Vol. 8, No. 2, April 2026

Journal of Information Systems and Informatics

Published By Asosiasi Doktor Sistem Informasi Indonesia

Coraza-Based WAF with OWASP CRS for SQL Injection in Multi-Domain Web System

Muhammad Zaedil[1], Irfan Syamsuddin[2], Muhammad Nur Yasir Utomo[3]

[1,2,3]Department of Informatics and Computer Engineering, Politeknik Negeri Ujung Pandang, Makassar, Indonesia

Received:
December 10, 2025
Revised:
March 10, 2026
Accepted:
April 11, 2026
Published:
April 26, 2026

Corresponding Author:
Author Name*:
Muhammad Zaedil
Email*:
aedills@poliupg.ac.id

DOI:
10.63158/journalisi.v8i2.1475

Abstract. This research aims to design and implement a Web Application Firewall (WAF) based on the OWASP Core Rule Set (CRS) to enhance web application protection against SQL Injection attacks. The study was conducted in the web environment of the State Polytechnic of Ujung Pandang, which has more than 80 active subdomains with uniform server configurations, mostly using vulnerable CMSs such as WordPress. The proposed solution integrates Coraza, a Go-based WAF engine, into the Nginx reverse proxy system. The system includes a web-based control panel, JSON-formatted logging, and Redis support for efficient traffic mapping and storage, enabling flexible management of multiple domains. A key contribution of this study is the implementation of a centralized WAF management approach capable of securing more than 80 subdomains within a unified configuration environment. Tests were carried out using five SQL Injection scenarios: URL parameters, form-data, x-www-form-urlencoded, JSON API, and automated tools such as SQLMap. Without WAF, all attacks successfully penetrated the system, whereas with WAF activated, all tested payloads were successfully blocked, manual and automated, was effectively blocked, indicating a significant improvement in defense capability. These results demonstrate that the developed WAF system provides strong protection against SQL Injection attacks and indicate strong potential for enhancing web application security against SQL Injection attacks.

Keywords: Web Application Firewall, Coraza, OWASP Core Rule Set, SQL Injection, Nginx, Reverse Proxy

ISI Journal of Information Systems and Informatics
Published By Asosiasi Doktor Sistem Informasi Indonesia

1. INTRODUCTION

The rapid growth of web-based systems has made websites a crucial component of modern information infrastructure, supporting communications, data management, and public services across a wide range of sectors [1][2]. Websites enable the rapid and efficient dissemination of information with broad accessibility, making them the platform of choice for developing institutional information systems [3][4]. However, the increasing reliance on web applications also increases security risks, as publicly accessible systems are constantly exposed to increasingly sophisticated cyber threats [5].

One of the most common and dangerous web-based attacks is the SQL injection attack [6][7]. This attack exploits improper input validation by injecting malicious SQL statements into application queries, allowing attackers to bypass authentication, exfiltrate sensitive data, or manipulate backend databases [8][9]. Furthermore, the effects of SQL injection attacks place database information at risk of exploitation, which compromises data integrity, disrupts server operations, and ultimately affects the organization's image [10]. Despite widespread awareness, SQL injection continues to rank among the most critical web vulnerabilities, and various detection approaches, including machine learning-based methods, have been proposed to address this issue [11].

This security issue is particularly critical in institutional web environments with homogeneous configurations. At the State Polytechnic of Ujung Pandang, 86 active subdomains operate under the same infrastructure, with many websites built using the same CMS platform, primarily WordPress. Recorded incidents show repeated SQL injection attempts targeting institutional subdomains within a short period of time, demonstrating that a single exploited vulnerability can lead to chain compromises across other services built on the same concept. Such conditions highlight the limitations of relying solely on application-level security controls, such as input sanitization or prepared statements, which are often inconsistently implemented [12].

Several studies have proposed mitigation strategies for SQL injection, including Intrusion Detection Systems (IDS) and Web Application Firewalls (WAF) [11][13][14]. While a WAF effectively operates at the application layer to inspect and block malicious traffic, the way these existing solutions are deployed often creates practical problems. For instance,

custom solutions like SEA WAF have demonstrated detection capabilities but fail to adopt standardized rulesets like OWASP CRS, limiting their adaptability [15].

On the other hand, studies using the standard OWASP CRS typically install the WAF directly on the web server. For example, previous studies evaluated ModSecurity embedded within cPanel [16] or Apache [17]. While these setups have strong detection rates, this approach has a major analytical drawback for large organizations [18]. Because the WAF is tied directly to the backend server, administrators must conFigure and manage security separately for every single website. For an institution with dozens of subdomains, this decentralized setup leads to a massive administrative workload, inconsistent security rules, and a greater risk of human error during maintenance [19]. Therefore, these prior studies highlight a critical need to shift from individual, server-based WAFs to a centralized security gateway that protects all domains from a single point.

To address these specific limitations in securing multi-domain environments, this study proposes a centralized architecture utilizing Coraza as the primary WAF engine. Coraza represents a significant shift from the traditional C-based ModSecurity, offering an Enterprise-grade Open-Source framework written in Go that provides inherent memory safety and superior performance in high-concurrency environments. Unlike legacy server-side embedded models that create operational silos and high administrative overhead, Coraza's cloud-native design allows for seamless integration at the reverse proxy layer [20]. This approach creates a single, unified security gateway capable of protecting multiple domains simultaneously, effectively decoupling security logic from backend application dependencies. Furthermore, Coraza's is compatibility with SecLang enables the optimal implementation of the OWASP Core Rule Set [21], providing a standardized security posture that is both scalable and easier to manage across complex institutional multi-domain ecosystems.

2. METHODS

The method used consists of several parts. The following explains the method used in this study.

2.1. Research Methods

The research methodology follows a structured workflow consisting of six main stages, as depicted in Figure 1.

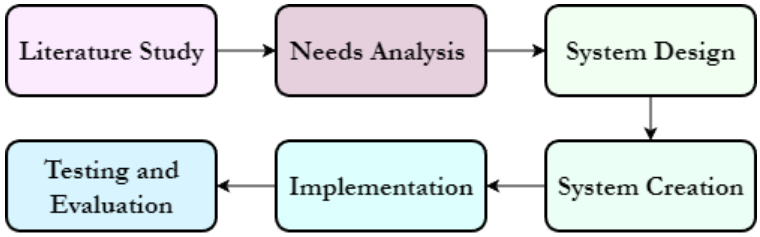

Figure 1. Research Method

First, a literature review is conducted to identify common SQL Injection attack patterns, vulnerabilities, and existing Web Application Firewall (WAF) implementation approaches. Second, a requirements analysis is conducted to determine specific system requirements, including the hardware, software, and network configurations required for deployment. Third, the system design phase maps out the architecture, detailing the integration of Nginx as a reverse proxy, the selection of Coraza as the core WAF engine, and the workflow of supporting components such as Redis and MySQL. Fourth, system buildout involves the development of required custom components, including the creation of a centralized dashboard for multi-domain management. Fifth, the implementation phase is executed by configuring Coraza with the OWASP Core Rule Set (CRS) and fully integrating it with the Nginx server in the deployment environment. Finally, testing and evaluation are conducted using mapped SQL Injection scenarios to measure the system's ability to detect and block malicious payloads.

2.2. System Design

The architecture presented in this study represents the proposed system for centralized multi-domain web application protection. The proposed system consists of three main components: reverse proxy (Nginx), Coraza WAF engine, and backend web servers. All incoming HTTP/HTTPS requests are first received by Nginx, which forwards them to Coraza for inspection using the OWASP Core Rule Set (CRS), which applies a rule-based detection mechanism based on known attack signatures to identify malicious web requests [22]. If the request is deemed malicious, it is blocked immediately; otherwise, it is relayed to the backend server.

2.2.1. Hardware Architecture

The system was deployed on a Proxmox-based virtual environment with two main hosts: (1) Proxy/WAF Server, running Nginx and Coraza; (2) Backend Server, running the web servers with multiple vulnerable web applications for testing. The hardware system overview is shown in Figure 2.

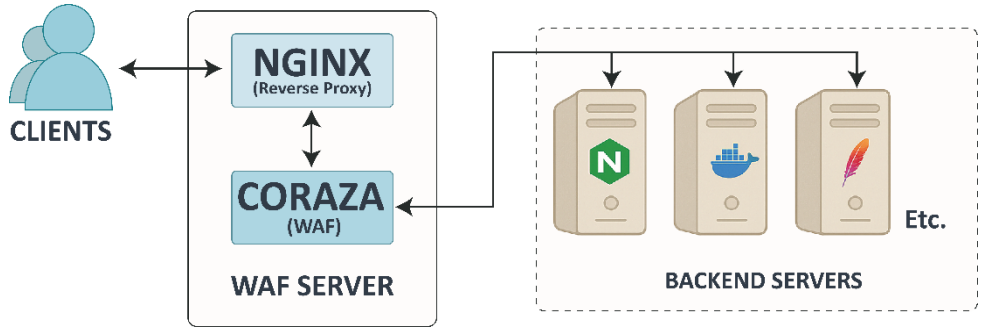

Figure 2. Hardware Architecture

As illustrated in Figure 2, incoming requests are first received by the Nginx reverse proxy, which acts as the primary entry point. The request is then forwarded to the Coraza WAF engine, where it is inspected using OWASP CRS rules. If the request matches a predefined attack pattern, it is immediately blocked and logged. Otherwise, the request is forwarded to the backend server. This mechanism ensures that malicious traffic is filtered before it reaches the application layer.

2.2.2. Software Architecture

The software layer integrates the following key technologies:

1) Nginx acts as a reverse proxy and routing layer, serving as the primary entry point for all incoming HTTP/HTTPS traffic. Previous studies have highlighted Nginx as a robust HTTP server with high performance and low resource consumption [23][24]. Nginx is responsible for forwarding requests to the WAF engine and routing validated traffic to the appropriate backend services based on the domain configuration.
2) Coraza functions as the main WAF engine, implementing the OWASP Core Rule Set (CRS) version 3 to inspect incoming requests and detect malicious patterns such as SQL Injection attacks. Coraza processes each request and determines whether it should be blocked or forwarded.

3) Redis is utilized as an in-memory data store to manage domain mapping and accelerate request processing. The implementation of Redis for such purposes is supported by recent studies demonstrating its capability to significantly decrease response times and increase system throughput in high-traffic environments [25]. Each incoming request is matched against domain configurations stored in Redis, allowing Coraza to dynamically route traffic to the appropriate backend service. Additionally, Redis is also used to store log data per unit of time before it's actually written to the database. This prevents programs from repeatedly querying the database, which can be incredibly time-consuming if there are hundreds or even thousands of requests per minute.
4) MySQL is used as the primary persistent storage for system configurations and indexed log data, enabling structured storage, querying, and long-term analysis of security events.
5) Docker is used to containerize each component of the system, ensuring isolation, portability, and scalability of services within the deployment environment.

Figure 3 illustrates the system's software architecture. Nginx functions as a reverse proxy layer, handling incoming traffic and forwarding it to the Coraza WAF engine. Coraza processes requests with several initializations, starting with connection processing and URI processing. The process then goes through five main phases, each of which uses the OWASP Core Rule Set (CRS) as a reference for evaluation:

1) Phase 1 (Request Headers): In this phase, the system processes rules that evaluate request headers. The variables analyzed include HTTP/S connection data (such as IP address, port, and protocol version), URI and GET arguments, and information within the request headers themselves, such as cookies, content-type, and content-length.
2) Phase 2 (Request Body): In this phase, the evaluation continues to the request body. The rules process POST arguments, arguments and files in multipart format, JSON and XML data payloads, and the raw request body.

ISI Journal of Information Systems and Informatics
Published By Asosiasi Doktor Sistem Informasi Indonesia

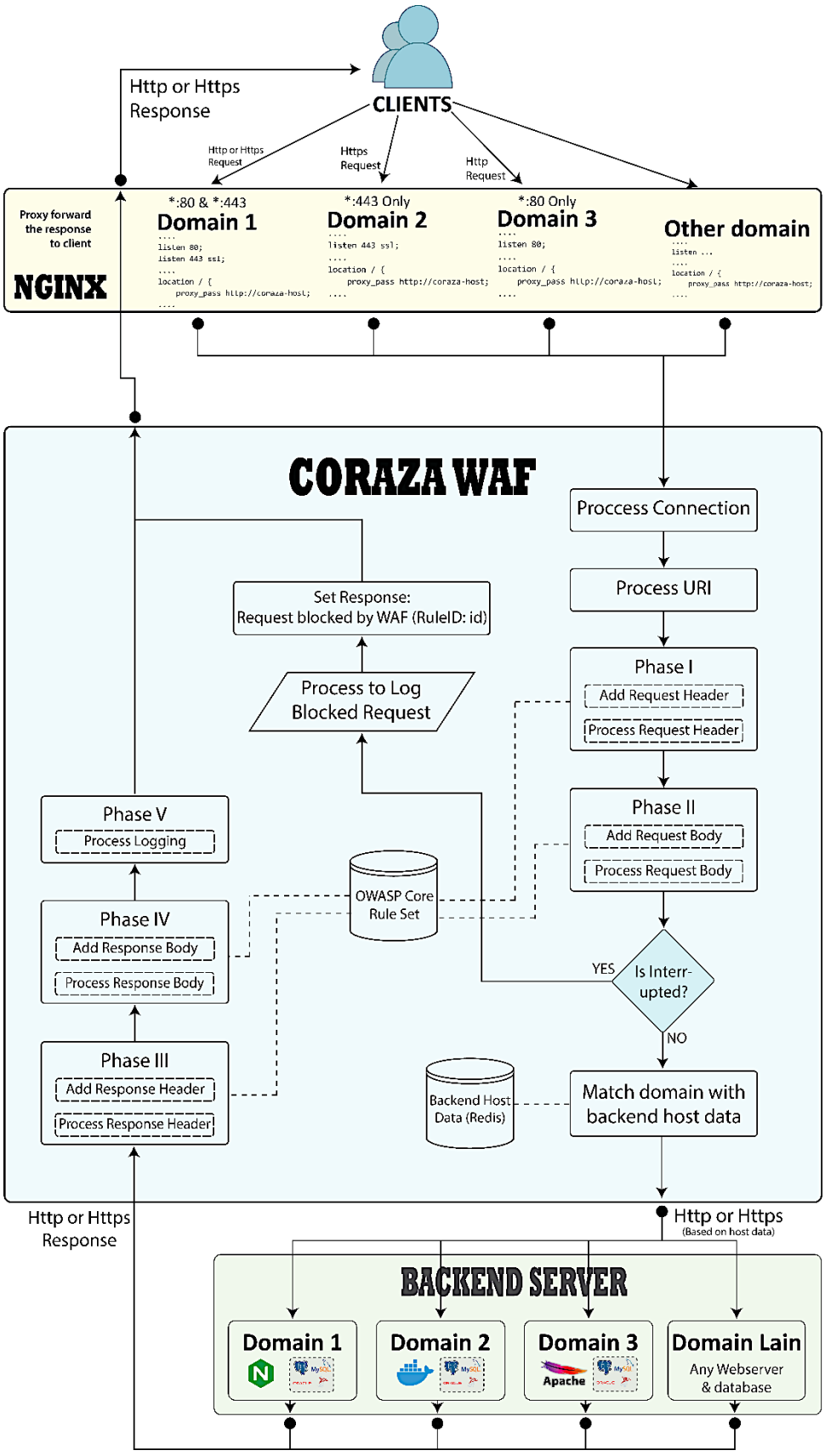

Figure 3. Software Architecture

Each evaluation result in Phases 1 and 2 will be stored in a variable. After these two phases are completed, the system will check for any interruptions triggered by the rule set. If an interruption occurs, the system will immediately redirect the process to the logging system and return a response with a request blocking status (HTTP 403 Forbidden). However, if there are no interruptions, the process will proceed to the forwarding preparation phase to the origin server.

In this phase, Coraza will map domain information by retrieving data stored in Redis using the domain name as a key. The information retrieved from Redis includes the origin server address, access method (HTTP/HTTPS), and other configuration information. Using this information, Coraza forwards the evaluated request to the origin server and waits for a response. Once a response from the origin server is received, the process continues to the next phase:

1) Phase 3 (Response Headers): This phase evaluates the headers from the origin server's response before returning it to the client. The variables processed include the response status code and response headers such as content-length and content-type.
2) Phase 4 (Response Body): In this phase, the system processes and evaluates rules related to the raw response body to ensure no sensitive data is leaked to the client.
3) Phase 5 (Logging): This final phase is non-disruptive (does not block traffic) and can be executed asynchronously after the response has been sent to the client. This phase evaluates logging rules, stores persistent data collections, and writes log entries for security audit purposes.

All evaluation results, whether in the form of a response from the origin server for valid requests or a rejection status (HTTP 403) for blocked requests, will be forwarded back to the client via Nginx as a reverse proxy.

2.3. Testing and Evaluation Procedures

The evaluation aimed to assess the effectiveness of the developed WAF in detecting and preventing SQL Injection attacks. Testing was performed under two environments:

1) Without WAF protection
2) With WAF enabled using Coraza and OWASP CRS.

Five attack scenarios were executed:

1) URL Parameter Injection (GET)
2) Form-data Injection (POST multipart/form-data)
3) x-www-form-urlencoded Injection (POST)
4) JSON API Injection (POST application/json)
5) Automated SQLMap Attack

Each scenario used the same set of 30 SQL Injection payloads categorized into Tautology-Based, Union-Based, Error-Based, Blind SQL, and Time-Based attacks. Table 1 provides a recap of the scenarios that will be used.

Table 1. Test Scenario Recap

No	Scenario	Method	Data Format	Endpoint	Testing
1.	URL Parameters	GET	Query String	https://sql.neofetch.my.id/vuln_get.php	Manual (Static Payloads)
2.	Input Forms: multipart/form-data	POST	multipart/form-data	https://sql.neofetch.my.id/vuln_post.php	Manual (Static Payloads)
3.	Input Forms: x-www-form-urlencoded	POST	application/x-www-form-urlencoded	https://sql.neofetch.my.id/vuln_post.php	Manual (Static Payloads)
4.	API JSON	POST	application/json	https://sql.neofetch.my.id/vuln_api.php	Manual (Static Payloads)
5.	SQLMap Tools	POST	(determined automatically by the tools)	https://sql.neofetch.my.id/vuln_post.php	Automatic (Dynamic Payloads)

Testing utilized Postman for manual payload delivery and SQLMap for automated penetration simulation. Success or failure of each payload was determined by backend responses and HTTP status codes:

1) Successful attack: visible SQL error, unauthorized data access, or login bypass.

2) Blocked attack: HTTP 403 (Forbidden) or terminated request.

The four manual attack scenarios will use the same SQL injection payload for each method. Table 2 lists the five main categories of SQL injection payloads used in the manual attack scenarios.

Table 2. Payloads SQL Injections

No.	Payload SQL Injection
Tautology-Based (Boolean-Based) Injection	
1.	OR 3409=3409 AND ('pytW' LIKE 'pytW
2.	OR 1=1
3.	AS INJECTX WHERE 1=1 AND 1=1
4.	ORDER BY 18#
5.	RLIKE (SELECT (CASE WHEN (4346=4346) THEN 0x61646d696e ELSE 0x28 END)) AND 'Txws'='
6.	IF(7423=7423) SELECT 7423 ELSE DROP FUNCTION xcjl--
Union Based Injection	
1.	ORDER BY 1,SLEEP(5)
2.	ORDER BY 1,SLEEP(5),BENCHMARK(1000000,MD5('A')),4,5,6,7,8
3.	UNION ALL SELECT 1,2,3,4,5,6
4.	UNION SELECT @@VERSION,SLEEP(5),USER(),BENCHMARK(1000000,MD5('A')),5
5.	UNION ALL SELECT @@VERSION,USER(),SLEEP(5),BENCHMARK(1000000,MD5('A'))--
6.	UNION ALL SELECT 'INJ'||'ECT'||'XXX',2,3,4
Error-Based Injection	
1.	1 AND EXTRACTVALUE(1, CONCAT(0x7e, VERSION(), 0x7e)) --
2.	1' AND EXTRACTVALUE(1, CONCAT(0x7e, DATABASE(), 0x7e)) --
3.	1' AND (SELECT COUNT(*) FROM (SELECT 1 UNION SELECT 1) x) --
4.	1 AND CONVERT(int, (SELECT TOP 1 name FROM sysobjects WHERE xtype='U')) --
5.	1 AND 1=CONVERT(int, (SELECT Table_name FROM information_schema.Tables LIMIT 1)) --
6.	1' AND (SELECT 1 FROM (SELECT COUNT(*), CONCAT(0x7e, DATABASE(), 0x7e), FLOOR(RAND(0)*2)) x FROM information_schema.Tables GROUP BY x) --
Blind SQL Injection	

1.	1' AND 1=1-- -
2.	1' AND ASCII(SUBSTRING(DATABASE(),1,1))=109-- -
3.	1' AND IF(1=1, SLEEP(5), 0)-- -
4.	1' AND IF(ASCII(SUBSTRING(USER(),1,1))=114, SLEEP(5), 0)-- -
5.	1' AND IF(LENGTH(DATABASE())=5, SLEEP(5), 0)-- -
6.	1' AND IF(SUBSTRING(DATABASE(),1,2)='my', SLEEP(5), 0)-- -
Time-Based Injection	
1.	1' AND SLEEP(5)-- -
2.	1' AND IF(1=1, SLEEP(5), 0)-- -
3.	1' AND IF((SELECT COUNT(*) FROM information_schema.Tables)>0, SLEEP(5), 0)-- -
4.	1' AND IF((SELECT Table_name FROM information_schema.Tables LIMIT 1)='users', SLEEP(5), 0)-- -
5.	1' AND IF(ASCII(SUBSTRING((SELECT Table_name FROM information_schema.Tables LIMIT 1),1,1))=117, SLEEP(5), 0)-- -
6.	1' AND IF((SELECT SUBSTRING(@@version,1,1))='8', SLEEP(5), 0)-- -

It is important to note that this evaluation focuses exclusively on the detection and prevention effectiveness of the WAF against SQL Injection attacks. This study does not assess system performance metrics, such as latency overhead, resource utilization (CPU/RAM), or network throughput. Furthermore, the analysis of false positive rates under legitimate traffic conditions is outside the scope of this evaluation.

3. RESULTS AND DISCUSSION

3.1 SQL Injection Testing Without WAF Protection

This phase of testing was conducted to establish a baseline security condition and to verify the vulnerability of the target web application to SQL Injection attacks prior to the deployment of the Web Application Firewall (WAF). In this scenario, all incoming HTTP requests were forwarded directly to the backend server without any application-layer inspection or filtering.

The evaluation covered four manual attack scenarios, including URL parameter injection, multipart form-data, x-www-form-urlencoded requests, and JSON-based API requests. A total of 120 SQL Injection payloads were tested across these scenarios, representing tautology-based, union-based, error-based, blind SQL Injection, and time-based injection techniques. The detailed payload composition and testing scenarios are described in Table 1 and Table 2.

The results show that the application was highly vulnerable when no WAF protection was applied. In URL parameter-based attacks, most payloads were successfully executed, producing database error messages and abnormal application responses, indicating direct interaction with the backend database. A detailed breakdown of these results is provided in Table 3.

Table 3. URL Access Test Result Without WAF

No.	Payload Category	Total Payload	Total Success	Total Failure	SQLi Prevention Percent
1.	Tautology-Based (Boolean-Based) Injection	6	5	1	16.6%
2.	Union-Based Injection	6	5	1	16.6%
3.	Error-Based Injection	6	6	0	0%
4.	Blind SQL Injection	6	6	0	0%
5.	Time-Based Injection	6	6	0	0%
	Total Percentage:	**30**	**28**	**2**	**6.64%**

Similar conditions were observed in POST-based attacks using multipart form-data and x-www-form-urlencoded formats. In both cases, nearly all payloads bypassed application controls and were executed successfully by the backend system, demonstrating the absence of effective input validation mechanisms. The complete results for these scenarios are summarized in Table 4 and Table 5.

Table 4. Data Form Access Test Result Without WAF

No.	Payload Category	Total Payload	Total Success	Total Failure	SQLi Prevention Percent
1.	Tautology-Based (Boolean-Based) Injection	6	6	0	0%
2.	Union-Based Injection	6	6	0	0%
3.	Error-Based Injection	6	6	0	0%
4.	Blind SQL Injection	6	6	0	0%
5.	Time-Based Injection	6	6	0	0%
	Total Percentage:	**30**	**30**	**0**	**0%**

Table 5. x-www-form-urlencoded Access Test Result Without WAF

No.	Payload Category	Total Payload	Total Success	Total Failure	SQLi Prevention Percent
1.	Tautology-Based (Boolean-Based) Injection	6	4	2	33.3%
2.	Union-Based Injection	6	5	1	16.6%
3.	Error-Based Injection	6	6	0	0%
4.	Blind SQL Injection	6	6	0	0%
5.	Time-Based Injection	6	6	0	0%
	Total Percentage:	**30**	**27**	**3**	**6.64%**

For JSON-based API requests, the vulnerability level was even more critical. All tested SQL Injection payloads were successfully executed without restriction, resulting in a zero percent prevention rate. This indicates that the application lacked any form of security control for JSON-formatted input. The corresponding results are presented in Table 6.

Automated testing using SQLMap further confirmed the severity of the vulnerabilities. SQLMap successfully identified injecTable parameters, detected the backend database management system, and executed multiple SQL Injection techniques, including boolean-

based, error-based, union-based, and time-based attacks. Detailed findings from the automated testing are provided in Table 7.

Table 6. API Access Test Result Without WAF

No.	Payload Category	Total Payload	Total Success	Total Failure	SQLi Prevention Percent
1.	Tautology-Based (Boolean-Based) Injection	6	6	0	0%
2.	Union-Based Injection	6	6	0	0%
3.	Error-Based Injection	6	6	0	0%
4.	Blind SQL Injection	6	6	0	0%
5.	Time-Based Injection	6	6	0	0%
	Total Percentage:	**30**	**30**	**0**	**0%**

Table 7. SQLMap Test Result Without WAF

No	Endpoint	Tool	Execution Status	Generated Output
1.	https://sql.neofetch.my.id/vuln_post.php	SQLMap	Success	1. Username parameter injecTable. 2. Backend DBMS detected: MySQL 5.6. 3. Injection techniques successful: Boolean based, Error based, Time based and Union based. 4. Backend technology information exposed (nginx, php).

Overall, the baseline testing results confirm that the target application environment was critically exposed to SQL Injection attacks across all tested input formats. These findings

highlight the limitations of relying solely on application-level controls and emphasize the necessity of deploying an application-layer protection mechanism, such as a Web Application Firewall, to mitigate both manual and automated SQL Injection threats.

3.2 SQL Injection Testing with WAF Protection

This section presents and discusses the results of SQL Injection testing conducted after the Web Application Firewall (WAF) was activated. All incoming HTTP/HTTPS requests were routed through the Nginx reverse proxy and inspected by the Coraza WAF engine using the OWASP Core Rule Set (CRS). The testing scenarios, payload sets, endpoints, and tools remained identical to those used in the baseline (non-WAF) evaluation to ensure result comparability.

3.3.1. URL Parameter Injection

When WAF was enabled, all SQL Injection payloads sent via URL parameters were successfully detected and blocked. As illustrated in Figure 4, malicious requests triggered CRS rules associated with tautology-based and union-based injection patterns, resulting in an HTTP 403 (Forbidden) response.

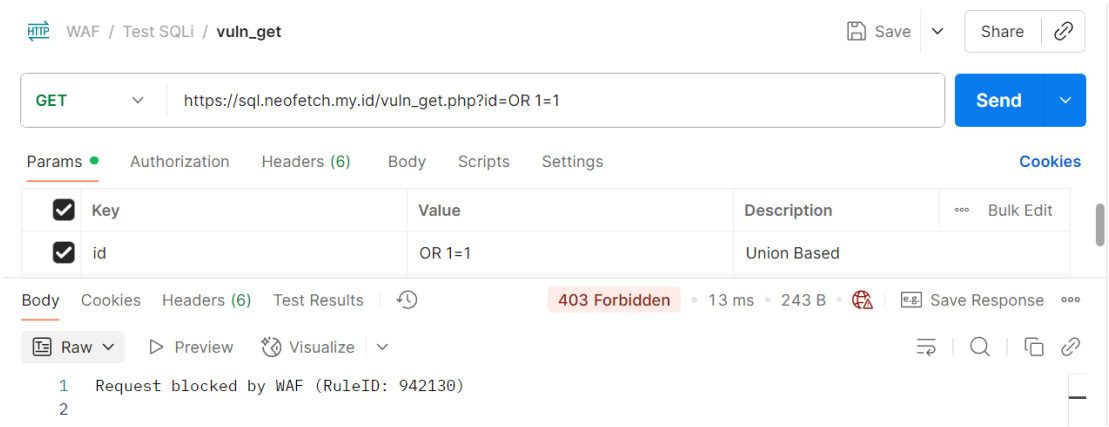

Figure 4. URL Access Test Result of One Payload (With WAF)

Detailed results summarized in Table 8 show that all 30 payloads across five attack categories (tautology-based, union-based, error-based, blind SQL, and timing-based injection) were blocked, achieving a 100% prevention rate. This represents a significant improvement compared to the condition without WAF, where most payloads were successfully executed.

These results demonstrate the ability of OWASP CRS to detect SQL Injection patterns embedded in query strings, which are commonly exploited in real-world attacks due to their visibility and ease of manipulation.

Table 8. URL Access Test Result With WAF

No.	Payload Category	Total Payload	Total Success	Total Failure	SQLi Prevention Percent
1.	Tautology-Based (Boolean-Based) Injection	6	0	6	100%
2.	Union-Based Injection	6	0	6	100%
3.	Error-Based Injection	6	0	6	100%
4.	Blind SQL Injection	6	0	6	100%
5.	Time-Based Injection	6	0	6	100%
	Total Percentage:	**30**	**0**	**30**	**100%**

3.3.2. Form-Data Injection (POST Multipart/Form-data)

For POST requests using multipart/form-data, the WAF consistently intercepted malicious payloads embedded in form parameters. As shown in Figure 5, requests containing SQL Injection patterns were immediately blocked, and corresponding CRS rule IDs were logged.

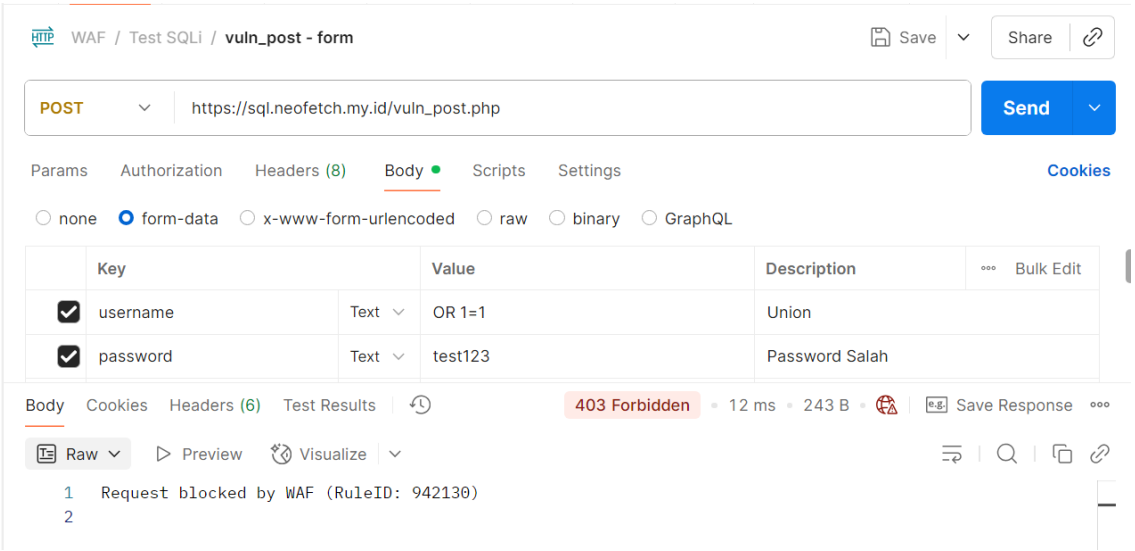

Figure 5. Form-Data Access Test Result of One Payload (With WAF)

Based on the results in Table 9, all 30 payloads were prevented from reaching the backend server, yielding a 100% mitigation rate.

This outcome highlights the ability of Coraza to inspect request bodies beyond simple URL parameters. Multipart parsing, which is often overlooked by less sophisticated security mechanisms, was handled effectively by the WAF engine.

Table 9. Form-Data Access Test Result with WAF

No.	Payload Category	Total Payload	Total Success	Total Failure	SQLi Prevention Percent
1.	Tautology-Based (Boolean-Based) Injection	6	0	6	100%
2.	Union-Based Injection	6	0	6	100%
3.	Error-Based Injection	6	0	6	100%
4.	Blind SQL Injection	6	0	6	100%
5.	Time-Based Injection	6	0	6	100%
	Total Percentage:	**30**	**0**	**30**	**100%**

3.3.3. x-www-form-urlencoded Injection

Similar results were observed for POST requests using the application/x-www-form-urlencoded format. As demonstrated in Figure 6, every malicious request resulted in an HTTP 403 response, indicating successful WAF intervention. The aggregated results in Table 10 confirm that all SQL Injection payloads across all categories were blocked, achieving full protection.

Compared to the baseline condition without WAF, where several payloads bypassed application logic, this finding confirms that CRS-based inspection remains effective even when payloads are encoded in standard form submission formats.

ISI Journal of Information Systems and Informatics
Published By Asosiasi Doktor Sistem Informasi Indonesia

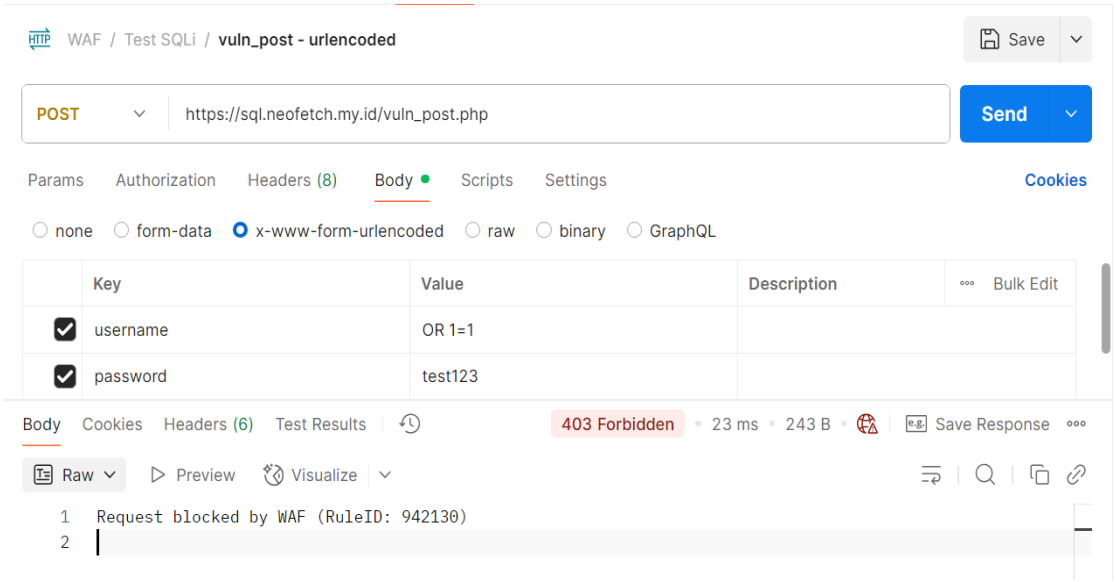

Figure 6. x-www-form-urlencoded Access Test Result with WAF

Table 10. x-www-form-urlencoded Access Test Result with WAF

No.	Payload Category	Total Payload	Total Success	Total Failure	SQLi Prevention Percent
1.	Tautology-Based (Boolean-Based) Injection	6	0	6	100%
2.	Union-Based Injection	6	0	6	100%
3.	Error-Based Injection	6	0	6	100%
4.	Blind SQL Injection	6	0	6	100%
5.	Time-Based Injection	6	0	6	100%
	Total Percentage:	**30**	**0**	**30**	**100%**

3.3.4. Json Based API Injection

SQL Injection attacks delivered via JSON payloads were also fully mitigated by the WAF. As shown in Figure 7, malicious JSON requests triggered CRS rules and were blocked before reaching the API endpoint.

ISI Journal of Information Systems and Informatics
Published By Asosiasi Doktor Sistem Informasi Indonesia

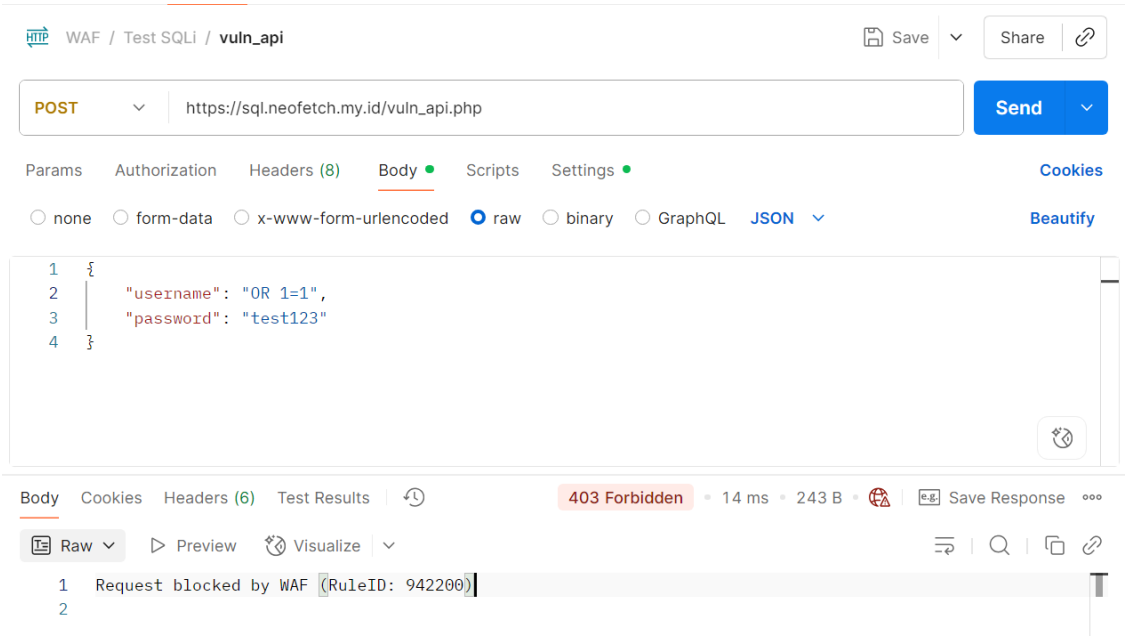

Figure 7. Json Based API Access Test Result of One Payload (With WAF)

According to Table 11, the prevention rate reached 100%, with no payload successfully executed.

Table 11. Json Based API Access Test Result with WAF

No.	Payload Category	Total Payload	Total Success	Total Failure	SQLi Prevention Percent
1.	Tautology-Based (Boolean-Based) Injection	6	0	6	100%
2.	Union-Based Injection	6	0	6	100%
3.	Error-Based Injection	6	0	6	100%
4.	Blind SQL Injection	6	0	6	100%
5.	Time-Based Injection	6	0	6	100%
	Total Percentage:	**30**	**0**	**30**	**100%**

This result is particularly significant because JSON-based APIs are frequently targeted in modern web and mobile applications, and many legacy security mechanisms fail to adequately inspect JSON request bodies. The findings demonstrate that Coraza and OWASP CRS are capable of providing robust protection for API-driven architectures.

ISI Journal of Information Systems and Informatics
Published By Asosiasi Doktor Sistem Informasi Indonesia

3.3.5. Automatic SQL Injection Using SQLMap

To evaluate resilience against automated exploitation, SQLMap was run against a protected endpoint. When the WAF was active, SQLMap failed to identify any injecTable parameters, as illustrated in Figure 8.

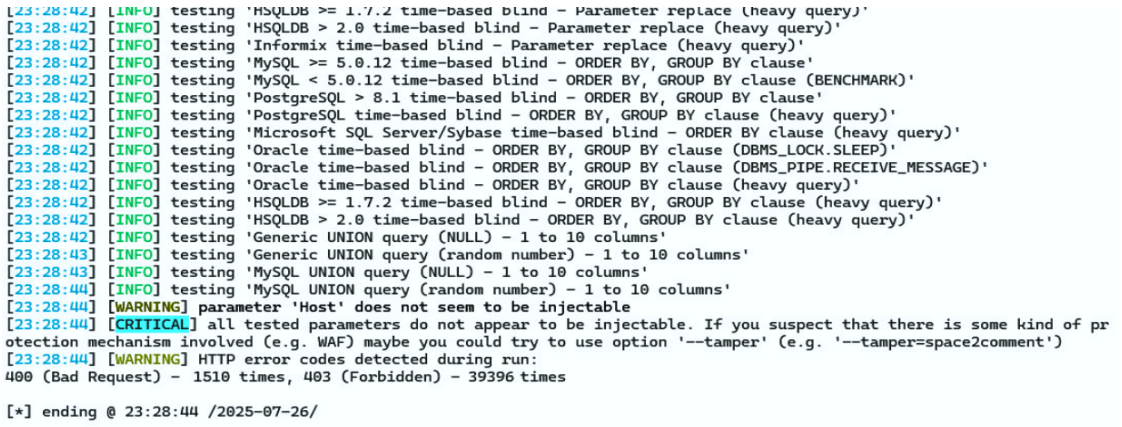

Figure 8. SQLMap Tool Test Result with WAF

The summary in Table 12 shows that SQLMap repeatedly encountered HTTP 403 and 400 responses and explicitly detected the presence of a Web Application Firewall. Unlike the baseline scenario, where SQLMap successfully enumerated database information, all automated exploitation attempts failed under WAF protection. This demonstrates that the WAF configuration is not only capable of preventing manual attacks but also resilient to aggressive tool-based SQL injection attempts.

Table 12. SQLMap Tools test Result with WAF

No	Endpoint	Tool	Execution Status	Generated Output
1.	https://sql.neofetch.my.id/vuln_post.php	SQLMap	Critical / Failed	1. All parameters are not injecTable 2. Error 403 occurred 39.396 times 3. Error 400 occurred 1.510 times 5. SQLMap detected the WAF mechanism

3.3. Evaluation

The evaluation was conducted to measure the ability of the proposed WAF system in mitigating SQL Injection attacks across multiple scenarios. Testing was performed using both manual techniques and automated tools such as SQLMap, covering various payload formats including URL parameters, form-data, x-www-form-urlencoded, and JSON-based requests.

Table 13. Summary of test results without WAF

No.	Access Method	Method	Success Rate	Prevention Rate
1.	URL Access	GET	93.3%	6.64%
2.	Data-Forms	POST	100%	0%
3.	Data-Forms (urlencoded)	POST	93.3%	6.64%
4.	JSON API	POST	100%	0%
	Average		96.68%	3.32%

Without WAF protection, Table 13 show that approximately 96.68% of SQL Injection payloads were successfully executed, while only an average 3.32% failed due to existing application-level or server-level protections. This indicates that the baseline system remains highly vulnerable to SQL Injection attacks.

Table 14. Summary of test results with WAF

No.	Access Method	Method	Success Rate	Prevention Rate
1.	URL Access	GET	0%	100%
2.	Data-Forms	POST	0%	100%
3.	Data-Forms (urlencoded)	POST	0%	100%
4.	JSON API	POST	0%	100%
	Average		0 %	100%

After enabling the WAF, Table 14 shows that all 120 tested payloads based on payload list in Table 2 were successfully blocked across all scenarios, resulting in a 100% prevention rate. This demonstrates a significant improvement in the system's defense capability compared to the baseline condition.

As shown in Table 14, the WAF consistently blocked all manual SQL Injection attempts across different payload formats. Similarly, Table 12 shows that automated attacks generated using SQLMap were also fully mitigated. These results confirm that the system

3.4. Discussion

The results demonstrate that the proposed WAF system effectively mitigates SQL Injection attacks across all tested scenarios. The consistent blocking of all tested payloads indicates the effectiveness of the OWASP Core Rule Set (CRS) in detecting a wide range of SQL Injection patterns, including variations in URL parameters, form-data, x-www-form-urlencoded, and JSON-based requests.

The high detection capability can be attributed to the rule-based detection and anomaly scoring mechanisms implemented in OWASP CRS, which allow the system to identify malicious patterns regardless of encoding or delivery format. This ensures consistent protection across diverse application structures and request types.

Compared to baseline conditions without WAF, where a small percentage of payloads failed due to existing application-level protections, the proposed system eliminates these inconsistencies by enforcing a centralized and uniform security layer. This demonstrates that the WAF not only enhances security but also standardizes protection across multiple domains.

In comparison with previous studies that rely on embedded WAF implementations such as ModSecurity, the proposed system offers improved flexibility and scalability through its reverse proxy architecture and centralized management approach. This makes it particularly suiTable for institutional environments with a large number of subdomains requiring consistent protection.

From a performance perspective, the implementation of the WAF introduces additional processing overhead due to request inspection. While the system remained functional during the testing phase, the specific impact on latency and throughput requires further empirical measurement. Future studies will focus on a detailed quantitative analysis to determine the system's performance limitations.

Additionally, the failed payloads observed in the baseline scenario without WAF protection may be attributed to default server configurations or built-in input validation mechanisms. However, such protections are inconsistent and insufficient compared to the comprehensive filtering provided by the WAF. Overall, the findings indicate that the proposed Coraza-based WAF architecture is effective for centralized deployment in institutional environments, providing scalable, consistent, and reliable protection against SQL Injection attacks.

4. CONCLUSION

This study demonstrates that a centralized Coraza-based WAF, integrated with the OWASP Core Rule Set via a Nginx as the reverse proxy, provides highly effective protection for multi-domain institutional environments. Experimental results showed a stark contrast: without the WAF, 96.68% of SQL Injection payloads executed successfully, whereas the implemented system achieved a 100% prevention rate across all tested manual and automated attack vectors. While this architecture significantly standardizes security enforcement, current evaluations are limited strictly to SQL Injection. Future research should expand the threat scope to vulnerabilities such as XSS and DDoS, while comprehensively analyzing false positive rates, specific latency overheads, and long-term operational stability in larger deployments.

REFERENCES

[1] M. Nawrocki and J. Kołodziej, "Vulnerabilities of Web Applications: Good Practices and New Trends," *Applied Cybersecurity & Internet Governance*, vol. 3, no. 2, pp. 122–143, 2024, doi: 10.60097/ACIG/199521.

[2] A. Wahyudi, "Digital Transformation in Public Service Management: Addressing Challenges in the Modern Era," *Sinomics Journal*, vol. 3, 2024, doi: 10.54443/sj.v3i4.409.

[3] R. Riche and S. H. Marpaung, "Pengembangan Website Sekolah SD-SMP Methodist Romalbest Medan," *Jurnal Pengabdian Masyarakat (ABDIRA)*, vol. 2, no. 4, pp. 62–70, 2022.

ISI Journal of Information Systems and Informatics
Published By Asosiasi Doktor Sistem Informasi Indonesia

[4] R. G. Mokosolang, A. Mewengkang, and O. E. S. Liando, "Analisis dan Perancangan Website Sekolah Menengah Pertama," *Edutik: Jurnal Pendidikan Teknologi Informasi Dan Komunikasi*, vol. 2, no. 1, pp. 141–146, 2022.

[5] I. R. N. Ardhian, "Dampak serangan siber dan kebocoran data pada perbankan syariah terhadap tingkat kepercayaan nasabah," *Maliki Interdisciplinary Journal*, vol. 1, no. 3, pp. 351–359, 2023.

[6] S. Tamilselvan and K. France, "SQL Injection Attack Detection in Web Applications Using Machine Learning Algorithms," in *International Conference on Trends in Electronics and Informatics (ICOEI)*, 2025, pp. 545–552. doi: 10.1109/ICOEI65986.2025.11013708.

[7] T. Muhammad and H. Ghafory, "SQL Injection Attack Detection Using Machine Learning Algorithm," *Mesopotamian Journal of CyberSecurity*, vol. 2022, pp. 5–17, 2022, doi: 10.58496/MJCS/2022/002.

[8] B. Wiguna et al., "Implementasi Web Application Firewall dalam Mencegah Serangan SQL Injection pada Website," *Jurnal Teknologi Informasi & Komunikasi*, vol. 11, no. 2, pp. 245-256, Nov. 2020. doi: 10.31849/digitalzone.v11i2.4867ICCS.

[9] W. G. J. Halfond, J. Viegas, and A. Orso, "A Classification of SQL Injection Attacks and Countermeasures," in *Proceedings of the IEEE International Symposium on Secure Software Engineering*, 2006.

[10] K. Ahmad and M. Karim, "A Method to Prevent SQL Injection Attack using an Improved Parameterized Stored Procedure," *International Journal of Advanced Computer Science and Applications*, vol. 12, no. 6, 2021.

[11] M. Curipallo Martínez, A. Guevara-Vega, A. Reyes Narváez, G. Raura, and H. Barba Molina, "Web Application Protection Optimization Through Coraza WAF: Performance Assessment Against OWASP Risks in Reverse Proxy Configurations," *Engineering Proceedings*, vol. 115, no. 1, 2025, doi: 10.3390/engproc2025115017.

[12] A. Riyanti, B. M. Rahmanto, D. R. Hardianto, R. D. A. Yuristiawan, and A. Setiawan, "Uji Penetrasi Injeksi SQL terhadap Celah Keamanan Database Website menggunakan SQLmap," *Journal of Internet and Software Engineering*, vol. 1, no. 4, p. 9, Jun. 2024, doi: 10.47134/pjise.v1i4.2623.

[13] I. Bilic, K. Josić, D. Pranic, and S. Ribaric, "Web Application Firewalls (WAFs) in Protecting Software," in *Proceedings of the DAAAM International Symposium*, 2024, pp. 306–311. doi: 10.2507/35th.daaam.proceedings.042.

[14] R. Riska and H. Alamsyah, "Penerapan Sistem Keamanan Web Menggunakan Metode Web Application Firewall," *Jurnal Amplifier: Jurnal Ilmiah Bidang Teknik Elektro Dan Komputer*, vol. 11, no. 1, pp. 37–42, 2021.

[15] J. Harefa, G. Prajena, A. Alexander, A. Muhamad, E. V. S. Dewa, and S. Yuliandry, "SEA WAF: The Prevention of SQL Injection Attacks on Web Applications," *Advances in Science, Technology and Engineering Systems Journal*, vol. 6, no. 2, pp. 405–411, Mar. 2021, doi: 10.25046/aj060247.

[16] M. Akbar and M. A. F. Ridha, "SQL Injection and Cross Site Scripting Prevention Using OWASP Web Application Firewall," *International Journal on Informatics Visualization*, vol. 2, 2018.

[17] B. I. Mukhtar and M. A. Azer, "Evaluating the Modsecurity Web Application Firewall against SQL Injection Attacks," in *Proceedings of ICCES 2020 - 2020 15th International Conference on Computer Engineering and Systems*, Institute of Electrical and Electronics Engineers Inc., Dec. 2020. doi: 10.1109/ICCES51560.2020.9334626.

[18] M. Alghawazi, D. Alghazzawi, and S. Alarifi, "Detection of SQL Injection Attack Using Machine Learning Techniques: A Systematic Literature Review," *Journal of Cybersecurity and Privacy*, vol. 2, no. 4, pp. 764–777, 2022, doi: 10.3390/jcp2040039.

[19] R. A. Muzaki, O. C. Briliyant, M. A. Hasditama, and H. Ritchi, "Improving Security of Web-Based Application Using ModSecurity and Reverse Proxy in Web Application Firewall," in *2020 International Workshop on Big Data and Information Security (IWBIS)*, 2020, pp. 85–90. doi: 10.1109/IWBIS50925.2020.9255601.

[20] M. H. Syed, "Benchmarking Open-Source WAF Engines Against Modern Evasion Payloads," *SSRN preprint*, 2026. doi: 10.2139/ssrn.6141529.

[21] S. Amelinckx, R. Sadre, C.-H. Bertrand, V. Ouytsel, E. Hegedüs, and S. Mihy, "Advancing continuous integration for WAF engines by developing the ModSecurity Regression Test Set," Master's thesis, UCLouvain, Belgium, 2025.

[22] A. MK, K. S. S. Bala, S. S. T. Sonti, and J. KP, "An empirical study on the evaluation and enhancement of OWASP CRS (Core Rule Set) in ModSecurity," *Comput. Secur.*, vol. 160, p. 104714, 2026, doi: 10.1016/j.cose.2025.104714.

[23] F. Agostini et al., "Enhancing StoRM WebDAV data transfer performance with a new deployment architecture behind NGINX reverse proxy," in Proceedings of Science, 2024.

ISI Journal of Information Systems and Informatics
Published By Asosiasi Doktor Sistem Informasi Indonesia

[24] M. Kazemi, "Optimizing Web Service Performance: A Comparative Analysis of Load Balancing Strategies Using NGINX and HAProxy with StoRM WebDAV Deployment," *Master's thesis, Telecommun. Eng., Univ. Bologna, Bologna, Italy*, 2024.

[25] L. Kaptosv, "Using Redis for caching optimization in high-traffic web applications," *International Journal of Advanced Multidisciplinary Research and Studies*, vol. 5, no. 4, pp. 1714–1722, 2025.