Improving ModSecurity WAF with Machine Learning Methods

Chapter · November 2020

DOI: 10.1007/978-981-33-4370-2_7

4 authors, including:

Tín Trần
University of Economics Ho Chi Minh City
2 PUBLICATIONS **25** CITATIONS
SEE PROFILE

Khuong Nguyen-An
University of Technology (HCMUT) VNU-HCM
42 PUBLICATIONS **175** CITATIONS
SEE PROFILE

Improving ModSecurity WAF with Machine Learning Methods

Ngoc-Tin Tran[1,2], Van-Hoa Nguyen[2], Thanh Nguyen-Le[2], and Khuong Nguyen-An[1(✉)]

[1] University of Technology (HCMUT), VNU-HCM, Ho Chi Minh City, Vietnam
{1613575,nakhuong}@hcmut.edu.vn
[2] Verichains Lab, Ho Chi Minh City, Vietnam
{tintn,vanhoa,thanh}@verichains.io
https://verichains.io

Abstract. Web Application Firewall (WAF) is a security technology that helps mitigate and prevent common attacks in web applications. Designed as a barrier between the web server and the user (possibly an attacker), the WAF analyzes, detects, and early warns of possible attacks. However, the current techniques for detecting web application attacks are still mainly based on available attack signatures. This method has the advantage of being simple and easy to apply in most cases, a typical implementation of this method is ModSecurity (`ModSecurity`: open source web application firewall, https://modsecurity.org.) with ModSecurity CRS (`ModSecurity CRS`: a set of generic attack detection rules for use with ModSecurity, https://coreruleset.org.). The main weakness of this approach is that it is unable to detect new attack vectors and mistakenly detects a normal user as an attacker also. In response to these issues, the goal of this paper is to develop a WAF system based on ModSecurity with ModSecurity CRS, which focuses on reducing the false-positive rate of ModSecurity CRS based on machine learning methods. Specifically, this is a method that combines ModSecurity CRS and machine learning.

Keywords: Web Application Firewall · ModSecurity · Machine learning · Security

1 Introduction and Related Work

With the rapid development of technology accompanied by sophistication and diversity in the types of cyber-criminal attacks, web application security has a more and more important role today. In order to combat the risk of attacks as well as data theft, organizations, businesses, etc. need to pay attention and have a proper investment in the level of security in their products, especially in web applications. However, in order to build and maintain a team of security experts to continually perform risk assessments in all products, a business needs to have a relatively high financial investment. This may not be affordable sometimes,

T. K. Dang et al. (Eds.): FDSE 2020, CCIS 1306, pp. 93–107, 2020.
https://doi.org/10.1007/978-981-33-4370-2_7

especially for small and medium businesses. There are two popular solutions today that can be applied separately or simultaneously for most cases.

Performing Security Testing. To ensure the system's safety before release, the web application should be performed vulnerability checking by third-party security companies or independent security experts in many bug bounty platforms.

Deploy a Defense System. In the course of operation, although security checks have already been performed, the remains of a few security vulnerabilities inside the application are surely inevitable. In addition, the process of patching new security bugs will take a long time to verify before being put into production. During this time, the web application can be completely exploited and compromised. To overcome this problem, WAF was deployed to temporarily fix existing security holes without interfering with the application source code. This is also our main research topic.

The main problem of WAF is detecting whether the input request is valid or not. For solving this problem, many research results have been published, in which two main approaches are rule-based and anomaly detection. Typical examples of the rule-based approach have been mentioned in [7] by Roger Meyer and Carlos Cid, wherein this study they have shown how to detect common attack vectors from OWASP Top 10 (2007) [11] through web server log file analysis. Next, Ryan Barnett published a paper [2] discussing how to use ModSecurity to *virtual patch* web applications, which clearly shows the usage of ModSecurity for applying in real cases. These rule-based approaches have the advantage of being easy to implement, but the downside of rule-based methods is that they have a high rate of false positives and cannot detect new attacks, typically zero-day attacks.

Several approaches based on anomaly detection using machine learning methods are also proposed. Specifically, Christopher Kruegel et al. mentioned an anomaly detection method based on the statistical properties of HTTP requests in [6]. This is an anomaly detection-based method, but this method only takes input from the access log, which mainly analyzes the request's URI. Thus, this method will omit some malicious requests, which may include attack vectors inside the request body (POST requests). Besides, Tammo Krueger et al. proposes a WAF system called TokDoc [1] that automatically replaces abnormal parts in the request to turn the malicious one into the normal one using various abnormally detection methods based on n-gram and Markov Chain models.

Some approaches based on neural network models have also been proposed as in [10,13], etc. Using neural network models, these WAF systems will have the ability to detect some new types of attacks that rule-based methods cannot. However, the more blocked attacks, the higher the false-positive rate will be, and the system availability will decrease. Moreover, the downside of using neural network models is that our WAF's behavior will become more unpredictable. In our opinion, a request should only be blocked if it contains a clear attack

signature and not by its strangeness. So, we still prefer a rule-based method when approaching this kind of problem. Also, Vartouni et al. [12] proposes an exciting approach that combines a Stacked-AutoEncoder neural network (SAE) for feature extraction and an Isolation Forest model for anomaly detection. The approach that combines of ModSecurity and machine learning algorithms is not a new one either. Betarte et al. [3] proposes a method that combines a one-class classifier for anomaly detection and ModSecurity CRS when deciding the outcome. In this study, we approach this problem as a two-class classification problem with the goal of classifying the input request as normal or attack based on the labeled training dataset and the ruleset of ModSecurity CRS.

2 Background

ModSecurity is an open-source, cross-platform WAF engine developed by Spider-Labs at Trustwave. This engine provides a simple and powerful syntax, allows us to quickly write security rules to protect web applications from various types of attacks, perform real-time analysis on HTTP requests. With over 10,000 installations worldwide, ModSecurity is arguably the most used WAF right now[1]. Below is an example rule, written in ModSecurity rule syntax, which will block (`deny`) any request whose uri (`REQUEST_URI`) matches the regular expression `<script>` and return a `403` status code.

```
SecRule REQUEST_URI "@rx <script>" "t:lowercase,log,deny,
    status:403"
```

In general, any ModSecurity rule syntax written with `SecRule` directive can be expressed as follows (there are many more directives in ModSecurity).

```
SecRule VARIABLES OPERATOR [TRANSFORMATIONS,ACTIONS]
```

Four components in SecRule syntax are `VARIABLES`, `OPERATOR`, `ACTIONS`, and `TRANSFORMATIONS`. `VARIABLES` is a set of input variables extracted from the request where the rule is checked, which can be the request headers, URI, form inputs, etc. For each value in the rule's variables, a list of transformation functions in `TRANSFORMATIONS` will be applied for preprocessing before further checks. `OPERATOR` defines how the rule is checked, which can be regular expression matching, or simple substring matching, etc. The remaining component, `ACTIONS`, is a list of actions that ModSecurity will execute if this rule matches the input request; a valid action can be logging, blocking, setting the returned status code, etc. These things are just some basic syntax of ModSecurity to introduce how ModSecurity works briefly; more details can be found in ModSecurity Handbook [9].

Combined with ModSecurity CRS, a set of generic attack detection rules, ModSecurity can help prevent the majority of common attacks listed in [11]. The principle of ModSecurity CRS is quite simple; when a request is analyzed, each rule in CRS will generate a corresponding score for that request, also known

[1] https://github.com/SpiderLabs/ModSecurity/wiki.

as anomaly-score. When the total score of this request exceeds a pre-configured threshold, it will be blocked. The score of each rule is decided based on the Paranoia Level. In ModSecurity CRS, there are four Paranoia Levels with increasing security levels from low to high (1 to 4). A rule with a higher Paranoia Level will have a more strict checking condition, which can easily be triggered with the normal request. The higher the Paranoia Level increases, the more the number of rules will be checked; in some cases, the false-positive rate can reach to 40% as mentioned in [4], which makes ModSecurity CRS challenging to apply to web applications in practice. We can say that ModSecurity CRS is a set of rules with a relatively high false-positive rate. For that reason, in the next section, we will propose a method that uses two machine learning models, Decision Tree and Random Forest, to reduce the false-positive rate of the ModSecurity CRS.

3 Our Proposed Method

The reason for the false detection of a normal HTTP request into attack one mainly comes from some quite strict rules in ModSecurity CRS. For illustration, we will consider the rule with id 920272 in the ModSecurity CRS, which has content as follows.

```
SecRule REQUEST_URI|REQUEST_HEADERS|ARGS|ARGS_NAMES|
    REQUEST_BODY "@validateByteRange 32-36,38-126" \
    "id:920272,\
    phase:2,\
    block,\
    t:none,t:urlDecodeUni,\
    msg:'Invalid character in request (outside of printable
chars below ascii 127)',\
    logdata:'%{MATCHED_VAR_NAME}=%{MATCHED_VAR}',\
    tag:'application-multi',\
    tag:'language-multi',\
    tag:'platform-multi',\
    tag:'attack-protocol',\
    tag:'OWASP_CRS',\
    tag:'OWASP_CRS/PROTOCOL_VIOLATION/EVASION',\
    tag:'paranoia-level/3',\
    ver:'OWASP_CRS/3.2.0',\
    severity:'CRITICAL',\
    setvar:'tx.anomaly_score_pl3=+%{tx.critical_anomaly_score
    }'"
```

This rule will check the value of some fields in the HTTP request through ModSecurity variables such as uri (`REQUEST_URI`), headers (`REQUEST_HEADERS`), request inputs (`ARGS`), etc. If any of the checked values that contain characters whose ASCII value is not between 32 and 36 or 38 to 126, this rule will be triggered, meaning the chance of being suspected as an attack of this request will be higher. However, while observing the actual daily traffic, we realized this rule

is triggered by many normal requests, which means that the false-positive rate of this rule is very high. Therefore, we think that ModSecurity CRS needs a more precise decision formula based on the triggered rules instead of the total score of each rule, which is pre-configured entirely manually. Assigning the abnormal score for each rule is not straightforward and inaccurate, so we should not do it manually, but let the machine learning models decide based on real data instead.

3.1 Data Preparation

At present, the most widely used dataset for this problem is HTTP CSIC 2010 [5]. However, during the initial experiment, we realized a few limitations of this dataset through a manual verification process as well as referencing the origin of the dataset.

- An anomaly request should not always be considered as an attack. There are many anomaly requests in the attack-label part of this dataset, which not really be attack requests and should have a normal-label; these wrong cases may produce a significant negative impact on the training results of the machine learning model.
- This dataset is entirely generated by an automation tool, and information is filled randomly. Therefore, the dataset is somewhat unrealistic and only suitable for research purposes.
- All generated HTTP request data belongs to a single e-commerce website. Therefore, this dataset is somewhat not generic, so trained machine learning models might not be appropriate for real cases since the characteristics of websites on the Internet are hugely diverse. For example, the HTTP request data of an e-commerce website will have completely different properties from social network websites. The diversity may come from user behavior, the technology used, or the type of website; if the training data is not generic enough, it will easily misunderstand the regular user's request into the attack.

In general, the input data of this problem is HTTP requests from the user, which can contain user sensitive data such as email, password, access token, etc. Therefore, having a free dataset generated from real websites available on the Internet is extremely difficult. If so, there is a high chance that the dataset is illegally published. Furthermore, data collected from actual operating websites will be completely unlabeled. Labeling HTTP requests as normal or attack will take much time; finding someone to label the dataset is also a difficult problem. It requires the labeler to have a certain level of knowledge about web application security. Therefore, this type of data will be much more challenging to handle than typical data such as images, voices, handwriting texts, etc. which most people are capable of labeling.

With the goal of building a new dataset for practical application, we have been setting up a proxy server and redirect all HTTP traffic in the local computer to the WAF system. Specifically, we have been redirected all of our daily web traffic to a previously configured WAF system. The goal is to use our own HTTP

traffic to train the machine learning models, which can solve the problem of HTTP traffic belongs to only one website so that the generated data will be more generic and realistic than the original dataset, HTTP CSIC 2010.

In order to build a WAF system that can analyze, monitor HTTP requests, and store them in a log database for further use, we propose a system architecture, as shown in Fig. 1. According to the proposed system architecture, we divide the system into many small services, in which each service only does a specific task; the use of each service can be briefly described as follows.

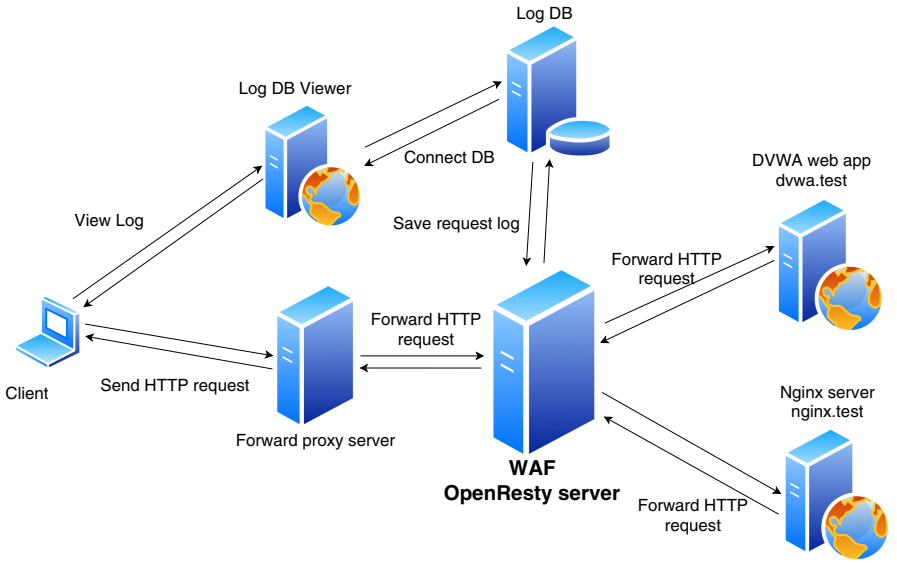

Fig. 1. Proposed system architecture

- OpenResty server with ModSecurity and CRS installed, all HTTP requests will be redirected to it for analyzing. This is the core component of the system, in which trained machine learning models will be integrated later.
- A server with DVWA[2] installed (`dvwa.test`), protected by the WAF server, which provides a testing environment for the WAF system.
- An empty Nginx server (`nginx.test`), used as a target for the data generation process.
- LogDB server with MongoDB[3] installed, this server is responsible for saving HTTP requests sent from the OpenResty server, which will be used to train the machine learning models.

[2] `Damn Vulnerable Web App (DVWA)`: http://www.dvwa.co.uk/.
[3] `MongoDB`: a document-based database, https://www.mongodb.com.

- LogDB viewer with Mongo Express[4] installed, this server provides a web-based user interface, making it easy to manage and manipulate the LogDB database.
- Forward proxy server, this server allows the client to redirect all HTTP traffic through the WAF server without the need to change local DNS config (via `/etc/hosts` file).

With the proposed system architecture above, we divide the data collection process into two steps as follows.

Generating Data with Normal-Label. At this step, we will first configure ModSecurity to be in the detection-only mode. In order to generate normal requests, we redirect all traffic from our browser to the WAF server through the proxy configuration. Next, we start browsing as many websites on the Internet as possible, like a typical user. Depending on daily web traffic, the number of recorded requests may be more or less. Currently, we have been capturing our web traffic data within five days and obtained a total of 182,103 normal HTTP requests on a total of 1,405 different websites (calculated according to the Host header in HTTP requests). The statistical result of 15 domains with the most number of requests in the dataset is shown in Table 1. In the next step, we will show the way we generate attack data.

Generating Data with Attack-Label. Currently, the simplest way to generate attack data is to use available open-source web application security tools such as SQLMap, WPScan, MetaSploit, etc. However, there are some notable characteristics and mechanisms of these tools. Most of these tools will conduct a scanning process to find some information about the target website before actually launching an attack. Therefore, among these attack requests, it is possible to mix quite a lot of normal requests, which will lead to mislabeling of request data. In order to use the above tools, first of all, it is necessary to have a particular mechanism to eliminate normal requests contained in it. After careful consideration, finally, the only tool we chose to generate attack data was FTW[5]. This tool was initially used for writing the unit test suite for ModSecurity CRS. Using this tool and the available attack test cases in the CRS test suite, we can quickly generate a large number of attack requests. Below is an example of a PHP Command Injection attack test case in ModSecurity CRS test suite.

```
...
tests:
- desc: PHP Injection Attack (933100) from old modsec
    regressions
  stages:
  - stage:
```

[4] `Mongo Express`: web-based MongoDB admin interface, https://github.com/mongo-express/mongo-express.

[5] `FTW`: a framework for testing WAFs, https://github.com/coreruleset/ftw.

```
      input:
        dest_addr: nginx.test
        headers:
          Accept: text/xml,application/xml,application/xhtml+
   xml,text/html;q=0.9,text/plain;q=0.8,image/png,*/*;q=0.5
          Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
          Accept-Encoding: gzip,deflate
          Accept-Language: en-us,en;q=0.5
          Host: nginx.test
          Keep-Alive: '300'
          Proxy-Connection: keep-alive
          User-Agent: Mozilla/5.0 (Windows; U; Windows NT
   5.1; en-US; rv
        method: GET
        port: 80
        uri: /?foo=<?exec('wget%20http://r57.biz/r57.txt%20-O
        version: HTTP/1.0
      output:
        response_contains: 403 Forbidden
  test_title: 933100-1
...
```

With our method of automatic data generation, as mentioned above, the initial results that we achieved are relatively satisfactory. Currently, we have collected 182,103 normal requests on a total of 1,405 different websites. As for attack data, we can only generate attack requests on our own web apps with a limitation of attack tools. Tools that only generate purely attack requests without being mixed with the normal ones are relatively rare. Furthermore, it is entirely illegal to use automatic tools to attack websites on the Internet arbitrarily without the owner's permission. Due to the above limitations, the number of attack requests in our dataset will be pretty much less than normal requests, only about 2,313 attack requests. Details of the generated dataset can be found in Table 2.

From the current datasets that we have, for normal data, we will use our generated data completely; the normal-labeled part of the HTTP CSIC 2010 dataset will not be used. Due to the lack of attack data, we will combine the attack dataset we generated with the attack-label part of the HTTP CSIC 2010 dataset. In short, the dataset used to train the models is summarized in Table 3.

Due to the privacy and sensitive nature of recorded data (which may contain access tokens, cookies, private information, login passwords, etc.), we will show our data generation method only but not the collected dataset. However, using our proposed data generation method, the readers can easily generate labeled data supporting their future research on this topic.

Table 1. List of 15 domains with the most number of requests

Host name	Number of requests
mail.google.com	28,644
www.youtube.com	22,776
www.google.com	15,535
static.xx.fbcdn.net	10,072
www.messenger.com	9,115
play.google.com	4,947
beacons.gcp.gvt2.com	4,886
github.com	4,224
github.githubassets.com	1,894
i.ytimg.com	1,818
calendar.google.com	1,568
r3---sn-n5pbvoj5caxu8-nbos.googlevideo.com	1,565
20.client-channel.google.com	1,468
www.google-analytics.com	1,453
edge-chat.messenger.com	1,360

Table 2. Summary of our datasets

Source	Type	Number of requests
Generated	Attack	2,313
Generated	Normal	182,103
CSIC 2010	Attack	21,451
CSIC 2010	Normal	66,958

Table 3. The dataset used to train the models

Source	Type	Number of requests	Ratio (%)
Generated, CSIC 2010	Attack	23,764	11,54%
Generated	Normal	182,103	88,46%

3.2 Proposed Model

Our original basic idea was to keep the entire ruleset of ModSecurity CRS, while also discovering which rules would often block ordinary users in order to reduce the impact of these rules. According to the way ModSecurity works, when a request is being analyzed, a particular set of rules in the ModSecurity CRS will be triggered for that request. From there, with this set of matched rules from ModSecurity CRS along with the HTTP method, we will take it as features for model training. With the above idea, we propose a method combining the

ModSecurity CRS with a classification algorithm (for example, Decision Tree), as illustrated in Fig. 2. Many machine learning algorithms can be applied in this case. Currently, we only experiment with two algorithms, Decision Tree and Random Forest. The Decision Tree is a fairly effective model for the classification problem. With fast training and prediction speeds, Decision Tree is the top choice for such problems, which requires real-time processing on each input request. Furthermore, the Decision Tree can help us identify which combinations of rules in the ModSecurity CRS are often triggered with normal requests as well as attack requests, which cannot be achieved via simple linear functions (for example, the sum of rules' scores).

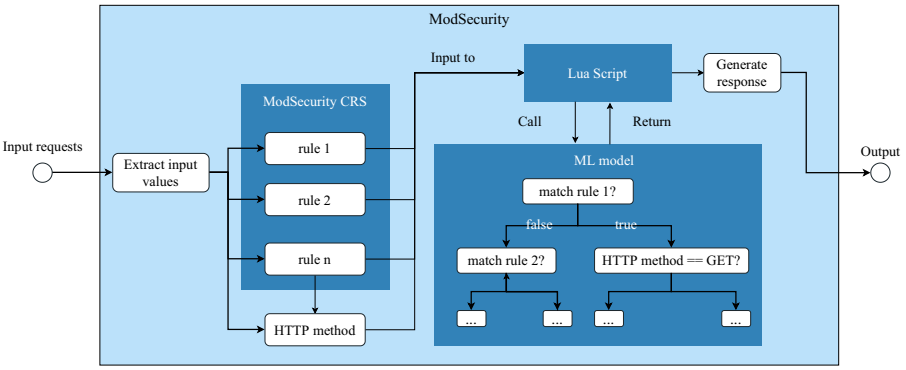

Fig. 2. Combining ModSecurity CRS with Decision Tree

In the current version of ModSecurity CRS that we use, there are a total of 156 rules denoted as r_1, r_2,..., r_{156}. For each of the above rules, we will map it as a feature in the training dataset. If a request x causes a rule r_i to be triggered, we will have feature $x_{r_i} = 1$, otherwise $x_{r_j} = 0$ for all rule r_j in the set of non-triggered rules. Additionally, we also use the HTTP request method as a training feature. The possible values of the HTTP request method in our collected dataset include `GET`, `POST`, `HEAD`, `OPTIONS`, `PUT`. For an input request x, which has the HTTP method GET, we will have the following feature x_{method} = `GET` and so on.

Using these extracted features, we apply two machine learning algorithms, Decision Tree and Random Forest, into the training dataset. To integrate the trained machine learning models into ModSecurity, we use the `SecRuleScript` directive syntax in ModSecurity. This feature allows us to write our custom rules using Lua script and invoke the compiled machine learning models as a shared library through Lua C API, which gains a relatively high performance. However, ModSecurity does not support retrieving triggered rules using the Lua C API

currently. Hence, we have made a few small modifications to the ModSecurity library. The `m.getTriggeredRules()` function has been implemented to get all the triggered rules of the current request. Below is a simple example that calls the `m.getTriggeredRules()` function from the Lua script to get all triggered rules and passing features to the compiled model. The implementation details can be found in our source code[6].

```
local ml_model = require("ml_model")

function main()
    -- logging
    m.log(1, "Starting Lua script")

    -- get all triggered rules
    local rules = m.getTriggeredRules()

    -- transform rules to model features
    local features = transform(rules)

    -- evaluate with ML model
    local ml_result = ml_model.predict(features)

    return ml_result > 0
end
```

4 Experimental Result

To start the process of training the machine learning models, we split the training dataset into two parts for training and testing with a 2:1 ratio. Due to the imbalance between the ratio of the two classes in the dataset, as shown in Table 3, we will split the dataset using the stratified train-test split technique. The first model that we experimented is the Decision Tree, which is a model where outputs are determined based on the tree data structure. Each node in a tree corresponds to a question; the result of that question will lead us to the corresponding sub-tree. This process continues until the leaf node is met; the value of the leaf node will decide which class the input data belongs to. There are many variations of the Decision Tree algorithm, such as ID3, C4.5, C5.0, and CART. Here, we only experiment with the CART algorithm according to the implementation version of the scikit-learn library [8]. After the training process, we obtained the following Decision Tree, as shown in Fig. 3.

[6] https://github.com/ngoctint1lvc/waf.

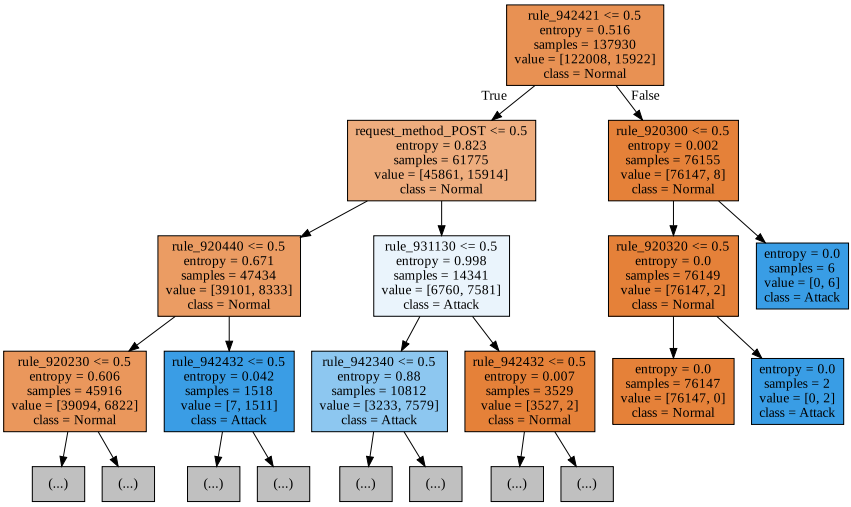

Fig. 3. The Decision Tree after training

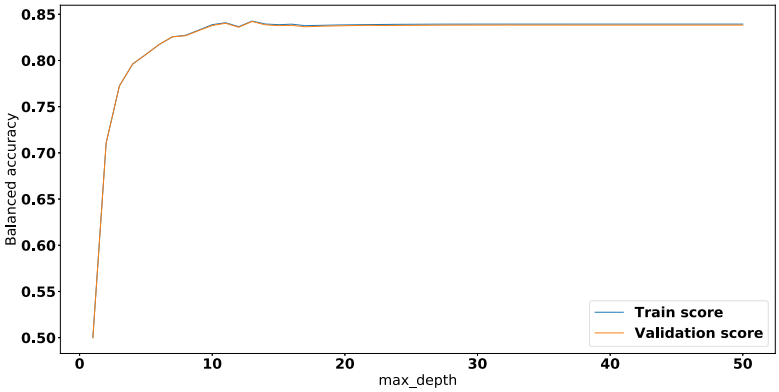

Fig. 4. Decision Tree validation curve with `max_depth` parameter

Additionally, to improve the Decision Tree model, we experimented with a popular ensemble model, Random Forest. Essentially, Random Forest is a model which makes the decision based on the average result of multiple Decision Trees. Figures 4 and 5 respectively show the validation curves of the max depth parameter of the Decision Tree model and the number of estimators of the Random Forest model compared to the balanced accuracy score. The balanced accuracy score here is the average of true-positive rate and false-positive rate, as shown in Eq. (1).

$$BAC\ (Balance\ Accuracy\ Score) = \frac{TPR + FPR}{2}. \tag{1}$$

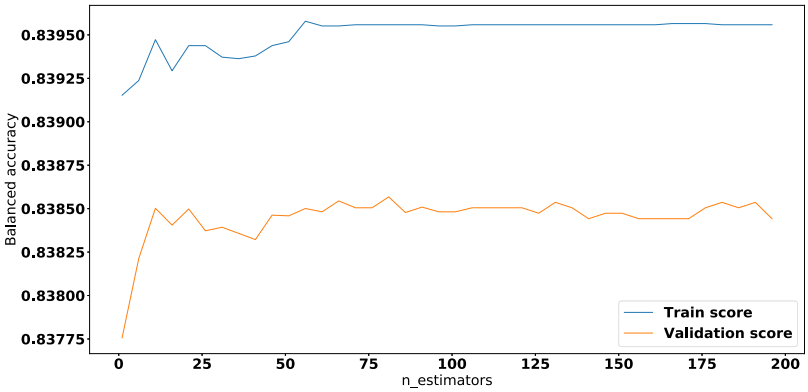

Fig. 5. Random Forest validation curve with `n_estimators` parameter

This type of score is suitable for the classification problem when two classes are sample imbalance, so we used this score to evaluate our models. Based on the validation curves in Figs. 4 and 5, we can see that the balanced accuracy score does not improve much with the max depth greater than 15 for the Decision Tree model and the number of estimators greater than 75 for Random Forest. With these parameters, our model evaluation results are shown in Table 4, and the confusion matrixes are shown in Figs. 6 and 7.

Based on the experimental results, they show that our proposed method has significantly improved the initial false-positive rate of the ModSecurity CRS. However, in order to reduce the false-positive rate to this level (about 1.26%), the true-positive rate had to decrease to 69.96% (with Random Forest model). It means that our WAF can only protect about 69.96% of attacks on web applications. The reason may come from the noisy training dataset, as mentioned in Subsect. 3.1, that many abnormal requests have been labeled attack in the CSIC dataset. Also, the ruleset in ModSecurity CRS still does not characterize between normal and attack requests well. An attack that does not trigger any rules in ModSecurity CRS will easily bypass the WAF system. Therefore, a more generic ruleset is required, which can easily be triggered by most types of attacks to prevent various tricky bypasses from hackers.

In addition to evaluating the models on the available dataset, we also tested our WAF system when accessing to real websites, which is called the *false-positive test*. The testing process is similar to the normal data collection process. Here, we

Table 4. Model evaluation results

Model	FPR	TPR (Recall)	TNR	BAC	Precision	F1-Score
Decision Tree	0.0128	0.6934	0.9872	0.8403	0.8762	0.7742
Random Forest	0.0126	0.6996	0.9874	0.8435	0.8788	0.7790

logged all access data and checked how many requests were accidentally blocked by our WAF. Specifically, the tasks that we tested include browsing blogs, commenting, logging in, posting, watching videos, listening to music, etc. on most popular websites like facebook.com, google.com, medium.com, youtube.com, etc. The testing process was performed on the WAF system integrating the Random Forest model. Table 5 below shows the test results.

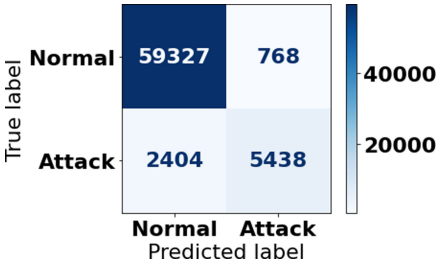

Fig. 6. Confusion matrix of Decision Tree model

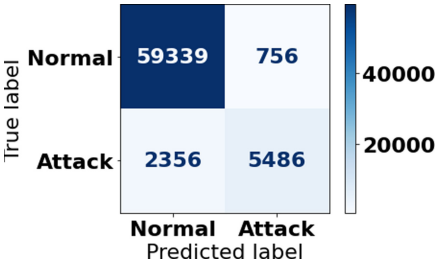

Fig. 7. Confusion matrix of Random Forest model

Table 5. False-positive test

Model	Passed requests	Wrong blocked requests
Random Forest	16,784 (99,71%)	48 (0,29%)

5 Conclusion and Future Work

In this research, we have proposed a relatively efficient method for generating labeled data so that it can easily be applied in future research on this topic. Based on the generated dataset together with the available data, we also proposed a method that combines Decision Tree and Random Forest machine learning models with ModSecurity and ModSecurity CRS. With the trained model, we have been significantly reducing the false-positive rate of ModSecurity CRS and thereby making it more suitable for practical use. However, the proposed models basically only improve the false-positive rate in ModSecurity CRS's final decision formula, not the ability to detect any new types of attacks. In future work, we can improve it by adding a more generic ruleset which better characterizes the input requests and returns a more detailed value rather than `1` or `0`. For example, a rule that returns the number of `SELECT *` keyword inside the request, which may indicate a malicious SQL Injection attack from the user. Besides, we can conduct more experiments that combine this generic ruleset with many other deep learning and machine learning methods. These approaches are

extremely exciting and promising. In the near future, we will proceed to deploy our WAF system to protect our blog websites in order to evaluate its capabilities in practice.

Acknowledgments. The authors would like to thank Mr. Van Minh Hao for his comments helping to improve the manuscript significantly.

References

1. Ammo, K., Christian, G., Konrad, R., Pavel, L.: TokDoc: a self-healing web application firewall. In: SAC'10, Sierre (2010)
2. Barnett, R.: WAF virtual patching challenge: securing WebGoat with modsecurity. Breach Security (2009)
3. Betarte, G., Giménez, E., Martínez, R., Pardo, Á.: Machine learning-assisted virtual patching of web applications. arXiv preprint arXiv:1803.05529 (2018)
4. Folini, C.: Handling false positives with the OWASP modsecurity core rule set. https://www.netnea.com/cms/nginx-tutorial-8_handling-false-positives-modsecurity-core-rule-set (2016). Accessed 04 Oct 2020
5. Giménez, C.T., Villegas, A.P., Marañón, G.Á.: HTTP data set CSIC 2010. Information Security Institute of CSIC (Spanish Research National Council) (2010)
6. Kruegel, C., Vigna, G., Robertson, W.: A multi-model approach to the detection of web-based attacks. Comput. Netw. **48**(5), 717–738 (2005)
7. Meyer, R., Cid, C.: Detecting attacks on web applications from log files. Sans Institute (2008)
8. Pedregosa, F., et al.: Scikit-learn: machine learning in Python. J. Mach. Learn. Res. **12**, 2825–2830 (2011)
9. Ristic, I.: ModSecurity Handbook. Feisty Duck, London (2010). GBR
10. Rong, W., Zhang, B., Lv, X.: Malicious web request detection using character-level CNN. In: Chen, X., Huang, X., Zhang, J. (eds.) ML4CS 2019. LNCS, vol. 11806, pp. 6–16. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-30619-9_2
11. Stock, A., Williams, J., Wichers, D.: Owasp top 10. OWASP Foundation (2007)
12. Vartouni, A.M., Kashi, S.S., Teshnehlab, M.: An anomaly detection method to detect web attacks using stacked auto-encoder. In: 2018 6th Iranian Joint Congress on Fuzzy and Intelligent Systems (CFIS), pp. 131–134. IEEE (2018)
13. Zhang, M., Xu, B., Bai, S., Lu, S., Lin, Z.: A deep learning method to detect web attacks using a specially designed CNN. In: Liu, D., Xie, S., Li, Y., Zhao, D., El-Alfy, E.-S.M. (eds.) ICONIP 2017. LNCS, vol. 10638, pp. 828–836. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-70139-4_84