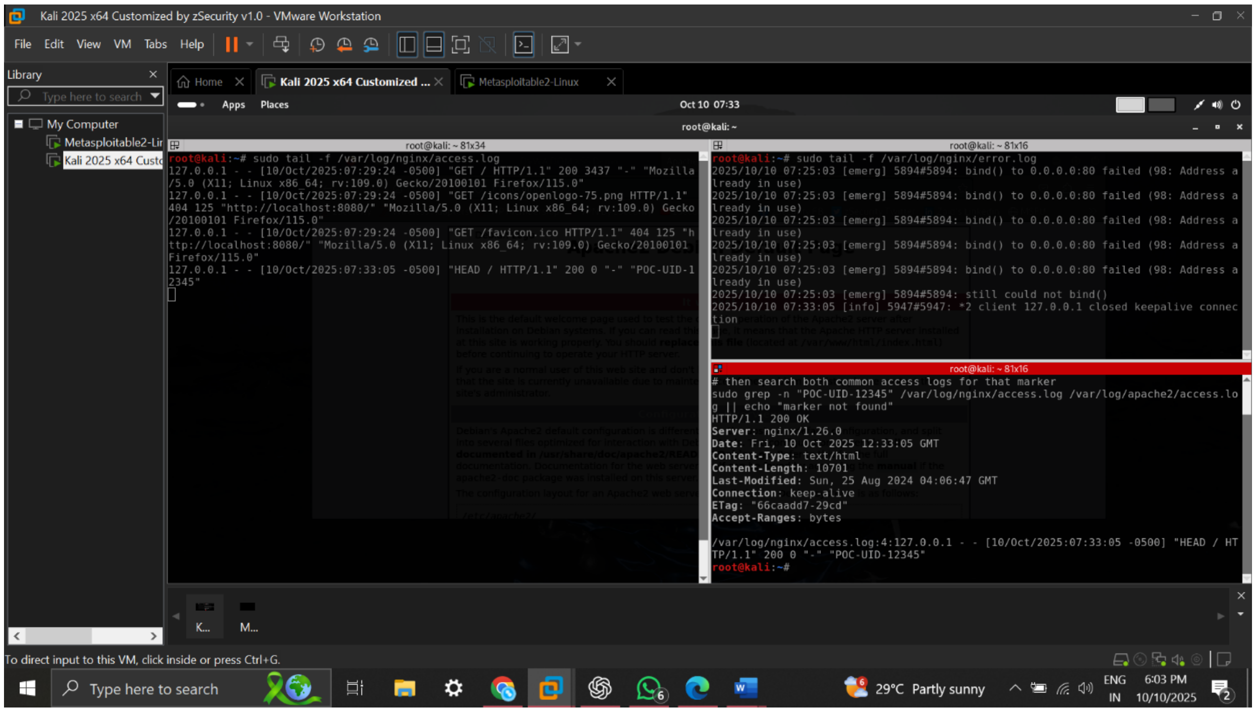
Content-Length: 24

{"poc":"POC-UID-40005"}

Logger console output example:

Listening on 8000

POC received: {'poc':'POC-UID-40005'}



```
root@kali:~# sudo tail -f /var/log/nginx/access.log
127.0.0.1 - - [10/Oct/2025:07:29:24 -0500] "GET / HTTP/1.1" 200 3437 "-" "Mozilla
/5.0 (X11; Linux x86_64; rv:109.0) Gecko/20100101 Firefox/115.0"
127.0.0.1 - - [10/Oct/2025:07:29:24 -0500] "GET /icons/openlogo-75.png HTTP/1.1"
404 125 "http://localhost:8080/" "Mozilla/5.0 (X11; Linux x86_64; rv:109.0) Gecko
/20100101 Firefox/115.0"
127.0.0.1 - - [10/Oct/2025:07:29:24 -0500] "GET /favicon.ico HTTP/1.1" 404 125 "h
ttp://localhost:8080/" "Mozilla/5.0 (X11; Linux x86_64; rv:109.0) Gecko/20100101
Firefox/115.0"
127.0.0.1 - - [10/Oct/2025:07:33:05 -0500] "HEAD / HTTP/1.1" 200 0 "-" "POC-UID-1
2345"
```

```
root@kali:~# sudo tail -f /var/log/nginx/error.log
2025/10/10 07:25:03 [emerg] 5894#5894: bind() to 0.0.0.0:80 failed (98: Address a
lready in use)
2025/10/10 07:25:03 [emerg] 5894#5894: bind() to 0.0.0.0:80 failed (98: Address a
lready in use)
2025/10/10 07:25:03 [emerg] 5894#5894: bind() to 0.0.0.0:80 failed (98: Address a
lready in use)
2025/10/10 07:25:03 [emerg] 5894#5894: bind() to 0.0.0.0:80 failed (98: Address a
lready in use)
2025/10/10 07:25:03 [emerg] 5894#5894: bind() to 0.0.0.0:80 failed (98: Address a
lready in use)
2025/10/10 07:25:03 [emerg] 5894#5894: still could not bind()
2025/10/10 07:33:05 [info] 5947#5947: *2 client 127.0.0.1 closed keepalive connec
tion
```

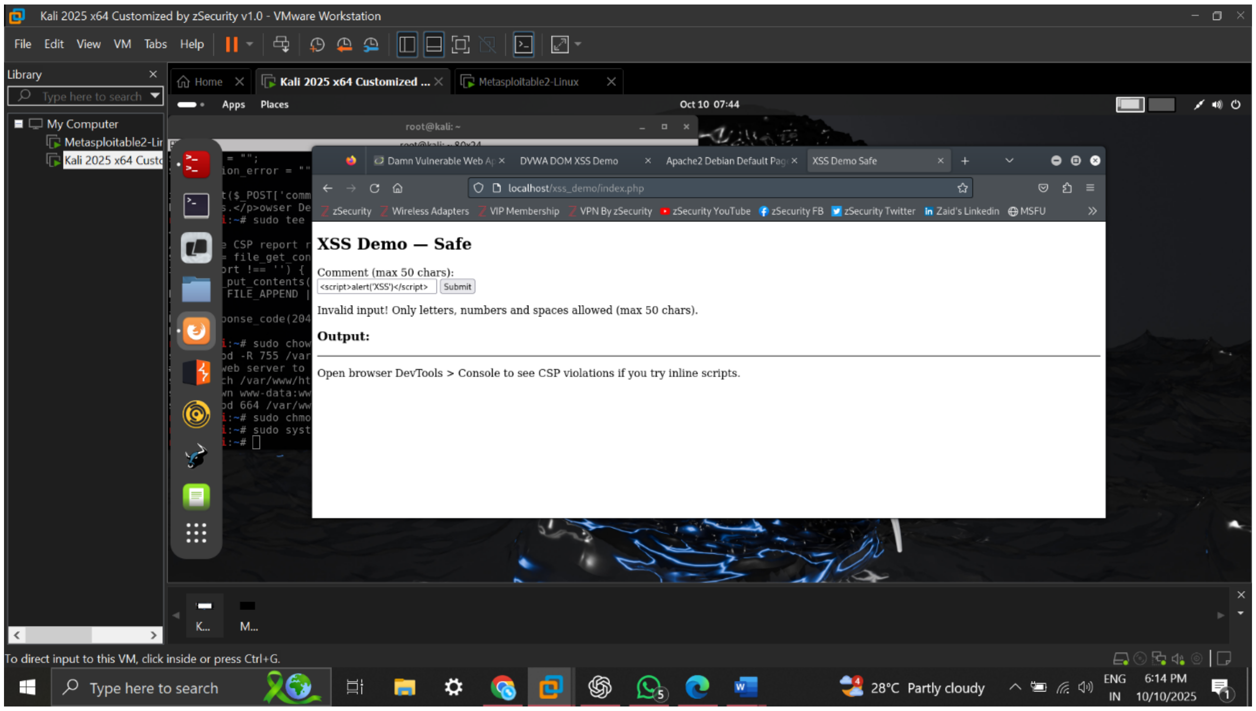
```
# then search both common access logs for that marker
sudo grep -n "POC-UID-12345" /var/log/nginx/access.log /var/log/apache2/access.lo
g || echo "marker not found"
HTTP/1.1 200 OK
Server: nginx/1.26.0
Date: Fri, 10 Oct 2025 12:33:05 GMT
Content-Type: text/html
Content-Length: 10701
Last-Modified: Sun, 25 Aug 2024 04:06:47 GMT
Connection: keep-alive
ETag: "66caadd7-29cd"
Accept-Ranges: bytes

/var/log/nginx/access.log:4:127.0.0.1 - - [10/Oct/2025:07:33:05 -0500] "HEAD / HT
TP/1.1" 200 0 "-" "POC-UID-12345"
root@kali:~#
```

<u>8. Detection & Forensics:</u>

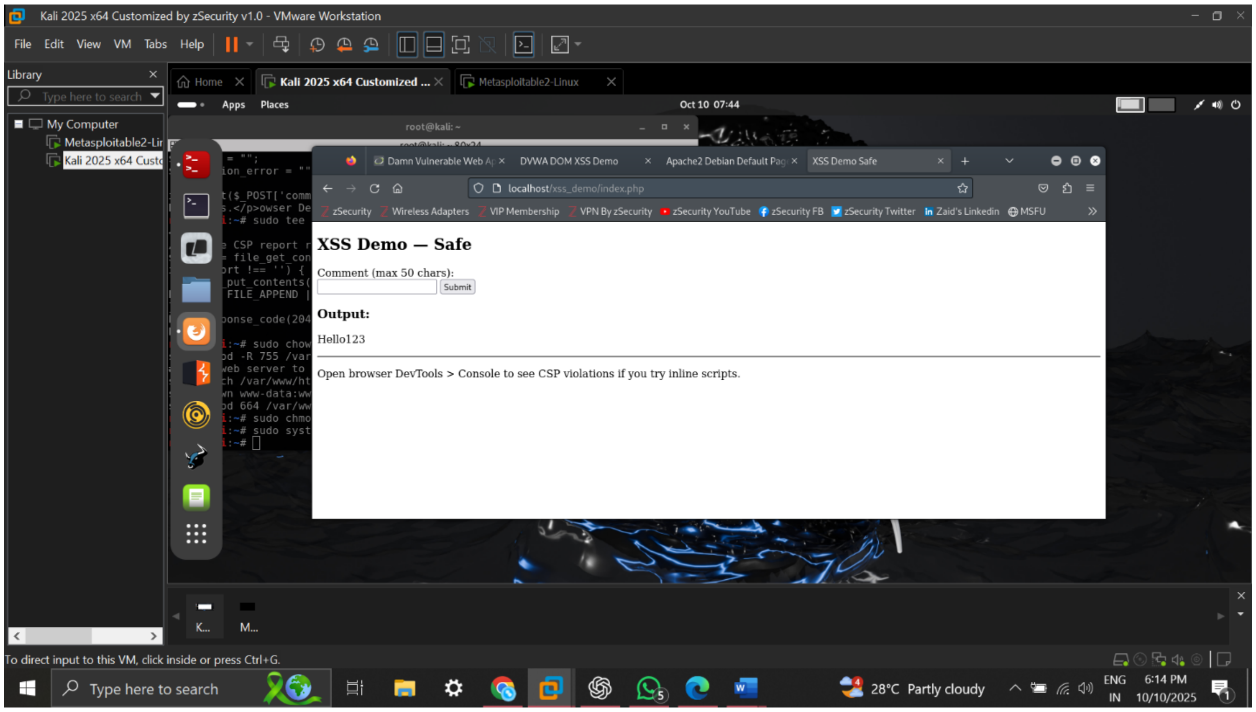
Logging approach:

- Enable Apache access logs with extended fields: client IP, timestamp, request-line, referer, user-agent.
- Enable application logs (DVWA PHP warnings/errors).
- Keep periodic browser DevTools HAR captures for each demonstration.
- Use unique POC markers (e.g., POC-UID-XXXXX) embedded in payloads for searchability.

Evidence correlation example:


Kali 2025 x64 Customized by zSecurity v1.0 - VMware Workstation
localhost/xss_demo/index.php
XSS Demo Safe

```
XSS Demo — Safe
Comment (max 50 chars): <script>alert('XSS')</script> Submit
Invalid input! Only letters, numbers and spaces allowed (max 50 chars).
Output:
Open browser DevTools > Console to see CSP violations if you try inline scripts.
```


Kali 2025 x64 Customized by zSecurity v1.0 - VMware Workstation
localhost/xss_demo/index.php
XSS Demo Safe

```
XSS Demo — Safe
Comment (max 50 chars): Submit
Output:
Hello123
Open browser DevTools > Console to see CSP violations if you try inline scripts.
```

9. Mitigations — Implementation & Validation (15 pts)

Each mitigation below includes: code/config snippet, deployment location.

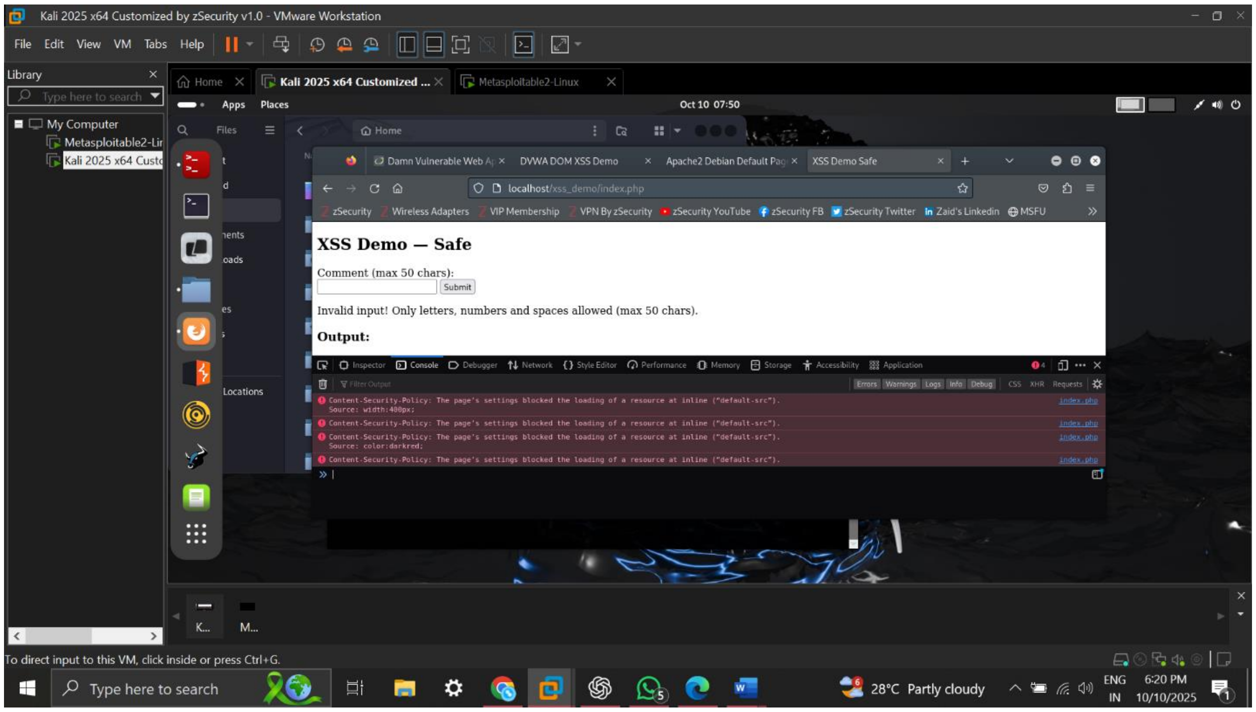
9.1 Output encoding / escaping (PHP)

What to change: Escape user-provided content at output.
Example PHP change:

```
// Before (vulnerable)

echo $user_input;


// After (safe)

echo htmlspecialchars($user_input, ENT_QUOTES | ENT_HTML5, 'UTF-8');
```

Validation:

After: payload shows escaped text &lt;script&gt;...&lt;/script&gt; and no JS execution.


Kali 2025 x64 Customized by zSecurity v1.0 - VMware Workstation
Damn Vulnerable Web Ap
DVWA DOM XSS Demo
Apache2 Debian Default Pag
XSS Demo Safe
localhost/xss_demo/index.php
Submit

XSS Demo — Safe

Comment (max 50 chars):

Invalid input! Only letters, numbers and spaces allowed (max 50 chars).

Output:

```
Content-Security-Policy: The page's settings blocked the loading of a resource at inline ("default-src").   index.php
Source: width:400px;
Content-Security-Policy: The page's settings blocked the loading of a resource at inline ("default-src").   index.php
Content-Security-Policy: The page's settings blocked the loading of a resource at inline ("default-src").   index.php
Source: color:darkred;
Content-Security-Policy: The page's settings blocked the loading of a resource at inline ("default-src").   index.php
```

9.2 Input validation (defense-in-depth):

- Constrain length, character set, and reject inputs with suspicious sequences (e.g., <script).
- Validation alone is not sufficient — use with output encoding.

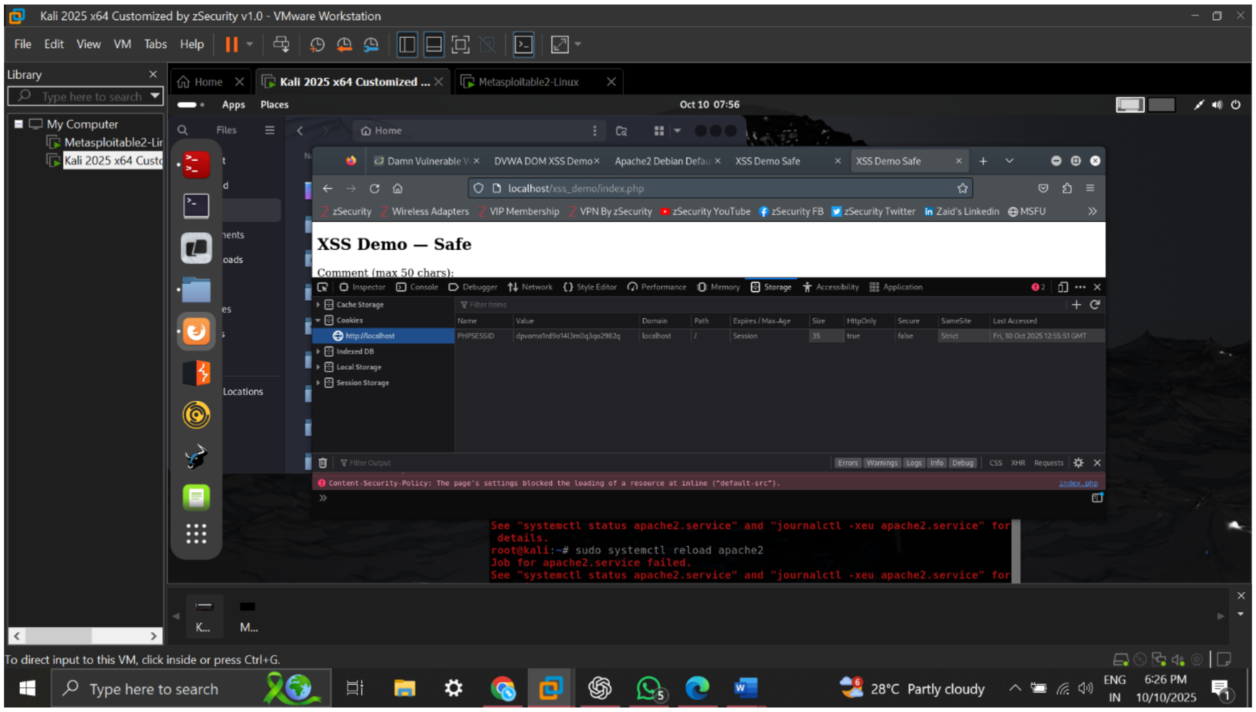
9.3 Content Security Policy (CSP)

Example header (strict):

Content-Security-Policy: default-src 'self'; script-src 'self'; object-src 'none'; base-uri 'self'; frame-ancestors 'none';

Deployment: Add to Apache virtualhost or via application headers.
Validation:

- After CSP applied, inline injected scripts should be blocked. Console shows Refused to execute inline script and CSP violation report if reporting endpoint configured.



Name	Value	Domain	Path	Expires / Max-Age	Size	HttpOnly	Secure	SameSite	Last Accessed
PHPSESSID	dpvomo1rd9o14l3m0q3qo2982q	localhost	/	Session	35	true	false	Strict	Fri, 10 Oct 2025 12:55:51 GMT

```
Content-Security-Policy: The page's settings blocked the loading of a resource at inline ("default-src").
```

```
See "systemctl status apache2.service" and "journalctl -xeu apache2.service" for
 details.
root@kali:~# sudo systemctl reload apache2
Job for apache2.service failed.
See "systemctl status apache2.service" and "journalctl -xeu apache2.service" for
```

9.4 Cookie hardening:

Set cookie flags: HttpOnly; Secure; SameSite=Strict for session cookies.
Example (PHP):

session_set_cookie_params(['httponly' => true, 'secure' => true, 'samesite' => 'Strict']);

session_start();

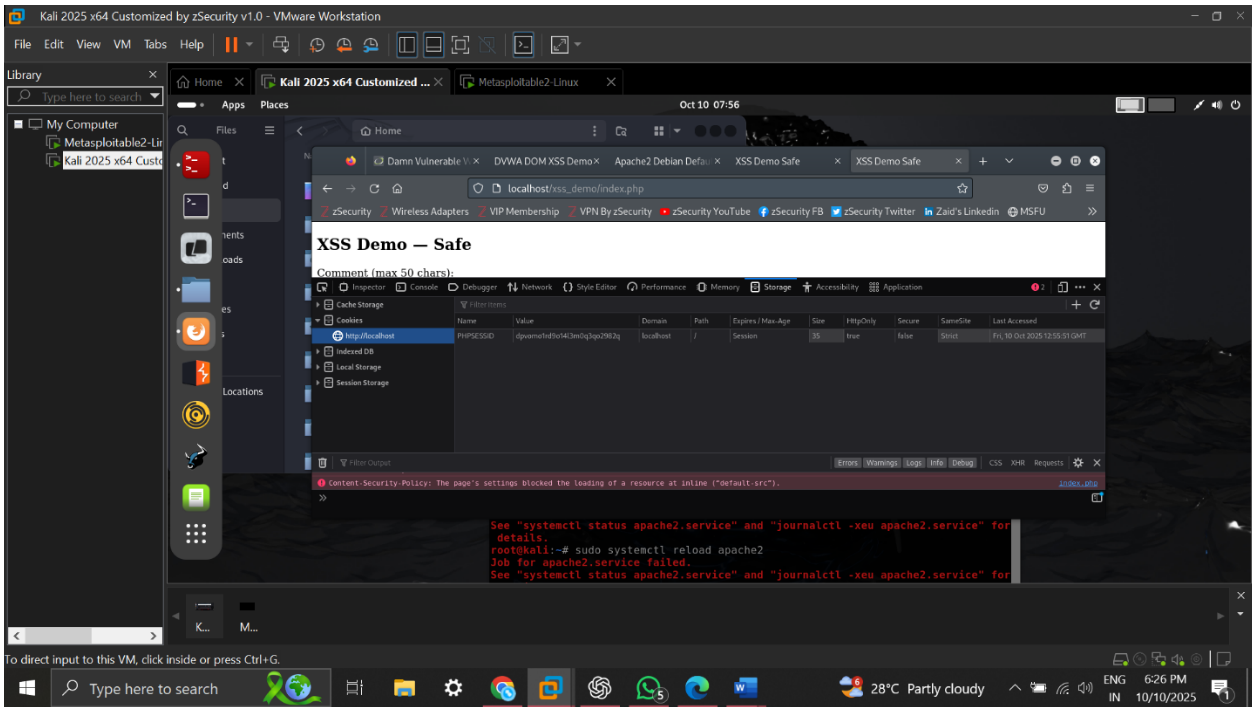
Validation:

- document.cookie in console does not show session cookie.

- Explain: HttpOnly prevents JS reads; Secure requires TLS (in a lab, note limitation) and SameSite reduces CSRF risk.

9.5 CSP Reporting & Monitoring:

- Optionally configure report-to or report-uri to send violation reports to VM3 for centralized monitoring.
- Example: Content-Security-Policy: ...; report-uri http://192.168.56.103:8000/report (configure server to accept reports).


Kali 2025 x64 Customized by zSecurity v1.0 - VMware Workstation
File Edit View VM Tabs Help
Library
My Computer
Metasploitable2-Linux
Kali 2025 x64 Customized
Home
Kali 2025 x64 Customized ...
Metasploitable2-Linux
Apps Places
Oct 10 07:56
localhost/xss_demo/index.php
XSS Demo — Safe
Comment (max 50 chars):
Inspector Console Debugger Network Style Editor Performance Memory Storage Accessibility Application
Cache Storage
Cookies
http://localhost
Indexed DB
Local Storage
Session Storage
To direct input to this VM, click inside or press Ctrl+G.
Type here to search
28°C Partly cloudy
ENG IN
6:26 PM 10/10/2025

Name	Value	Domain	Path	Expires / Max-Age	Size	HttpOnly	Secure	SameSite	Last Accessed
PHPSESSID	dpvomo1rd9o14l3m0q3qo2982q	localhost	/	Session	35	true	false	Strict	Fri, 10 Oct 2025 12:55:51 GMT

```
Content-Security-Policy: The page's settings blocked the loading of a resource at inline ("default-src").   index.php
```

```
See "systemctl status apache2.service" and "journalctl -xeu apache2.service" for
 details.
root@kali:~# sudo systemctl reload apache2
Job for apache2.service failed.
See "systemctl status apache2.service" and "journalctl -xeu apache2.service" for
```

---

10. Conclusion:

This project demonstrated, in a safe and fully isolated lab, how reflected, stored, and DOM-based XSS work, how they can escalate from a nuisance (alerts) to meaningful impact (actions performed in an authenticated user context), and how defenders can detect, investigate, and mitigate them. Using DVWA inside host-only VMs, benign proof-of-concept payloads and synthetic tokens, we reproduced realistic attack mechanics while strictly avoiding any real-data exfiltration. We also produced reproducible forensic artifacts (browser captures, server logs, DB rows, and logger receipts) that show how incidents can be reconstructed and attributed.

Key findings

- **Different attack surfaces:** Reflected XSS appears in request/response reflection, stored XSS persists in server-side data, and DOM XSS arises entirely in client-side scripts. Each requires different discovery and mitigation approaches.
- **Impact without theft:** JavaScript executing in a victim's browser can perform actions as that user (CSRF-like effects) even without cookie theft; simulating those actions safely demonstrates real-world risk.
- **Layered mitigation works best:** Output encoding, strict CSP, input constraints, cookie hardening (HttpOnly/Secure/SameSite), and safe templating together greatly reduce exploitability. No single control is sufficient on its own.
- **Forensics is practical:** Embedding unique POC markers, keeping verbose server logs, and capturing browser network/DevTools evidence provides a clear timeline for triage and root-cause analysis.
- **Testing discipline is essential:** Host-only networks, VM snapshots, and synthetic tokens keep research ethical and legally safe.

Practical recommendations

1. **Adopt output encoding by default** (e.g., htmlspecialchars or framework auto-escaping) for all user-supplied content.
2. **Deploy a strict CSP** that disallows inline scripts and only allows trusted script sources; enable reporting to a monitoring endpoint.
3. **Set secure cookie attributes** (HttpOnly, Secure, SameSite) for session cookies and require TLS in production.
4. **Instrument logging and monitoring** to capture payload strings, timestamps, and referers; correlate client-side captures with server logs during investigations.
5. **Automate regression tests** that inject benign payloads in CI to prevent reintroduction of XSS via code changes.
6. **Train developers** on secure output handling and threat modeling so XSS prevents are baked into the development lifecycle.

Final statement

When responsibly executed, hands-on labs like this provide the most effective way to understand both attacker capabilities and defender controls. This project not only taught how XSS works, but produced repeatable methods for demonstrating impact without harm, collecting forensic evidence, and validating layered mitigations — all of which are directly transferable to secure development and incident response practices.