

Design and Implementation of a Custom Web Application Firewall Middleware Using Node.js and React

Author: **KATTA VENKATA RAMANA¹** (MCA student), **Mr .M. Sreenivasu²** (Assistant Professor)

Master Of Computer Applications, GIET ENGINEERING COLLEGE (A), Rajahmundry, AP.

Corresponding Author: Katta Venkata ramana

(email-id: kattaramana139@gmail.com)

ABSTRACT: With the rapid growth of web applications, protecting them from cyberattacks has become a major challenge. Attackers frequently exploit vulnerabilities in web applications to gain unauthorized access, steal sensitive information, or disrupt services. To address these security concerns, this project presents the design and implementation of a custom Web Application Firewall (WAF) developed using Node.js and the Express framework.

The proposed WAF acts as a middleware layer between clients and web applications, monitoring and analyzing every incoming HTTP request before it reaches the application. The system follows a six-stage inspection process that includes IP blacklist verification, rate limiting, header inspection, URL analysis, request body inspection, and final access control decisions.

A regex-based rule engine is used to identify common web attacks such as SQL Injection, Cross-Site Scripting (XSS), Path Traversal, Command Injection, LDAP Injection, and XML Injection. In addition, the system supports automatic IP blacklisting, intelligent rate limiting, structured attack logging using Winston, and persistent storage of attack records using MongoDB. To provide real-time visibility into security events, a monitoring dashboard was developed using React.js. The dashboard displays attack statistics, blocked requests, attack trends, and suspicious IP activity in an interactive format.

The implementation results demonstrate that the proposed WAF effectively detects and blocks major web application attacks while allowing legitimate user requests to pass without interruption. The system offers a cost-effective, customizable, and lightweight security solution suitable for modern web applications.

KEYWORDS: Web Application Firewall, Node.js, Express Middleware, SQL Injection, Cross-Site Scripting, Rate Limiting, IP Blacklisting, React Dashboard, MongoDB, Cybersecurity, HTTP Security, Attack Detection.

I. INTRODUCTION

web applications are widely used in areas such as banking, e-commerce, education, healthcare, and social networking. As the dependence on web-based services continues to grow, the number of cyberattacks targeting

these applications has also increased significantly. Attackers often attempt to exploit security vulnerabilities in web applications to gain unauthorized access, steal sensitive information, manipulate data, or disrupt normal services.

Although traditional network firewalls provide

protection against network-level threats, they are not capable of analyzing the actual content of HTTP requests. As a result, many application-layer attacks such as SQL Injection and Cross-Site Scripting (XSS) can bypass traditional security mechanisms and directly target web applications.

To overcome this challenge, Web Application Firewalls (WAFs) have become an important component of modern web security. A WAF operates at the application layer and examines incoming HTTP requests, including URLs, headers, query parameters, and request payloads. By analyzing request content before it reaches the application, a WAF can identify and block potentially malicious activities, thereby reducing the risk of security breaches.

This project focuses on the design and implementation of a custom Web Application Firewall using Node.js and Express. The firewall is developed as middleware that intercepts every incoming request and performs multiple security checks before allowing access to the application. The system is designed with a modular and lightweight architecture, making it easy to integrate into existing Node.js applications.

In addition to attack detection and prevention, the project includes features such as rate limiting, IP blacklisting, attack logging, and a real-time monitoring dashboard developed using React.js. These features help administrators monitor suspicious activities and gain better visibility into the security status of the application. The project demonstrates a practical approach to strengthening web application security using modern web technologies.

Objectives of the Project

The primary objective of this project is to design and develop a custom Web Application Firewall (WAF)

that enhances the security of web applications by detecting and

preventing common web-based attacks. The specific

objectives of the project are as follows:

- To develop a middleware-based Web Application Firewall that monitors and analyzes incoming HTTP requests before they reach the web application.
- To create an efficient rule-based detection mechanism using regular expressions (Regex) for identifying common web attacks such as SQL Injection, Cross-Site Scripting (XSS), Path Traversal, Command Injection, LDAP Injection, and XML Injection
- To implement an IP-based rate limiting feature that helps prevent brute-force attempts, denial-of-service activities, and excessive request flooding.
- To design an automated IP blacklisting mechanism that identifies suspicious behavior and blocks repeated malicious requests from the same source.
- To maintain detailed security logs using Winston, recording important information such as attack type, source IP address, severity level, and timestamp for security analysis.
- To integrate MongoDB for storing attack records and security events, enabling future investigation and historical analysis of attack patterns.
- To develop a real-time monitoring dashboard using React.js that provides visual insights into attack statistics, blocked requests, and overall system security status.
- To provide a cost-effective and customizable security solution that can be easily integrated into existing Node.js web applications
- To demonstrate the practical implementation of web application security concepts and improve protection against common vulnerabilities listed in the OWASP security guidelines.

II. Literature Survey:

2.1 OWASP Top Ten Web Application Security Risks:

The OWASP Top Ten is a widely recognized security standard that identifies the most critical vulnerabilities affecting web applications. It highlights threats such as Injection Attacks, Cross-Site Scripting (XSS), Broken Authentication, and Security Misconfiguration. The OWASP guidelines helped in identifying the major attack categories addressed in the proposed system.

2.2 SQL Injection Attack Classification:

Halfond, Viegas, and Orso presented a detailed study on SQL Injection attacks and their prevention techniques. Their research explains various attack methods used to manipulate database queries and gain unauthorized access to sensitive information. This study provided valuable insights for designing the SQL Injection detection module of the firewall.

2.3 Cross-Site Scripting (XSS) Prevention:

Bisht and Venkatakrishnan proposed techniques for preventing Cross-Site Scripting attacks through dynamic analysis. Their work discusses common XSS attack vectors and the security risks associated with malicious scripts. The concepts presented in this research were useful for developing XSS detection rules within the proposed system.

2.4 Web Application Vulnerabilities:

The Web Application Hacker's Handbook provides detailed information about web application security weaknesses and attack techniques. It covers vulnerabilities such as Path Traversal, Command Injection, LDAP Injection, and XML-based attacks. This resource helped in understanding attacker behavior and developing effective attack detection patterns.

2.5 Firewall Security Guidelines:

The NIST guidelines on firewall technologies explain the importance of network security and firewall policies. The study highlights the limitations of traditional network firewalls and emphasizes the need for application-layer protection mechanisms. These findings supported the development of a dedicated Web Application Firewall.

2.6 Existing WAF Solutions:

Several existing Web Application Firewall solutions such as ModSecurity, Cloudflare WAF, and AWS WAF were studied. These systems provide protection through predefined security rules and traffic analysis. However, factors such as cost, deployment complexity, and limited customization encouraged the development of a lightweight and flexible alternative.

2.7 Express.js Middleware Architecture:

The Express.js framework provides a middleware-based architecture that allows requests to be processed before reaching application routes. This feature forms the foundation of the proposed system, where multiple security checks are performed on each incoming request.

2.8 Rate Limiting and Brute Force Protection:

Various studies show that limiting repeated requests from the same IP address can significantly reduce brute-force and credential-stuffing attacks. Based on these findings, an IP-based rate limiting mechanism was incorporated into the proposed firewall.

2.9 Structured Logging and Monitoring:

Security monitoring relies on accurate and structured logging of events. Research suggests that maintaining detailed records of attack activities improves incident analysis and threat investigation. This concept influenced the implementation of Winston logging and MongoDB-based storage.

2.10 Real-Time Security Dashboard:

Modern security systems commonly use dashboards to visualize attack statistics and security events. React.js provides an efficient framework for building interactive and responsive dashboards. Therefore, it was selected for developing the monitoring interface of the proposed system.

III. SYSTEM ANALYSIS AND DESIGN:

3.1 Existing System

Web application security is commonly achieved through

a combination of network firewalls, application-level validations, and commercial security solutions. Although these approaches provide a certain level of protection, they still have several limitations when dealing with modern web-based attacks.

1.Traditional Network Firewalls:

Traditional firewalls primarily operate at the network and transport layers of the OSI model. Their main function is to filter incoming and outgoing traffic based on factors such as IP addresses, protocols, and port numbers. While they are effective in controlling network access, they cannot analyze the actual content of HTTP requests. As a result, they are unable to identify application-layer attacks such as SQL Injection, Cross-Site Scripting (XSS), and other web-based threats.

2.Manual Input Validation:

Many web applications depend on developers to implement input validation within the application code. Although input validation is an important security practice, it requires careful implementation throughout the entire application. Any overlooked input field or coding mistake can create security vulnerabilities. This approach is time-consuming, difficult to maintain, and heavily dependent on the developer's security awareness and experience.

3.Commercial Web Application Firewalls:

Several commercial WAF solutions are available in the market, offering advanced security features and managed protection services. While these solutions provide strong security capabilities, they often involve subscription costs, licensing fees, and vendor dependency. For small businesses, educational projects, and individual developers, these solutions may not be financially practical. Additionally, externally hosted services can sometimes introduce additional network latency.

4.Framework-Level Security Features:

Modern web frameworks provide built-in security mechanisms such as input sanitization, security headers, and authentication support. However, these features alone may not be sufficient to defend against all types of web application attacks. They are generally designed to address specific security concerns and may not offer centralized monitoring, attack logging, or customizable threat detection capabilities.

Drawbacks of Existing System

- Unable to effectively detect application-layer attacks such as SQL Injection and Cross-Site Scripting (XSS).
- High implementation and maintenance costs for small organizations and individual developers.
- Limited flexibility for customization based on specific application requirements.
- Lack of real-time monitoring and visualization of security threats.
- Absence of centralized logging and attack analysis mechanisms.
- No automated IP blacklisting to handle repeated malicious activities.
- Dependence on external security services and third-party providers.

- Limited visibility into attack patterns and security incidents.

3.2 Proposed System

The proposed system is a custom Web Application Firewall (WAF) developed using Node.js and Express.js to enhance the security of web applications. The firewall acts as a middleware layer between users and the web application, inspecting incoming HTTP requests before they reach the application. The system is designed to provide protection against common web-based attacks while remaining lightweight, flexible, and easy to integrate into existing applications.

1.Multi-Stage Request Inspection:

Every incoming HTTP request passes through a series of security checks before being processed by the application. These checks include IP blacklist verification, rate limiting, header inspection, URL analysis, and request body inspection. This layered approach improves the overall effectiveness of attack detection and prevention.

2.Regex-Based Attack Detection:

The system uses a rule-based detection mechanism built with regular expressions (Regex) to identify suspicious patterns in incoming requests. It can detect common web application attacks such as SQL Injection, Cross-Site Scripting (XSS), Path Traversal, Command Injection, LDAP Injection, and XML Injection.

3.Rate Limiting Mechanism:

To prevent abuse and excessive traffic, the firewall implements IP-based rate limiting. Different request limits can be configured for general pages, login endpoints, and API routes. This helps reduce the risk of brute-force attacks and request flooding.

4.Automatic IP Blacklisting:

The system continuously monitors suspicious activities. IP addresses that repeatedly generate malicious requests are automatically added to a blacklist,

preventing further access and reducing the impact of persistent attackers.

5.Logging and Database Storage:

All blocked requests and security events are recorded using Winston logging. Important information such as source IP address, attack type, timestamp, and severity level is captured and stored in MongoDB. This allows administrators to analyze attack trends and maintain historical security records.

6.Real-Time Monitoring Dashboard:

A user-friendly dashboard developed using React.js provides real-time visibility into security events. The dashboard displays information such as total requests, blocked requests, attack categories, frequently targeted endpoints, and suspicious IP addresses. This helps administrators monitor application security more effectively.

Advantages of Proposed System

- Detects common web application attacks such as SQL Injection, XSS, Path Traversal, Command Injection, LDAP Injection, and XML Injection.
- Free and open-source solution.
- Easy to integrate into Node.js and Express applications.
- Provides real-time monitoring and visualization of security events.
- Stores attack history and security logs in MongoDB.
- Supports automatic IP blacklisting for repeated attackers.
- Allows customization of security rules and thresholds.
- Lightweight and does not significantly affect application performance.
- Improves overall web application security.Reduces manual effort in threat monitoring and response.

3.3 System Architecture

1.Client Layer

The Client Layer represents all sources of HTTP requests interacting with the web application. These

requests may originate from legitimate users accessing the application through web browsers, API clients, or mobile applications. It also includes requests generated by attackers using testing tools, automated scanners, or malicious scripts. All incoming traffic first enters the system through this layer.

2.WAF Inspection Layer

The WAF Inspection Layer serves as the core security component of the system. It is implemented as Express.js middleware and examines every incoming request before it reaches the application. The request passes through multiple security checks, including IP blacklist verification, rate limiting, header inspection, URL analysis, and request body inspection.

The Rule Engine analyzes request data using predefined attack detection rules, while the IP Management module tracks request counts and manages blacklisted IP addresses. If any malicious activity is detected, the request is immediately blocked and an appropriate response is returned to the client.

3.Application Layer

The Application Layer contains the actual web application and its functionalities, such as user authentication, search features, and API endpoints. Only requests that successfully pass all security checks in the WAF layer are allowed to access this layer. This helps ensure that malicious requests are filtered before reaching the application.

4.Logging and Monitoring Layer

The Logging and Monitoring Layer is responsible for recording security events and maintaining attack information. Winston is used to generate structured log files containing details of blocked requests and detected attacks. MongoDB stores attack records and blacklist information for future analysis and reporting purposes.

This layer also provides statistical data that can be accessed by the monitoring dashboard, enabling administrators to track security events and system

activity.

5.React Dashboard Layer

The React Dashboard provides a user-friendly interface for monitoring application security in real time. It displays important information such as total requests, blocked requests, attack categories, attack trends, suspicious IP addresses, and recent security events. The dashboard communicates with the backend through APIs and updates information at regular intervals, helping administrators monitor threats effectively.

IV. PROPOSED METHODOLOGY

4. 1. Request Interception

The first step in the methodology is intercepting all incoming HTTP requests. A global middleware is implemented in the Express.js application using the `app.use()` function. This middleware ensures that every request passes through the firewall before reaching the application routes. The client's IP address is extracted from the request and used for further security analysis.

4.2.IP Verification and Rate Limiting

After interception, the system checks whether the incoming IP address is present in the blacklist. If the IP is found in the blacklist, the request is immediately blocked and a 403 Forbidden response is returned.

If the IP is not blacklisted, the system verifies whether the request exceeds predefined rate limits. Different request limits are configured for general pages, login endpoints, and API routes. Requests exceeding the allowed threshold are blocked with a 429 Too Many Requests response, helping to prevent brute-force attacks and request flooding.

4. 3. Header Inspection

In this phase, request headers are analyzed to identify suspicious activity. The User-Agent header is checked

for known scanning and penetration testing tools such as sqlmap, nikto, and nmap. Additional headers are inspected for unusual values or potentially malicious content that may indicate an attack attempt.

4.4. URL and Query Parameter Analysis

The request URL and query parameters are examined using the Rule Engine. Before inspection, encoded values are decoded to prevent attackers from bypassing detection mechanisms through URL encoding techniques.

4.5. Request Body Inspection

For requests containing form data or JSON payloads, the request body is analyzed using the same detection rules applied during URL inspection. Each input value is scanned for malicious content to ensure that attacks hidden within form fields or API payloads are detected and prevented.

4.6. Decision Making, Logging, and Blacklisting

Based on the results of the previous inspection stages, the system decides whether to allow or block the request.

If the request is considered safe, it is forwarded to the application using the next() function and recorded as an allowed request.

If malicious activity is detected, the request is blocked and the attack details are logged using Winston. Information such as IP address, attack type, severity level, timestamp, and detected payload is stored for future analysis.

The system also monitors repeated attack attempts. IP addresses that continuously generate malicious requests are automatically added to the blacklist, preventing

future access without requiring manual intervention.

4.7. Real-Time Monitoring and Visualization

The final phase focuses on monitoring and reporting. The backend provides API endpoints that generate statistics related to total requests, blocked requests, attack categories, and suspicious IP addresses.

A React.js dashboard retrieves this information and displays it using charts, graphs, and tables. Administrators can view attack trends, monitor blocked requests, and analyze security events in real time, enabling faster response to potential threats.

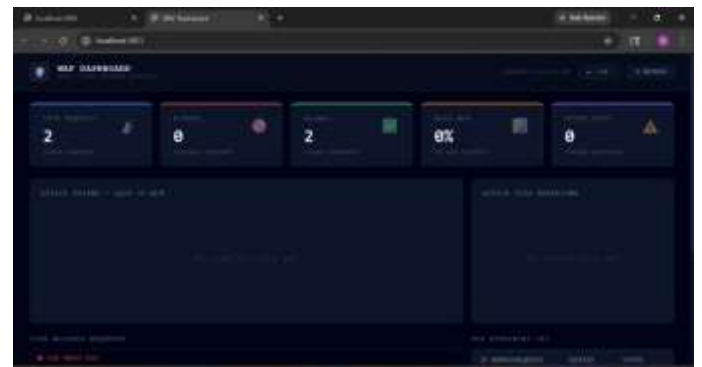


Fig1: This figure shows the before request

V. IMPLEMENTATION

5.1. Development Environment Setup

The backend was built using Node.js with the Express framework. The project was organized into separate folders for middleware, rule definitions, utility functions, and database models to keep the code modular and maintainable. Required packages including express, express-rate-limit, winston, helmet, cors, uuid, dotenv, and mongoose were installed. The frontend dashboard was built as a separate React application, with the recharts library added for charts and graphs.

5.2. Implementing the Rule Engine

The rule engine was implemented first since it forms the core detection logic. Six attack categories were defined — SQL Injection, Cross-Site Scripting, Path Traversal, Command Injection, LDAP Injection, and XML Injection — each with a set of regex patterns and an assigned severity level. A scanning function decodes the input value to handle URL-encoding evasion and tests it against every pattern in all categories, returning the matched attack type, severity, and location if a threat is found. Separate checks were also added for malicious user-agent strings such as sqlmap and nikto, and for suspicious header values.

5.3. Implementing the IP Manager

The IP manager module tracks three things using in-memory data structures — a blacklist of IP addresses with reasons and expiry times, a record of request timestamps per IP for rate limiting, and a record of blocked-request timestamps per IP for automatic blacklisting. Sliding-window logic was used so that only requests within the relevant time window (such as the last 15 minutes) are counted, with old entries automatically expiring.

5.4. Implementing the WAF Middleware Pipeline

The core WAF middleware was implemented as a single function that runs on every incoming request before it reaches any application route. It executes six stages in strict order — checking the IP blacklist, checking the rate limit, inspecting headers, scanning the URL and query parameters, scanning the request body, and finally allowing the request to proceed if it passes all checks. At any stage, if a threat is detected, the request is immediately blocked with an appropriate error response and the process stops there.

5.5. Implementing Logging and Database Persistence

A structured logging system was implemented using Winston, writing all requests to one log file and only

blocked/threat requests to a separate log file in JSON format. An in-memory statistics object was also maintained to track total requests, blocked requests, allowed requests, attack-type counts, and a timeline of blocked requests over the last 60 minutes for fast access by the dashboard.

For permanent storage, database models were created for attack records and blacklisted IPs, and connected to a local MongoDB database. Whenever a threat is blocked, the attack details — including IP address, attack type, severity, matched payload, and timestamp — are saved to the database asynchronously, ensuring a permanent historical record of all attacks.

5.6. Implementing the Server and Dashboard API

The main server file initializes the Express application, applies security-related middleware, registers the WAF middleware globally so it applies to all routes, and defines example protected application routes such as login, user listing, product listing, and search. A separate set of dashboard API routes was created to expose live statistics, the list of blacklisted IPs, the list of top attacking IPs, and endpoints to manually add or remove an IP from the blacklist.

5.7. Implementing the React Dashboard

The dashboard was built as a single-page React application that periodically fetches data from the backend's statistics API at fixed time intervals. The fetched data is displayed using summary cards for total, blocked, and allowed requests; a line chart showing attack volume over the last 60 minutes; a pie chart showing the breakdown of attack types; a table listing the top attacking IP addresses; and a live-updating feed of the most recently blocked requests. A pause and refresh control was also added so the user can stop automatic polling when needed.

5.8. Testing and Validation

An automated test script was implemented to validate the system. It sends a series of predefined requests

covering all six attack categories along with several legitimate requests to the running server, and automatically checks that every attack request is blocked while every legitimate request is allowed through, confirming both the accuracy and usability of the WAF.

VI. RESULTS AND ANALYSIS

6.1. Attack Detection Results

The firewall was tested against various attack payloads representing SQL Injection, Cross-Site Scripting (XSS), Path Traversal, Command Injection, LDAP Injection, and XML Injection attacks. The results showed that the system successfully identified and blocked malicious requests while allowing legitimate traffic to access the application normally.

Common attack payloads such as SQL Injection queries, malicious script tags, encoded path traversal strings, and command injection attempts were detected by the rule engine and blocked with appropriate HTTP responses. At the same time, normal user requests and API calls were processed without interruption, indicating that the detection mechanism was functioning as expected.

6.2. Rate Limiting Results

The rate limiting module was evaluated by sending a large number of requests from the same IP address within a short period of time. Requests within the configured limit were processed successfully, while requests exceeding the threshold were blocked with a 429 (Too Many Requests) response.

The results confirmed that the rate limiting mechanism effectively prevents excessive traffic, helping to reduce the risk of brute-force attacks and request flooding.

6.3. Automatic IP Blacklisting Results

Repeated malicious requests were generated from the same IP address to verify the automatic blacklisting functionality. After exceeding the predefined threshold, the IP address was automatically added to the blacklist.

Once blacklisted, all subsequent requests from the same source were immediately rejected without undergoing further inspection. This behavior demonstrated the effectiveness of the automated response mechanism in handling persistent attackers.

6.4. Logging and Database Storage Results

All detected threats were successfully recorded using the Winston logging framework. Security events were stored in structured log files containing information such as timestamp, IP address, attack type, and severity level.

Additionally, the attack records were stored in MongoDB for future analysis. Verification of the stored data confirmed that the database records matched the generated log entries, ensuring reliable and consistent security event tracking.

6.5. Dashboard Monitoring Results

The React.js monitoring dashboard successfully displayed real-time security statistics retrieved from the backend APIs. Information including total requests, blocked requests, attack categories, attack trends, and suspicious IP addresses was updated dynamically during testing.

The dashboard provided a clear visualization of security events and enabled easy monitoring of application activity and attack patterns.

6.6. Performance Analysis

Performance testing indicated that the firewall introduced minimal overhead to normal application requests. Since the detection process is based on lightweight regular expression matching and early-stage request filtering, request processing remained efficient even when security checks were applied.

Requests originating from known malicious IP addresses were blocked at the initial stages of inspection, reducing unnecessary processing and improving overall system efficiency.

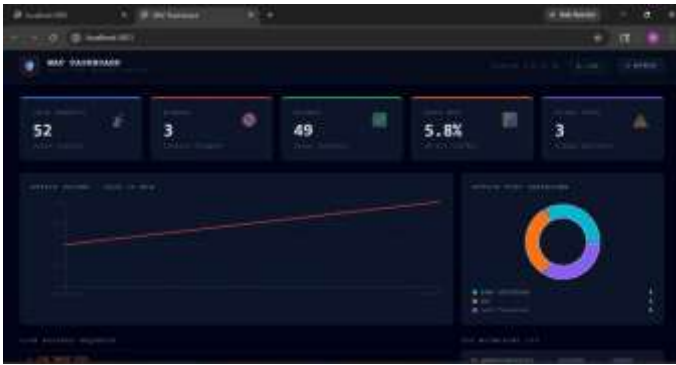


Fig 2: This figure shows the dashboard of WAF

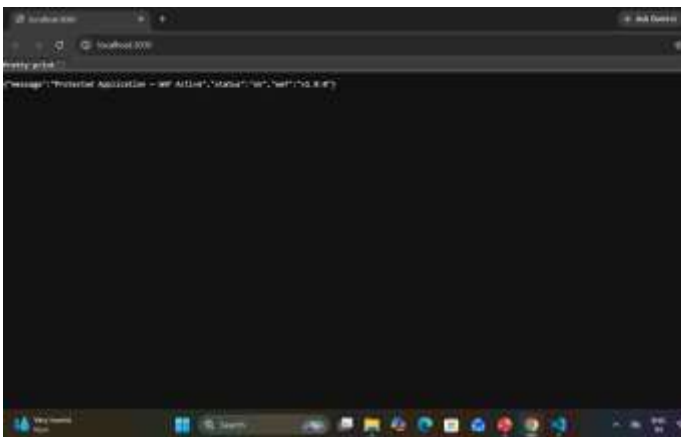


Fig3: figure shows the backend result

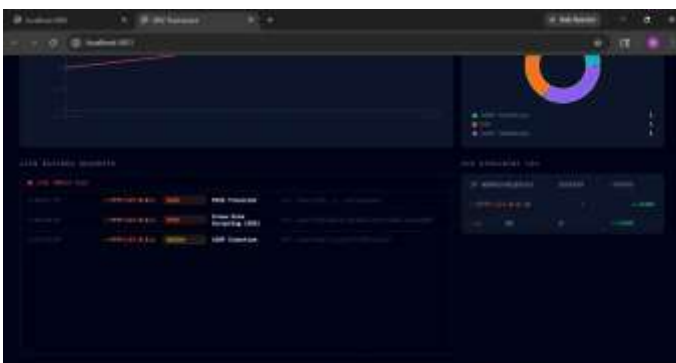


Fig4: figure shows the attacks

VII. CONCLUSION

This project presents the design and implementation of a custom Web Application Firewall (WAF) using Node.js, Express.js, React.js, and MongoDB. The developed system provides an additional layer of security by inspecting incoming HTTP requests before they reach the web application. Through a multi-stage

inspection process, the firewall helps identify and block various malicious activities that may compromise application security.

The implemented rule engine successfully detects common web application attacks such as SQL Injection, Cross-Site Scripting (XSS), Path Traversal, Command Injection, LDAP Injection, and XML Injection. Additional security features including rate limiting, automatic IP blacklisting, structured logging, and attack data storage further enhance the effectiveness of the system.

The integration of MongoDB enables the storage of security events for future analysis, while the React-based dashboard provides real-time visibility into attack statistics and suspicious activities. These features help administrators monitor security threats and respond to potential risks more effectively.

The project demonstrates how modern web technologies can be used to build a practical, customizable, and cost-effective security solution. It also provides valuable insights into web application security, attack detection techniques, and defensive security mechanisms. Overall, the developed WebShield system serves as a strong foundation for improving web application security and can be extended with advanced features as future security requirements evolve.

VIII. REFERENCES

- [1] OWASP Foundation, "OWASP Top Ten Web Application Security Risks," Open Web Application Security Project, 2021. [Online]. Available: <https://owasp.org/www-project-top-ten/>
- [2] R. Fielding and J. Reschke, "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing," Internet Engineering Task Force (IETF), RFC 7230,

June 2014. [Online]. Available:
<https://tools.ietf.org/html/rfc7230>

[3] W. Stallings, Cryptography and Network Security: Principles and Practice, 8th ed. Hoboken, NJ, USA: Pearson, 2019.

[4] OWASP Foundation, "SQL Injection Prevention Cheat Sheet," Open Web Application Security Project, 2023. [Online]. Available:
https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html

[5] OWASP Foundation, "Cross Site Scripting Prevention Cheat Sheet," Open Web Application Security Project, 2023. [Online]. Available:
https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html

[6] OWASP Foundation, "Path Traversal," Open Web Application Security Project, 2023. [Online]. Available:
https://owasp.org/www-community/attacks/Path_Traversal

[7] Node.js Foundation, "Node.js Documentation," OpenJS Foundation, 2023. [Online]. Available:
<https://nodejs.org/en/docs/>

[8] Express.js Contributors, "Express.js Web Framework Documentation," OpenJS Foundation, 2023. [Online]. Available: <https://expressjs.com/>

[9] W. G. J. Halfond, J. Viegas, and A. Orso, "A Classification of SQL Injection Attacks and Countermeasures," in Proc. IEEE Int. Symp. Secure Software Engineering, Arlington, VA, USA, Mar. 2006, pp. 13–15.

[10] P. Bisht and V. N. Venkatakrishnan, "XSS-GUARD: Precise Dynamic Prevention of Cross-Site Scripting Attacks," in Proc. 5th Int. Conf. Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA), Paris, France, 2008, pp. 23–43.

[11] National Institute of Standards and Technology (NIST), "Guidelines on Firewalls and Firewall

Policy," Special Publication 800-41 Rev. 1, Gaithersburg, MD, USA, 2009. [Online]. Available:
<https://csrc.nist.gov/publications/detail/sp/800-41/rev-1/final>

[12] React Contributors, "React — A JavaScript Library for Building User Interfaces," Meta Open Source, 2023. [Online]. Available: <https://react.dev/>

[13] Winston Contributors, "Winston — A Logger for Just About Everything," npm Registry, 2023. [Online]. Available:
<https://github.com/winstonjs/winston>

[14] MongoDB Inc., "MongoDB Documentation," MongoDB, 2023. [Online]. Available:
<https://www.mongodb.com/docs/>

[15] D. Stuttard and M. Pinto, The Web Application Hacker's Handbook: Finding and Exploiting Security Flaws, 2nd ed. Indianapolis, IN, USA: Wiley, 2011.