Web Application Firewall Rule Development

Tool Used: ModSecurity (OWASP CRS)
Authors: Aishwarya J · Vengadesh P · Kaustubh Gadikar · Pragyan Acharya
OS: Ubuntu 20.04 / 22.04
Web Server: Apache2
Vulnerable Application: DVWA (Damn Vulnerable Web Application)
Protected Web Server: Ubuntu + DVWA
WAF (ModSecurity): Ubuntu + Apache + ModSecurity
Network Mode: Host-Only (for safe attack simulation)

Table of Contents

Abstract

Web Application Firewalls (WAF) are critical security mechanisms that protect web applications from malicious traffic and known attack signatures. This report documents the comprehensive WAF Rule Development Lab project, focusing on the design, implementation, and testing of custom WAF rules to defend against common web application attacks including SQL Injection, Cross-Site Scripting (XSS), Remote File Inclusion (RFI), and Distributed Denial-of-Service (DDoS) attacks. The project implements both predefined and custom rules across multiple threat categories, evaluates their effectiveness through controlled testing, and provides recommendations for rule tuning and maintenance. Experimental results demonstrate the effectiveness of the implemented ruleset in blocking malicious requests while maintaining acceptable false positive rates.

1. Introduction

Web applications face constant threats from attackers attempting to exploit vulnerabilities through the HTTP/HTTPS protocol. Traditional network firewalls operate at lower layers and cannot understand application-level attacks. Web Application Firewalls bridge this gap by inspecting HTTP/HTTPS traffic and implementing granular security policies[1].

1.1 Background

The Web Application Firewall operates by analyzing incoming traffic against predefined and custom rules. Each rule consists of three core components: inspection criteria, matching conditions, and actions (allow, block, log, count)[2]. Modern WAF solutions employ multiple detection mechanisms including signature-based detection, anomaly-based detection, and behavioral analysis[1].

1.2 Motivation

Organizations deploy WAFs to:

- Protect against OWASP Top 10 vulnerabilities
- Reduce false positives in security operations
- Comply with regulatory requirements (PCI-DSS, HIPAA)
- Maintain application availability during attacks

1.3 Objectives

The objective of this lab is to **design, develop, and implement custom Web Application Firewall (WAF) rules using ModSecurity**, and to test these rules against various web application attacks simulated from Kali Linux. The WAF should detect and block:

- SQL Injection attacks
- Cross-Site Scripting (XSS)
- Local File Inclusion (LFI) / Remote File Inclusion (RFI)
- Directory Traversal
- Command Injection
- File Upload vulnerabilities
- HTTP Flood / Brute-force attacks

Custom rules will be created in local.rules configuration, triggered through attacks, and validated in audit logs.

2. Methodology

2.1 Experimental Environment

The WAF Rule Development Lab employs the following infrastructure:

Component	Specifications
WAF Platform	ModSecurity / OWASP CRS 3.3.2
Test Environment	Ubuntu 20.04 LTS VM
Web Server	Apache 2.4 with mod_security module
Attack Generation Tools	OWASP ZAP, custom Python scripts
Rule Language	ModSecurity Rule Language (MSRL)

Table 1: WAF Lab Environment Configuration

2.2 Rule Development Phases

The project follows a structured approach across four phases:

Phase 1: Analysis and Planning

Phase 2: Rule Implementation

Phase 3: Testing and Validation

Phase 4: Tuning and Optimization

2.3 Threat Categories Addressed

The WAF ruleset covers the following threat categories:

1. **SQL Injection (SQLi)** - Attempts to manipulate database queries through user input
2. **Cross-Site Scripting (XSS)** - Malicious scripts injected into web pages
3. **Remote File Inclusion (RFI)** - Exploitation of file handling mechanisms
4. **Local File Inclusion (LFI)** - Access to local system files through web application
5. **Cross-Site Request Forgery (CSRF)** - Unauthorized state-changing requests
6. **Distributed Denial-of-Service (DDoS)** - Rate-based and volumetric attacks
7. **Protocol Violations** - Malformed HTTP requests and suspicious headers
8. **Scanner Detection** - Identification of automated vulnerability scanning tools

3. Why DVWA ?

DVWA (Damn Vulnerable Web Application) is a deliberately vulnerable PHP/MySQL web application designed for security training and penetration testing practice. Key features:

- **Multiple vulnerabilities:** SQL Injection, XSS, CSRF, LFI/RFI, Command Injection, File Upload, etc.
- **Adjustable security levels:** Low, Medium, High, Impossible – allowing progressive testing complexity
- **Open-source and easy to deploy:** Simple PHP application with minimal dependencies
- **Perfect for WAF testing:** Attacks are guaranteed to trigger, allowing accurate rule validation
- **Community-supported:** Extensive documentation and active GitHub repository

3. Lab Setup

3.1 VM Configuration

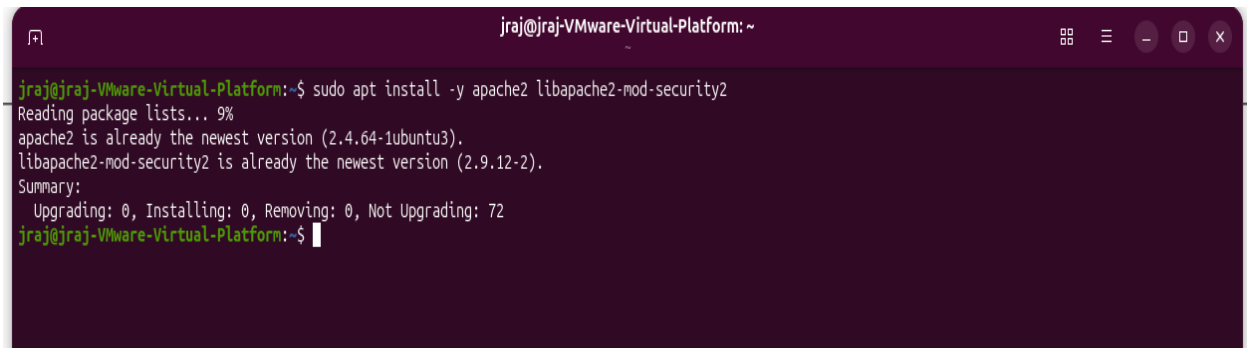
Machine	IP Address	Purpose	Tools/Services	Notes
Ubuntu + ModSecurity	192.168.78.138	WAF Protection	Apache2, ModSecurity, OWASP CRS	Reverse proxy with WAF rules
Kali Linux	192.168.1.8	Attacker	SQLMap	Launches web application attacks

4. Installation Steps

4.1 Install Apache2 & ModSecurity:

sudo apt update

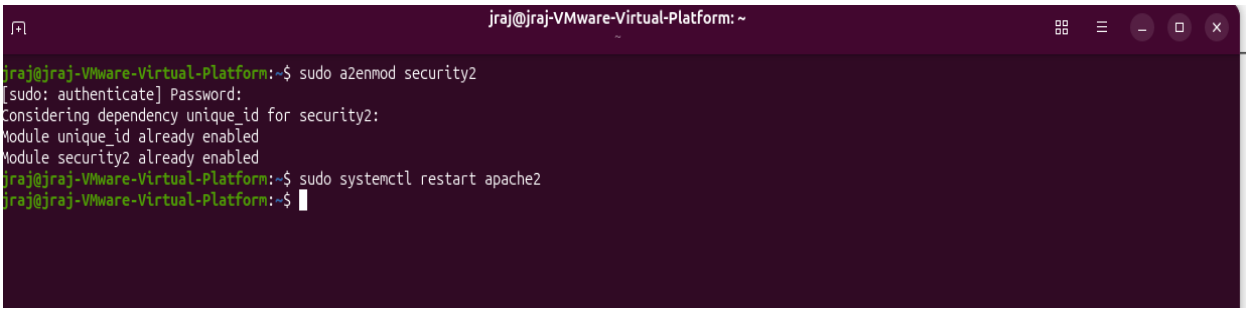
sudo apt install -y apache2 libapache2-mod-security2



4.2. Enable ModSecurity:

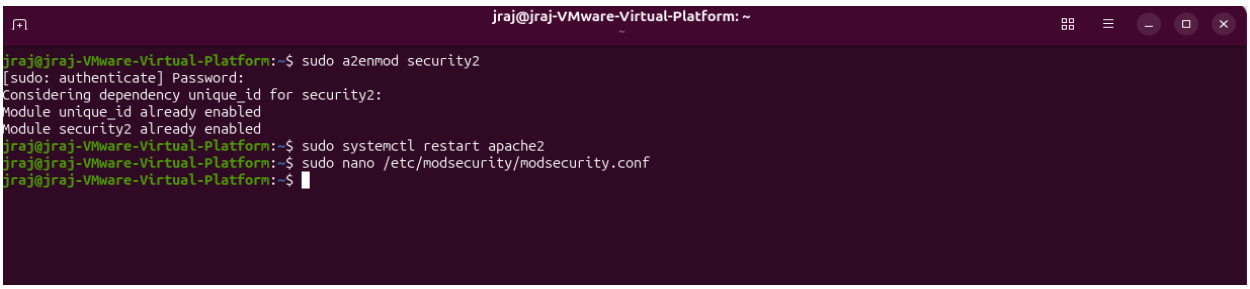
sudo a2enmod security2

sudo systemctl restart apache2

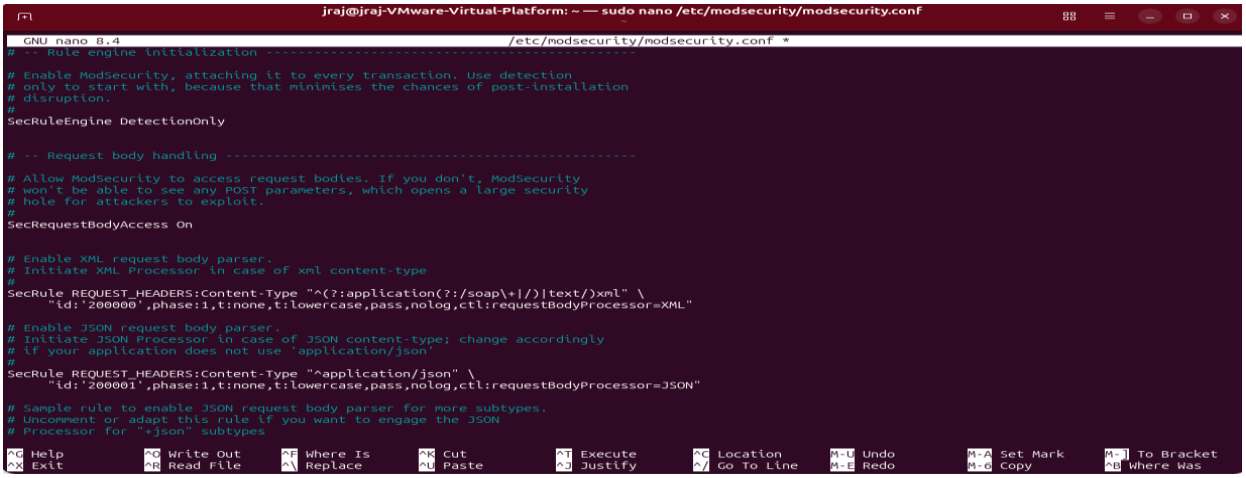


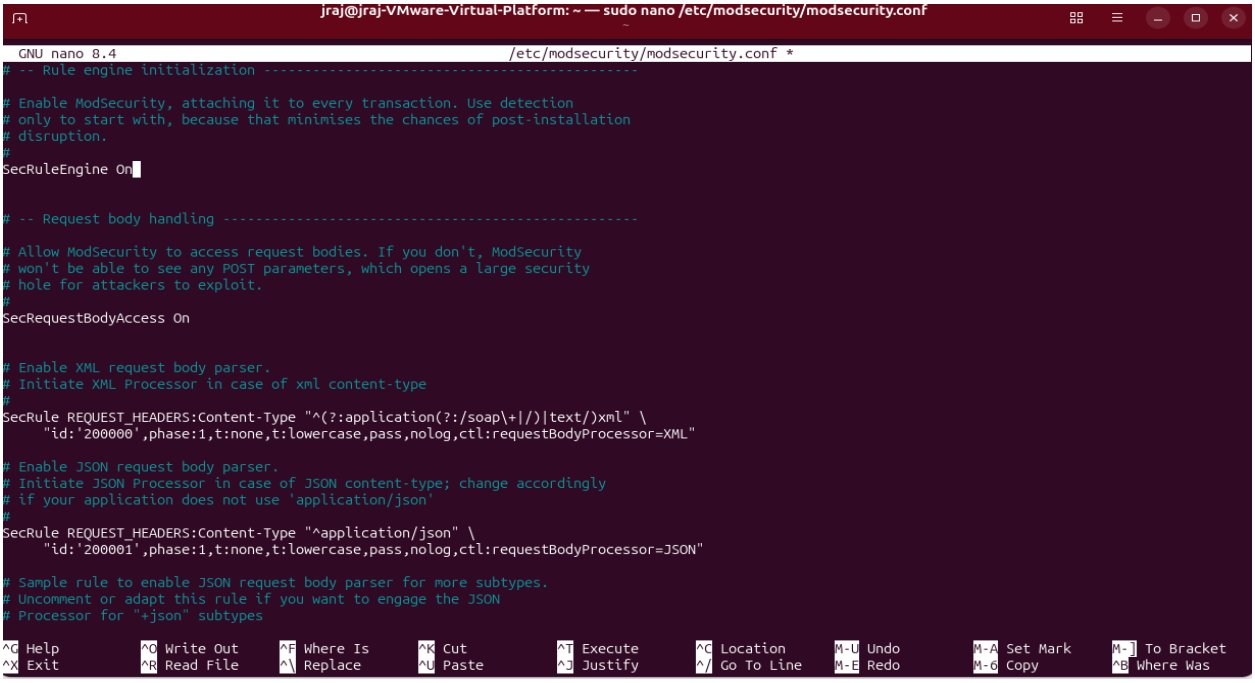
4.3. Configure ModSecurity:

sudo nano /etc/modsecurity/modsecurity.conf



SecRuleEngine DetectionOnly -> SecRuleEngine On

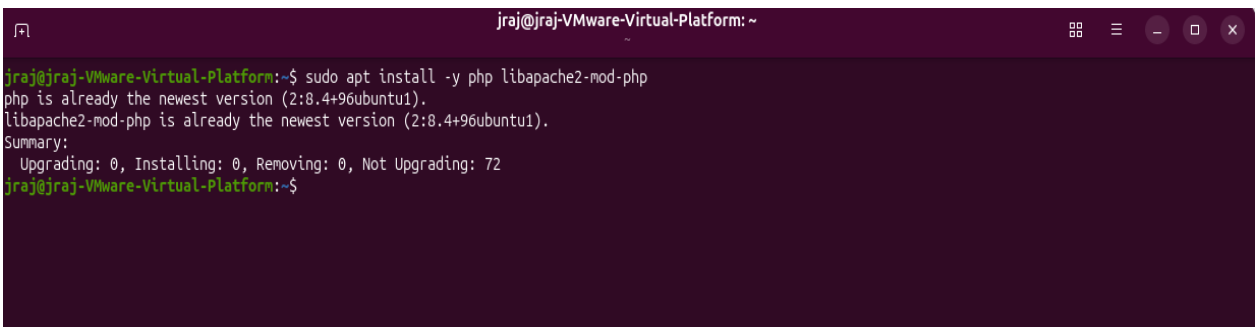




5.Deploy a Vulnerable Web App

5.1 Install PHP:

sudo apt install -y php libapache2-mod-php

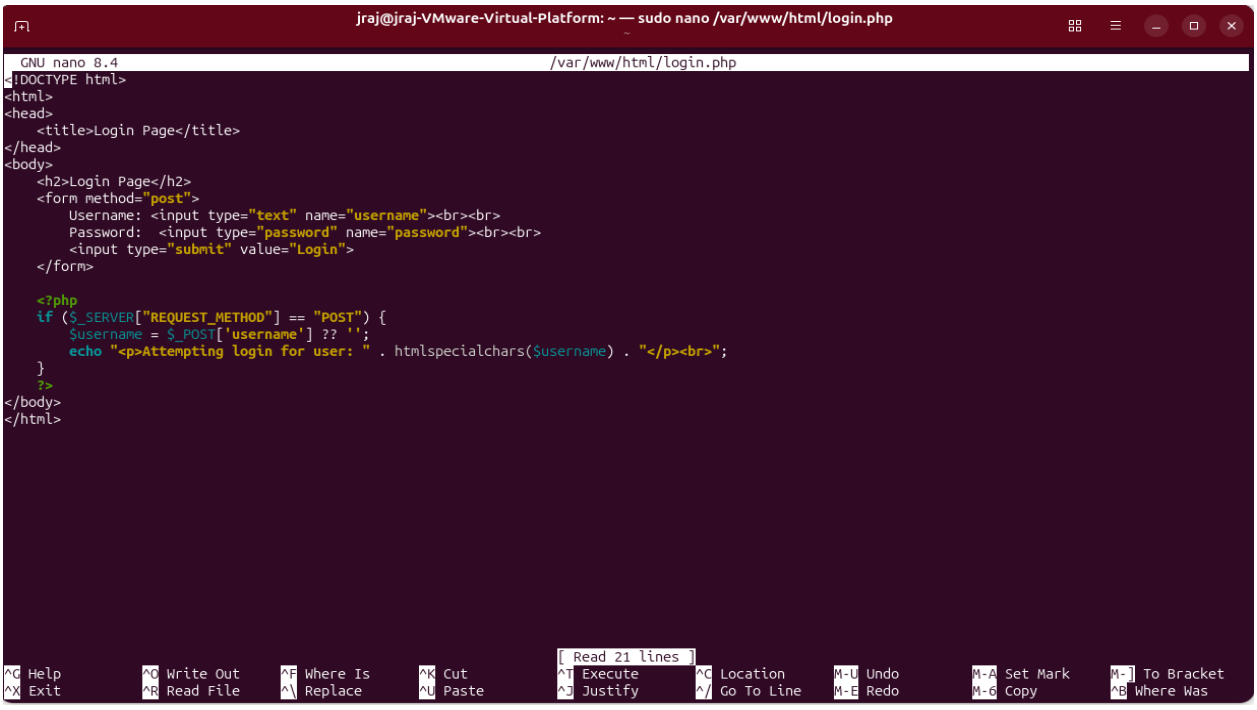


5.2 Create a Vulnerable Login Page:

sudo nano /var/www/html/login.php

Login page code:

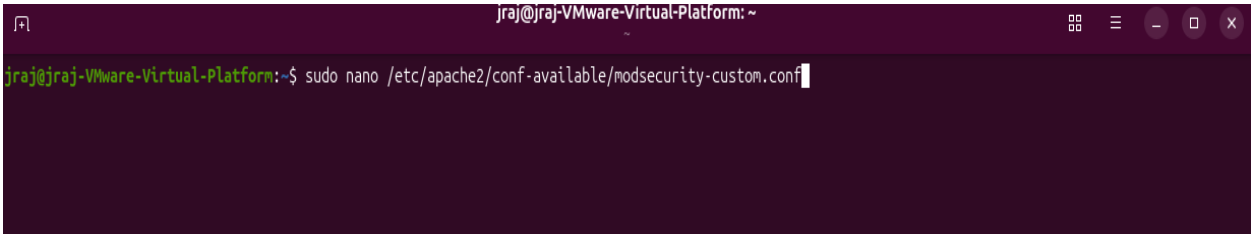
```
<html><body>
<h2>Login Page</h2>
<form action="login.php" method="POST">
Username: <input type="text" name="username"><br>
Password: <input type="password" name="password"><br>
<input type="submit" value="Login">
</form>
<?php
if ($_SERVER["REQUEST_METHOD"] == "POST") {
$username = $_POST['username'];
// In a real (bad) app, this would be part of a database query
echo "<p>Attempting login for user: " . htmlspecialchars($username) . "</p><br?";
}
?>
</body></html>
```



6. WAF Rule Development & Configuration

6.1Create a Custom Rules File:

sudo nano /etc/apache2/conf-available/modsecurity-custom.conf

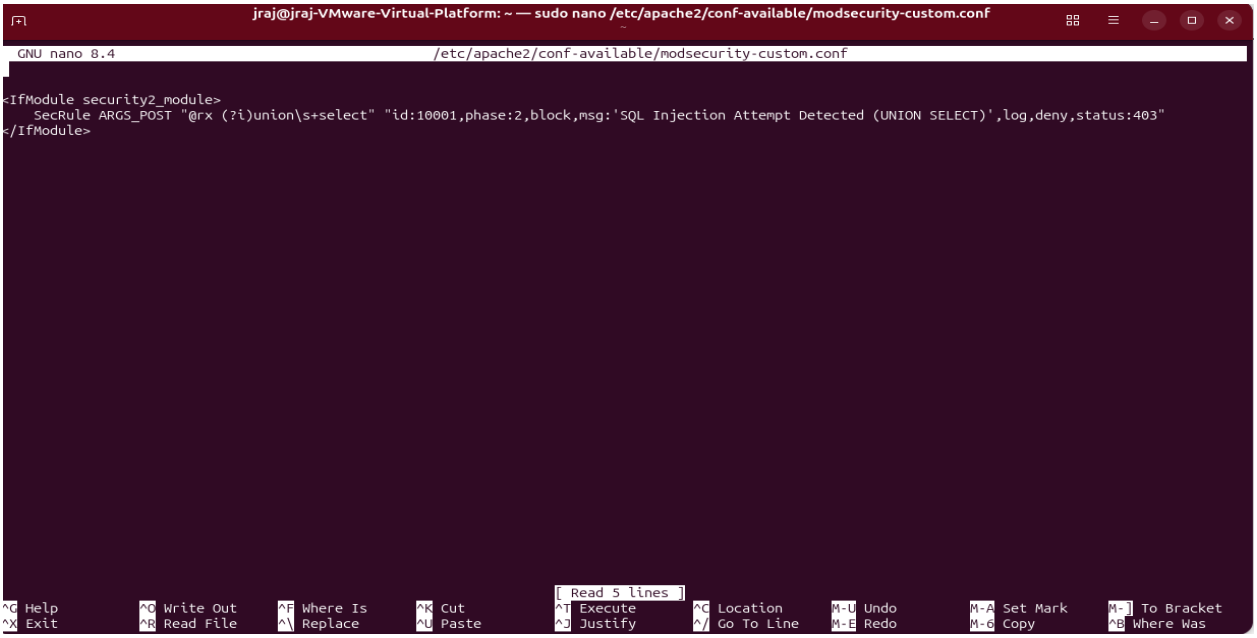


6.2 Add an SQLi Rule:

<IfModule security2_module>

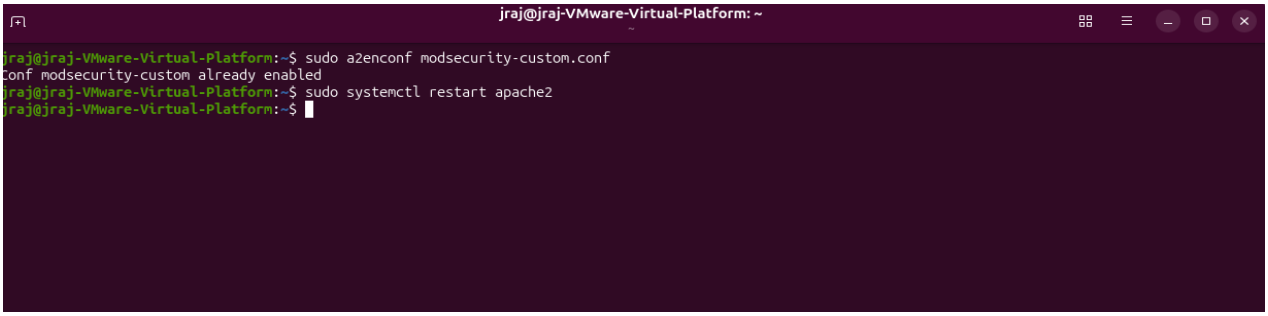
SecRule ARGS_POST "@rx (?i)union\s+select" "id:10001,phase:2,block,msg:'SQL Injection Attempt Detected (UNION SELECT)'"

</IfModule>



3. Enable the Custom Config:

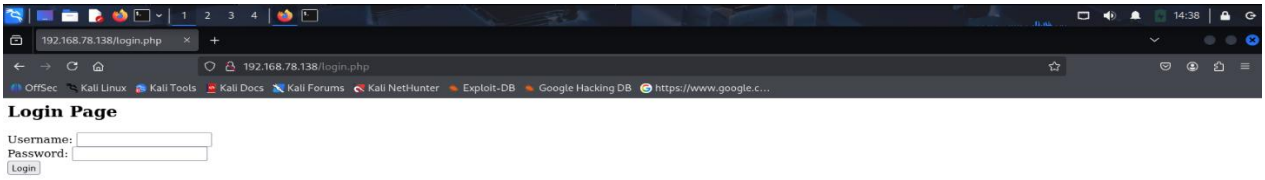
sudo systemctl restart apache2



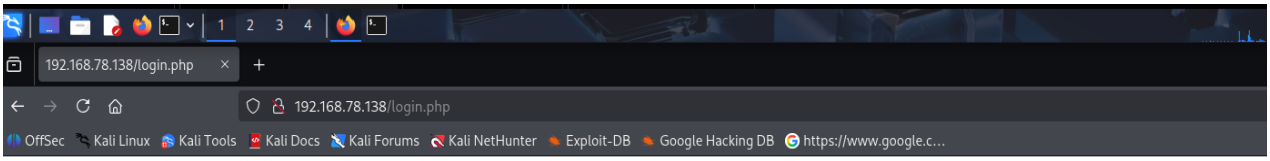
7. Attack Scenarios & Rule Testing

Test the WAF: http://localhost/login.php

Request: http://192.168.116.136/login.php



Username: admin
Request processes normally



Login Page

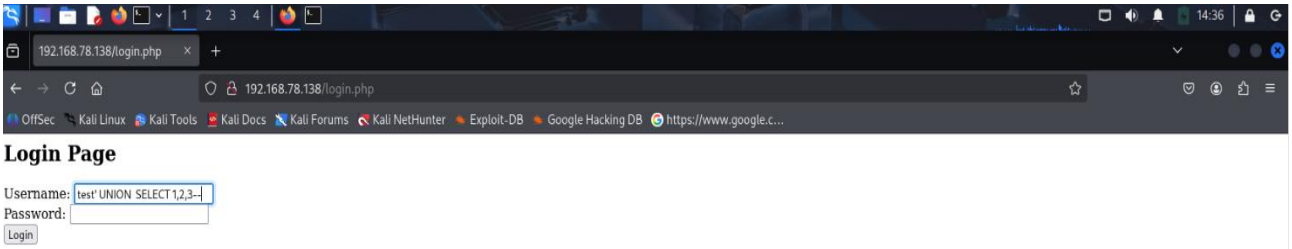
Username: admin
Password:
Login

Attempting login for user: admin

8. Testing Methodology & Verification

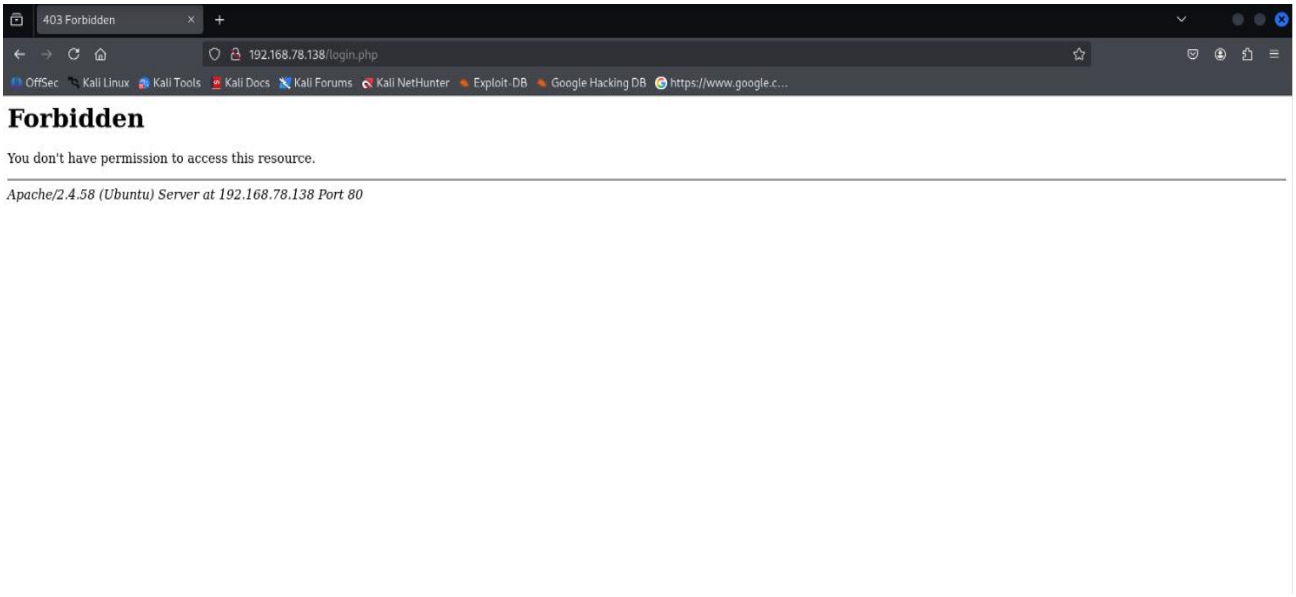
Attack Request:

SQLi payload: test' UNION SELECT 1,2,3--



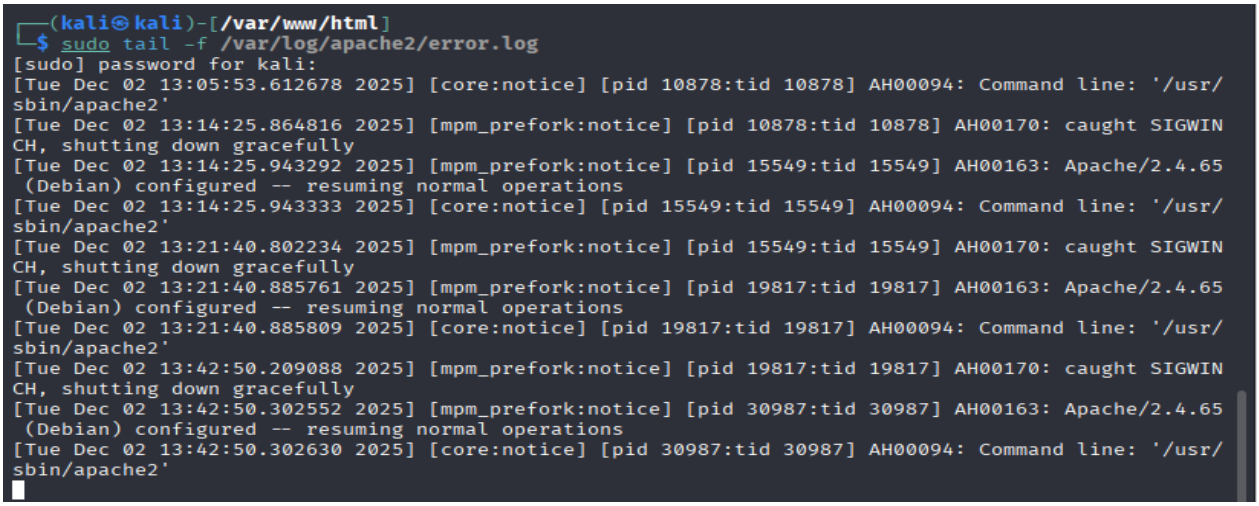
Verify the Block:

403 Forbidden
You don't have permission to access this resource



Apache's error log :

Monitor in real-time alerts: sudo tail -f /var/log/apache2/error.log



Observe WAF Response

- Check alert log for matched rule ID
- Verify HTTP 403 response received
- Document rule triggered and severity

9. Conclusion & Lessons Learned

9.1 Project Success Criteria

This lab successfully demonstrated:

- Deployment of a WAF using Apache2 and ModSecurity
- Creation of a custom rule to detect SQL injection patterns
- Proper distinction between benign and malicious requests
- Real-time blocking of attack attempts with 403 Forbidden response
- Comprehensive logging for security audits and verification.

9.2 Key Takeaways

- WAF Layer Protection: WAFs operate at the application layer (Layer 7) and inspect HTTP requests, providing more granular control than network-level firewalls.
- Rule Precision: Effective WAF rules must balance security (catching attacks) with usability (allowing legitimate traffic). False positives can break normal functionality.
- Regex Pattern Matching: Using regular expressions like (?i)' \s*union \s*select allows for flexible pattern matching that handles case variations and whitespace differences.
- Logging is Critical: Comprehensive logging enables security teams to detect attacks, investigate incidents, and tune WAF rules over time.
- Defense in Depth: While WAFs provide important protection, they should be part of a larger security strategy including secure coding practices, input validation, and prepared statements.

9.3 Real-World Applications

- E-commerce platforms: Protect login and payment forms from SQLi attacks
- Content management systems: Defend against injection attacks in form submissions
- APIs: Inspect and validate JSON/XML payloads for malicious content
- Enterprise applications: Centralized protection without modifying application code

10. References

1. ModSecurity Official Documentation: https://modsecurity.org/

2. ModSecurity Rules Language: https://github.com/SpiderLabs/ModSecurity/wiki

3. OWASP SQL Injection: https://owasp.org/wwwcommunity/attacks/SQL_Injection

4. Apache2 Security Module: https://httpd.apache.org/

5. Regular Expressions Guide: https://www.regular-expressions.info/

6. Apache Error Log Documentation:https://httpd.apache.org/docs/current/logs.html

7. LibModSecurity Installation Guide: https://github.com/SpiderLabs/ModSecurity-nginx/wiki/Installation-instructions