Analysis of the Defense Effectiveness of Cloud WAFs (AWS, GCP, ModSecurity) Against Command Injection Attacks*

Ivan Fedorenko[1,†], Dmytro Hasilin[1,*,†], Roman Banakh[1,†] and Ihor Zhuravel[1,†]

[1] Lviv Polytechnic National University, 12 Stepana Bandery St., Lviv, 79013, Ukraine

Abstract

This paper investigates how web application protection systems based on Web Application Firewalls work in cloud environments, specifically Amazon Web Services and Google Cloud Platform. The emphasis is on understanding the fundamental protective mechanisms and how they operate in real-world circumstances.

Along with the overall examination, consideration is given to potential methods such systems can be circumvented. In actuality, even well-configured protection technologies may contain flaws that can be exploited under certain circumstances. Identifying such scenarios allows us to better assess the level of security provided by these systems.

Several common methods for evading WAF security were investigated during the investigation. The findings demonstrate that traditional procedures, while effective in many cases, are insufficient when confronted with more adaptive or non-standard assault techniques.

The paper describes the key steps of the research and summarizes the findings. Based on this, a number of recommendations are made. It is especially reasonable to combine WAF with other security solutions and use a multi-layered strategy including K-Nearest Neighbors classification algorithm, which can considerably improve the overall resilience of online applications.

Keywords

Web Application Firewall, Machine Learning, Command Injection, ModSecurity.

1. Introduction

Web applications today are the basis of a significant part of digital services used both in business and in government information systems. Along with their spread, the number of cyberattacks aimed at exploiting vulnerabilities in the web environment is also growing. One of the most common mechanisms for protecting such systems is the Web Application Firewall (WAF), which analyzes HTTP/HTTPS traffic and blocks suspicious requests before they reach the server part of the web application.

Cloud providers very often take responsibility for security, but this happens due to the Shared responsibility model, and in the case of incorrect configuration of an application running in a cloud environment, all responsibility falls on the person who deployed this application [1, 2].

This paper examines two common cloud solutions of this class – AWS WAF and Google Cloud Armor. The study focuses on the analysis of request filtering mechanisms, typical protection rules, and the practical effectiveness of blocking potentially dangerous actions. Special attention is paid to identifying weaknesses in such systems and analyzing approaches that can be used by attackers to bypass protective mechanisms even with an active WAF.

* The Third International Conference of Young Scientists on Artificial Intelligence for Sustainable Development: Intelligent Systems and Artificial Intelligence (YAISD-WS 2026:ISAI) May 7-8, 2026, Ternopil, Ukraine

[1,*] Corresponding author.

[†] These authors contributed equally.

ivan.fedorenko.kb.2022@lpnu.ua (I. Fedorenko); dmytro.l.hasilin@edu.lpnu.ua (D. Hasilin); roman.i.banakh@lpnu.ua (R. Banakh); ihor.m.zhuravel@edu.lpnu.ua (I. Zhuravel)

0009-0003-5165-5429 (I. Fedorenko); 0009-0000-4636-7666 (D. Hasilin); 0000-0001-6897-8206 (R. Banakh); 0000-0003-1114-0124 (I. Zhuravel)

© 2026 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR Workshop Proceedings ceur-ws.org ISSN 1613-0073

published 2026-11-11

CEUR-WS.org/Vol-4326/paper8.pdf

Most modern WAF solutions use signature analysis and regular expressions to detect malicious requests. This approach has a number of limitations: it requires regular rule updates, complex manual configuration, and is often ineffective in the case of zero-day attacks or the use of various circumvention techniques. As noted in [3], one of the directions of development of WAF technologies is the transition to adaptive systems that use machine learning methods to analyze network traffic and detect anomalies.

Research in this area covers various approaches. For example, WAFFLED [4] proposed using pre-normalization of HTTP requests to eliminate discrepancies between how a request is interpreted by a WAF and how it is processed by the server. Other research focuses on using machine learning methods to analyze web traffic. In particular, [5] shows the effectiveness of feature engineering to improve the accuracy of attack classification, and [6] proposes the use of ensemble models based on LSTM, GRU and autoencoders to detect anomalies in HTTP request sequences. A significant development in this direction is presented in the work of Wang, Zhai and Yang [7], where the CCBA model is described, which, thanks to the use of convolutional networks and the attention mechanism, achieves an accuracy of 99.3% in detecting Command Injection attacks. Despite the development of intelligent traffic analysis methods, the problem of detecting OS Command Injection (CI) remains relevant. Research shows that modern WAFs can be bypassed using various obfuscation techniques, in particular the use of Unicode transformations or various encoding schemes, such as Base64 [8]. Similar bypass approaches related to OS Command Injection, SQL Injection, XSS, CSRF and Local File Inclusion are also considered in [9]. In response to these challenges, specialized frameworks are proposed, such as GenXSS for automated detection of XSS attacks [10], as well as the WAFBooster system, which uses neural networks to generate adversarial attacks and optimize WAF rules [11]. However, most of these solutions are focused on offline training, which limits their use in real-time environments. More flexible approaches, as shown in [12], involve iterative training of models during system operation, but this complicates the process of their deployment.

Experimental results using standardized datasets show that the effectiveness of ML approaches can vary significantly depending on the type of attack and does not always exceed 95% accuracy [13, 14]. This indicates the existence of a research problem associated with the search for effective methods for detecting obfuscated Command Injection attacks without significantly increasing traffic processing delays.

In view of this, the purpose of this work is to analyze the limitations of existing cloud WAF solutions and develop an approach based on mathematical models that allows detecting and blocking obfuscated Command Injection attacks that can bypass traditional protection mechanisms.

2. Materials and methods

This section will provide a detailed overview of the tools and methods used to conduct the research. The choice of specific tools is due to the fact that they are leading WAFs. Below, we will describe the technical characteristics of the environment used, justify the choice of our methods, and explain how these tools contributed to achieving the goal of the work.

2.1. Testing environment

To conduct the study, a test environment was deployed, which included:

1. Damn Vulnerable Web Application (DVWA) – as a web application that was deployed on cloud platforms such as AWS, GCP, and on a virtual machine with the Ubuntu Server operating system. It is employed as a controlled sandbox environment to identify bypass techniques, as meaningful WAF testing requires vulnerable application logic.
2. To protect the models that were deployed on cloud platforms, AWS WAF/GCP WAF Cloud Armour was used.

3. ModSecurity was used for off-cloud simulation.

The simulation of machine learning models was performed using the Python 3 language and the scikit-learn library [15].

2.2. Testing data

Test data were collected from well-known sources and processed in order to divide them into three categories. The primary source was the request collection recommended by the WAF Efficacy Framework, namely nuclei [16, 17, 18]. Only the modules related to Command Line Injection vulnerabilities were selected for use. An additional source of attack signatures was the widely known "Web Application Payloads Dataset" [19]. Together, these two datasets provide relatively good coverage; however, they also have several limitations that we attempted to address.

Like most attack-focused datasets, they exhibit a significant class imbalance, with a strong bias toward malicious requests. This makes proper model training and evaluation difficult. In an attempt to mitigate this imbalance, and also to better approximate realistic conditions where malicious commands occur tens or even hundreds of times less frequently than legitimate ones, the dataset was expanded using the following sources:

- Infinite-dataset-hub/Linux_Command_Classification [20]
- Linux Terminal Commands Dataset [21]

This made it possible to better balance the dataset. However, the distribution can still be made closer to real-world conditions by adding a larger volume of legitimate traffic.

Another issue with commonly used datasets is data freshness, since publicly available datasets are often outdated. In the datasets used in this study, the most recent updates ranged from 10 months to 4 years old. For this reason, the oldest records were analyzed, revealing an excessive focus on outdated technologies, particularly attempts to interact with PHP and other legacy applications. These records were not removed from the dataset; instead, the data were divided into three labeled subsets.

In addition to the legacy and publicly available data, a small set of anonymized commands was collected. These commands reflect operations commonly used in the development of modern services based on Python and Java, as well as infrastructure tools such as Google Cloud Platform (GCP), BigQuery, and Airflow. The dataset also includes commands related to the server's domain-specific environment–in this case, the Fast Healthcare Interoperability Resources medical data standard [22]– as well as tools used to interact with these resources within the GCP environment.

The data collection process was performed on a functioning UAT (User Acceptance Testing) service. All commands were anonymized and standardized to prevent models from memorizing file names or other environment-specific markers that are not relevant to the learning task.

Due to their irrelevance, the sets of records that were marked as outdated when modeling the "old" system participated in the training, but all testing for comparison was carried out based on current records. Figure 1 shows the test data sets.

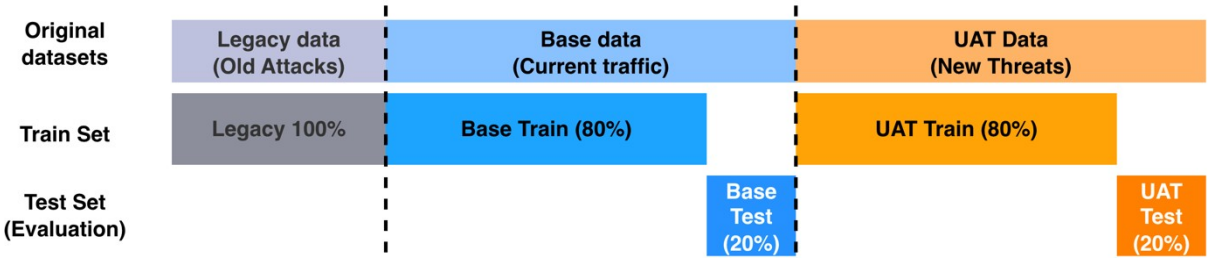


Figure 1: Dataset Split Strategy for Model Evaluation

To update the data, queries from the work of Tyshyk I. [9] were used, and our own attacks were developed and tested, which allow us to bypass current WAFs.

The experimental samples conform to the established data formats utilized by the referenced sources. The dataset is organized into discrete files, categorized by their ground-truth labels as either malicious payloads (attacks) or legitimate commands (benign traffic). Each file consists of multiple entries, with each row containing a single, isolated command. This architectural separation is intentional; it ensures the preservation of special characters and shell metacharacters that are essential for testing obfuscation techniques. Consolidating the entire dataset into a single comma-separated values (CSV) format often introduces parsing errors that could alter the command's signature.

Example of UAT true positive entries:

```
127.0.0.1; setenforce 0; wget http://evil.com/shell.sh -O /tmp/shell.sh; sh /tmp/shell.sh

echo YmFzaCAtaSA+JiAvZGV2L3RjcC8xMjcuMC4wLjEvNjY2Ng== | base64 -d | bash kill -p 1234
```

Examples of false-positive entries:

```
head -n 5 access.log

mv dir1 /archive

/opt/jdk-17/bin/java -jar my_application.jar
```

2.3. Testing infrastructure

To evaluate the WAF, we set up identical environments, both on-premises and in the cloud. This approach allowed us to concentrate on the performance of the protective tools themselves, minimizing the impact of external factors. The cloud environment was set up using Amazon Web Services and Google Cloud Platform. Following this, a WAF was added to the standard web application deployment infrastructure. Then, the local environment was set up to replicate this architecture, including a web server, network settings, and traffic filtering rules. In the course of the experiment, both environments were subjected to identical requests, encompassing both standard and attack-simulating traffic. This approach facilitated an assessment of the WAF's threat detection capabilities, while also allowing for the identification of potential operational issues contingent upon the deployment context.

The infrastructure in the AWS environment was built according to the best practices proposed in the AWS Well Architected Framework (Fig. 2).

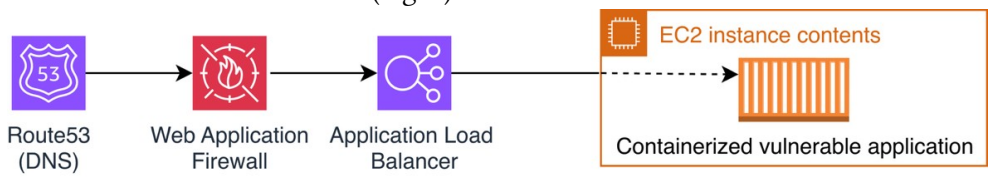


Figure 2: AWS infrastructure architecture with WAF as a web attacks protection mechanism

The AWS VPC cloud infrastructure was used with network segmentation into public and private subnets in multi-AZ mode. Computing power is provided by two Amazon EC2 instances with a deployed DVWA Docker container. Traffic management is implemented via an Application

Load Balancer with active Health Checks and Sticky Sessions. Perimeter security is guaranteed by Security Groups for port isolation and AWS WAF with configured rules against Command injection.

There is no separate WAF service on the GCP platform, instead, the use of a comprehensive Cloud Armor solution is proposed, which was done in this study. This solution focuses on ensuring the security of the available network infrastructure graph in different parts of the modern security landscape and allows flexible and dynamic customization from the user side as well as from the Google SOC side. The latter simplifies the updating of security rules, but the results may differ after adding new rules from the GCP side.

Additionally, the managed Cloud Run service with a pre-deployed Docker image was used (Fig. 3).



Figure 3: GCP Infrastructure architecture with WAF as a web attacks protection mechanism

The latest available version of the vulnerable/web-dvwa Docker image was selected, which already contains the dependencies necessary for operation, including the database and the application runtime. The services were deployed using a Terraform script to simplify automation and support.

The deployment for this model took place on a virtual machine running the Ubuntu Server 24.04.03 LTS operating system. To ensure the normal operation of the model, the following amount of hardware resources was allocated:

1. CPU (Core): 4 cores,
2. RAM: 4 GB.

The DVWA web application was chosen as the object for simulating attacks. We deployed it using Docker containerization technology (Fig. 4).

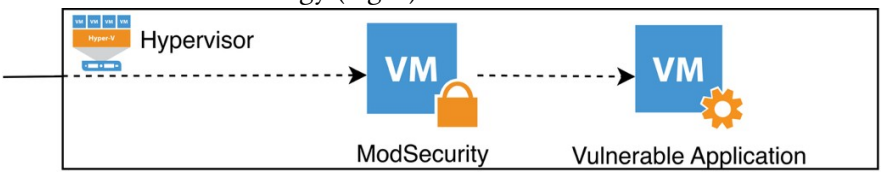


Figure 4: On-premisse infrastructure architecture as a web attacks protection mechanism

The network topology was built according to the Reverse Proxy scheme. The entry point for external traffic was Apache HTTP Server, on which the ModSecurity module was integrated.

3. Attack vectors

To test the system's resistance to Command injection attacks, a test suite was created containing various filtering-bypass techniques. The attack vectors were divided into three categories: obfuscation via string concatenation, Base64 encoding, and the use of shell metacharacters.

3.1. Obfuscation via string concatenation

This technique is commonly used to bypass signature-based detection mechanisms implemented in WAFs. Instead of sending a completely malicious command or keyword in a single form, the attacker splits it into several smaller fragments. These fragments are stored as separate strings or variables and later combined during execution. As a result, the final command is reconstructed only at runtime.
Most WAF systems analyze incoming requests in their original textual form and attempt to match them against predefined attack signatures or known patterns. When potentially dangerous expressions–such as typical shell commands or sensitive file paths–are divided into multiple parts, they may no longer match these signatures. Because the malicious payload does not appear in its entirety in the initial request, the filtering system may fail to detect it. Such an example provided in the following code snippet.

```
a=/; b=etc; c=passwd; cat “$a$b$a$c”
```

This approach is frequently observed in OS Command Injection attacks, where attackers dynamically assemble commands inside the execution environment. Constructing the payload via concatenation makes it significantly harder for security mechanisms to detect malicious intent during the initial inspection phase. The actual harmful command emerges only after the fragments are combined and interpreted by the target system.

3.2. Base64 encoding

Another technique used to evade signature-based detection involves encoding malicious commands using Base64. Instead of transmitting the command in its original readable form, the attacker encodes it as a sequence of characters. From the perspective of a filtering system, such a sequence often appears as a harmless string and may not match known attack signatures. Such an example provided in the following code snippet.

```
$(echo ‘asdfghj’ | base64 -d)
```

To execute the payload, attackers typically rely on built-in system utilities that can decode Base64 data, such as base64, openssl, or similar tools available in many operating environments. During execution, the encoded string is decoded back into its original form and then interpreted as a command by the system.
This approach allows the malicious instruction to be delivered as an apparently benign block of encoded text. Because the dangerous command is revealed only after decoding, security mechanisms that analyze requests in their original form may fail to recognize the payload's underlying intent.

3.3. Shell metacharacters

Another commonly observed obfuscation technique involves using shell metacharacters. These characters have special meaning in command-line interpreters and can alter how commands are parsed and executed. Attackers exploit this behavior to construct commands that do not explicitly contain suspicious keywords, making them less likely to match predefined detection patterns.
A typical approach relies on wildcards and substitution characters, which allow a system to reference files or commands without specifying their full names. For example, wildcard characters can be used to match parts of filenames or command names, allowing the shell to resolve the final target at execution time. As a result, the attacker can indirectly invoke a command or access a resource while avoiding the explicit strings that many security filters are designed to detect.

```
cat /etc/passw*
```

Because the actual command is reconstructed by the shell at runtime, this method can bypass security mechanisms that rely primarily on static pattern matching within the original request.

4. Attack detection using machine learning

In the proposed model, the calculation of both entropy metrics begins with determining the character proportions: the algorithm counts the occurrences of each unique character and divides it by the total length of the payload. Then the processes diverge:

- Shannon entropy: the algorithm takes the list of proportions for absolutely all characters, multiplies each by its binary logarithm, and sums these values with a minus sign.
- Min-entropy: the algorithm selects only the single largest proportion from the list (which corresponds to the most frequent character) and calculates its negative binary logarithm, completely ignoring the rest of the characters.

These entropy threshold values are calculated dynamically during training using the three-sigma rule. The algorithm finds the mean μ and standard deviation σ of the legitimate sample, establishing the normal limits as μ + 3σ for Shannon entropy and μ - 3σ for min-entropy. This approach was chosen due to three key advantages:

- Adaptability: the model automatically adjusts the limits for each parameter without relying on hard-coded constants.
- Optimal balance: the statistical coverage of 99.7% of the norm minimizes False Positives, leaving the thresholds strict enough to block attacks.
- Performance: calculating μ and σ are lightweight mathematical operations that require minimal resources and ensure fast model training.

Similarly, the structural trigram base is formed during training as a reference "whitelist" of legitimate patterns. An incoming request is considered anomalous if more than 20% of its trigrams are missing from this base. This threshold is chosen for an optimal balance: it minimizes false positives caused by minor deviations yet guarantees the detection of real cyberattacks. Since such attacks radically change the payload syntax, they consistently generate a large number of new, atypical trigrams, the proportion of which significantly exceeds the established 20% margin.

$$F_\beta = (1+\beta^2)\frac{\text{Precision} \cdot \text{Recall}}{(\beta^2 \cdot \text{Precision}) + \text{Recall}} \quad (1)$$

where β explicitly control the relative weight of recall over precision based on the specific misclassification costs of our domain.

$\beta = 2$ ensures that, within the evaluation reward system, detecting an attack is weighted as twice as important as falsely blocking a legitimate request. Both models showed promising performance during initial evaluation as shown in Figure 5.

The method was implemented in Python programming language using standard libraries for machine learning such as sklearn, numpy, pandas, matplotlib, etc [23, 24, 25, 26]. The developed model is a conceptually simplified implementation for the verification of the mathematical core of the algorithm; it analyzes only the payload, excluding the stage of parsing HTTP headers.

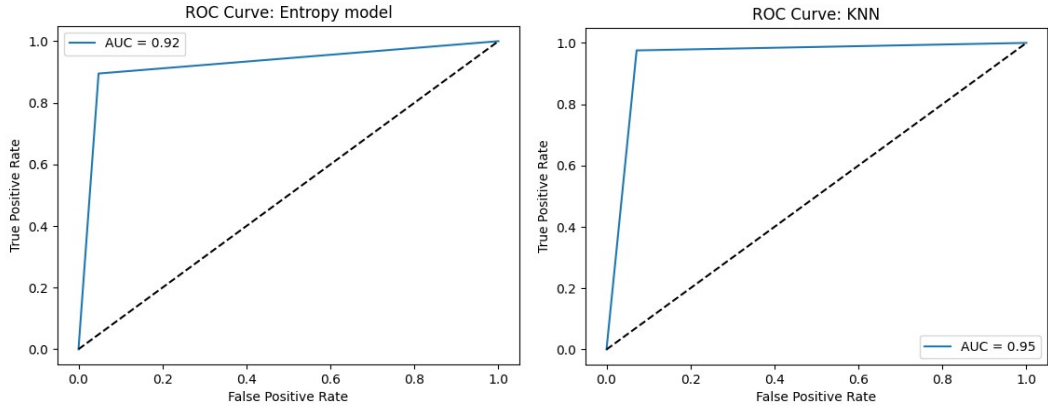


Figure 5: ROC for Entropy and KNN models.

Identical experiments were conducted across all models (Fig. 6).

- Cross-validation training and testing on the Legacy and Base datasets (legacy). To demonstrate the original performance of the ML model and the expectations established post-training when tested on the available data.
- Cross-validation training and testing on the Base and UAT datasets (updated). This one allows for a comparison and provides an understanding of how the model might perform, taking into account the adapted and augmented test dataset.

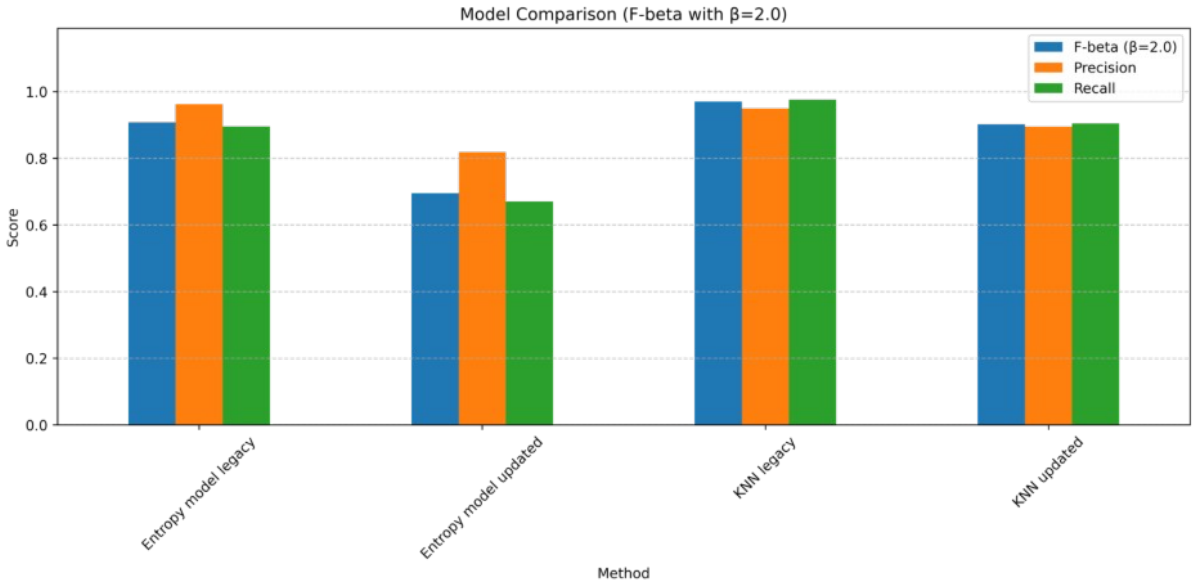


Figure 6: Model comparison, legacy and updated datasets.

The baseline KNN model, although relatively computationally intensive due to the need to evaluate distances to many records, demonstrated strong results in both experiments. The entropy model, despite stable general performance, showed a significant degradation in metrics, especially significant at the dataset drift experiment.

5. Discussion

Given the limitations mentioned, the ability to create highly specialized templates via the built-in administrative panel editors is insufficient to analyze and block unwanted traffic fully. At this stage, the primary objective of a WAF is to detect malicious traffic that could lead to vulnerability exploitation or that is frequently used in DDoS attacks.

Following the development of three datasets and their application in cross-validation to model the efficacy of machine learning models as an extension of core WAF functionality, the following conclusions can be drawn:

- The integration of ML models is a promising way to improve WAF services.
- Updating the dataset is crucial, especially after internal technology changes or major changes to technology standards.
- The computational efficiency of the algorithms is very important, given the role of a WAF as a high-performance security system.

Considering the above, the prospects for machine learning development for WAF at the current stage of model development depend more on the model's performance, data quality, and completeness than on its architecture.

To ensure comprehensive protection, specialized modules should be considered and developed to integrate and expand the capabilities of existing WAFs. The reviewed solutions, due to their closed nature and architecture, have low flexibility and limited integration capability. ModSecurity (SpiderLabs, n.d.) can be chosen as the basis for implementing such an improvement. This is an open-source solution that supports extension via custom modules. It allows not only the execution of a custom pipeline, which is inaccessible to template engines, but also the analysis of dynamic requests.

Further development of WAF may involve creating a comprehensive system that enhances endpoint protection across the infrastructure against a wide range of attacks. For example, the formalization of an API for the integration of third-party modules with a pre-prepared staging environment, in which complex parts of requests selected for detailed control, in particular binary files or multipart form content, are analyzed asynchronously (delayed) in the context of a pre-configured environment, optionally a container.

An example of such an application is combining it with an antivirus to analyze potential malware. Another example could be solving the complex task of using steganalysis tools or integrating ML models to analyze the content of hidden malicious commands. After all, composing system utilities and models into a ready-made module could significantly simplify the integration and implementation of such detectors. The latter could complement and enhance template checks, while also protecting against the vulnerabilities mentioned in this work.

Conclusion

The efficiency of extending WAF functionality through machine learning was analyzed. The most common algorithms were modeled, and a model-selection methodology was developed to ensure the quality of attack detection in a gradually changing data environment.

A further direction of research could be to improve the proposed methods by enriching the vectorization parameters through LLM contextual semantic analysis of the command. Such an extension could significantly improve detection by allowing the input set to be expanded and reducing the impact of minor syntactic changes in the command on the result.

Declaration on Generative AI

The author(s) have not employed any Generative AI tools.

References

[1] D. Tykholaz, R. Banakh, L. Mychuda, A. Piskozub, R. Kyrychok, Incident response with AWS detective controls, in: CEUR Workshop Proceedings 3826, 2024, pp. 190–197.

[2] O. Volotovskyi, R. Banakh, A. Piskozub, Z. Brzhevska, Automated security assessment of Amazon Web Services accounts using CIS Benchmark and Python 3, in: CEUR Workshop Proceedings 3826, 2024, pp. 363–371.

[3] S. Applebaum, T. Gaber, A. Ahmed, Signature-based and Machine-Learning-based Web Application Firewalls: A Short Survey, Procedia Computer Science 189 (2021) 359–367. doi:10.1016/j.procs.2021.05.105.

[4] S. Akhavani, B. Jabiyev, B. Kallus, C. Topcuoglu, S. Bratus, E. Kirda, WAFFLED: Exploiting Parsing Discrepancies to Bypass Web Application Firewalls, arXiv (2025). doi:10.48550/arXiv.2503.10846.

[5] A. Shaheed, M.-B. Kurdy, Web Application Firewall Using Machine Learning and Features Engineering, Security and Communication Networks 2022 (2022). doi:10.1155/2022/5280158.

[6] V. Babaey, H. Faragardi, Detecting Zero-Day Web Attacks with an Ensemble of LSTM, GRU, and Stacked Autoencoders, Computers 14 (2025) 205. doi:10.3390/computers14060205.

[7] X. Wang, J. Zhai, H. Yang, Detecting command injection attacks in web applications based on novel deep learning methods, Scientific Reports 14 (2024). doi:10.1038/s41598-024-74350-3.

[8] P. Andorno, Research, Testing, and Mitigation Solutions for Web Application Firewalls Evasion Techniques, Master's thesis, Politecnico di Torino, 2024.

[9] I. Tyshyk, Research of web application firewall bypass mechanisms, Cybersecurity: Education, Science, Technology 3(31) (2025) 86–99. doi:10.28925/2663-4023.2025.31.1008.

[10] V. Babaey, A. Ravindran, GenXSS: an AI-Driven Framework for Automated Detection of XSS Attacks in WAFs, arXiv (2025). doi:10.48550/arXiv.2504.08176.

[11] C. Wu, J. Chen, S. Zhu, W. Feng, R. Du, Y. Xiang, WAFBOOSTER: Automatic Boosting of WAF Security Against Mutated Malicious Payloads, arXiv (2025). doi:10.48550/arXiv.2501.14008.

[12] T. Korobeinikova, N. Kravchuk, ML-WAF information technology for classification and blocking of injection attacks in a containerized Web environment, Radiotekhnika 223 (2025) 100–110. doi:10.30837/rt.2025.4.223.11.

[13] Cloud Armor best practice, 2026. URL: https://docs.cloud.google.com/armor/docs/best-practices.

[14] T. Korobeinikova, N. Kravchuk, Application of the KNN method with feature enrichment for injection detection, Measuring and Computing Devices in Technological Processes 84(4) (2025) 473–479. doi:10.31891/2219-9365-2025-84-58.

[15] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. VanderPlas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, E. Duchesnay, Scikit-learn: Machine learning in Python, Journal of Machine Learning Research 12 (2011) 2825–2830. URL: http://jmlr.org/papers/v12/pedregosa11a.html.

[16] Fastly, WAF Efficacy Framework, 2026. URL: https://github.com/fastly/wafefficacy.

[17] ProjectDiscovery, Nuclei vulnerability scanner, 2026. URL: https://docs.projectdiscovery.io/opensource/nuclei/overview.

[18] MGM Security Partners, WAF Payload Collection, GitHub repository, 2026. URL: https://github.com/mgm-sp/WAF-Payload-Collection. Last accessed: 2026/03.

[19] CyberPrince, Web Application Payloads Dataset, Kaggle dataset. URL: https://www.kaggle.com/datasets/cyberprince/web-application-payloads-dataset. Last accessed: 2026/03.

[20] Infinite-Dataset-Hub, Linux Command Classification, Hugging Face dataset, 2026. URL: https://huggingface.co/datasets/infinite-dataset-hub/Linux_Command_Classification.

[21] Nabeel H., Linux Terminal Commands Dataset, Kaggle dataset, 2026. URL: https://www.kaggle.com/datasets/nabeel965/linux-terminal-commands-dataset.

[22] HL7 International, FHIR: Fast Healthcare Interoperability Resources, 2026. URL: https://www.fhir.org/.
[23] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. VanderPlas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, E. Duchesnay, Scikit-learn: Machine learning in Python, Journal of Machine Learning Research 12 (2011) 2825–2830.
[24] C. R. Harris, K. J. Millman, S. J. van der Walt et al., Array programming with NumPy, Nature 585(7825) (2020) 357–362.
[25] J. D. Hunter, Matplotlib: A 2D graphics environment, Computing in Science & Engineering 9(3) (2007) 90–95.
[26] The pandas development team, pandas-dev/pandas: Pandas, Zenodo, version latest, 2020. doi:10.5281/zenodo.3509134.