Project 2 — Web Application Firewall (WAF) Rule Development Lab

Full Name(s): Shreyas Gurrapu, Folasade Nasir, Raphooko Phooko

Mentor: Vasudev Jha

Date: October 28, 2025

Abstract

This report documents the development, deployment, and evaluation of a focused Web Application Firewall (WAF) rule implemented with ModSecurity to detect and mitigate a targeted SQL injection pattern (UNION SELECT). The laboratory exercise employed a deliberately vulnerable PHP login page hosted on an Ubuntu server and an attacker machine executing crafted requests. Observations, log evidence, bypass attempts, and recommendations for hardening are presented for supervisory review.

Objectives

• Deploy ModSecurity on an Ubuntu web server and enable request inspection.

• Host a deliberately PHP endpoint to serve as a test target.

• Develop and apply a custom WAF rule to detect UNION SELECT SQL injection attempts.

• Execute controlled attacker payloads to validate rule effectiveness and to assess evasion techniques.

• Produce evidence and recommendations based on observed results.

Test Environment

Target System: Ubuntu Server running Apache2 and PHP (mod_php).

ModSecurity (libapache2-mod- security2) was installed on the target and configured for request inspection.

A simple vulnerable page (login.php) that echoes user-supplied input served as the test application.

Attacker System: Kali Linux was used as the attacker host. The testing methodology relied on curl to issue HTTP requests and capture request/response data.

Logging and Evidence: Apache error logs and ModSecurity audit logs were collected and analyzed as primary evidence of rule matches and blocking actions.

Custom WAF rule deployed

A single, focused rule was created to detect UNION SELECT occurrences in POST parameters. The rule used in the lab was:

SecRule ARGS_POST "RX (?i)union\s+select" "id:10001,phase:2,deny,status:403,msg:'SQL Injection Attempt Detected (UNION SELECT)'"

Notes:

- ARGS_POST restricts matching to POST parameters (login form scenario).

- RX is the regex operator accepted by the installed ModSecurity build; @rx is the modern operator on other builds. We used RX

- (?i) makes the regex case-insensitive.

- deny,status:403 causes ModSecurity to block matching requests with HTTP 403. During tuning, a log,pass variant is recommended to avoid unintended blocks.

Methodology

A focused approach was employed. A single custom ModSecurity rule was created to detect the literal pattern 'UNION SELECT' within POST parameters.

Controlled requests—benign and malicious—were issued from the attacker host to the target endpoint.

1. Start with ModSecurity enabled and the custom rule in place; use log,pass for tuning.**(sudo tail -F /var/log/apache2/error.log /var/log/modsec_audit.log)**

2. From Kali, issue HTTP requests (curl) against login.php and capture verbose output **(curl -i -X POST 'http://<target ip>/login.php' -d "username=test UNION SELECT 1,2,3--&password=x")**

Findings

The custom rule successfully detected and prevented baseline SQL injection probes containing the literal sequence 'UNION SELECT'. Evidence of a successful block was observed in the ModSecurity audit log and the server returned HTTP 403 for matching requests. Benign traffic continued to be served normally.

Several evasion techniques were exercised to assess the rule's resilience. URL encoding, mixed-case alteration, comment insertion, and double-encoding were among the techniques tested. Results varied by technique: some obfuscated payloads were detected when input normalization was applied (e.g., URL decoding and case normalization), whereas double-encoded payloads and certain function-based encodings (CHAR()/CONCAT) bypassed the basic keyword-based rule.

Evidence and Artifacts

Representative evidence collected during testing includes:

• curl request/response captures for benign and malicious test cases.

• ModSecurity audit log blocks indicating Rule ID 10001 and the matched data string.

• Apache error log entries showing the blocking action and relevant transaction identifiers.

A redacted example of an audit block is provided in the appendix. All captured artifacts are
available as attachments to this submission.

Analysis

The rule provided effective coverage against straightforward keyword-based SQL injection attempts.
However, the evaluation highlighted limitations inherent to single-signal detections.

Notable weaknesses include:

• Reliance on literal keyword matching which can be evaded by encoding or by employing alternative SQL constructs.

• Single-stage decoding that permits double-encoding bypasses.
• Susceptibility to payload splitting and function-based encodings that avoid literal keywords.

These observations indicate the need for input normalization, layered detection strategies, and the incorporation of heuristic or community-maintained rule sets for broader coverage.

Recommendations

1. Apply input normalization transforms within rules (e.g., URL decode, lowercase, whitespace compression) to reduce simple obfuscation bypasses.

2. Adopt a phased tuning process: begin in detection/logging mode, analyze false positives, then transition to blocking.

3. Introduce layered detection mechanisms—combine keyword detection with function-name checks, length heuristics, and anomaly scoring.

4. Evaluate and integrate the OWASP Core Rule Set (CRS) and libinjection to supplement handcrafted rules.

5. Scope rules to specific parameters (whitelisting) where possible to minimize false positives.

6. Maintain secure application coding practices (parameterized database queries, proper output encoding) as primary controls.

Conclusion

The exercise demonstrates that a focused ModSecurity rule can be effective at blocking simple SQL injection probes. At the same time, practical testing exposed a range of evasion techniques that necessitate normalization and more robust detection strategies. A combined approach—application hardening plus tuned WAF rules and community rule sets—is recommended to achieve a resilient defensive posture.