Break the Wall from Bottom: Automated Discovery of Protocol-Level Evasion Vulnerabilities in Web Application Firewalls

Qi Wang*, Jianjun Chen*†✉ , Zheyu Jiang*, Run Guo*, Ximeng Liu‡, Chao Zhang*† and Haixin Duan*†
* *Tsinghua University*
† *Zhongguancun Laboratory*
‡*Fuzhou University*

***Abstract*—Web Application Firewalls (WAFs) are a crucial line of defense against web-based attacks. However, an emerging threat comes from protocol-level evasion vulnerabilities, in which adversaries exploit parsing discrepancies between the WAF HTTP parser and those of web applications to circumvent WAFs. Currently, uncovering these vulnerabilities still depends on manual, ad hoc methods. In this paper, we propose WAF Manis, a novel testing methodology to automatically discover protocol-level evasion vulnerabilities in WAFs. We evaluated WAF Manis against 14 popular WAFs including Cloudflare and ModSecurity and 20 popular web frameworks including Laravel and Spring. In total, we discovered 311 protocol-level evasion cases affecting all tested WAFs and applications. Due to the generic nature of protocol-level evasions, these evasion vulnerabilities do not hinge on specific payload patterns and can transmit any malicious payloads - for instance, SQL injection, XSS, or Log4jShell - to the target websites. We further analyzed these vulnerabilities and identified three primary reasons contributing to WAF evasions. We have reported those identified vulnerabilities to the affected providers and received acknowledgments and bug bounty rewards from Cloudflare WAF, Fortinet WAF, Alibaba Cloud WAF, Huawei Cloud WAF, ModSecurity, Go security Team, and the PHP security team.**

1. Introduction

Web Applications Firewalls (WAFs) have become fundamental building blocks of modern application security. As an increasing number of websites transition to cloud-based platforms, and the prevalence of Web attacks continues to rise, more and more websites are relying on web application firewalls (WAFs) to ensure the security of their web applications. WAFs provide an additional layer of protection to web applications by intercepting and scrutinizing inbound web traffic to detect and block malicious requests. Web administrators can utilize WAFs as virtual patches to prevent various attacks without altering the underlying codebase of the Web application. Due to those advantages, the deployment of WAFs is mandated by compliance regulations. For example, the PCI standard, set for organizations handling credit card transactions, dictates that any application facing the internet should either be protected by a WAF or successfully pass a code review process [5]. The growing reliance on WAFs has led to a substantial market value, projected to reach USD 21.05 billion by 2030 [38].

Given the popularity and importance of WAFs, significant research has been dedicated to auditing and testing WAF rulesets. These investigations typically involve modifying the payload of a malevolent request to evade the pattern recognition of WAF rule sets. For example, SQLMap [43] obfuscated attacking payloads to evaluate WAFs against SQL injection attacks. Luca et al. [18] utilize an adversarial machine learning algorithm to alter the original malicious payload and bypass WAFs. Qiu et al. [36] use a Monte Carlo tree search guided approach to automatically find SQL injection bypass payloads. WAF vendors also actively incentivize the discovery of WAF evasion techniques. For instance, Alibaba Cloud provides a bug bounty rewards of approximately USD 800 for each discovered WAF evasion vulnerability [2].

However, prior studies have shown that protocol-level WAFs evasion vulnerabilities have emerged as a growing threat [8], [25], [26], [39]. These vulnerabilities stem from the discrepancies in parsing HTTP requests between WAFs and applications. Attackers can exploit these inconsistencies to manipulate protocol-level operations, such as how HTTP requests are structured and parsed. This allows attackers to effectively 'hide' any malicious payloads within the operations of the protocol itself. As a result, these vulnerabilities can be used to transmit any type of malicious content, including but not limited to SQL injection, XSS, Log4jShell, etc., effectively bypassing the protection provided by WAFs. This makes them a potent tool for attackers, as they can adapt to different attack payloads, making them more challenging to detect and prevent.

Despite posing a significant threat, the discovery of these protocol-level evasion vulnerabilities is still reliant on manual, ad hoc methods. They are difficult to find with state-of-the-art testing tools, mainly due to three key challenges: First, existing WAF testing tools focus on generating malicious payloads rather than malformed HTTP requests, which is *insufficient* for discovering protocol-level evasions. Second, previous works usually generate test cases blindly because most commercial WAFs are closed-source products, and can only be interacted with remotely, which makes

✉ *Corresponding author: jianjun@tsinghua.edu.cn*

testing *ineffective*; Third, there is a *lack of vulnerability detector* to examine the HTTP parsers of web applications to detect protocol-level evasion vulnerabilities.

To tackle those challenges, we propose WAF Manis, a novel testing methodology to automatically discover protocol-level evasion vulnerabilities in web application firewalls. For the first challenge, we design a grammar tree-based payload-aware generation approach. This method allows us to generate and mutate high-quality HTTP requests that contain testing payloads and ensure these payloads are not modified during mutation. For the second challenge, we combine the strengths of white-box and black-box testing. We first leverage open-source web applications for white-box testing, using their code coverage to guide the generation of testing HTTP requests. We then forward these requests to commercial WAFs for black-box testing. For the third challenge, we design a testing harness to detect protocol-level evasion vulnerabilities. This harness extracts parameters and form data from different backend applications and checks for embedded malicious payloads. If a test request can pass WAF inspection, while the web application recognizes the embedded malicious payload from parsed data, we use this disparity to identify potential protocol-level evasion vulnerabilities.

We developed an automated testing tool, WAF Manis, and evaluated it against 14 popular WAFs, including 8 commercial WAFs and 6 open-source WAFs, along with 20 popular backend applications. In total, we discovered 311 bypass vulnerabilities affecting all tested WAFs and frameworks. Due to the universal nature of protocol-level evasion vulnerabilities, these vulnerabilities can be used to transmit any malicious payloads to the target websites. We further analyzed these vulnerabilities and identified three primary causes leading to protocol-level WAF evasions: (1) Parameter type Confusion; (2) Differences in parsing malformed parameter structure; (3) Inconsistencies in supporting RFCs. Some evasion cases are even rooted in the PHP and Go programming languages. We have responsibly reported all identified vulnerabilities to the affected providers and received acknowledgments and bug bounty rewards from Cloudflare WAF, Fortinet WAF, Alibaba Cloud WAF, Huawei Cloud WAF, ModSecurity, the Go security team, and the PHP security team.

Contributions. In summary, we make the following contributions:

- *New automated approach to find protocol-level WAF evasions.* We introduced WAF Manis , a novel testing methodology to automatically discover protocol-level WAF evasion vulnerabilities that broadly threaten web applications.
- *New Implementation and Findings.* We implemented our methodology and evaluated it on 14 well-known WAFs and 20 popular web frameworks. We found 311 new vulnerabilities which falls into three categories: `Parameter Type Confusion, Malformed Parameter Structure and RFC Support Gaps`. Those vulnerabilities can be exploited to bypass popular WAFs, including Cloudflare, Modsecurity, Huawei Cloud WAF and Alibaba Cloud WAF.
- *Responsible Disclosure.* We responsibly reported our findings to affected vendors and received positive feedback.

2. Background

2.1. Web Application Firewall

Web applications receive a variety of *HTTP parameters* from users. For instance, in a GET request, the parameters are typically included in the URL as query parameters, whereas in a POST request, the parameters are sent in the request body with the Content-Type header indicating the body's mime type. However, attackers may exploit these parameters to deliver *malicious payloads* intended to compromise web applications. Such payloads might include harmful SQL code, cross-site scripts, or forced commands embedded within these parameters, leading to various attacks including SQL injection, Cross-Site scripting (XSS), and command injections.

To mitigate these threats, Web Application Firewalls (WAFs) are widely deployed to safeguard web applications. A WAF serves as a protective intermediary between clients and web applications. It works at the application layer (HTTP/HTTPS), filtering, monitoring, and blocking HTTP(S) traffic to and from the web application, thereby providing a robust defense against malicious threats.

WAFs primarily operate by inspecting HTTP parameters to detect and block malicious payload in HTTP(S) requests. This is usually achieved by implementing a set of rules known as a *ruleset*. A ruleset comprises patterns that are dangerous or anomalous. If any HTTP parameters coincide with a pattern within the ruleset, the WAF either flags or blocks the associated HTTP request.

Currently, the most influential WAF ruleset is the OWASP Core Rule Set (CRS) [17], which is widely used in various commercial WAFs, including Google Cloud Armor, AWS WAF, Azure WAF, etc. The CRS provides detection rules for use with ModSecurity and aims to protect the web application from common attacks such as those defined in the OWASP Top Ten.

A typical ruleset consists of three core elements: 1) *Parameters*: These are HTTP parameters like URL parameters and form data that the WAF extracts for rule application; 2) *Patterns*: These are payload patterns that the rule matches against, such as signatures of SQL injection payloads; and 3) *Action*: These are the steps taken when a request matches the rule, such as rejection or logging. The following example presents a CRS rule that denies all the requests with "`'or(1)#`" in HTTP parameters, which can be used to block SQL injection:

```
ARGS "@contains 'or(1)#" "deny"    (1)
```

Figure 1 illustrates the workflow of WAF. In figure 1a when a user transmits json `{"id":"1"}`, the WAF detects the json parameter id value of 1, which does not match

the rules, so the request is forwarded to the WebApp. In figure 1b when an attacker sends malicious payload `{"id":"'or(1)#"}`, the WAF parses the request, identifies that the URL parameter 'id' matches the rule, and subsequently rejects the request in accordance with the rule's defined action.

2.2. WAF Evasion Attack

With web applications providing high-value services, attackers are persistently developing new web attack techniques or variants to bypass the WAFs.

Payload-Level Evasion. A common way to bypass a WAF is by obfuscating or encoding the malicious payload. Attackers can alter the malicious payload of the request to evade the pattern recognition of WAF. Figure 1c shows such an example. In this scenario, an attacker exploits the fact that the SQL interpreter of the target WebApp is not case-sensitive. Thus an attacker can modify the case of the original payload to bypass the rule and allow the SQL injection request to reach the WebApp undetected. By obfuscating the original payload without altering its semantics, attackers can effectively trick the WAF into disregarding the malicious payload. Recent work has developed several techniques or tools to uncover these payload-level WAF evasions [18], [36], [43]. However, payload-level WAF evasions can be effectively mitigated by implementing strict input validation rules. For example, WAFs can prevent the technique in Figure 1c by limiting input parameters to accept only numerical values. The OWASP Core Rule Set (CRS) also offers a broad and robust set of rules to defend against such evasion techniques.

Protocol-Level Evasion. Protocol-level WAF evasion modifies HTTP requests rather than malicious payloads to bypass WAFs. Figure 1d shows an example. In HTTP, the Content-Type header indicates the format of the data being sent in the message body. For indicating json payload, the appropriate Content-Type value to use is `application/json`. However, the attacker can confuse the WAF by modifying the Content-Type header to `application/x-whatever-json` which WAF cannot recognize and parse HTTP parameters. Nevertheless, the web application, such as those based on the Flask framework, can recognize both Content-Type headers and extract the malicious payloads, consequently triggering SQL injection vulnerabilities.

As protocol-level evasion exploits weaknesses in the HTTP parsers, it provides a more universal approach to deliver any malicious payloads, thus posing severe threats to the Web. Yet, currently, there is still a lack of an automatic and efficient approach to discovering the protocol-level evasions, which has motivated our study.

2.3. Fuzz Testing

Fuzz testing, also known as fuzzing, is a software testing technique that involves generating enormous inputs to a program to uncover bugs in an automatic way. Over the past years, Fuzz testing has proven highly successful in discovering bugs in software systems. Generally, fuzz testing approaches can be divided into two categories: black-box fuzzing and white-box fuzzing.

Black-box fuzzing refers to testing without using the source code of the target program or runtime-generated information, relying solely on blind input generation to test target programs. Early fuzzing techniques [21], [29] predominantly belong to the realm of black-box fuzzing, generating inputs in a completely random manner. However, this approach can be time-consuming and inefficient to trigger deep logical problems since randomly generated inputs may not effectively cover the specific paths, states, or conditions required to trigger deep logical problems within the target program.

White-box fuzzing, on the other hand, leverages the source code of the target program or runtime-generated information, to guide input generation and accelerate the discovery of vulnerabilities. A widely used approach is *Coverage-Guided Fuzzing [20], [34], [46] (CGF)*, which incorporates code coverage information generated during the execution phase of the target program. The main idea behind CGF is to select test inputs that have the potential to explore new areas of the code, leading to the likelihood of finding more bugs or vulnerabilities in unexplored areas. CGF fuzzer feeds test inputs to the target program and monitors its execution. The code coverage information of the executing target program can be obtained from specific instructions inserted during compilation [46], directly inserted into the binary target program [19], or the support of specific hardware features [4]. Fuzzer tends to retain inputs that generate new code coverage and mutate them, with the expectation that mutated new inputs will reach unexplored areas in the program. AFL (American Fuzzy Loop) [46], originally developed by Michal Zalewski, is one of the most popular and widely used tools for CGF.

3. WAF Manis Overview

3.1. Threat Model

Generally, the detection process of WAF can be divided into three phases: (1) *Parameter parsing:* When receiving raw HTTP inputs from the clients, the WAF first parses them to recognize HTTP parameters; (2) *Pattern matching:* The WAF checks whether the parsed parameters are matched by the pattern in WAF security policies, such as the CRS rulesets; (3) *Actions:* If any parameter matches a pattern in the WAF rulesets, WAF applies the actions in matched rules, such as rejection. Otherwise, the requests pass through WAF inspection and are forwarded to web applications. Thus, parameter parsing forms the first step before detection techniques can be applied to a suspicious HTTP request. Failing to parse the parameters and extract the payloads could lead to protocol-level WAF evasions.

Figure 2 presents a real-world case we discovered. Both two HTTP requests contain the same malicious payload, but

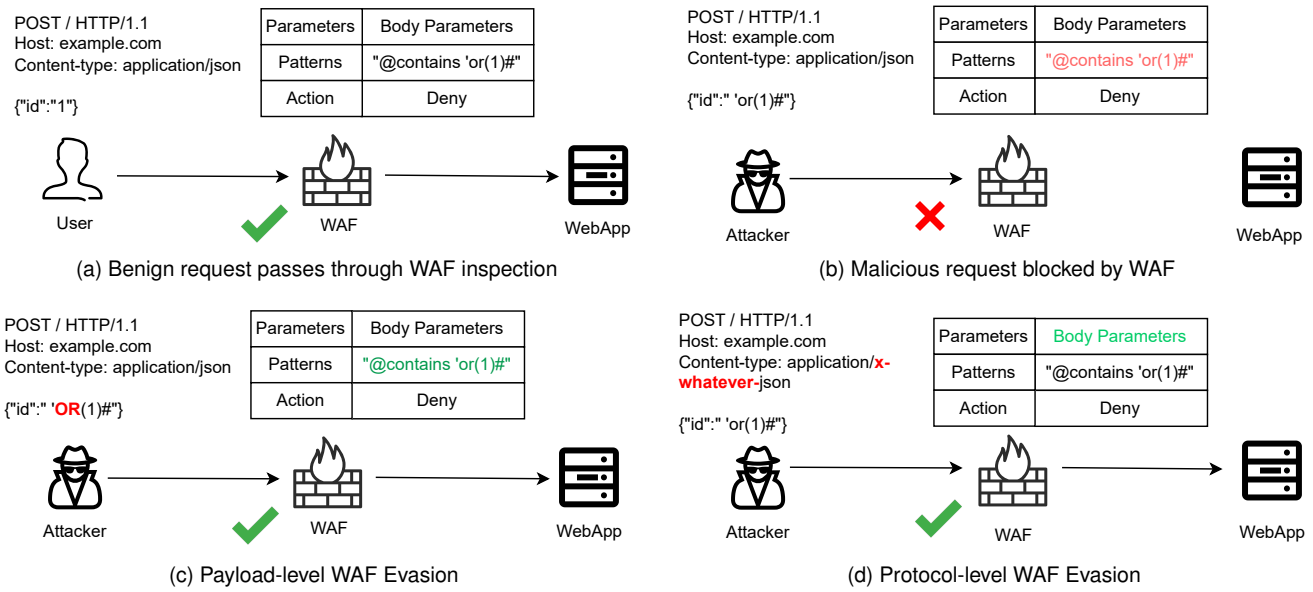


Figure 1. Four examples illustrating the WAF workflow, payload-level WAF evasion, and protocol-level evasions.

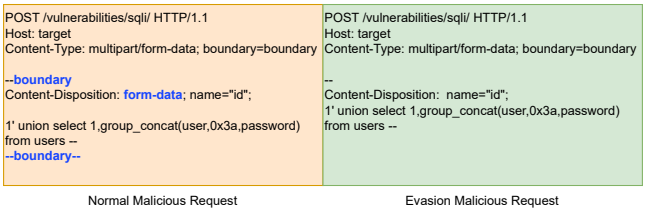


Figure 2. A motivating example of protocol-level WAF evasion we discovered. The malformed request on the right can bypass most major WAFs to exploit PHP-based applications.

the HTTP structures of the two requests are different. The request on the left, embedding the malicious payloads in a standard HTTP form, is typically rejected by the WAFs. In contrast, the request on the right can pass through the WAF, and the payload can be recognized by PHP-based applications with `$_POST`. This is due to the built-in HTTP parser of the PHP programming language, which exhibits high tolerance for HTTP protocol.

In essence, protocol-level evasions are rooted in the HTTP parsers of the WAF and the web applications. These vulnerabilities are typically *general WAF evasions* and can be exploited to deliver arbitrary malicious payloads. Furthermore, many are difficult to mitigate by simply updating the rules, thus posing severe security consequences on web applications.

3.2. Challenges

In this study, we propose to develop a novel fuzzing testing methodology to automatically discover protocol-level evasion vulnerabilities in WAFs to address the rising threat. The core idea is to generate a number of malformed requests to identify the parsing differences between the WAFs and the WebApps and detect protocol-level evasion vulnerabilities. However, there are three major challenges in developing this approach.

Challenge 1: How to generate and mutate testing requests efficiently?

The first challenge is to generate a high number of high-quality HTTP testing requests capable of triggering protocol-level evasion vulnerabilities. These requests should meet two requirements: (1) the testing requests should conform to, or closely approximate, the grammar of the HTTP protocol because HTTP messages are structured data. Invalid HTTP requests will be rejected by the WAF or web application without further processing; (2) the testing requests should always contain a specific payload, and this payload should not be modified during the mutation process. Prior WAF testing tools [18], [36], [43] concentrated on generating malicious payloads rather than malformed HTTP requests, which is insufficient for discovering protocol-level evasions.

To address this challenge, we have designed a grammar tree-based payload-aware generation approach, which includes two phases: generation and mutation. In the first phase, we generate initial requests as seeds. We first construct grammar trees based on the HTTP grammar from RFC documents and traverse the grammar trees to generate testing requests. This process starts from the root node and selects one of the corresponding grammar rules to generate its child nodes, iteratively expanding the tree until it reaches the terminators to generate requests.

In the second phase, we mutate the initial seeds to generate additional malformed data. This includes two types of mutations: (1) grammar-level mutation, achieved by ma-

nipulating the nodes on the grammar tree, and (2) byte-level variations, based on the grammar tree, i.e., mutating the bytes of leaf nodes in the grammar tree representing terminal symbols. This phase ensures the discovery of parsing vulnerabilities caused by non-standard RFC implementations.

To ensure our testing requests always contain the attacking payload, we insert special nonterminal symbols into the grammar tree, representing our predefined malicious payload. When traversing the grammar tree, the special terminal symbols are always included in the HTTP request. During the mutation of requests, we avoid performing deletion or modifying mutations at the grammar level, and we also do not perform byte mutations except when using the encoding operator on the terminal symbol. This ensures that our predefined malicious payload remains unaltered during the mutation process.

Challenge 2: How to test black-box WAFs effectively?

Many popular WAFs, especially Software as a Service (SaaS) WAFs, are provided as cloud services, for which we cannot get access to their source code or even the binary.

Black-box fuzzing tools, like Boofuzz [33], primarily rely on monitoring application crashes or error messages as indicators of potential vulnerabilities. However, protocol-level WAF evasions do not lead to crashes, and the HTTP response errors can only offer limited feedback. The lack of fine-grained feedback renders the testing ineffective to find vulnerabilities. On the other hand, white-box fuzzing, like Coverage Guided Fuzzing (CGF), has proven highly successful in discovering bugs. CFG selects seeds for the next round of mutations based on the program code coverage collected in each round. To collect code coverage from running programs, CGF requires patching the target program, which is not feasible for black-box WAFs.

To overcome this challenge, we combine the strengths of both white-box and black-box testing. Firstly, we leverage open-source web applications for white-box testing, using their code coverage to guide the generation of testing HTTP requests. If the requests pass web applications validation and the attacking payload can be recognized by these applications, we then forward these requests to commercial WAFs for black-box testing.

The key observation behind this approach is that both the HTTP parsers in applications and WAFs adhere to HTTP protocol standards. Therefore, using open-source web application codes can assist in efficiently creating high-quality HTTP test requests. This approach also reduces the testing requests for commercial WAFs and accelerates the fuzzing process, as invalid requests are rejected by web applications in our local environments before being forwarded to WAFs.

Challenge 3: How to detect protocol-level WAF evasion automatically?

Previous fuzzing approaches like AFL have achieved great success in identifying memory vulnerabilities by monitoring program exceptions or memory errors to determine whether a vulnerability has been triggered [20], [32], [34], [46], [48]. However, protocol-level WAF evasions typically don't trigger these exceptions and these approaches will miss these evasions. On the other hand, previous WAF testing work [18], [43] focuses on payload-level evasions and doesn't examine the HTTP parsing behaviors of different web application frameworks.

To address this challenge, we have developed a new vulnerability detector to detect parsing vulnerabilities between the WAF and the web applications in a timely manner. For a given testing request, we consider it as a valid protocol-level WAF evasion if both the following requirements are satisfied: (1) the request containing the malicious payload can pass through the WAF to the web applications; (2) the native built-in interface of web application frameworks can recognize our predefined malicious payload in HTTP parameters.

Thus, our vulnerability detector includes two parts: (1) *WAF Validator*, which checks if the request can pass through any WAF, and (2) *WebApp Validator*, which extracts HTTP parameters from different web application frameworks, including path parameters, query parameters, header parameters, and body parameters, to check if any match our predefined payload. As some real-world WAFs modify the message when forwarding requests, we save the forwarded request data in the *WAF Validator* and then forward it to the *WebApp Validator* for the 2-step validation.

4. Design and Implementation

4.1. Workflow

Figure 3 shows the workflow of *WAF Manis*, which can be divided into 9 steps: (1) We collect the grammar rules, such as ABNF rules, from the HTTP RFC documents. (2) These collected rules are forwarded to the*Generator*, which constructs grammar trees and generates HTTP requests as initial seeds. These initial requests are stored in the *Corpus*. (3) In the evolutionary fuzz loop, the program selects one testing request from the *Corpus* to be mutated by the *Mutator*. (4) The mutated requests are forwarded to the*WebApp Executor* for execution and then to the *WebApp Validator* for validation. (5) The *WebApp Executor* collects the code coverage information during execution, filtering out good testing requests that contribute to code coverage and adding them to the *Corpus*. (6) If a testing request passes the *WebApp Validator* – meaning the predefined malicious payload can be recognized – it is then forwarded to the*WAF Validator* for black-box testing. (7) If a testing request passes *WAF Validator*, it is then forwarded to the *Evasion Sample Centrifuge*, and the bypassed WAF is recorded. (8) The *Evasion Sample Centrifuge* replays and minimizes this evasion example to validate the vulnerability. The validated testing request is then added to the *Corpus* for further mutation. (9) The corresponding raw HTTP message will be saved in *Evasion Samples*.

To summarize, the *Generator* and *Mutator* are designed to generate and mutate high-quality HTTP requests containing malicious payloads for testing. *WebApp Executor* sends these mutated HTTP requests to the*WebApp Validator* and tracks their code coverage during execution. The *WebApp*

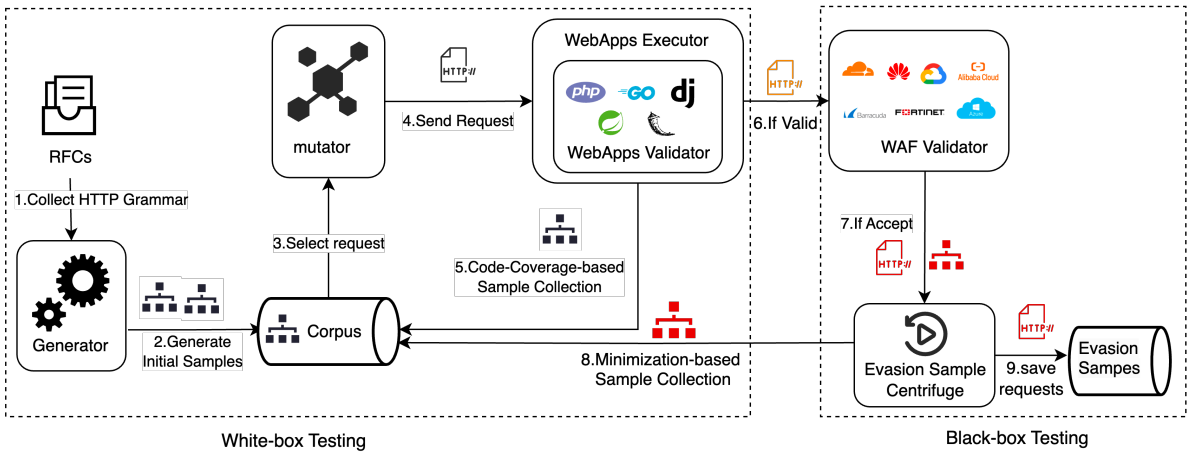


Figure 3. WAF Manis Workflow

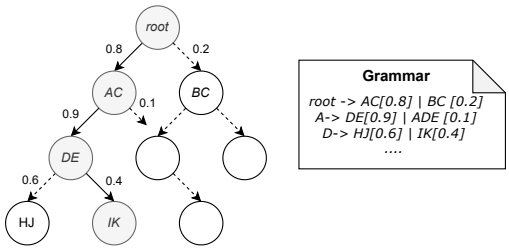


Figure 4. Generating new sample of initial corpus

Validator and *WAF Validator* evaluate whether the request, with its target payload, can be recognized by the web application or successfully bypass the WAF inspection. Lastly, the *Evasion Sample Centrifuge* is designed to minimize and re-validate the WAF evasion samples. We will elaborate on these modules in the sections below.

4.2. Generator and Mutator

The *Generator* and *Mutator* are developed to generate and mutate high-quality HTTP requests for testing, as described in section 3. We extracted the grammar rules of HTTP requests from the RFCs related to the HTTP protocol (including RFC 1867, 2046, 2231, 2616, 7578, RFC 7230-7240). Then, we provide these grammar rules to the *Generator* to construct the grammar tree. We define an intermediate variable serving as the link between the mutation and the further generation of the corresponding byte stream. This intermediate variable is stored and represented in the structure of a grammar tree, with any non-leaf node representing a nonterminal symbol and any leaf node representing a terminator. In particular, we set two special nonterminal symbols, `taint_key` and `taint_value`, representing the key-value pairs of our predefined malicious parameters. We extract context-free grammar (CFG) productions from related RFC standards for any nonterminal symbol, which are shown in Figure 4. Each sample of the initial corpus is generated by creating a tree with the start symbol as the root. As the algorithm described in Appendix 1, the *Generator* will expand nonterminal symbols in the tree by randomly selecting possible right-hand sides in the production according to their respective weights. Once the tree is fully expanded, the terminal symbols are combined to form the HTTP request.

By repeating the generation several times, *WAF Manis* collects all the generated samples as the initial corpus. Based on the initial corpus, the *Mutator* randomly selects a field from an original request and applies either grammar-level or byte-level mutations to it. For grammar-level mutation, the mutator selects a non-leaf node from the grammar tree represented by the intermediate variable and applies the following two strategies: 1) delete the grammar node: remove the sub-tree of the non-leaf node from the grammar tree, and 2) duplicate the grammar node: copy the sub-tree of the non-leaf node and add the sub-tree under the parent node of the node. For byte-level mutation, the mutator can select a leaf node from the grammar tree represented by the intermediate variables and apply the following three strategies: 1) add a character: insert a random character at a random position in the leaf node, 2) delete a character: delete a character at a random position in the leaf node and 3) encode characters: apply encoding rules to characters, including urlencode, quote-printable, base64, and other encoding methods. Specifically, to ensure that the contents of our predefined malicious parameters remain unchanged during the mutation process, we do not perform deletion mutation on the two special nodes including `taint_key` and `taint_val`, and also do not perform byte mutation except encoding operator on the terminal symbol pointed to by these two nodes. Otherwise, it may cause false positives due to missing malicious parameters in the requests.

4.3. WebApp Executor

To ensure the smooth operation of each fuzzing iteration, the Executor needs to satisfy at least three conditions: 1) The Executor should launch the WebApp in a refreshed state for each fuzzing iteration; 2) The Executor should return the final execution result and timely terminate each fuzzing iteration; 3) the Executor can obtain the code coverage information of the WebApp during each round of Fuzz.

In the classical CGF model, the input samples are passed through shared memory or input files, and since the test target is stateless, the executor only needs to wait for the main function under test to finish processing before proceeding to the next round. However, for WebApps whose input is passed through network protocols, the expected end signal or end time is uncertain. HTTP protocols are stateless protocols, but in the fuzzing process, the request sample may break the structure of the HTTP protocol when getting mutated. This can cause the target program to close the socket earlier than expected or keep waiting for the remaining parts of the message. Moreover, for a complex web application framework, it is challenging to locate the address of the program where the socket will be released in advance. As a solution, we measure in advance the average processing time of the HTTP request by the target program, and we set double the average time as the timeout time. This helps to provide a reasonable timeout period that allows the target program to proceed even if there are variations in the processing time caused by mutations.

For the last goal, we have selected the corresponding state-of-the-art (SOTA) fuzzing framework for different development languages. For example, we use LibAFL to fuzz PHP and Atheris to fuzz Python. To collect code coverage information during execution, we modify the target WebApps by inserting special instructions or using dynamic binary instrumentation tools. In the fuzzing process, when a sample is sent to WebApps, the Executor records the code coverage information to the shared memory. After WebApps return the response or the timeout is exceeded, the coverage feedback module collects the coverage information and clears the shared memory. This approach allows us to effectively collect code coverage data for further analysis and improvements in the fuzzing process.

4.4. WebApp Validator and WAF Validator

To detect protocol-level WAF evasion, we utilize the interfaces and functions provided by the web framework which we refer to as *GetParameter Functions*. These functions are used to extract the target parameter from HTTP requests. Once we obtain the target parameter, we perform validation to check if it matches the predefined parameter key-value pairs: `taint_key` and `taint_value`.

When the obtained parameter key-value pairs match the key-value pairs we filled in beforehand when generating the sample, we consider the sample to be parsed properly and we return the response with a specific status code S_{WEBAPP_PASS} to indicate the end state of successful parsing. The constant S_{WEBAPP_PASS} we defined is not a standard status code, which helps avoid the interference of the server's original status code. Similarly, we design the WAF-protected WebApp (WAF Validator) to return the status code S_{WAF_PASS} regardless of any HTTP request received and save the raw request for further analysis.

The advantage of this strategy is that the reject characteristics of different WAFs may be inconsistent, but the accepted characteristics must be the response from the WAF-protected WebApp. Therefore, when the response status code S_{WAF_PASS} is received in the WAF verification process, the request sample can be considered to have successfully bypassed the WAF. To further improve efficiency, we can simultaneously send the sample to different WAF deployment addresses. If the sample successfully bypasses any of the WAFs, we put it into the *Evasion Sample Centrifuge* with the corresponding WAF identifier attached. Additionally, we save the raw request message from the WAF to monitor whether the request sample gets modified by the WAF.

4.5. Evasion Sample Centrifuge

It seems that as long as the sample has passed both the *WebApp Validator* and the *WAF Validator*, we can conclude that the sample can bypass the WAF and transmit malicious parameters to the WebApp. However, there are two problems in practice, one is that in the real world, a significant number of cloud WAFs will modify the original HTTP request in some way, and the WebApp may receive a modified request sample rather than the original request sample. Therefore, the above judgment may still be at risk of false positives and false negatives. The other issue is that a mutated sample may have multiple factors that contribute to the WAF evasion, or may simply add some redundant fields to the original bypass factor. For Example, sample 5a is the initial sample, which becomes sample 5b after mutation, which is able to bypass the WAF. By minimizing the bypass sample b, we can find that there are actually two factors that determine the ability of sample 5b to bypass the WAF, which can be represented by the minimized sample 5c and the minimized sample 5d, respectively. If we keep mutating the sample 5b, it will not only make it difficult for us to identify the true influencing factors, but it will also result in new samples of variation that may appear inconsistent but share the same influencing factors.

To ensure the accuracy of the samples and isolate the different WAF bypass factors in a single mutated sample, we developed the *Evasion Sample Centrifuge* module. The main idea behind this module is to replay and minimize the evasion sample by iteratively removing nodes from the grammar tree until the corresponding request sample cannot bypass the WAF or fails to be correctly parsed by the WebApp. The replay and minimizing process can be described as algorithm 2 as shown in the Appendix. To ensure that the samples can bypass the WAF and can be correctly parsed by the WebApp, we send the request samples saved by the *WAF Validator* to the WebApp for

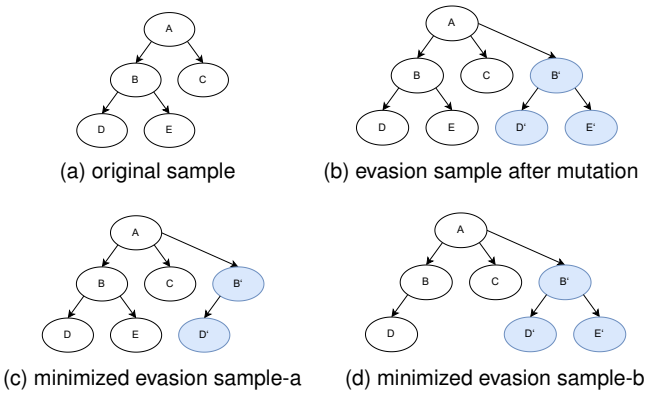


Figure 5. Evasion Sample Minimization

secondary verification, instead of sending the original samples directly to the *WebApp Validator*. Specifically, we also deploy the *WebApp Validators* in this module. Throughout the continuous node deletion process, we send the sample after node deletion to the corresponding WAF, whose *WAF Validator* will save the real request sample that the WAF will forward to the WebApp, and this sample will be sent to the deployed *WebApp Validator*. If this sample gets correctly parsed by the WebApp validator, then we can conclude that this sample can cause an effective WAF bypass in the real world.

After minimization, samples in c_{min} will be added to the corpus to undergo further mutation, and samples in s_{min} will be stored in *Evasion Samples*. Furthermore, the above algorithm is only for one combination of one specific WAF with a single *WebApp Validator*. Since we save bypass samples in the corresponding WAF Validator, we can test multiple *WebApp Validator* in parallel, which can help us test some web frameworks written in programming languages that do not yet have a mature Coverage-Guided Fuzz framework, such as ruby.

In general, this module provides two optimizations for *WAF Manis*. For one thing, the minimization is supplementary to the pursuit of high code coverage. According to Figure 3, we actually establish two corpus collection approaches to determine which sample should be added to the corpus and undergo mutation first: 1) code coverage information in WebApp Executor and WAF evasion status. The increment in code coverage means that the sample triggers more code paths, which may include new unstable parsing features or old deprecated features that are still present in the code. This can lead to inconsistencies in the semantic understanding of the same HTTP request samples across implementations, resulting in WAF evasion. 2) "reject or not" information that we can get from the black-box WAFs to guide the sample mutation, which can help WAF Manis to find samples with incomplete protocol structure that may have low code coverage but can also bypass WAF while being parsed correctly by WebApp, such as the evasion example shown in the figure 2.

For another, the minimization makes it easy to classify different evasion samples. As shown in Figure 5, after minimization, we separate two different types of evasions samples 5c and 5d from sample 5b. In addition, it helps avoid a large number of mutated samples sharing the same factors that contribute to WAF evasion. Since the samples we add to the corpus are those that remove the key nodes and contribute to the WAF evasion, they could explore more possible evasion approaches after further mutation.

5. Evaluation and Findings

5.1. Methodology and Testing Environment

Web Framework	Language	Version	Github Star
Laravel	PHP	9.19	73.8k
Django	Python	4.15	71.6k
Gin	Go	1.8.1	69.6k
Spring-boot	Java	2.7.5	68k
Flask	Python	2.1.3	63.4k
Express	Node.js	4.18.2	61.2k
Fastapi	Python	0.88.0	59.5k
Nest	Node.js	9.0.0	57.6k
Rails	Ruby	7.0.4	53.1k
Meteor	Node.js	2.8.0	43.5k
Koa	Node.js	2.14.1	34.1k
ASP.NET Core	.NET	6.0.12	32k
Beego	Go	2.0.1	29.9k
Symfony	PHP	6.2.4	28.5k
Fastify	Node.js	4.11.0	27.7k
Echo	Go	4.10.0	25.9k
Sails	Node.js	1.5.3	22.6k
Rocket	Rust	0.5.0-rc2	20.9k
CodeIgniter	PHP	4.0	18.2k
Webpy	Python	0.62	5.8k

TABLE 1. TESTED OPEN SOURCE WEB FRAMEWORKS. (GITHUB STAR COUNT AS OF JUNE 25, 2023)

To evaluate WAF Manis, we systematically analyze 8 commercial WAFs, 6 open-source WAFs, and 20 popular web frameworks, as shown in table 2 and table 1. We collected commercial WAFs according to the global WAF market share report [38] and the Forrester WAF report [12] and chose those WAFs that we could register accounts and perform security testing. For open-source WAF, our list was collected by exploring the "WAF" topic on GitHub [3] and subsequently selected those projects that have garnered the most stars. We collected top-tier frameworks according to the rankings from OSSinsight [31], which evaluates frameworks based on GitHub stars, pull requests, and issues.

In this evaluation, We developed *WebApp Validator* for 20 web frameworks and implemented *WebApp Executors* to collect code coverage of *WebApp Validators* based on PHP, Python, and Rust languages. For each *WAF Manis* process, we chose one *WebApp Validators* to collect code coverage for guiding the mutation, while we tested all the *WebApp Validators* in the *Evasion Sample Centrifuge* Module of each *WAF Manis* process to find WAF evasion vulnerabilities of the 20 web frameworks. Full tested *GetParameter* Function list is in Appendix table 5.

Type	WAF	Evasion Samples	Affected Web Framework[1]
Commercial	Microsoft Azure WAF	5	13/20
	Google Cloud Armor	5	13/20
	Alibaba Cloud WAF	21	20/20
	Cloudflare WAF	38	20/20
	Huawei Cloud WAF	40	20/20
	Safeline WAF	22	20/20
	Fortinet WAF	40	20/20
	Barracuda WAF	8	20/20
Open Source	ModSecurity	2	2/20
	Naxis	2	2/20
	OpenWAF	13	20/20
	Janusec	21	17/20
	WAFbrain	49	20/20
	HiHTTPs	45	20/20

TABLE 2. SUMMARY OF PROTOCOL-LEVEL WAF EVASION VULNERABILITIES WE DISCOVERED
1. THE AMOUNT INCLUDES ALL EVASION CASES FOR THE CORRESPONDING WAF

Comparison with State-of-the-Art Tools. Prior to deploying our tool for testing various WAFs, we initially utilized two state-of-the-art (SOTA) WAF testing tools: xwaf [6] (a wrapper of SQLmap [43] specifically designed to identify WAF evasions) and WAFNinja [27] to assess our collection of WAFs. These results illustrate that both xwaf and WAFNinja were unable to bypass the evaluated WAFs, as shown in Appendix table 3.

5.2. Findings

After about three days of fuzzing, WAF Manis had generated around 108668 samples for each WebApp Validator.In total,*WAF Manis* found 311 protocol-level evasion cases affecting all tested WAFs and web applications. We list some of the mutated samples found by WAF Manis in Appendix table 4, and the final results of the vulnerabilities found are shown in table 2.

We classified those evasion cases into three categories based on the possible causes and WAFs' behaviors: (1) Parameter Type Confusion; (2) Malformed Parameter Structure; (3) RFC Support Gap. We present the examples of the three evasion types in Figure 6 , and the details of each category are illustrated in the following.

Category 1: Parameter Type Confusion. To parse the content of a parameter correctly, the first primary requirement is to identify the type of the parameter accurately. However, there are multiple fields and values in the HTTP implementation standard that can be used to indicate the type of the parameter, so if there are semantic gaps between the WAF and the WebApp, an attacker can craft a malicious payload to confuse the WAF to parse the payload content with the incorrect parser, causing a WAF evasion.

Multiple Content-Type Headers. According to RFC 7230, A sender *MUST NOT* generate multiple header fields with the same field name in a message unless either the entire field value for that header field is defined as a comma-separated list or the header field is a well-known exception. However, many implementations still tolerate these behaviors. When dealing with multiple Content-Type headers, some WAFs, such as ModSecurity, will use the value of the first header as the basis for selecting a parameter parser, while some web frameworks, such as Flask, will use the last one as the basis for selecting the parser. So an attacker can confuse the WAF into thinking that the request body is JSON but smuggle data as urlencoded-form by setting two Content-Type Headers, the first one is `application/json`, the second one is `application/x-www-form-urlencoded`. Figure 7a shows an example produced by *WAF Manis*.

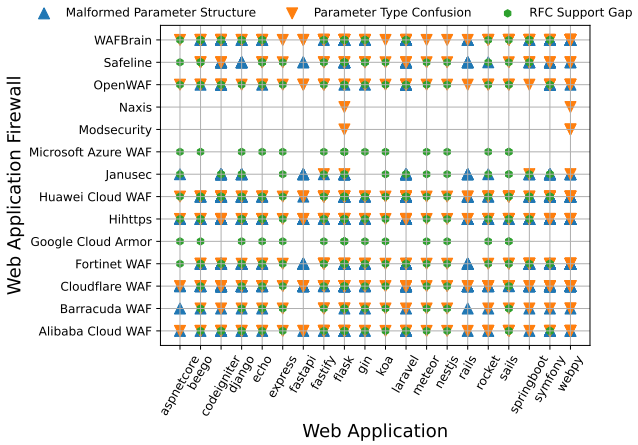
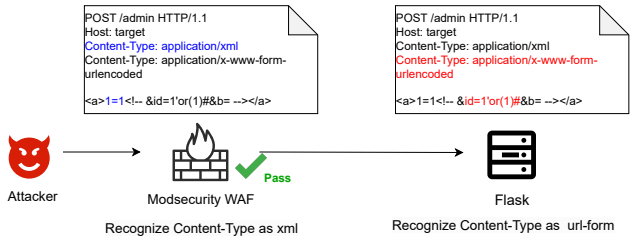


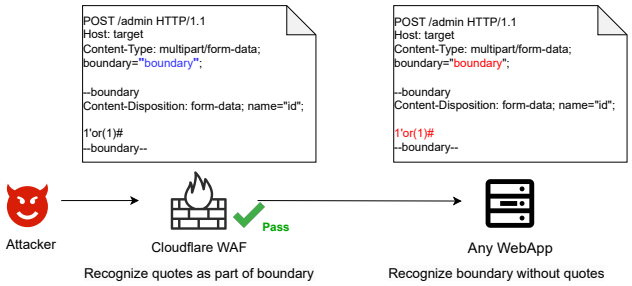
Figure 6. WAFs/WebApps pairs affected by three types of attacks

Content-Type Variant. According to RFC 7230 and RFC 2045, The Content-Type header consists of a media type followed by optional parameters. The media type is typically represented as a MIME type, which is a standardized format for describing the nature and format of a file. The HTTP protocol does not restrict the media type for submitting parameters, but the current de facto standards include `application/x-www-form-urlencoded`, `application/json`, `application/xml` and `multipart/form-data`. Most WAFs currently support the detection of these parameters (Google Cloud armor only supports `application/x-www-form-urlencoded` and `application/json` [16]). However, in practice, the actual payload and these protocol names are not in the one-to-one correspondence. For example, as shown in Figure 6, flask will treat the body as JSON when Content-Type values start with `application/x-` and end with `+json`. And webpy will treat arbitrary mime values starting with `multipart` as `multipart/form-data`, so an attacker can construct a Content-Type value that can be accepted by the WebApps but confusing the WAFs from choosing a correct parser, resulting in the evasion.

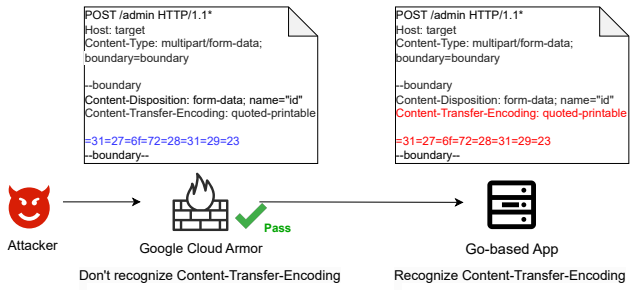
Category 2: Malformed Parameter Structure. Besides tricking the WAF into using the wrong parser to parse HTTP requests, an attacker can also construct malformed data so that the WAF will not find the parameter containing the malicious payload while the WebApp takes them. By benefiting from the mutation and minimization of the *WAF Manis*, we have found many malformed samples that are capable of interfering with the parsing process of WAF, which is hard to discover through manual auditing.



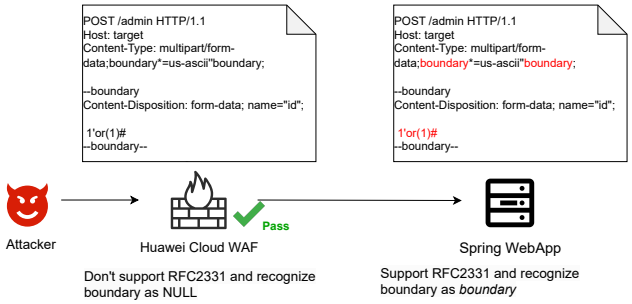
(a) Different Preferences of Content-Type headers (CVE-2023-38199).



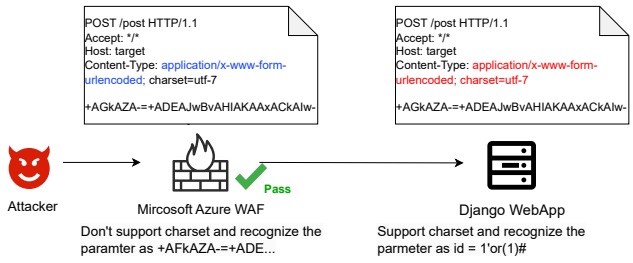
(b) Differences in parsing boundary parameter, which can exploit all web applications behind the Cloudflare WAF.



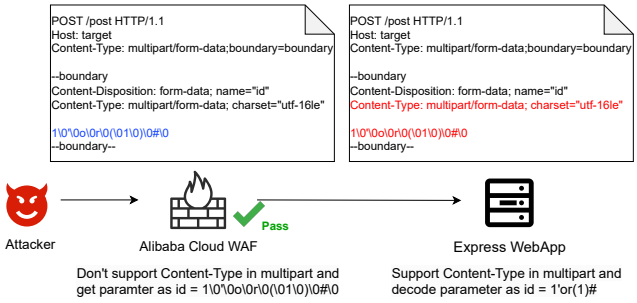
(c) Differences in recognizing Content-Transfer-Encoding (CTE) header, which bypasses major WAFs including Google Cloud, Fortinet, and Azure WAF to exploit Go-based web applications.



(d) Differences in supporting RFC 2331 standard, which can bypass Huawei Cloud WAF to exploit Java Spring applications.



(e) Differences in recognizing charset parameter header, which bypasses major WAFs including Microsoft Azure WAF to exploit Django web applications.



(f) Differences in supporting Content-Type header in multipart data, which can bypass Alibaba Cloud WAF to exploit Express-based applications.

Figure 7. Samples of protocol-level WAF evasion found by *WAF Manis*.

Malformed boundary parameter. According to RFC 2046, the body of a "multipart" media type field must contain one or more body parts, each preceded by a boundary delimiter and the last one followed by a closing boundary delimiter line. The boundary value is defined in the Content-Type header. During our evaluation, Cloudflare WAF will not parse `boundary="boundary"` value as `boundary`, so Cloudflare WAF will not parse the corresponding multipart data while all most popular web frameworks like Laravel, Springboot, Gin, and Flask will parse these parts. Figure 7b shows how the evasion occurs.

Malformed boundary separator. Through RFC 2046 indicates that the boundary delimiter line is defined as a line consisting entirely of two hyphen characters with a terminating CRLF, we found that many web framework implementations (eg. Django) can tolerance incomplete CRLF tokens with only one CR or one LF token while many WAFs including Fortinet, Huawei Cloud, and Alibaba Cloud can not parse them correctly and thus they get bypassed.

Category 3: RFC Support Gap. It is essential that all parties involved adhere to the specifications outlined in the RFC to ensure interoperability and security. However, during our evaluation, we found a number of WAFs and WebApps don't follow the current RFC standard. Apart from crafting malformed data which may throw warnings in the parsers, attackers can leverage the RFC support gaps between the WAFs and WebApps to make a legitimate-looking request while getting parsed by the WAFs and WebApps differently.

RFC 2231 Support. Based on RFC 2388, if the file name of the sender's operating system is not in US-ASCII,

the file name can be encoded with the method of RFC 2231. However, during our evaluation, the fuzzing results of *WAF Manis* show that not all WAFs and WebApps correctly implement this feature. With the grammar-level Mutation of *WAF Manis*, we found that some Web framework implementations even support other parameters to be encoded. Figure 7d shows the sample that can confuse WAFs to choose the correct subpart in `multipart/form-data` protocol by encoding boundary parameter or name parameter in RFC 2331 as `boundary*=us-ascii''boundary`, which results in the value of the boundary to `boundary`.

Deprecated Content-Transfer-Encoding Header. Once, it was recommended in RFC standards that the "content-transfer-encoding" header be supplied if the value of that part does not conform to the default encoding in RFC 2388. But in RFC 7578, the recommendation got deprecated and senders *SHOULD NOT* generate any parts with a Content-Transfer-Encoding header field. Notably, web frameworks such as Gin, Beego, Echo, and Flask still support these features during our evaluation. As illustrated in Figure 7c, an attacker can set `Content-Transfer-Encoding: quoted-printable` header to transfer encoded malicious payload in `multipart/form-data` protocol to bypass almost every WAF in the real world.

Charset Support. According to RFC 1866, there is no clue that `application/x-www-form-urlencoded` media type supports body encoding. In most WAFs and web frameworks, parameters on this MIME type are ignored. In particular, this MIME type does not support the charset parameter. However, as is shown in figure 7e there are some web frameworks (e.g Django) that will use the charset parameter in the Content-Type header to decode the request body, which means an attacker can encode malicious payload with utf-7 with Content-Type `application/x-www-form-urlencoded; charset=utf-7` to evade the detection of the WAF.

Content-Type supprot in multipart data. Notably, during our evaluation, we found that web frameworks including express, nest, koa, fastify, and sails support content-type in multipart data. As illustrated in Figure 7f, these web frameworks parse the `Content-Type` header in the part and use `charset` parameter in this header to decode the value of the part, so attackers can encode their malicious payload to evade the detection from almost every WAF, leaving these web frameworks in threats.

5.3. Case Study

We manually review the found evasion samples to estimate their real-world impact. Notably, we found 2 cases that are rooted in the PHP and Go programming languages, which can bypass major WAFs to exploit web applications written in those languages.

Case Study 1: RFC 1867 violations in PHP language. After manually reviewing the evasion samples for Laravel, Symfony, and Codeigniter, we found the common root cause of the evasions is that the parser built in PHP is using the "best effort parsing" mode to parse HTTP requests, which means the parser attempts to process input data to the best of its ability, even if the input contains errors or is not fully compliant with the expected syntax. As shown in Figure 2, there are a number of unexpected parsing behaviors against the RFC 1867. First, according to `find_boundary` function defined in `main\rfc1867.c`, PHP tries to find the boundary ignoring the padding lines before multipart data. Second, PHP only seeks for `name` and `filename` parameters in `Content-Disposition` header, which means `form-data` token defined in RFC 1867 can be omitted. Third, when the CRLF token between part headers and part body is missing, PHP will treat the first line after `Content-Disposition` header as the value of that part and stop parsing the remaining parts. Finally, PHP will not throw any exception when the terminal boundary is missing. Since the parsing behavior is built in the PHP engine, these evasion tactics can affect almost any web framework written in PHP.

Real-world Experiment. Given the potential ethical risks of conducting exploitation attacks against real-world websites, we conducted exploitation experiments against our websites deployed behind commercial WAFs. We conducted a real-world experiment to exploit this vulnerability for SQL injection attacks. We set up a WordPress website on our own VPS server, one with the vulnerability (CVE-2022-33965). Then we deploy Cloudflare WAF to protect our website. By exploiting the WAF evasion vulnerability, we are able to deliver a payload for CVE-2022-33965 to obtain sensitive data from the database, such as passwords.

Case Study 2: RFC 2616 and RFC 7578 violations in Go language. As shown in Figure 7c, some web frameworks like Gin, Beego, and Echo will decode quoted-printable form parameters, which can be leveraged to bypass WAF. The unexpected parsing behaviors arise from the standard libraries of Go: `http` and `mime`. `Request.ParseMultipartForm` method in `http` library will call `Part` struct in `mime` library to parse multipart parameters. However, according to RFC 2616, unlike MIME, HTTP *does not* use Content-Transfer-Encoding, and *does use* Transfer-Encoding and Content-Encoding. Furthermore, according to RFC 7578 section 4.7 Senders *SHOULD NOT* generate any parts with a Content-Transfer-Encoding header field. This evasion tactic has a huge impact on WebApps based on Go because the wrong behavior is introduced in the standard library.

Real-world Experiment. We conducted a real-world experiment to demonstrate the threat. We set up a Gogs (A popular Git service written in Go language) instance on our own VPS server, which has the vulnerability (CVE-2022-0415). Subsequently, we deployed a Fortinet WAF to safeguard our Gogs website. By leveraging the evasion vulnerability, we successfully delivered the malicious payload of CVE-2022-0415 to the Git service, enabling us to gain complete control over the target system, such as accessing sensitive data, modifying or deleting files, and executing commands.

6. Discussion

6.1. Responsible Disclosure

Ethical Consideration. In the whole process of our experiments, we try our best to follow the best industry practice of security research. First, we set up popular open-source WAFs on our controlled servers to verify the effectiveness of our WAF Manis, throughout this work. Second, for well-known commercial WAF providers like Cloudflare, Fortinet, Huawei Cloud, and Alibaba Cloud, we strictly follow their bug bounty rules to perform controlled experiments by sending small-scale traffic to our own websites. Third, both open-source and cloud WAFs encourage security tests through bug bounty programs, and we responsibly disclosed the details to them. Our contact results are summarized as follows.

Cloudflare. They acknowledged our report and rewarded us for reporting the issue of WAF evasion. They told us that they generally don't consider WAF evasion bugs for bug bounty purposes, but as our report provided a more notable finding than most, they offered us a cash reward in thanks.

Fortinet. They accepted our report and confirmed the vulnerability. The vulnerability now has been fixed.

Alibaba Cloud. They accepted our reports as critical vulnerabilities and provided us $900 bug bounty rewards for reporting the vulnerabilities.

Huawei Cloud. They appreciated our work [44], accepted our reports and provided us $550 bug bounty rewards for reporting the vulnerabilities.

ModSecurity&Core Rule Set. They confirmed our report and fixed the vulnerabilities. CVE-2023-38199 is assigned for the vulnerabilities.

PHP Security Team. They thanked our report and confirmed our findings as valid security issues.

Go Security Team. They expressed their gratitude for our report and would address it as a hardening measure.

Others. We have contacted other relevant WAF vendors and are looking forward to receiving their feedback.

6.2. Mitigation

Our work underscores the unfortunate fact that the parsing process has become the Achilles heel in WAF defense. Our study discovered a number of protocol-level evasions. We suggest possible mitigations as follows.

One possible approach is *normalization*. Previous work on TCP/IP protocol ambiguities [22] has proposed normalization and canonicalization methods to remove potential ambiguities to mitigate NIDS evasions. However, applying such techniques to HTTP protocol may be not infallible. Because the nature of HTTP as a text-based protocol, coupled with its extensive flexibility and redundancy, makes correct normalization challenging. Furthermore, even if WAFs enforce correct canonicalization or allowlist-based mitigation, web applications may have their "dialect" and interpret the request differently with the WAFs. For instance, in our evaluations, we observed that some WAFs attempted to normalize HTTP requests when forwarding, but still fail to prevent evasions.

Another possible approach is *Runtime Application Self-Protection (RASP)*. This approach integrates security policies directly into the web application's runtime environment, actively analyzing the parameters and application logic to identify and mitigate potential threats. As having this vantage point, RASP can directly collect the same parameters from Web applications, thus avoiding potential protocol parsing ambiguities. However, RASP also has its limitations, such as the complexity of deep integration, performance impact, and limited protection scope.

The third approach is *fuzzing WAF implementations with methodologies like WAF Manis.* The vulnerabilities in this research arise from the parsing inconsistency between the WAFs and the WebApps, which cannot be revealed with traditional fuzzing tools that only explore either WAF or WebApp. To address this gap, our tool, WAF Manis, has been designed to specifically target the interaction between WAFs and WebApps, enabling a more effective identification of potential vulnerabilities. We will open source our tool at https://github.com/EkiXu/WAFManis once all the identified vulnerabilities are fixed by affected vendors.

At high-level, we suggest several broader considerations when implementing and designing protocols.

In implementation, follow RFC standards and be consistent in HTTP-level parsing. First, both WAFs and WebApps should strictly follow related implementation standards in the RFC. Second, for ambiguous or undetailed definitions in the RFCs, we recommend following well-known HTTP implementations as industry standards. In these ways, protocol-level evasion caused by inconsistency can be mostly avoided.

In design, keep simplicity and apply secure defense. HTTP implementations normally follow the Postel's law of "be liberal in what you receive" to keep robustness, however, the primary goal of WAF is to defend against possible attacks, thus the WAF parser should keep simplicity in its core function implementation, and apply secure defense to avoid any HTTP-level confusion concerning either parameter type, malformed structure, or any support gaps.

Above all, as these vulnerabilities are caused by semantic gaps among multiple implementations with different understandings of the same data, a concerted effort and a systematical security view are needed to mitigate this problem.

7. Related Work

In this section, we present work related to WAF evasion and network fuzzing.

WAF Evasion. Prior research [6], [27], [43] achieved success in automating the discovery of payload-level evasions. They simply facilitate the encoding and obfuscating approaches of the original attack payload in a known bypass cheat sheet in order to bypass the WAF. To cope with more complex WAFs, researchers [7], [18], [23], [36] have applied

evolution algorithms on payloads. These evasion methods typically begin by generating a malicious payload that is initially non-evasive based on the underlying grammar. Subsequently, evolutionary algorithms are applied to mutate the payload multiple times, guided by specific metrics that facilitate the transformation process, ultimately enabling the payload to transit from its initial non-evasive state to an evasive state that can bypass WAFs. However, these researches mainly focus on detecting SQL injection evasion samples and their techniques cannot discover protocol-level evasions.

Previous studies identified protocol-level semantic gaps, which indicates possible WAF evasions. HTTP Parameter Pollution Attack (HPP) [11] is a classical type of protocol-level evasion. HPP occurs when an attacker deliberately modifies the parameter structure or introduces duplicate or conflicting parameters to confuse the WAF and server-side processing. If the WAF parses these parameters differently from the WebApps, it is possible for the WAF to overlook malicious payloads within the parameters, thus leading to a bypass. Balduzzi et al. [9] proposed an automated approach for the discovery of HPP by scanning and analyzing parameters. However, HPP is limited to parameter-based attacks, whereas many potential vulnerabilities can only be triggered by requests in formats such as `form-data` and others.

Differential testing is a software testing technique that focuses on comparing the behavior or output of two or more similar implementations of a program or system. which can indicate protocol-level evasion. T-Reqs [24] presents a grammar-based differential fuzzer to find HTTP Request Smuggling (HRS) [28] samples. It first tests each web middleware target in isolation, and then compares responses from targets to identify the pairs that behave differently, indicating potential HTTP semantic gaps. Unfortunately, T-Reqs is limited to HRS and lacks testing on common web application frameworks.

Previous work has proposed protocol grammar-based fuzzing approaches to identify censorship evasions. For example, geneva [10] and CenFuzz [37] also utilized HTTP grammars to generate test cases and fuzz censorships. Our work differs from them in two aspects: (1) the vulnerability detector is different. Geneva and CenFuzz focus on bypassing censorship, while our work focuses on web application firewalls evasion which require examining web applications for parametric integrity detection. (2) Approach to fuzzing is different. Our work employs the code coverage of open source web frameworks to guide new testing request generation, while Geneva and CenFuzz operate within black box testing.

Generally, protocol-level WAF evasion vulnerabilities fall under the category of semantic gap attacks. Similar semantic gap attacks have been identified in various systems, including HTTP implementations [13], [41], CDN systems [15], [47], and email systems [14]. The methodologies we have proposed could potentially be adapted and applied to these systems as well, addressing similar discrepancies in interpretation.

Network Fuzzing. Previous studies have also attempted to leverage fuzzing techniques to discover vulnerabilities in Network services. However, previous fuzzers like AFL [46] are primarily designed for file format fuzzing, which requires additional modifications or tooling to support complex network protocols. The most straightforward approach is to interact with the target WebApp over the network [1], [45]. AFL++ [20] integrates preeny [42] that converts socket-based I/O to file-based I/O, which provides basic support for network service fuzzing.

Another challenge faced by network protocol fuzzing is that vanilla fuzzers are not designed for stateful network protocols. Thus, the generated inputs are likely to fail to comply with the required format or order of the protocol, making it difficult to reach deep areas in the target program. Pham et al. [34] proposed the AFLNet which addresses the limitations of AFL for network protocol fuzzing. AFLNet aims to discover vulnerabilities in network-based applications with complex protocols. Using response codes to represent states, AFLNet is capable of automatically inferring the state model of the target service and generating input sequences through mutations that can reach deeper states. To address the issue of insufficient information in response codes, STATEAFL [30] and NSFuzz [35] proposed methods that utilize memory states and program variables to represent service states. TCP-Fuzz [48] aims to discover semantic gaps between different TCP stacks using differential fuzzing. Nyx-Net [40] applies snapshots on network service fuzz to improve efficiency. However, none of these published research and tools can uncover the semantic gap of HTTP parsers between WAFs and WebApps.

8. Conclusion

In this paper, we have introduced WAF Manis, a novel automated tool designed to detect web application firewall evasions. These evasions exploit differences in protocol parsing between the WAFs and the WebApps. Our evaluation of 280 combinations (14×20) of real-world deployed systems demonstrates WAF Manis' ability to effectively detect 311 protocol-level evasion cases, affecting all tested WAFs and web applications. Our tool can assist developers in detecting vulnerabilities before they are exploited by attackers. We have responsibly disclosed all identified vulnerabilities to the affected providers, receiving acknowledgments and bug bounty rewards from Cloudflare WAF, Fortinet WAF, Alibaba Cloud WAF, Huawei Cloud WAF, ModSecurity, the Go security team, and the PHP security team. We hope this work can inspire the community to discover and reduce semantic gap attacks between WAFs and WebApps.

9. Acknowledgement

We sincerely thank all anonymous reviewers and our shepherd for their insightful and constructive feedback to improve the paper. This work was supported by the National Natural Science Foundation of China (grant #62272265).

References

[1] "Aflplusplus fuzzing network services," https://github.com/AFLplusplus/AFLplusplus/tree/networking.

[2] "Alibaba waf challenge game," https://security.alibaba.com/online/detail?spm=0.0.0.0.4oWvph&type=1&id=147&tab=1.

[3] "Github waf topic," https://github.com/topics/waf.

[4] "Perf tools support for intel® processor trace," https://perf.wiki.kernel.org/index.php/Perf_tools_support_for_Intel%C2%AE_Processor_Trace#What_is_Intel.C2.AE_Processor_Trace.

[5] "Information Supplement: Application Reviews and Web Application Firewalls Clarified," PCI Security Standards Council, Standard, Oct. 2008.

[6] 3xp10it, "xwaf," https://github.com/3xp10it/xwaf.

[7] D. Appelt, C. D. Nguyen, A. Panichella, and L. C. Briand, "A machine-learning-driven evolutionary approach for testing web application firewalls," *IEEE Trans. Reliab.*, vol. 67, no. 3, pp. 733–757, 2018. [Online]. Available: https://doi.org/10.1109/TR.2018.2805763

[8] E. Athanasopoulos, V. P. Kemerlis, M. Polychronakis, and E. P. Markatos, "ARC: protecting against HTTP parameter pollution attacks using application request caches," in *Applied Cryptography and Network Security - 10th International Conference, ACNS 2012, Singapore, June 26-29, 2012. Proceedings*, ser. Lecture Notes in Computer Science, F. Bao, P. Samarati, and J. Zhou, Eds., vol. 7341. Springer, 2012, pp. 400–417. [Online]. Available: https://doi.org/10.1007/978-3-642-31284-7_24

[9] M. Balduzzi, C. T. Gimenez, D. Balzarotti, and E. Kirda, "Automated discovery of parameter pollution vulnerabilities in web applications." in *NDSS*, 2011.

[10] K. Bock, G. Hughey, X. Qiang, and D. Levin, "Geneva: Evolving censorship evasion strategies," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 2199–2214. [Online]. Available: https://doi.org/10.1145/3319535.3363189

[11] L. Carettoni and S. di Paola, "Http parameter pollution," https://owasp.org/www-pdf-archive/AppsecEU09_CarettoniDiPaola_v0.8.pdf.

[12] S. Carielli, "Now tech: Web application firewalls, q2 2022," https://www.forrester.com/report/now-tech-web-application-firewalls-q2-2022/RES177433.

[13] J. Chen, J. Jiang, H. Duan, N. Weaver, T. Wan, and V. Paxson, "Host of troubles: Multiple host ambiguities in http implementations," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 1516–1527.

[14] J. Chen, V. Paxson, and J. Jiang, "Composition kills: A case study of email sender authentication," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 2183–2199.

[15] J. Chen, X. Zheng, H.-X. Duan, J. Liang, J. Jiang, K. Li, T. Wan, and V. Paxson, "Forwarding-loop attacks in content delivery networks." in *NDSS*, 2016.

[16] G. Cloud, "Post body inspection limitation," https://cloud.google.com/armor/docs/security-policy-overview#post-body.

[17] corerulest, "Owasp modsecurity core rule set," https://coreruleset.org/.

[18] L. Demetrio, A. Valenza, G. Costa, and G. Lagorio, "Waf-a-mole: evading web application firewalls through adversarial machine learning," in *SAC '20: The 35th ACM/SIGAPP Symposium on Applied Computing, online event, [Brno, Czech Republic], March 30 - April 3, 2020*, C. Hung, T. Cerný, D. Shin, and A. Bechini, Eds. ACM, 2020, pp. 1745–1752. [Online]. Available: https://doi.org/10.1145/3341105.3373962

[19] S. Dinesh, N. Burow, D. Xu, and M. Payer, "Retrowrite: Statically instrumenting cots binaries for fuzzing and sanitization," in *2020 IEEE Symposium on Security and Privacy (SP)*, 2020, pp. 1497–1511.

[20] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++: Combining incremental steps of fuzzing research," in *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, Aug. 2020.

[21] J. E. Forrester and B. P. Miller, "An empirical study of the robustness of windows nt applications using random testing," ser. WSS'00. USA: USENIX Association, 2000, p. 6.

[22] M. Handley, V. Paxson, and C. Kreibich, "Network intrusion detection: Evasion, traffic normalization, and end-to-end protocol semantics," in *Proceedings of the 10th Conference on USENIX Security Symposium - Volume 10*, ser. SSYM'01. USA: USENIX Association, 2001, p. 9.

[23] M. Hemmati and M. A. Hadavi, "Using deep reinforcement learning to evade web application firewalls," in *18th International ISC Conference on Information Security and Cryptology, ISCISC 2021, Isfahan, Iran, Islamic Republic of, September 1-2, 2021*. IEEE, 2021, pp. 35–41. [Online]. Available: https://doi.org/10.1109/ISCISC53448.2021.9720473

[24] B. Jabiyev, S. Sprecher, K. Onarlioglu, and E. Kirda, "T-reqs: Http request smuggling with differential fuzzing," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 1805–1820. [Online]. Available: https://doi.org/10.1145/3460120.3485384

[25] J. Kettle, "How to identify and exploit http host header vulnerabilities," https://portswigger.net/web-security/host-header/exploiting#send-ambiguous-requests.

[26] ——, "Top 10 web hacking techniques of 2020," https://portswigger.net/research/top-10-web-hacking-techniques-of-2020.

[27] khalilbijjou, "Wafninja," https://github.com/khalilbijjou/WAFNinja.

[28] C. Linhart, A. Klein, R. Heled, and S. Orrin, "Http request smuggling," https://www.cgisecurity.com/lib/HTTP-Request-Smuggling.pdf.

[29] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of unix utilities," vol. 33, no. 12, p. 32–44, dec 1990. [Online]. Available: https://doi.org/10.1145/96267.96279

[30] R. Natella, "Stateafl: Greybox fuzzing for stateful network servers," *Empirical Softw. Engg.*, vol. 27, no. 7, dec 2022. [Online]. Available: https://doi.org/10.1007/s10664-022-10233-3

[31] OSSInsight, "Web framework - ranking," https://ossinsight.io/collections/web-framework.

[32] H. Peng and M. Payer, "Usbfuzz: A framework for fuzzing USB drivers by device emulation," in *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, S. Capkun and F. Roesner, Eds. USENIX Association, 2020, pp. 2559–2575. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/peng

[33] J. Pereyda, "boofuzz: Network protocol fuzzing for humans," https://github.com/BBVA/waf-brain.

[34] V.-T. Pham, M. Böhme, and A. Roychoudhury, "Aflnet: A greybox fuzzer for network protocols," in *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, 2020, pp. 460–465.

[35] S. Qin, F. Hu, Z. Ma, B. Zhao, T. Yin, and C. Zhang, "Nsfuzz: Towards efficient and state-aware network service fuzzing," *ACM Trans. Softw. Eng. Methodol.*, mar 2023. [Online]. Available: https://doi.org/10.1145/3580598

[36] Z. Qu, X. Ling, and C. Wu, "Autospear: Towards automatically bypassing and inspecting web application firewalls," https://i.blackhat.com/Asia-22/Thursday-Materials/AS-22-Qu-AutoSpear-Towards-Automatically-Bypassing-and-Inspecting-Web-Application-Firewalls.pdf.

[37] R. S. Raman, M. Wang, J. Dalek, J. Mayer, and R. Ensafi, "Network measurement methods for locating and examining censorship devices," in *Proceedings of the 18th International Conference on Emerging Networking EXperiments and Technologies*, ser. CoNEXT '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 18–34. [Online]. Available: https://doi.org/10.1145/3555050.3569133

[38] Research and Markets, "Global web application firewall market estimated to reach $21.05 billion by 2030 amid growing concerns of cyberattacks," https://finance.yahoo.com/news/global-application-firewall-market-estimated-104800918.html.

[39] I. Ristic, "Protocol-level evasion of web application firewalls," https://media.blackhat.com/bh-us-12/Briefings/Ristic/BH_US_12_Ristic_Protocol_Level_Slides.pdf.

[40] S. Schumilo, C. Aschermann, A. Jemmett, A. Abbasi, and T. Holz, "Nyx-net: Network fuzzing with incremental snapshots," in *Proceedings of the Seventeenth European Conference on Computer Systems*, ser. EuroSys '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 166–180. [Online]. Available: https://doi.org/10.1145/3492321.3519591

[41] K. Shen, J. Lu, Y. Yang, J. Chen, M. Zhang, H. Duan, J. Zhang, and X. Zheng, "Hdiff: A semi-automatic framework for discovering semantic gap attack in http implementations," in *2022 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2022, pp. 1–13.

[42] Y. Shoshitaishvili, "preeny," https://i.blackhat.com/us-18/Thu-August-9/us-18-Kettle-Practical-Web-Cache-Poisoning-Redefining-Unexploitable.pdf.

[43] sqlmap, "sqlmap: Automatic sql injection and database takeover tool," https://sqlmap.org/.

[44] H. P. S. I. R. Team, "Huawei cloud security huawei bug bounty program 2023q1 acknowledgement announcement," https://bugbounty.huawei.com/hbp/#/announcement/detail?code=HBPA23-0010.

[45] thuanpv, "Aflnwe," https://github.com/thuanpv/aflnwe.

[46] M. Zalewski, "american fuzzy lop," https://lcamtuf.coredump.cx/afl/.

[47] L. Zheng, X. Li, C. Wang, R. Guo, H. Duan, J. Chen, C. Zhang, and K. Shen, "Reqsminer: Automated discovery of cdn forwarding request inconsistencies with differential fuzzing," in *NDSS*, 2024.

[48] Y.-H. Zou, J.-J. Bai, J. Zhou, J. Tan, C. Qin, and S.-M. Hu, "TCP-Fuzz: Detecting memory and semantic bugs in TCP stacks with fuzzing," in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, Jul. 2021, pp. 489–502. [Online]. Available: https://www.usenix.org/conference/atc21/presentation/zou

Appendix A.

A.1. Baseline Test

WAF	WAFNinjia Result	SQLMap (xwaf) Result
Mircosoft Azure WAF	Failed	Failed
Google Cloud Amor	Failed	Failed
Alibaba Cloud WAF	Failed	Failed
Cloudflare WAF	Failed	Failed
Huawei Cloud WAF	Failed	Failed
Safeline WAF	Failed	Failed
Fortinet WAF	Failed	Failed
Barracuda WAF	Failed	Failed
ModSecurity WAF	Failed	Failed
Naxis	Failed	Failed
OpenWAF	Failed	Failed
Janusec	Failed	Failed
WAFbrain	Failed	Failed
HiHTTP	Failed	Failed

TABLE 3. BASELINE TEST RESULTS

A.2. Algorithms

Algorithm 1 Generation Algorithm

Require: grammar rules G
Ensure: sample_tree t sample_message m

$t.root \leftarrow root$
$m \leftarrow empty\_string$
$q \leftarrow$ new Queue
q.push_tail($t.root$)
while !q.empty() **do**
  $c_{now} \leftarrow q$.pop_tail()
  if len($G[c_{now}]$)> 0 **then**
    $children \leftarrow$weighted_random_choice($G[c_{now}]$)
    $c_{now}.chidlren \leftarrow children$
    $children \leftarrow$ reverse($children$)
    for all $child$ **in** $children$ **do**
      q.push_head($child$)
    end for
  else
    $c_{now}.is\_terminal \leftarrow true$
  end if
end while
q.push_tail($t.root$)
while !q.empty() **do**
  $c_{now} \leftarrow q$.pop_tail()
  if $c_{now}.is\_terminal$ **then**
    $m \leftarrow m + c_{now}$.to_str()
  end if
  $children \leftarrow$reverse($c_{now}.children$)
  for all $child$ **in** $children$ **do**
    q.push_head($child$)
  end for
end while
return t, m

Algorithm 2 Evasion Sample Centrifuge

Require: Evasion sample t
Ensure: corpus samples c_{min} evasion samples s_{min}

$q \leftarrow$ new Queue;
q.push(t)
while !q.empty() **do**
  $node\_pool \leftarrow$ []
  $t_{now} \leftarrow q$.pop_tail()
  for all $node$ **in** t_{now} **do**
    if !$node$.visited **and** !$node$.is_leaf **and** !$node$.deleted **then**
      $node\_pool$.append($node$)
    end if
  end for
  if len($node\_pool$) = 0 **then**
    s_{min}.append(t_{now}.dump_to_raw_packet())
  end if
  for all $node$ **in** $node\_pool$ **do**
    $target$.visited $\leftarrow$ true
    $target$.deleted $\leftarrow$ true
    $req \leftarrow t_{now}$.dump_to_raw_packet()
    $accepted, req \leftarrow$ waf_verfication(req)
    if $accepted$ **then**
      if !webapp_verfication(req) **then**
        continue
      end if
    else
      c_{min}.append(t_{now})
      $target$.deleted $\leftarrow$ false
    end if
    q.push(t_{now})
  end for
end while

A.3. Evaluation Details

HTTP Field	Descripition	Example	Vulnerability
Content-Type Header	malformed boundary token	multipart/form-data; boundary*=”boundary”	MPS
	malformed boundary parameter separator	multipart/form-data&boundary=boundary&	MPS
	urlencoded-form with charset parameter	application/x-www-from-urlencoded; charset=utf-7	RSG
	variant json content type	application/x-a-json	PTC
Content-Disposition Header	malformed form-data token	fo-data; name=”taint_key”	MPS
	fake file indicator	form-data; name=”taint_key”; Content-Disposition:;filename*=””	PTC
Boundary Seperator	formdata boundary terminator missing lf	–boundary–\r	MPS
	formdata boundary startline missing cr	–boundary\n	MPS
	empty boundary token	–\r\n	MPS

TABLE 4. EXAMPLES OF PARSING VULNERABILITIES FOUND BY WAF MANIS

Web Framework	Path parameters	Query parameters	Header parameters	Body parameters
Laravel	Route::get(’/{taint_key}’)	$request->query(’taint_key’);	$request->header(’header’);	$request->input(’taint_key’);
gin	router.GET(”/:taint_key”, func(c *gin.Context) { value := c.Param(”taint_key”) })	Context.query(”taint_key”)	Context.GetHeader(”taint_key”)	Context.PostForm(”taint_key”)
beego	Controller.Ctx.Input. Param(”:taint_key”)	Controller.GetString(”taint_key”)	Controller.Ctx.Request. Header[’taint_key’]	Controller.Parseform(&message); message.taint_key
echo	c.Param(”taint_key”)	c.QueryParam(”taint_key”)	c.Request().Header[”taint_key”]	c.FormValue(”taint_key”)
springboot	@PathVariable String taint_key	@RequestParam(”taint_key”)	@RequestHeader(”taint_key”)	@RequestParam(”taint_key”)
express	req.params[”taint_key”]	req.query[”taint_key”]	req.headers.[”taint_key”]	req.body[”taint_key”]
codeigniter	$routes->get(’/(:taint_key)’, ’controller:method’)	$request->getGet(’taint_key’)	$request->header(’taint_key’)	$request->getPost(’taint_key’)
symfony	@Route(”{taint_key}”	$request->query- >get(’taint_key’)	$request->header- >get(’taint_key’)	$request->request- >get(’taint_key’)
flask	@app.route(”/ <taint_key>”)	flask.request.args[’taint_key’]	flask.request.headers[’taint_key’]	flask.request.form[’taint_key’]
django	path(”/ <taint_key>”)	request.GET.get(’taint_key’)	request.header.get(’taint_key’)	request.POST.get(’taint_key’)
fastapi	@app.get(”/{taint_key}”)	taint_key:Annotated[str | None, Query()]	taint_key: Annotated [str | None, Header()]	taint_key: Annotated[str | None, Form()]
webpy	regex match from path	web.input()[’taint_key’]	web.ctx.env.get(’TAINT_KEY’)	web.input()[’taint_key’]
rocket	#[get(”/ <taint_key>”)]fn handler(taint_key: &str)	#[get(”/? <taint_key>”)]fn handler(taint_key: &str) {}	Request <’r>::headers().get(”taint_key”) .collect()	form: Form <MyForm>; form.taint_key
rails	get ’/:taint_key’	params[:taint_key]	request.headers[”taint_key”]	params[:taint_key]
koa	router.get(’/:taint_key’, (ctx, next) =>{ ctx.params .taint_key});	ctx.query.taint_key	ctx.request.header.taint_key	ctx.request.body.taint_key
nestjs	@Param() params.taint_key	@Query() query.taint_key	@Headers() header.taint_key	@Body() body.taint_key
meteor	FlowRouter.getParam(’taint_key’)	FlowRouter.getQueryParam (’taint_key’)	req.headers.taint_key	req.body.taint_key
Fastify	fastify.get(’/:taint_key’, function (request, reply) {const { taint } = request.params;})	request.query.taint_key	request.headers.taint_key	request.body.taint_key
sails	req.param.taint_key	req.query.taint_key	req.headers.taint_key	req.body.taint_key
aspnetcore	[HttpGet(”{taint_key}”)]	[FromQuery(Name = ”taint_key”)] string taint_key	Request.Headers.TryGetValue (’taint_key’, out var headerValue)	[FromForm]myForm myForm.taint_key

TABLE 5. TESTED GETPARAMETER FUNCTION OF EACH WEB FRAMEWORK

Appendix B.
Meta-Review

The following meta-review was prepared by the program committee for the 2024 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

B.1. Summary

This paper proposes a new automated approach to detect protocol-level evasion vulnerabilities in web application firewalls (WAFs). The approach, implemented as WAS Manis, takes in a manually-constructed HTTP grammar from RFCs, and generates and mutates different requests that might cause web applications to not detect a malicious payload. A key insight of the work is that black-box fuzzing of commercial WAFs is not effective, so instead they use open-source WAFs to perform initial payload generation for testing against commercial WAFs. The authors use their approach to test 14 popular WAFs and 20 web frameworks, uncovering vulnerabilities across all tested targets. The authors analyzed the discovered vulnerabilities and identified three underlying reasons contributing to WAF evasions.

B.2. Scientific Contributions

- Addresses a Long-Known Issue
- Identifies an Impactful Vulnerability
- Provides a Valuable Step Forward in an Established Field

B.3. Reasons for Acceptance

1) The work addresses a long-known issue by improving and automating detection of protocol evasion attacks against web application firewalls. By leveraging transferability of inputs between WAFs the proposed technique improves the efficiency of exploring the input space.
2) The work identifies many instances of exploitable protocol evasion attacks by applying the proposed techniques. While the existence of this class of attacks is previously-known, the breadth of concrete examples motivates the impact of these vulnerabilities and accompanying disclosures improve real-world software security.
3) The work provides a valuable step forward by leveraging input transferability between algorithm implementations, allowing open source software to be used for improved fuzzing of closed-source implementations.

B.4. Noteworthy Concerns

WAFs are of limited tangible security value, as they are known to be relatively easy to evade. There is little evidence that WAFs slow sophisticated adversaries, limiting the impact of the work.

Appendix C.
Response to the Meta-Review

We sincerely thank the reviewers for their valuable feedback. In response to the noteworthy concern:

We acknowledge that WAFs that lack well-maintained rules could be trivial to bypass. However, a well-configured and diligently maintained WAF can offer significant security protection. The baseline experiments show that established security tools like SQLmap and WAFNinja cannot bypass commercially deployed WAFs today. Moreover, WAF vendors like Alibaba Cloud offers a bug bounty reward of approximately USD 800 for each identified WAF evasion vulnerability, which emphasizes the industry's recognition of WAF evasion and the difficulty of bypassing a robustly configured WAF.