

Article

Log Analysis and Evaluation of DoS Attacks in ModSecurity-Based Web Application Firewalls

Mustafa Kara

Department of Computer Engineering, Turkish Air Force Academy National Defence University,
Istanbul 34149, Türkiye; mustafa.kara@msu.edu.tr

Abstract

Within the scope of this article, security threats against web applications and the effects of Web Application Firewall (WAF) usage on security were examined. The aim of the study is to evaluate the effectiveness of the open-source ModSecurity-based WAF structure against application layer attacks and Denial of Service (DoS) attacks. For this purpose, a test environment with an Ubuntu and Apache2-based reverse proxy architecture was created, and HTTP traffic was analyzed using the OWASP Core Rule Set (CRS), custom security rules, and the mod_qos module. In the experimental studies, SQL Injection, Cross-Site Scripting (XSS), and DoS attack scenarios were applied, and custom ModSecurity rules operating on HTTP parameters, URI, and request body were also developed and tested. The obtained results were evaluated through log records and system performance metrics. When ModSecurity was disabled, it was observed that total CPU usage reached the level of 83% during the DoS attack. After ModSecurity and mod_qos configurations were enabled, CPU usage was determined to have decreased to the level of 25%. Log analyses showed that ModSecurity and OWASP CRS rules successfully analyzed attack traffic and detected abnormal request behaviors and protocol violations. In addition, it was verified that the developed custom rules successfully blocked requests with the HTTP 403 status code in tests performed on HTTP parameters, URI, and request body. It was observed that while the security mechanisms limited attack traffic, they did not block legitimate user access and normal client requests continued to be processed successfully. The experimental results show that the ModSecurity-based WAF architecture provides an effective security solution in terms of detecting attack traffic, protecting system resources, and ensuring service continuity.

Keywords: Web Application Firewall; ModSecurity; OWASP Core Rule Set; DoS Attack; Log Analysis



Academic Editor: Christos Bouras

Received: 31 May 2026

Revised: 10 June 2026

Accepted: 22 June 2026

Published: 29 June 2026

Copyright: © 2026 by the author.

Licensee MDPI, Basel, Switzerland.

This article is an open access article distributed under the terms and conditions of the [Creative Commons Attribution \(CC BY\)](https://creativecommons.org/licenses/by/4.0/) license.

1. Introduction

Web applications have become one of the fundamental components of the digital ecosystem today [1]. Many transactions such as banking, healthcare, education, public services, and electronic commerce are carried out through web-based systems [2]. Developments in Internet technologies and the widespread adoption of online services have increased dependence on web applications. While this situation has increased the traffic intensity of web applications, it has also significantly raised security requirements [3].

Web applications operate at the application layer of the OSI model. These applications manage data exchange by processing user requests. Since traditional network firewalls operate at the network layer, they cannot analyze traffic occurring at the application level

in detail [4]. Therefore, they cannot effectively prevent application layer attacks such as SQL Injection, Cross-Site Scripting, and similar threats [5]. The increase in application-level threats has increased the need for security solutions specifically designed for web applications [6].

Since web applications are accessible over the Internet, they are generally located in the Demilitarized Zone. Unlike internal network systems, they are exposed to many security threats because they receive traffic directly from external networks. Security vulnerabilities in web applications may lead to unauthorized access to sensitive information, modification or deletion of data, and unauthorized use of user accounts [7]. The resulting security problems can affect not only the applications themselves but also user data and institutional information assets.

The increasing security threats against web applications have been demonstrated by various studies [8,9]. Web application security in the information technology sector has shown a negative trend compared to previous years. It has been determined that approximately half of the examined applications have a low level of security. In addition, it has been identified that a large portion of the applications contain security vulnerabilities based on code injection [10]. Approximately twenty-nine percent of these vulnerabilities may lead to serious consequences, such as attacks against internal networks. It has been reported that critical web application security vulnerabilities are at the level of sixty-two percent. The same report stated that eighty-three percent of web applications contain security vulnerabilities resulting from improper security configurations [11].

Attack trends targeting web applications are also observed in distributed denial-of-service attacks [12]. Figure 1 presents HTTP(S)-based DDoS attack statistics reported by Kaspersky for the third quarter of 2022. HTTP(S)-based attack rates increased significantly during the third quarter of 2022 compared to previous periods. The increase in HTTP(S) attacks, which reached approximately one hundred and forty-seven percent, indicates that the application layer has become an important target for attackers. The results demonstrate the importance of security mechanisms at application level [13].

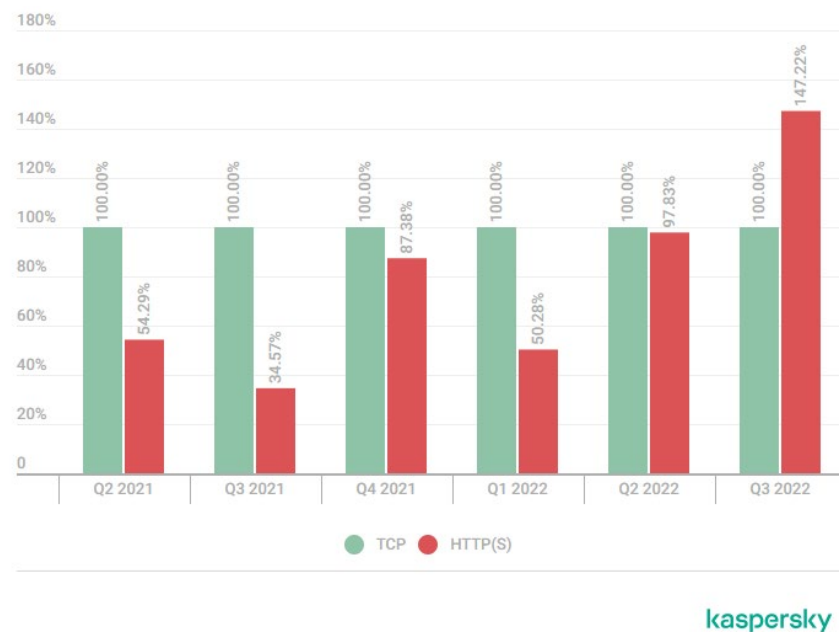


Figure 1. HTTP(S) DDoS attack graph for the third quarter of 2022 obtained with Kaspersky [13].

Recent threat intelligence reports indicate that DDoS attacks continue to increase in both scale and sophistication. According to Cloudflare's DDoS Threat Report for Q3 2025 [14], 8.3 million DDoS attacks were mitigated during a single quarter, representing a

15% increase quarter-over-quarter and a 40% increase year-over-year. In addition, approximately 2.4 million HTTP DDoS attacks were recorded during the same period, highlighting the continued prevalence of application-layer threats. Similarly, StormWall reported that global DDoS activity increased by 168% year-over-year in the first quarter of 2026, with a significant rise in multi-vector attacks targeting critical sectors such as telecommunications, finance, and government services [15]. These results demonstrate that HTTP(S)-based and application-layer attacks remain a significant cybersecurity challenge and emphasize the importance of effective protection mechanisms such as Web Application Firewall (WAF).

WAF are widely used to prevent attacks against web applications. Unlike network firewalls, WAFs can analyze HTTP traffic and filter attacks occurring at the application level [4]. In addition to protecting web applications, these systems also protect the servers, databases, and user data located behind the applications. Furthermore, they increase the overall security level of systems by reducing the impact of common security vulnerabilities [16].

ModSecurity is one of the most widely used WAF solutions due to its open-source structure, OWASP Core Rule Set (CRS) support, and customizable rule architecture [17]. ModSecurity can analyze HTTP traffic in detail and allows security policies to be extended according to requirements through user-defined custom rules. OWASP CRS is an open-source rule set that aims to detect common web application attacks such as SQL Injection, Cross-Site Scripting, and similar attacks, and it is widely used together with ModSecurity.

There are different studies in the literature on WAFs. Sobola et al. [18] examined throughput, transaction rate, and concurrency values under DDoS attacks on a ModSecurity firewall configured with the OWASP CRS. In the study conducted by Anuvashini et al. [17], the performance and attack detection success of the ModSecurity WAF integrated with OWASP CRS were evaluated. In the study, attack payloads that could not be detected by the existing rule set were analyzed, and 146 new custom rules were developed to address these deficiencies, significantly increasing detection accuracy. Muzaki et al. [19] analyzed SQL Injection and Cross-Site Scripting attacks by using the ModSecurity solution in a reverse proxy server structure. Clincy and Shahriar [20] showed that default firewall settings may cause security vulnerabilities. Erden [21] compared the features of 19 different open-source WAFs and evaluated the strengths and weaknesses of these solutions. Praseed and Thilagam [22] proposed an approach using HTTP request patterns for the early detection of application layer DDoS attacks. In their study, they aimed to enable faster detection of attacks by developing an Early Detection Module that can be integrated into existing WAFs.

When the studies in the literature are examined, it is seen that the research largely focuses on attack detection mechanisms, performance evaluations, and firewall comparisons. However, it is observed that studies on the detailed examination of audit log records generated by ModSecurity, the evaluation of the log-level behavior of OWASP CRS rules, and the analysis of the effects of custom security rules on traffic are limited. This situation reveals the importance of log analysis studies in terms of better understanding security incidents and developing security policies.

The aim of this study is to analyze the log records generated on the ModSecurity-based WAF and to evaluate the behavior of security mechanisms under different attack scenarios. In this context, the open-source ModSecurity solution was preferred, and HTTP traffic was examined using the OWASP ModSecurity Core Rule Set and custom security rules developed by the user. Within the scope of the study, a test environment with a reverse proxy architecture was created, application layer attacks and DoS attack scenarios were carried out, and the obtained audit log records and system performance data were analyzed. Thus, it was aimed to reveal the effects of both default security rules and custom rules on attack traffic and to contribute to reducing security risks in web applications.

The remainder of this paper is organized as follows. Section 2 presents the architecture of WAFs and their fundamental security mechanisms. Section 3 discusses open-source WAF solutions. Section 4 describes the ModSecurity installation process and the experimental test environment. Section 5 presents the ModSecurity log analysis and evaluates the results obtained from the conducted attack scenarios. Finally, Section 6 concludes the paper and provides recommendations for future work.

2. Web Application Firewall Architecture and Fundamental Security Mechanisms

2.1. Web Application Firewall

The increasing development and widespread use of web applications have caused a significant increase in attacks targeting the application layer [23]. Web applications continuously perform data exchange between users and servers. Secure management of this data exchange is critically important for the protection of web applications. For this purpose, the WAF solution has been developed.

A WAF is a security mechanism that filters incoming and outgoing data and can block network traffic considered dangerous according to the applied rules [19]. Web applications are located at the application layer of the OSI reference model. Security vulnerabilities in applications have led to the emergence of attacks targeting the application layer. Since network firewalls cannot analyze threats at the application layer in detail, the WAF solution has emerged.

Figure 2 shows the HTTP GET request, HTTP POST request, and the payload structure carried within the POST request.

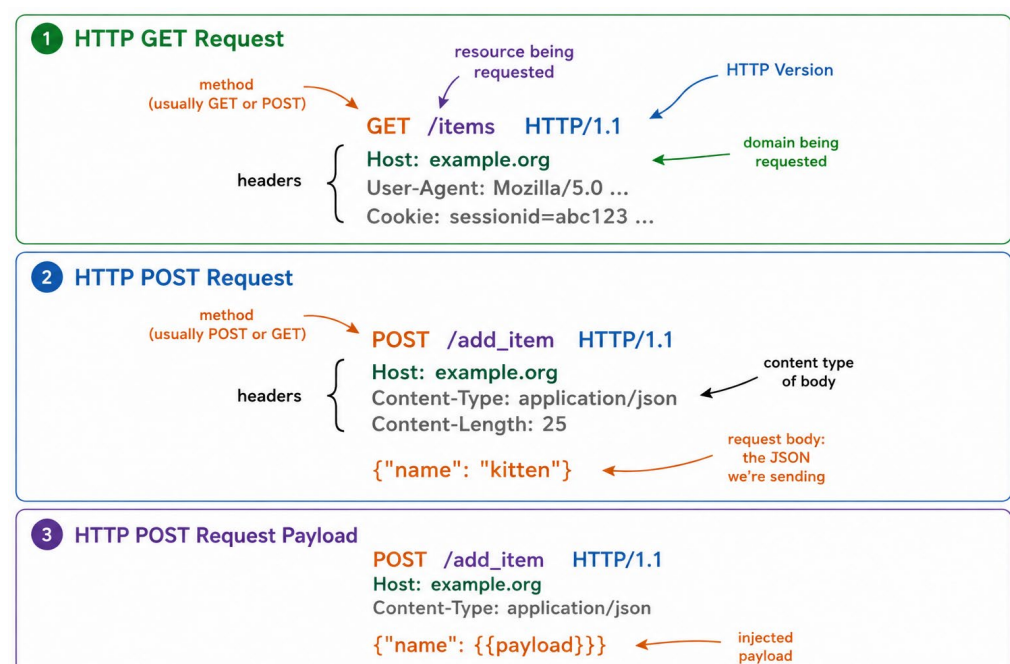


Figure 2. HTTP Request Components and Payload Analysis.

The WAF monitors incoming and outgoing HTTP/HTTPS traffic over the Internet before it reaches the application server and examines the contents of incoming request packets in detail. In this context, the main components analyzed are GET and POST requests.

GET requests are generally used to request data from the server. A GET request consists of domain name, source, method, HTTP version, and header fields. Domain name refers to the requested domain name, source refers to the resource to be accessed, and

method refers to request types such as GET, POST, or PUT. Headers contain additional information sent to the server [24]. POST requests are used to send data to the server. Unlike GET requests, POST requests may include additional header fields such as Content-Type and Content-Length and mostly contain a request body. This body enables the transmission of user data or application parameters.

In HTTP requests, the URL, request body, cookies, and other header fields can be exploited by attackers. Therefore, one of the main objectives of WAFs is to determine whether malicious content or payload exists in different parts of the request. Payload values added to a parameter within a POST request can be used in attacks such as SQL Injection, Cross-Site Scripting, or command injection. Therefore, detailed analysis of HTTP request parameters is critically important.

A WAF creates a security layer between the Internet and the web application. Thus, it can filter malicious HTTP or HTTPS traffic that may cause the exploitation of vulnerabilities in applications before it reaches the application server. This approach protects not only the web application but also application servers and sensitive user data. A set of rules called policies is defined on the WAF. With the help of these rules, fake or malicious requests are detected. When the condition defined in a rule occurs, the relevant action is applied. The effectiveness of WAFs largely depends on the quality of the rule sets used. Rule definitions used in different products may vary [25].

The basic operating mechanism of WAFs consists of the following stages [26].

1. Parsing incoming HTTP packets into their components,
2. Normalization of the data (decoding),
3. Execution of filtering rules (regex),
4. Deciding whether the incoming traffic is malicious or harmless.

In the first stage, HTTP packets are parsed and the fields to be analyzed are determined. Then, encoded data is decoded and converted into a standard format. In the next stage, the defined filtering rules are executed. In the final stage, whether the traffic is secure or not is determined according to the obtained results.

WAFs have different features from network firewalls. They can enable many types of attacks to be prevented. They also support the inspection and limitation of HTTP requests such as GET and POST. In addition, rules can be defined for fields such as file transfer size, URL parameter length, and similar areas [27].

A network firewall and a WAF work together to form complementary security solutions. The network firewall protects the data flow between servers. The WAF filters application traffic. The network firewall helps prevent threats located at the lower layers of the OSI model. However, it cannot detect attacks occurring at the application layer, such as XSS, SQL Injection, and HTTP DDoS. Detecting and preventing these attacks is one of the main tasks of the WAF.

2.2. Web Application Firewall Topologies

As shown in Figure 3, WAF that will first handle incoming requests can be deployed in 4 ways.

1. **Inline (Bridge):** The WAF is placed in front of the web server. When the client wants to connect to the web server, the request first passes through the WAF and then accesses the web application. If the request is blocked, the fake request cannot access the host where the web application is located.
2. **Offline (Out of Band):** Requests can directly reach the web server, and a copy of the requests is forwarded to the WAF. If the response that the web firewall sends for malicious requests remains after the response of the web server, the web firewall may become nonfunctional [28].

3. **Integrated (Agent):** The WAF is located within the web application as a program library. The client connects to the web application, and the web application sends the requests to the firewall [25].
4. **Reverse Proxy:** The WAF is located between the client and the web server. The WAF and the web server are separate hosts. The client connects directly to the IP address of the WAF. The WAF connects to the IP address of the web server. The request passes through the WAF, and if a response returns from the web server, the WAF also returns this response to the client. If the request is blocked, fake requests can never go to the web server host. The purpose is to hide the web server [19]. The client can never send a request directly to the web server; instead, it is directed to the WAF, which has the role of a proxy server. The client only knows the IP address information of the WAF and can never know the location of the web server.

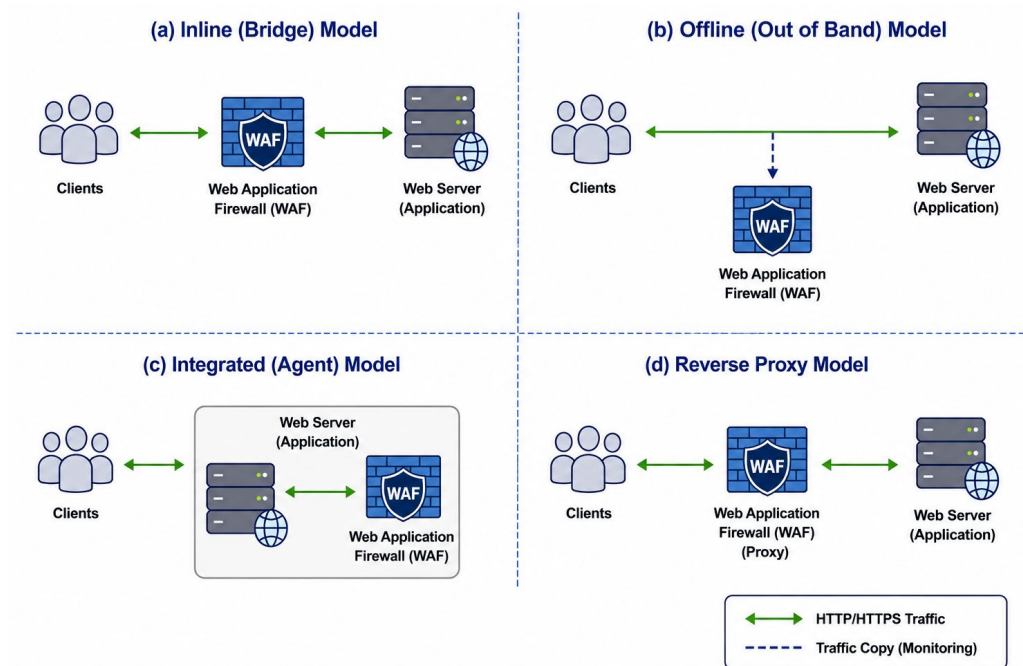


Figure 3. Web Application Firewall Topologies: (a) Inline Model, (b) Offline Model, (c) Integrated Model, (d) Reverse Proxy Model.

2.3. Web Application Firewall Operating Models

WAFs use different security models to distinguish normal requests from malicious requests. These models are basically divided into two as Positive Security Model and Negative Security Model, depending on the type of rule defined.

Positive Security Model is an allowlist-based security approach. In this model, only web traffic permitted in the rules is allowed to pass. All other web traffic is blocked [20]. The allowlist model accepts only predefined web traffic that is considered secure. For example, it can be configured to allow only HTTP GET requests from specific IP addresses. This approach can be effective in preventing possible cyberattacks. However, if the permitted traffic is not defined correctly, it may cause many legitimate requests to be blocked. Therefore, allowlist-based firewalls are considered more suitable especially for web applications located in the internal network.

Negative Security Model is a blocklist-based security approach. In this model, all web traffic is allowed, and only web traffic representing malicious rules is blocked [20]. The blocklist model uses signatures that define malicious traffic to prevent attacks that exploit web application security vulnerabilities. Therefore, it is considered a suitable option for web applications on the Internet. It can also be effective against DDoS attacks, which are one of

the most common types of attacks. For example, a rule that blocks all `<script>*</script>` inputs can be shown as an example of a blocklist-based security approach [29].

In practice, many WAFs do not depend on only a single security model. Negative and positive rule definitions are generally used together. This approach creates a more balanced security structure by both blocking known malicious requests and limiting only traffic considered secure.

2.4. X-Forwarded-For

One of the important settings to be considered in the use of a WAF is the X-Forwarded-For information located in the HTTP request header. HTTP headers are used to transmit additional information in the communication between the client and the server. These headers enable additional information belonging to the client or the server to be carried within an HTTP request or response. X-Forwarded-For is a request-type HTTP header field. If the client connects to a website through a proxy, it is used to identify the original IP address of the client. In other words, the X-Forwarded-For header enables the real source IP address of the request to be determined.

This header becomes especially important in structures where the WAF is positioned as a Reverse Proxy. In the Reverse Proxy model, the client connects not directly to the web server but to the WAF. The WAF then forwards the request to the web server. In this case, the IP address of the WAF may be seen in the source IP field of the request reaching the web server instead of the real IP address of the client. Therefore, in the Reverse Proxy architecture, the X-Forwarded-For feature must be enabled on the WAF to preserve the real IP address of the client. Thus, the web server can correctly determine from which client the request comes. The X-Forwarded-For header is generally used in the format X-Forwarded-For: <client>, <proxy>.

2.5. False Positive and False Negative Conditions

False Positive is the detection of a situation that does not actually exist as a security problem. From the perspective of a WAF, a False Positive occurs when a legitimate request sent by a user is evaluated as an attack and the user is blocked. This situation may especially arise in areas where user inputs are analyzed. For example, entering different characters into an input field where a phone number is expected may be perceived as suspicious behavior by the WAF. However, this does not always indicate a real attack. A high false positive rate poses a significant threat to a properly functioning website. Blocking a large number of users or frequently redirecting users to CAPTCHA pages negatively affects the user experience. This situation may cause financial losses and damage the reputation of the website. Therefore, it is necessary to prevent the activation of conditions that produce a high rate of false positives.

To evaluate the effectiveness of a WAF, two different conditions should be tested for each type of attack. These are true positives and false positives. A true positive test determines whether attack payloads are correctly detected. A false positive test reveals whether acceptable payloads are incorrectly identified as attacks. Figure 4 shows False Positive and True Positive conditions together. This illustration explains the difference between correctly detecting attack traffic and incorrectly blocking legitimate traffic by the firewall.

A False Negative condition is when a malicious request cannot be detected by the WAF and can reach the web application. This situation is considered one of the most critical problems in terms of security. Because although the attack successfully reaches the target system, it is not noticed by the security mechanism. Therefore, the success of a WAF is evaluated not only by the false positive rate but also by the low false negative rate.

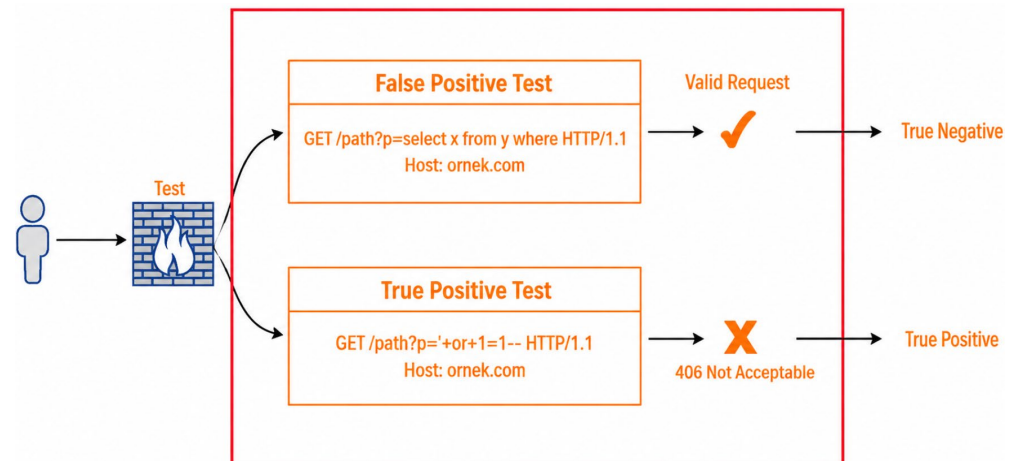


Figure 4. False Positive and True Positive Conditions.

2.6. OWASP

The Open Web Application Security Project (OWASP) is an online community that produces freely available articles, methodologies, documents, tools, and technologies in the field of web application security [29,30]. On its website, it explains in detail the vulnerabilities in web applications, how these vulnerabilities occur, which weaknesses they originate from, how these vulnerabilities can be exploited, and how these vulnerabilities can be prevented. OWASP leads security-related infrastructures. Through hundreds of local chapters worldwide, tens of thousands of members, and leading education and training conferences, the OWASP infrastructure is a resource for developers and technology experts to secure the Web.

OWASP publishes a Top 10 list of the most critical issues for web application security vulnerabilities every few years. The most recently published Top 10 list for 2021 is [31]:

1. Broken Access Control
2. Cryptographic Failures
3. Injection
4. Insecure Design
5. Security Misconfiguration
6. Vulnerable and Outdated Components
7. Identification and Authentication Failures
8. Software and Data Integrity Failures
9. Security Logging and Monitoring Failures
10. Server-Side Request Forgery.

The OWASP ModSecurity Core Rule Set (CRS) is a set of generic attack detection rules that can be used with ModSecurity or compatible WAFs. This rule set helps protect web servers. These rules are continuously updated by OWASP. CRS aims to protect web applications from a wide range of attacks, including the OWASP Top 10, with minimal false alerts. CRS provides protection against many common attack categories such as SQL Injection, Cross-Site Scripting, and Local File Inclusion [32]. It is the most robust open-source WAF rule set available on the Internet.

The OWASP ModSecurity Core Rule Set is a solution that provides web security and covers web application security vulnerabilities by protecting websites or web applications from malicious attacks that may result in significant losses for institutions and companies [33].

3. Open-Source Web Application Firewalls and OWASP Core Rule Set

3.1. ModSecurity

Open-source WAFs are widely used due to their cost advantage, customizable rule structures, and ability to integrate with different web servers. Among these solutions, ModSecurity stands out because it can monitor HTTP traffic in real time, perform logging, and work with web servers such as Apache, NGINX, and IIS. ModSecurity can be deployed in reverse proxy or embedded mode; it offers security features such as regular expression-based filtering, URL and Unicode encoding validation, auditing, prevention of null byte attacks, and masking of server identity.

Apart from ModSecurity, there are also different open-source WAF solutions such as NAXSI, Shadow Daemon, OWASP Coraza, and Vulture [34]. NAXSI is a WAF solution that works with NGINX and was developed especially against XSS and SQL Injection attacks. Instead of classical signature-based approaches, it uses a security model based on defining permitted requests. Shadow Daemon focuses on filtering the dangerous parts of the request rather than completely blocking malicious requests and supports PHP-, Perl-, and Python-based applications. OWASP Coraza is a modern WAF framework developed in the Go language and supporting ModSecurity's SecLang rule language. Its ability to work compatibly with OWASP CRS makes Coraza one of the alternative open-source solutions to ModSecurity. Vulture is an open-source WAF platform that works with a ModSecurity-based reverse proxy architecture and offers additional features such as TLS, authentication, caching, and traffic distribution.

Although these solutions have different architectural and security approaches, their common purpose is to detect and filter malicious HTTP/HTTPS traffic targeting web applications. The main reason for preferring Mod-Security within the scope of this study is that it can be widely used with the OWASP CRS, can be easily integrated with the Apache web server, produces detailed audit logs, and allows the development of user-defined custom rules. Compared with lightweight alternatives such as NAXSI and Shadow Daemon, ModSecurity offers more comprehensive request inspection and rule customization capabilities. However, this flexibility may require additional tuning to reduce false positives and maintain stable performance.

3.2. OWASP Core Rule Set

CRS is a set of predefined security rules that analyzes web traffic to detect and block malicious requests. Paranoia Level (PL) is the security sensitivity level used within CRS. PL1 is the default protection level and aims to detect common web attacks with a low false positive rate. PL2 includes additional security controls and detects more attack techniques; however, the probability of false positives increases. PL3 applies stricter inspections and focuses on detecting complex or obfuscated attacks, generally requiring additional tuning. PL4 is the highest security level; it uses the strictest rules, aims to provide maximum protection, and should be configured carefully due to its high false positive rate.

Increasing the CRS Paranoia Level generally enhances detection effectiveness by activating additional inspection rules and increasing the sensitivity of request analysis. However, this improvement is often accompanied by a higher false-positive rate and additional tuning requirements. Therefore, selecting an appropriate paranoia level involves balancing security effectiveness and operational usability. For this reason, PL1 is commonly used as a baseline configuration, whereas higher levels (PL2–PL4) are preferred in environments requiring stricter security controls.

CRS rules do not make decisions based solely on a single rule match. Each rule match generates a specific anomaly score. When the total score generated for a request exceeds the defined threshold value, the request is considered malicious and is blocked. CRS uses rules

with different severity levels to determine the risk level of a request. A match at the Critical level generates 5 points, the Error level generates 4 points, the Warning level generates 3 points, and the Notice level generates 2 points. When a request matches multiple rules, these scores are accumulated, and if the total score exceeds the defined threshold, the request may be considered malicious and blocked. In the default CRS configuration, the inbound anomaly threshold value is defined as 5. If the total score exceeds this value, the request is blocked [17].

Figure 5 shows the anomaly scoring-based decision mechanism used by OWASP CRS. The HTTP request coming from the client is first handled by ModSecurity and analyzed using CRS rules. As a result of the checks performed on request headers, URI parameters, and request body, each matched rule generates an anomaly score depending on its severity level. The obtained scores are accumulated, and the total anomaly score is calculated.

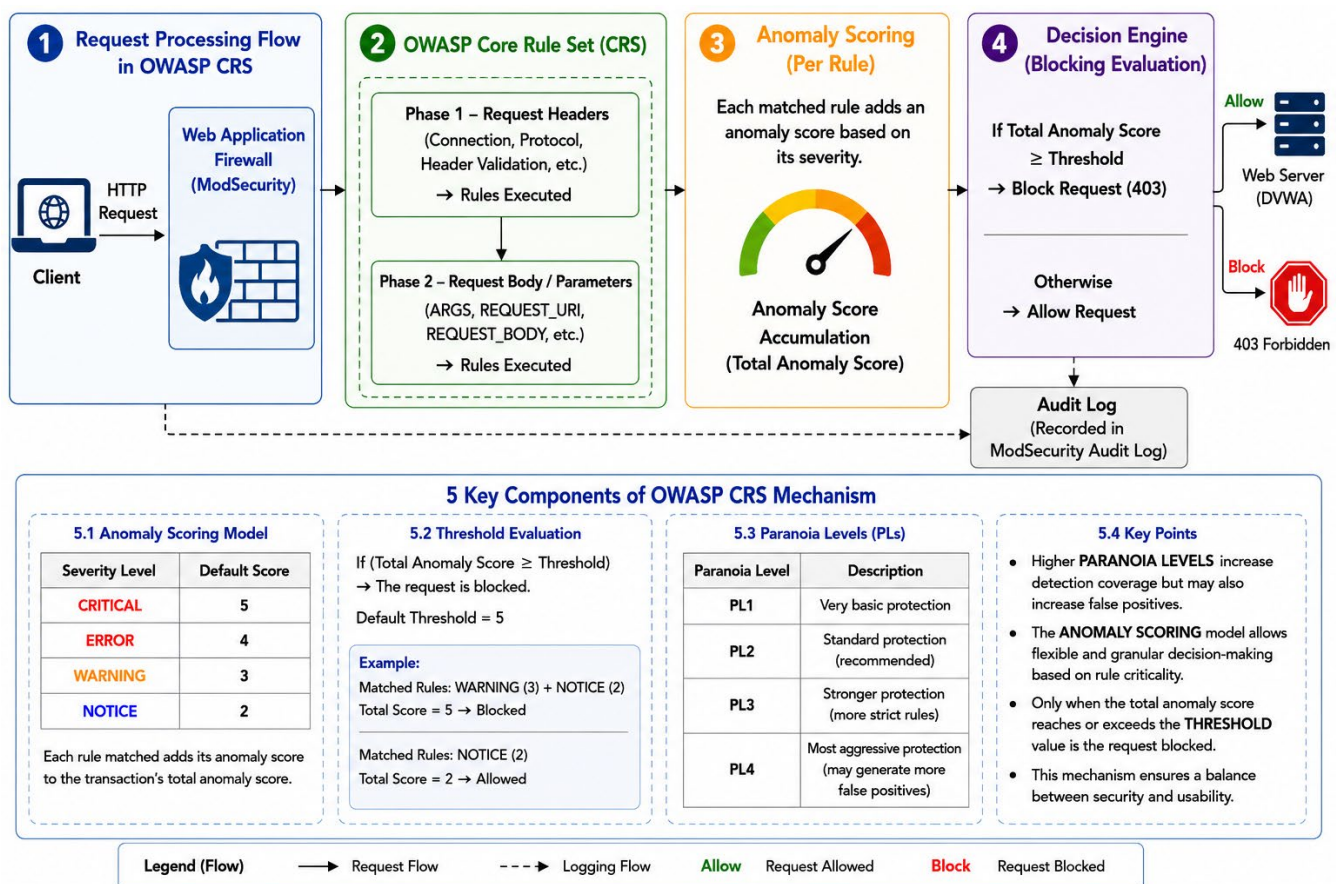


Figure 5. OWASP CRS Working Mechanism.

The calculated value is compared with the configured threshold value. If the total score reaches or exceeds the threshold value, the request is blocked with an HTTP 403 Forbidden response; otherwise, it is forwarded to the web application. This approach provides a more flexible and reliable attack detection mechanism based on multiple rule evaluation rather than decision-making based on a single rule match.

4. ModSecurity Installation and Test Environment

4.1. Test Environment and System Architecture

In this section, the experimental environment created to test the ModSecurity-based WAF, the system components used, and the configuration steps are explained. The test environment was designed to enable the examination of the behaviors of both the default

OWASP CRS rules and the specially defined rules by passing attack traffic through the WAF. In this context, a reverse proxy architecture was preferred, and the vulnerable DVWA environment was used as the web application.

The network architecture test environment established for the WAF log analysis study is shown in Figure 6. Open-source software components were preferred in the test environment created for the experimental study. On the WAF server, Apache HTTP Server 2.4.63 and ModSecurity 2.9.8 were run on the Ubuntu Server 24.04.2 LTS operating system. OWASP Core Rule Set 4.10.0 was used to apply security rules. For the implementation of attack scenarios, the Kali Linux 2025.2 system was preferred, and the Metasploitable 2 virtual machine hosting the DVWA application was used as the target system. A reverse proxy architecture was implemented to enable all HTTP traffic to pass through the WAF and be analyzed.

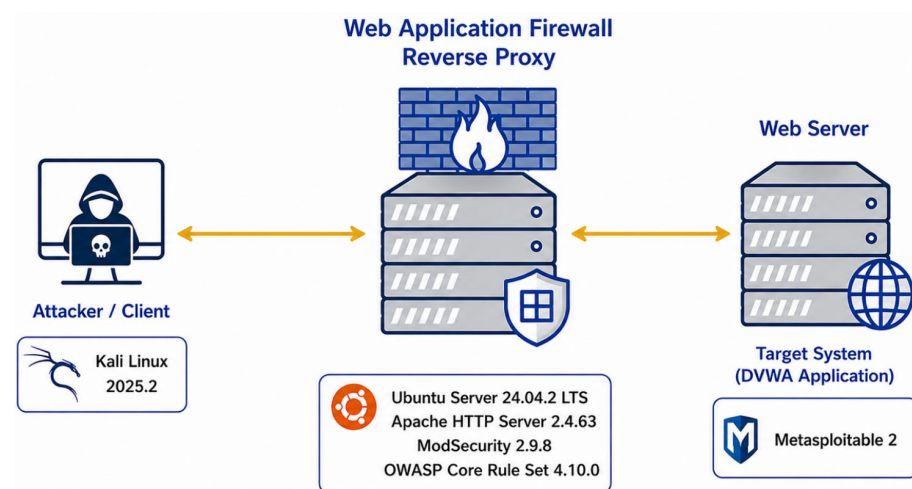


Figure 6. Test Environment Network Topology.

The WAF server was configured on the Ubuntu Server operating system, and the Apache2 web server and ModSecurity module were installed to operate together. The reverse proxy topology was preferred for the server serving as the web firewall and was configured accordingly. It was configured so that incoming traffic would first be handled by the web firewall and then forwarded to the web server in the background. The main reason for preferring this architecture is to prevent client traffic from reaching the web server directly and to ensure that all HTTP requests are first analyzed by ModSecurity.

The web server is Metasploitable 2, a vulnerable Ubuntu Linux virtual machine designed to test common security vulnerabilities. Tests will be performed on DVWA (Damn Vulnerable Web Application), which is the web application located on the Metasploitable 2 machine. After ModSecurity was installed, the OWASP ModSecurity Core Rule Set was configured.

The IP information in the established test environment is presented in Table 1.

Table 1. Server and IP information used in the experimental study.

Servers	IP Addresses
WAF Server (Apache2, ModSecurity)	192.168.56.114
Web Server (Metasploitable 2)	192.168.56.104
Kali Linux Server	192.168.56.108

4.2. ModSecurity and OWASP CRS Configuration

The 000-default.conf configuration file located under the /etc/apache2/sites-available/ directory must be modified so that the Apache2 web server can operate as a reverse proxy.

Within this file, the ProxyPreserveHost On directive is defined under the VirtualHost block to preserve the Host header received from the client. In addition, the ProxyPass and ProxyPassReverse directives are used to forward requests received from the client to the web server running in the background and to relay the responses returned from the web server back to the client. In this configuration, the specified IP address must belong to the web server to which the traffic will be forwarded. Through this configuration, clients can access only the WAF server, while the web server located in the background remains closed to direct access. Thus, all HTTP traffic is analyzed by ModSecurity and security policies are enforced.

The libapache2-mod-security2 package must first be installed on the system for the ModSecurity module to be installed on the Apache2 web server. The installation is performed using the package manager with the command `apt install libapache2-mod-security2`. After the package installation is completed, the ModSecurity module is enabled on Apache2 using the `a2enmod security2` command. Once ModSecurity is activated, HTTP requests can be inspected at the application layer level and filtered according to the defined security rules. In the final step, the Apache2 service must be restarted for the configuration changes to take effect.

After ModSecurity is activated, the default configuration file must be created so that the predefined rules or rules to be added later can operate. For this purpose, the `/etc/modsecurity/modsecurity.conf-recommended` file is copied within the same directory as `modsecurity.conf`. The copy operation is performed using the command `cp /etc/modsecurity/modsecurity.conf-recommended /etc/modsecurity/modsecurity.conf`. Then, the `modsecurity.conf` file created under the `/etc/modsecurity/` directory is opened, and the `SecRuleEngine DetectionOnly` value is changed to `SecRuleEngine On`, enabling ModSecurity to operate in active protection mode.

In this study, the OWASP CRS was used. OWASP CRS is an open-source rule set that works with ModSecurity and enables the detection of common web application attacks. The rule set must be downloaded from the GitHub repository before it can be added to the system. This process is performed using the command `git clone https://github.com/SpiderLabs/owasp-modsecurity-crs.git` (accessed on 31 May 2026). After the download process is completed, the OWASP CRS is enabled on ModSecurity so that the security rules can be applied. The reason for preferring OWASP CRS is that it provides ready-to-use and continuously updated security rules against common web application attacks such as SQL Injection, Cross-Site Scripting (XSS), Local File Inclusion, and similar threats.

The rules directory contained in the downloaded OWASP CRS must be moved to the appropriate location so that ModSecurity can use these security rules. This process is performed using the command `cp -r owasp-modsecurity-crs/rules /etc/modsecurity/crs/`. Thus, all security rules contained in the OWASP CRS are copied into the `/etc/modsecurity/crs/` directory and become available for use by ModSecurity.

The `/etc/apache2/mods-available/security2.conf` file was modified so that Apache2 can load the main configuration files defined under ModSecurity, custom rule files, and OWASP CRS rules. The `Include` directives used in this file incorporate the ModSecurity operating configuration, custom rule files, and CRS rules into the Apache2 configuration. Thus, when the Apache2 service is started, the ModSecurity engine can read the relevant rule files and analyze incoming HTTP requests according to these rules.

Table 2 presents the main directives used in the `security2.conf` file and their technical explanations.

ModSecurity examines and analyzes HTTP and HTTPS traffic in five main phases. These phases consist of request headers, request body, response headers, response body, and logging processes. Each phase enables different parts of the traffic to be analyzed,

allowing the communication between the client and the server to be evaluated in detail. Through this multi-phase analysis structure, ModSecurity can examine not only the URL part of the HTTP request but also header information, request body, and server responses. Thus, the probability of detecting different types of attacks carried out at the application layer is increased.

Table 2. ModSecurity Configuration Directives Used in security2.conf File.

Directive	Description
<IfModule security2_module>	Ensures that the relevant configuration block is executed if the ModSecurity Apache2 module is loaded.
SecDataDir /var/cache/modsecurity	Specifies the directory where persistent data used by ModSecurity will be stored.
Include/etc/modsecurity/modsecurity.conf	Includes the main ModSecurity configuration file in the Apache2 configuration.
Include/etc/modsecurity/modsecurity_custom_rules.conf	Enables the loading of custom ModSecurity rules created by the user.
Include/etc/modsecurity/crs/rules/*.conf	Enables the loading of the main OWASP CRS configuration files.
</IfModule>	Enables Apache2 to load all rule files contained within the OWASP CRS.
	Terminates the configuration block belonging to the ModSecurity module.

4.3. HTTP Traffic Analysis Process

ModSecurity evaluates HTTP and HTTPS traffic by passing it through multiple analysis phases. Each phase is responsible for examining different HTTP components, and the defined security rules are executed in the relevant analysis phases. Thus, the communication between the client and the server can be evaluated not only at the URI or parameter level but also by considering all components of the HTTP protocol.

Figure 7 shows the phases used by ModSecurity to analyze HTTP requests and responses. The first phase is REQUEST_HEADERS and includes the examination of HTTP header information received from the client. The second phase is the REQUEST_BODY phase and enables the analysis of the parameters contained in the request body. The third phase is the RESPONSE_HEADERS phase and performs the evaluation of response headers returned from the server. In the fourth phase, RESPONSE_BODY, the response body is examined. The final phase, LOGGING, ensures that the performed operations and detected events are recorded.

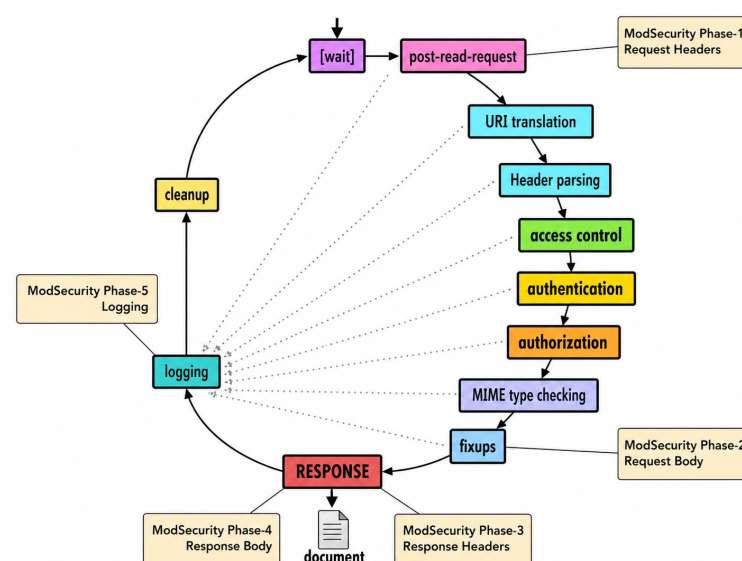


Figure 7. ModSecurity HTTP(S) Traffic Analysis Phases.

To determine in which phase a rule will be executed, the relevant phase information must be specified during rule definition. If no phase is defined within the rule, the rule is

executed in the default phase specified in SecDefaultAction. Therefore, applying the rule in the correct phase is important for accurately detecting attacks and effectively performing log analyses. Some of the custom rules developed in this study were configured to operate on the REQUEST_URI and ARGS fields, while some were configured to operate on the REQUEST_BODY field. Therefore, examining the analysis phases in which rule matches occur is important for correctly interpreting the log records generated by ModSecurity.

4.4. Custom Rule Development and Validation

In addition to the default rule sets, custom rules can be defined on ModSecurity. In this study, a configuration file named `modsecurity_custom_rules.conf` was created under the `/etc/modsecurity/` directory to create custom rules. Custom SecRule definitions that check specific parameters, URI fields, and the request body were used within the created file. The custom rules were prepared to demonstrate the analysis processes performed by ModSecurity on different HTTP components. In this context, different analysis scenarios were evaluated by creating parameter-based, URI-based, and request body-based filtering examples.

The rule creation structure in ModSecurity is generally based on the SecRule VARIABLES OPERATOR [ACTIONS] format. In this structure, the VARIABLES field refers to the HTTP component to be checked, the OPERATOR field refers to the matching condition, and the ACTIONS field refers to the operation to be applied in case of a match. In this context, matching checks were performed on request parameters, REQUEST_URI, and REQUEST_BODY fields as examples in the study. The defined rules were configured to block the request, create a log record, and return the HTTP 403 status code in case of a match.

WAF log files contain records related to the applied filtering rules together with the request information received by the web server. In this study, the logs generated by ModSecurity were examined through the `modsec_audit.log` file located under the `/var/log/apache2/` directory. Thus, the effects of both the default rule sets and the custom-defined rules on HTTP traffic could be analyzed.

Controlled test scenarios were applied to validate the developed custom rules. For each rule, the pattern defined in the relevant HTTP component was sent, and the responses generated by ModSecurity were examined. The obtained validation results are presented in Table 3. In all three test scenarios, the pattern defined in the relevant HTTP component was successfully detected, and an HTTP 403 Forbidden response was generated by ModSecurity. The results show that the developed rules can correctly detect matches performed in parameter fields, on the URI, and within the HTTP request body.

In the first custom rule created, the aim was to block the request if the `testparam` parameter contained the test expression. This rule was configured to examine the specified parameter value, stop the request in case of a match, and return an HTTP 403 Forbidden response.

Table 3. Validation Results of Custom ModSecurity Rules.

Rule ID	Target Field	Test Value	Analysis Phase	HTTP Response	Result
2545	ARGS:testparam	test	Phase 2	403 Forbidden	Passed
2555	REQUEST_URI	;	Phase 2	403 Forbidden	Passed
2565	REQUEST_BODY	some_bad_string	Phase 2	403 Forbidden	Passed

The ModSecurity audit log record belonging to the first custom rule created is presented in Table 4. When the log record is examined, it is seen that the request was evaluated in the second analysis phase (phase 2), the test expression was detected in the `ARGS:testparam` field, and the request was blocked due to the defined rule. In addition,

the rule ID, matched expression, URI information, and performed action were recorded in the log record.

Table 4. ModSecurity Audit Log Record of the First Custom Rule.

Message: Access denied with code 403 (phase 2) String match "test" at ARGS:testparam [file "/etc/modsecurity/modsecurity_custom_rules.conf"] [line "1"] [id "2545"] [msg "Test Successful"] Apache-Error: ModSecurity: Access denied with code 403 Action: Intercepted (phase 2) URI: /dvwa/login.php Host: 192.168.56.114
--

In the second custom rule, the aim was to block the request if the ';' character was present in the URI for testing purposes. While the first rule performed parameter-based filtering, the second rule performs URI-based filtering. In this context, the REQUEST_URI field was checked, and when the semicolon character was detected, the request was stopped with an HTTP 403 Forbidden response. The ModSecurity audit log record belonging to the second custom rule is presented in Table 5.

Table 5. ModSecurity Audit Log Summary of the Second Custom Rule.

Message: Access denied with code 403 (phase 2) String match ";" at REQUEST_URI [file "/etc/modsecurity/modsecurity_custom_rules.conf"] [line "2"] [id "2555"] [msg "Test URI Successful"] Apache-Error: ModSecurity: Access denied with code 403 Action: Intercepted (phase 2) URI: /dvwa/login.php; Host: 192.168.56.114 Action: Intercepted (phase 2) URI: /dvwa/login.php Host: 192.168.56.114
--

When the log record is examined, it is seen that the request was evaluated in the second analysis phase, the ';' character was detected in the REQUEST_URI field, and the request was blocked due to the relevant custom rule. This result shows that URI-based custom rules operate successfully on ModSecurity.

In the third custom rule, the aim was to detect and block a specific expression located in the body part of the HTTP request. In this context, the REQUEST_BODY field was checked, and the request was blocked when the some_bad_string expression was found in the request body. In case of a match, ModSecurity was configured to stop the request, and an HTTP 403 Forbidden response was returned to the client.

The ModSecurity audit log record belonging to the third custom rule is presented in Table 6.

Table 6. ModSecurity Audit Log Record of the Third Custom Rule.

Message: Access denied with code 403 (phase 2) Pattern match "some_bad_string" at REQUEST_BODY [file "/etc/modsecurity/modsecurity_custom_rules.conf"] [line "3"] [id "2565"] [msg "Matched some_bad_string"]
Apache-Error: ModSecurity: Access denied with code 403
Action: Intercepted (phase 2) URI: /dvwa/vulnerabilities/xss_s/ Host: 192.168.56.114

When the log record is examined, it is seen that the request was evaluated in the second analysis phase and that the some_bad_string expression was detected within the REQUEST_BODY field. This result shows that custom rules defined on the HTTP request body operate successfully.

The SecAuditLogParts directive is used in the modsecurity.conf file to determine which sections will be recorded in ModSecurity audit log files. In this study, the SecAuditLogParts ABCEFHJKZ value was defined so that the audit log content could be examined in detail. With this configuration, the transaction header, request headers, request body, response headers, audit log closing information, and details related to rule matches are included in the log file. Each letter in this setting represents a different section in the audit log file. Thus, the HTTP transactions analyzed by ModSecurity can be evaluated in detail not only in terms of the blocking result but also in terms of the request field in which the rule match occurred. This configuration was used to validate custom rules and to conduct the log analysis process more effectively.

5. ModSecurity Log Analysis and Experimental Results

In this section, the experimental results were evaluated based on three main criteria. First, the effect of the DoS attack on system resources was analyzed by examining the processor usage values of the WAF server during the attack. Second, the triggered rules, HTTP status codes, source IP addresses, and request URI information were evaluated through ModSecurity audit log records. Third, after the security mechanisms were enabled, whether legitimate user requests were processed successfully was examined. Thus, both the effect of attack traffic and the result of the security configuration on system performance and availability were evaluated.

In this study, an experimental attack scenario was applied to examine the behavior of the ModSecurity WAF against DoS attacks. Within the scope of the experiment, a DoS attack was performed against the target web application using the SlowHttpTest tool, and ModSecurity's ability to block attack traffic was analyzed. The SlowHttpTest tool was preferred because it can create long-duration HTTP connections using low bandwidth and realistically simulate application layer DoS attacks.

In the first stage, ModSecurity was disabled. In this stage, the response of the web application during the attack and the CPU usage values of the WAF were examined. Thus, the behavior of the system against the DoS attack was observed while the firewall was not active. The GET-based attack mode of the SlowHttpTest tool was used to perform

the DoS attack. In this context, 3000 simultaneous connections were created, requests were sent at a rate of 200 connections/second, and the connections were kept open for 24 s. In addition, a new connection was created every 10 s, and the target was set as <http://192.168.56.114/dvwa/>. The selected parameters were determined to increase resource consumption by creating a large number of concurrent connections on the web server and to observe the behavior of the security mechanisms under attack. With this configuration, it was aimed to keep client connections open for a long time and to consume the resources of the target system.

The experimental outputs obtained after the attack are presented in Table 7. According to the SlowHttpTest output, the attack was performed using the GET method, the Content-Length header value was set to 4096 bytes, and the follow-up data size was set to 52 bytes. During the test, the connection timeout was applied as 3 s and the total test duration as 240 s. Each DoS attack scenario was repeated 20 times, and CPU utilization was monitored throughout the 240 s test duration. Similar CPU usage values were observed across repeated runs. Therefore, the reported CPU values represent the average observed utilization rather than a single visual snapshot. At the end of the test, it was observed that 5 connections remained open, 2995 connections were closed, the number of errors was 0, and the service status was reported as “NO”. In addition, the occurrence of the “Connection was reset” error on the client side shows that the target web application could not process incoming connections properly during the attack. These results show that when ModSecurity and additional protection mechanisms are disabled, the application layer DoS attack has a significant effect on the availability of the web application.

Table 7. Observations Recorded After the SlowHttpTest Attack.

Observation	Value/Status
Number of active connections	5
Number of errors	0
Number of closed connections	2995
Service availability	No
Browser output	Connection was reset
Test result	Access to the web application was interrupted

Figure 8 shows the CPU usage values of the WAF server during the attack. Processor usage values were used as a performance indicator to measure the effect of the attack on the system. The change in CPU usage was analyzed to evaluate the effect of the attack on system resources and the load reduction success of the applied security mechanisms.

It is seen that the total CPU usage reached the level of 83% during the attack. When examined on a core basis, Core1 reached 80%, Core2 81%, Core3 80%, and Core4 91% usage values. These values show that the DoS attack performed with SlowHttpTest created a significant processor load on the WAF server.

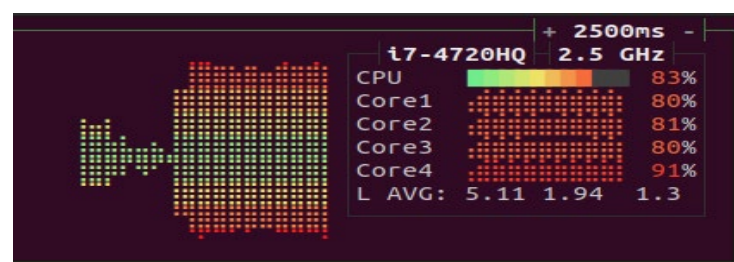


Figure 8. WAF Server CPU Value During the Attack.

When the log records generated during the attack process were examined, it was observed that the requests were sent from the source IP address 192.168.56.108 to the target web application. The records showed that the target resource was the /dvwa/ directory and that the requests were transmitted in the format GET /dvwa/ HTTP/1.1. In addition, HTTP 408 status codes were detected. This status code indicates that the client connections were not completed within the specified period and that a timeout occurred on the server side. These records indicate that the attack was successful.

To prevent DoS attacks, ModSecurity was first activated. In addition, the mod_qos library was used to limit traffic parameters such as the number of connections and request rates. The mod_qos module was installed using the apt-get install libapache2-mod-qos command, and the required settings were configured in the mod_qos.conf file. If the module is not active, it should be enabled using the a2enmod qos command. Although ModSecurity is effective in detecting application layer attacks, additional mechanisms may be required to limit the number of connections and request rates. Therefore, additional restrictions on connection and request rates were applied in this study using the mod_qos module.

Table 8 presents the mod_qos.conf configuration parameters used against DoS attacks. After the DoS prevention settings were applied, the SlowHttpTest attack was performed again with the same scenario. While ModSecurity was active, it was observed that the attack traffic did not cause interruption on the connections and that the CPU usage values of the WAF server decreased significantly.

Table 8. DoS Mitigation Configurations Used in the mod_qos.conf File.

Parameter	Value	Description
QS_SrvRequestRate	120	Specifies the minimum data rate per second that will be accepted while reading requests.
QS_SrvMaxConn	10	Limits the maximum number of connections allowed for the relevant virtual server.
QS_SrvMaxConnClose	300	Ensures that keep-alive support continues until the server reaches 600 connections
QS_SrvMaxConnPerIP	5	Limits the maximum number of connections that can originate from a single IP address.

Figure 9 shows that the total CPU usage was at the level of 25%. When examined on a core basis, Core1 had a usage value of 24%, Core2 29%, Core3 25%, and Core4 23%. These values show that the system load decreased significantly compared to the 83% total CPU usage observed in the previous attack case. In other words, they show that the security mechanisms not only analyzed the attack traffic but also contributed to more efficient use of system resources.

Also, we analyzed each security mechanism respectively ModSecurity Only and mod_qos only, because we need to separate the impact of these two tools. Additional experiments were conducted under four different configurations. As shown in Table 9, the highest CPU utilization (83%) was observed when no protection mechanism was enabled. Enabling ModSecurity alone reduced CPU utilization to 68%, indicating that request inspection and rule-based filtering provided a moderate improvement. When only mod_qos was enabled, CPU utilization decreased more significantly to 41%. This experiment demonstrates the effectiveness of connection and request-rate control against Slow HTTP-based attacks. The lowest CPU utilization (25%) was achieved when both ModSecurity and mod_qos were enabled simultaneously. As a result, we conclude that the combined deployment provides the most effective mitigation strategy in the evaluated environment.

The results also highlight the different operational roles of the two security mechanisms. While ModSecurity performs application-layer inspection through OWASP CRS and custom security rules, mod_qos primarily mitigates Slow HTTP-based DoS attacks by enforcing connection and request-rate limitations. The major reduction in CPU utiliza-

tion was mainly associated with the connection control capabilities of mod_qos, whereas ModSecurity contributed to attack detection, traffic inspection, and audit logging.

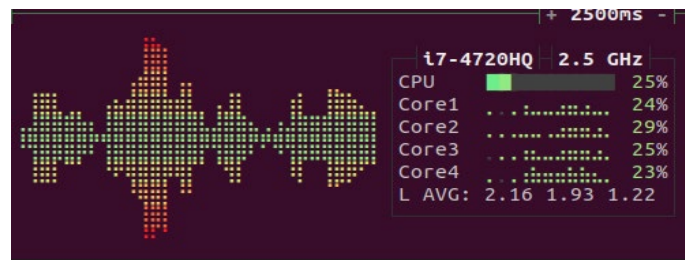


Figure 9. CPU Usage of the WAF Server with ModSecurity.

Table 9. CPU utilization under different protection configurations during the SlowHttpTest attack.

Configuration	CPU Usage (%)
No Protection	83
ModSecurity Only	68
mod_qos Only	41
ModSecurity + mod_qos	25

This study mainly evaluated CPU utilization, attack detection, HTTP status codes, and service availability. Response latency, throughput, and request-processing performance are also important operational metrics. Detailed request inspection and strict CRS rules may introduce additional processing overhead. Therefore, these metrics should be considered together when evaluating the overall performance of ModSecurity-based WAF deployments.

During the experiments conducted, legitimate user requests were successfully processed with HTTP 200 OK responses while attack traffic was detected and limited by the applied security mechanisms. The primary focus of this study was the evaluation of DoS mitigation, CPU utilization, log analysis, and service availability. Therefore, a systematic false-positive and false-negative assessment was outside the scope of the experimental design. Nevertheless, the results provide evidence of effective attack mitigation and service availability, although detection accuracy was not quantified in terms of FP and FN metrics.

When the log records were examined, it was determined that the request was sent from the source IP address 192.168.56.108 to the target server 192.168.56.114. The request was transmitted in the format GET /dvwa/ HTTP/1.1. The HTTP headers include the Host, User-Agent, and Referer fields. The User-Agent field contains information belonging to the client browser. The expression TESTING_PURPOSES_ONLY is contained in the Referer field.

The HTTP 302 Found status code was generated in the server response. This status code indicates that the request received from the client was redirected to a different resource. The Location: login.php information in the response headers shows that the request was redirected to the login page. In addition, PHPSESSID and security = high values were transmitted to the client through Set-Cookie headers. This indicates that the application was operating at the high security level. In the ModSecurity analysis records, pattern matching was performed on the REQUEST_HEADERS:Host field. In this context, the rule with ID 920350 in the REQUEST-920-PROTOCOL-ENFORCEMENT.conf rule file was triggered. The rule message contains the expression and the host header is a numeric IP address. This message shows that using a direct IP address instead of a domain name in the Host header was evaluated by ModSecurity as a protocol violation. The records include the OWASP_CRS, PROTOCOL_VIOLATION/IP_HOST, OWASP_TOP_10/A7, and WASC/21

tags. These tags show that the request was classified as a protocol violation within the scope of OWASP CRS. The severity level of the rule was determined as WARNING. In addition, the Engine-Mode: ENABLED value shows that the ModSecurity engine was operating in active mode. The phase:1 expression in the SecAction record indicates that the operation was evaluated in the first analysis phase. These results show that the ModSecurity and OWASP CRS configurations analyzed the attack traffic and recorded abnormal request behaviors.

When the log records were examined, it was observed that access to the web application was successfully performed by other clients. It was determined that the requests were sent in the format GET /dvwa/vulnerabilities/xss_s/ HTTP/1.1 and were answered by the server with the HTTP 200 OK status code. The 200 OK status code shows that the request received from the client was processed successfully and access to the relevant resource was provided.

When the response headers were examined, it was seen that the server returned HTML content with the Content-Type: text/html value. In addition, the Transfer-Encoding: chunked, Keep-Alive, and Connection: Keep-Alive information shows that the connection was actively maintained. It was also observed that cookie information containing PHPSESSID and security = high values was transmitted to the client. When these results are evaluated together, it is seen that the applied ModSecurity and mod_qos configurations did not block legitimate user requests while limiting attack traffic. In other words, while the security mechanism controlled the clients performing the attack, it was able to maintain normal user traffic access to the web application.

6. Conclusions

Recently, many critical operations, especially banking, shopping, and commercial transactions, are carried out through web applications. This situation makes ensuring the security of web applications important. The location of web application servers in the DMZ area causes application layer security vulnerabilities to be targeted by attackers. Therefore, WAF solutions that can provide protection at the application layer level have become an important security component.

In this study, ModSecurity, an open-source WAF solution, was used. HTTP traffic was analyzed with the help of the OWASP CRS and the custom rules created, and attack scenarios targeting URI, request header, and request body fields were evaluated. In addition, the effect of ModSecurity and mod_qos configurations on the DoS attack performed using the SlowHttpTest tool was examined.

The experimental results showed that the applied security mechanisms were effective in detecting and limiting attack traffic. When ModSecurity was disabled, the total CPU usage of the WAF server reached 83% during the DoS attack. After ModSecurity and mod_qos configurations were enabled, the total CPU usage decreased to 25% in the same attack scenario. This result shows an approximately 58 percentage-point reduction in processor load.

Log analyses showed that HTTP 408 status codes occurred during the attack. After the security mechanisms were enabled, the attack traffic was brought under control, and legitimate users were able to access the web application with HTTP 200 OK responses. In addition, the log analyses performed within the scope of the study showed that the audit records generated by ModSecurity can be used not only to detect attacks but also to examine in detail the sources of security incidents, the HTTP components used, and attack behaviors. The obtained findings show that the use of ModSecurity and OWASP CRS can provide effective protection against application layer attacks. However, the effectiveness of WAFs depends not only on the software used but also on configuration parameters, the

currency of rule sets, and application-specific security requirements. Therefore, regularly reviewing and updating security rules is important for maintaining the level of protection against new attack techniques.

The custom rules developed within the scope of the study produced successful results in tests performed on URI, request header, and request body fields. This finding indicates that defining application-specific rules in addition to default security rules can increase the attack detection rate. However, the obtained results are based on tests conducted in a controlled laboratory environment. Different results may be obtained in real-world systems due to variations in application architectures, traffic densities, and usage scenarios. Future studies may include more comprehensive testing against different web application attacks, evaluation of OWASP CRS at different security levels, and integration of machine learning-based anomaly detection methods with ModSecurity. Since DoS mitigation requires adaptive resource management and traffic prioritization, recent K-NN- and clustering-based optimization studies provide a useful perspective on improving resource utilization, resilience, delay performance, and secure decision-making [35,36]. Based on this perspective, future WAF architecture may benefit from intelligent and adaptive traffic management mechanisms.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The original contributions presented in the study are included in the article; further inquiries can be directed to the corresponding author.

Conflicts of Interest: The author declares no conflicts of interest.

References

1. Liu, B.; Penaka, S.R.; Lu, W.; Feng, K.; Rebbling, A.; Olofsson, T. Data-driven quantitative analysis of an integrated open digital ecosystems platform for user-centric energy retrofits: A case study in northern Sweden. *Technol. Soc.* **2023**, *75*, 102347. [CrossRef]
2. Alazmi, S.; De Leon, D.C. A systematic literature review on the characteristics and effectiveness of web application vulnerability scanners. *IEEE Access* **2022**, *10*, 33200–33219. [CrossRef]
3. Altulaihan, E.A.; Alismail, A.; Frikha, M. A survey on web application penetration testing. *Electronics* **2023**, *12*, 1229. [CrossRef]
4. Dawadi, B.R.; Adhikari, B.; Srivastava, D.K. Deep learning technique-enabled web application firewall for the detection of web attacks. *Sensors* **2023**, *23*, 2073. [CrossRef] [PubMed]
5. Kaya, M.O.; Dagdogan, H.A.; Ozdem, M.; Das, R. An effective new penetration test approach to detect web attacks on web applications. *Expert Syst. Appl.* **2025**, *298*, 129623. [CrossRef]
6. Román-Gallego, J.Á.; Pérez-Delgado, M.L.; Viñuela, M.L.; Vega-Hernández, M.C. Artificial Intelligence Web Application Firewall for advanced detection of web injection attacks. *Expert Syst.* **2025**, *42*, e13505.
7. Qureshi, R.; Koo, I. A Comprehensive Survey of Cybersecurity Threats and Data Privacy Issues in Healthcare Systems. *Appl. Sci.* **2026**, *16*, 1511. [CrossRef]
8. Shahid, J.; Hameed, M.K.; Javed, I.T.; Qureshi, K.N.; Ali, M.; Crespi, N. A comparative study of web application security parameters: Current trends and future directions. *Appl. Sci.* **2022**, *12*, 4077. [CrossRef]
9. Sadqi, Y.; Maleh, Y. A systematic review and taxonomy of web applications threats. *Inf. Secur. J. A Glob. Perspect.* **2022**, *31*, 1–27.
10. Mani, K.; Shenoy, A.K.B. Machine learning models in web applications: A comprehensive review. *ICT Express* **2025**, *11*, 1110–1119. [CrossRef]
11. Positive Technologies. Web Vulnerabilities 2020–2021. Available online: <https://www.ptsecurity.com/ww-en/analytics/web-vulnerabilities-2020-2021/> (accessed on 29 November 2025).
12. Jaafar, G.A.; Abdullah, S.M.; Ismail, S. Review of recent detection methods for HTTP DDoS attack. *J. Comput. Netw. Commun.* **2019**, *2019*, 1283472. [CrossRef]
13. Kaspersky. DDoS Report Q3 2022. Available online: <https://securelist.com/ddos-report-q3-2022/107860/> (accessed on 31 December 2025).
14. Cloudflare Radar. DDoS Threat Report 2025 Q3. Available online: <https://cloudflare.com> (accessed on 31 May 2026).

15. StormWall. *DDoS Attacks in Q1 2026: Trends & Analysis*; StormWall: Amsterdam, The Netherlands, 2026. Available online: <https://stormwall.network/resources/blog/ddos-attacks-in-q1-2026> (accessed on 31 May 2026).
16. Aslan, Ö.; Aktuğ, S.S.; Ozkan-Okay, M.; Yilmaz, A.A.; Akin, E. A Comprehensive Review of Cyber Security Vulnerabilities, Threats, Attacks, and Solutions. *Electronics* **2023**, *12*, 1333. [\[CrossRef\]](#)
17. Anuvarshini, M.K.; Bala, K.S.S.; Sonti, S.S.T.; Jevitha, K.P. An Empirical Study on the Evaluation and Enhancement of OWASP CRS (Core Rule Set) in ModSecurity. *Comput. Secur.* **2025**, *160*, 104714. [\[CrossRef\]](#)
18. Sobola, T.D.; Zavarisky, P.; Butakov, S. Experimental Study of ModSecurity Web Application Firewalls. In Proceedings of the IEEE International Conference on Big Data Security on Cloud, High Performance and Smart Computing and Intelligent Data and Security, Baltimore, MD, USA, 25–27 May 2020; pp. 209–213.
19. Muzaki, R.A.; Briliyant, O.C.; Hasditama, M.A.; Ritchi, H. Improving Security of Web-Based Application Using ModSecurity and Reverse Proxy in Web Application Firewall. In Proceedings of the International Workshop on Big Data and Information Security, Depok, Indonesia, 17–18 October 2020; pp. 85–90.
20. Clincy, V.; Shahriar, H. Web Application Firewall: Network Security Models and Configuration. In Proceedings of the IEEE 42nd Annual Computer Software and Applications Conference, Tokyo, Japan, 23–27 July 2018; pp. 835–836.
21. Erden, A. Modern Web Application Security in Practice: National-Scale Measurement and Managerial Insights from Türkiye. *Comput. Secur.* **2026**, *164*, 104847. [\[CrossRef\]](#)
22. Praseed, A.; Thilagam, P.S. HTTP request pattern based signatures for early application layer DDoS detection: A firewall agnostic approach. *J. Inf. Secur. Appl.* **2022**, *65*, 103090. [\[CrossRef\]](#)
23. Aladi, C.C. Web application security: A pragmatic exposé. *Digit. Threat. Res. Pract.* **2024**, *5*, 1–9. [\[CrossRef\]](#)
24. Tadhani, J.R.; Vekariya, V.; Sorathiya, V.; Alshathri, S.; El-Shafai, W. Securing web applications against XSS and SQLi attacks using a novel deep learning approach. *Sci. Rep.* **2024**, *14*, 1803. [\[CrossRef\]](#) [\[PubMed\]](#)
25. Schmitt, I.; Schinzel, S. WAFFle: Fingerprinting Filter Rules of Web Application Firewalls. In Proceedings of the WOOT 2012: USENIX Workshop on Offensive Technologies, Bellevue, WA, USA, 6–7 August 2012; pp. 34–40.
26. Akhavani, S.A.; Jabiyeve, B.; Kallus, B.; Topcuoglu, C.; Bratus, S.; Kirda, E. Waffled: Exploiting parsing discrepancies to bypass web application firewalls. *arXiv* **2025**, arXiv:2503.10846.
27. Fuertes, W.; Zambrano, P.; Sánchez, M.; Santillán, M.; Villacís, C.; Toulkeridis, T.; Torres, E. Repowering an open source firewall based on a quantitative evaluation. *Int. J. Comput. Sci. Netw. Secur.* **2014**, *14*, 118. [\[CrossRef\]](#)
28. Berner, J. Building Your Own Web Application Firewall as a Service and Forgetting About False Positives. *Magdebg. J. Zur Sicherheitsforschung* **2020**, *19*, 987–994.
29. Pantoulas, E. Description, Analysis and Implementation of a Web Application Firewall (WAF): Creation of Attack Scenarios and Threat Prevention. Master's Thesis, University of Piraeus, Piraeus, Greece, 2022.
30. OWASP Foundation. OWASP. Available online: <https://owasp.org/> (accessed on 28 November 2025).
31. OWASP Foundation. OWASP Top 10. Available online: <https://owasp.org/www-project-top-ten/> (accessed on 25 November 2025).
32. OWASP Foundation. OWASP ModSecurity Core Rule Set. Available online: <https://owasp.org/www-project-modsecurity-core-rule-set/> (accessed on 29 November 2025).
33. Akbar, M.; Ridha, M.A.F. SQL Injection and Cross Site Scripting Prevention Using OWASP ModSecurity Web Application Firewall. *JOIV Int. J. Inform. Vis.* **2018**, *2*, 286–292. [\[CrossRef\]](#)
34. Wu, C.; Chen, J.; Zhu, S.; Feng, W.; He, K.; Du, R.; Xiang, Y. Wafbooster: Automatic boosting of WAF security against mutated malicious payloads. *IEEE Trans. Dependable Secur. Comput.* **2024**, *22*, 1118–1131. [\[CrossRef\]](#)
35. Juwaied, A.; Romanowski, A. Intelligence-Driven Leader Selection in PEGASIS: A Data-Driven Machine Learning Framework for Sustainable and Secure Wireless Sensor Networks. *Electronics* **2026**, *15*, 1686. [\[CrossRef\]](#)
36. Juwaied, A. EMO-PEGASIS: A Dual-Phase Machine Learning Protocol for Energy Delay Optimisation in WSNs. *Sensors* **2026**, *26*, 611. [\[CrossRef\]](#) [\[PubMed\]](#)

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.