WAFFLED: Exploiting Parsing Discrepancies to Bypass Web Application Firewalls

1st Seyed Ali Akhavani
Northeastern University
Boston, USA
sadatakhavani.s@northeastern.edu

2nd Bahruz Jabiyev
Northeastern University
Boston, MA, USA
jabiyev.bahruz@gmail.com

3rd Ben Kallus
Dartmouth College
Hanover, NH, USA
benjamin.p.kallus.gr@dartmouth.edu

4th Cem Topcuoglu
Northeastern University
Boston, MA, USA
topcuoglu.c@northeastern.edu

5th Sergey Bratus
Dartmouth College
Hanover, NH, USA
sergey.orcid@gmail.com

6th Engin Kirda
Northeastern University
Boston, MA, USA
e.kirda@northeastern.edu

***Abstract*—Web Application Firewalls (WAFs) have been introduced as essential and popular security gates that inspect incoming HTTP traffic to filter out malicious requests and provide defenses against a diverse array of web-based threats. Evading WAFs can compromise these defenses, potentially harming Internet users. In recent years, parsing discrepancies have plagued many entities in the communication path; however, their potential impact on WAF evasion and request smuggling remains largely unexplored. In this work, we present an innovative approach to bypassing WAFs by uncovering and exploiting parsing discrepancies through advanced fuzzing techniques. By targeting non-malicious components such as headers and segments of the body and using widely used content-types such as `application/json`, `multipart/form-data`, and `application/xml`, we identified and confirmed 1207 bypasses across 5 well-known WAFs, AWS, Azure, Cloud Armor, Cloudflare, and ModSecurity. To validate our findings, we conducted a study in the wild, revealing that more than 90% of websites accepted both `application/x-www-form-urlencoded` and `multipart/form-data` interchangeably, highlighting a significant vulnerability and the broad applicability of our bypass techniques. We have reported these vulnerabilities to the affected parties and received acknowledgments from all, as well as bug bounty rewards from some vendors. Further, to mitigate these vulnerabilities, we introduce `HTTP-Normalizer`, a robust proxy tool designed to rigorously validate HTTP requests against current RFC standards. Our results demonstrate its effectiveness in normalizing or blocking all bypass attempts presented in this work.**

***Index Terms*—Web Security, Web Application Firewalls, Fuzzing, HTTP, Parsing Discrepancies, Request Smuggling, WAF Evasion**

1. Introduction

The widespread adoption of web applications has made them prime targets for cyberattacks. To protect these applications, Web Application Firewalls (WAFs) have been introduced as essential and popular security gates. These systems inspect incoming HTTP traffic to filter out malicious requests, and provide defenses against a diverse array of web-based threats, ranging from SQL injection to Cross-Site Scripting attacks, and beyond.

Despite their critical role, WAFs are not immune to evasion. Traditional WAF evasion techniques often rely on distorting attack payloads to bypass detection rules while ensuring the payloads remain executable by web applications. Attackers usually either obfuscate the payload with encoding schemes or inject new characters into payloads to bypass WAF rules. However, WAF vendors have already taken measures against most of these bypass techniques, which have been known for many years now. Also, these types of attacks assume the ability of the target web application to parse the encoded, or obfuscated payload.

As the threat landscape evolves, HTTP Request Smuggling (HRS) attacks have gained significant attention. HRS exploits discrepancies in the interpretation of HTTP requests between different entities in the communication chain, such as servers, proxies, and WAFs. These attacks can have severe consequences, including unauthorized access to sensitive information, session hijacking, and server compromise. The increasing complexity of web applications and their reliance on intermediary components have increased the risk of HRS vulnerabilities. Recent studies have investigated the importance of HRS, identifying new variants of attack and suggesting defense mechanisms [1], [2], [3], [4].

Parsing discrepancies, caused by inconsistencies in the interpretation of HTTP requests, play a critical role in enabling attacks such as HRS. These discrepancies first appeared in the communication path between servers, but WAFs, which are integrated to this path, may themselves be vulnerable to such inconsistencies. As a layer between the client and the web application, WAFs must correctly interpret HTTP requests to protect against malicious activities. However, vulnerabilities in their parsing mechanisms can allow attackers to exploit these discrepancies, bypassing

the WAF, and allowing attacks to reach the web application. Building on this understanding of the limitations of current WAF defenses and the emerging threat of HRS, we present a novel, real-world approach to bypassing WAFs. Our method exploits content parsing discrepancies between WAFs and web application frameworks. Unlike traditional evasion tactics, we keep the attack payload intact, and focus on mutating specific content elements, such as the boundary in `multipart/form-data`, or namespace feature in `application/xml`, causing the WAF to misinterpret the content. This misinterpretation allows the payload to pass through, while the web application framework correctly parses and executes the attack.

Our work tested a wide range of combinations of popular WAFs, including Google Cloud Armor, Cloudflare, AWS WAF, Azure WAF, and ModSecurity on NGINX, alongside widely-used web application frameworks such as Flask, Laravel, FastAPI, Gin, Express, and Spring Boot. We focused on three complex content types: `multipart/form-data`, `application/xml`, and `application/json`. By repurposing a grammar-based and structure-aware HTTP fuzzer, we identified implementation differences in how these WAFs and frameworks parse these content types. Our findings reveal that most WAF-framework pairs can be bypassed using various content distortions, highlighting a significant vulnerability in current WAF implementations.

In this work, we summarize our contributions as follows:

- We introduce a fundamentally new approach that uses content parsing discrepancies to bypass web application firewalls.
- We present a practical methodology for automatically finding new discrepancy-based bypass vectors using black-box fuzzing techniques.
- We design and implement a pipeline that tests these bypass vectors against popular web application firewalls and frameworks.
- We demonstrate successful discrepancy-based bypass instances on the pairs of most popular web application firewalls and frameworks, and we coordinate mitigation efforts with the affected technology vendors.
- We analyze the interchangeability of our bypass techniques using real-world data from PublicWWW, demonstrating that our findings are widely-applicable and practical across real-world web applications.
- We present the first tool, `HTTP-Normalizer`, that ensures that all proposed bypasses are avoidable by enforcing proper techniques.

Availability. Our results and source code, including fuzzer input grammars and WAF configurations are publicly available[1]. Sensitive bypass requests will remain restricted until the vulnerabilities are resolved.

1. https://github.com/sa-akhavani/waffled

2. Background

2.1. Web Content Types and RFCs Explained

Our work focuses on using specific features of content-types to bypass a malicious request. Thus, it is crucial to inspect the details of the content and media types discussed in this paper.

RFCs (Request for Comments). RFCs [5] are a series of documents that define protocols, procedures, and conventions used on the Internet and networking standards. They are published by the Internet Engineering Task Force (IETF) and related organizations. Each RFC is assigned a unique number, and these documents serve as the authoritative source of information on various Internet standards and protocols. For instance, the structure of HTTP headers, MIME types, and various other web technologies are defined by respective RFCs.

ABNF (Augmented Backus-Naur Form). ABNF grammar rules [6] are a formal notation used to specify the syntax of RFCs. ABNF extends the basic BNF notation to provide a more flexible and precise way to define the syntax of Internet protocols. It is particularly important in our work as it allows for precise definitions of content types and headers, which we exploit to identify parsing discrepancies in WAFs. By understanding and manipulating ABNF rules, we can generate requests that challenge the ability of WAFs to consistently parse and enforce these standards.

Multipart. Multipart content-types allow HTTP messages to include multiple entities with varying media types in a single message body. They are crucial for handling file uploads and complex data structure support. These content-types, such as multipart/form-data, multipart/related, and multipart/mixed, are governed by specific RFCs (e.g., RFC 2387, RFC 7578, RFC 2045-2047) [7], [8], [9], [10], [11] that define their structure and usage scenarios. Our research investigates the complexities of multipart/form-data requests, exploiting features such as boundary definitions, charset, content-disposition, and other custom headers.

XML. Extensible Markup Language serves as a flexible format for structured data representation, commonly used in data exchange protocols. XML-based requests, defined in RFC 7303 [12], are characterized by elements such as DOCTYPE declarations, schemas, and CDATA sections. Our approach includes mutating these structural elements to investigate WAF responses to XML-specific parsing matters. For example, variations in DOCTYPE declarations and CDATA usage are tested to identify vulnerabilities in WAF handling of XML content.

JSON. JavaScript Object Notation, described in RFC 8259 [13], is a lightweight data interchange format widely used in modern web applications. JSON requests are characterized by their simple structure based on key-value pairs. We manipulate JSON object formatting and nested structures to assess WAF handling of JSON parsing anomalies.

How WAFs Work. A WAF operates by inspecting HTTP traffic between clients and web applications, filtering out potentially harmful requests to prevent attacks such as SQL injection, cross-site scripting (XSS), and malicious script uploads. Positioned inline with the traffic flow, the WAF

acts as a barrier between users and the application server, and is able to analyze requests in real-time before they reach the protected application. To make filtering decisions, WAFs operate by scanning incoming HTTP requests, analyzing headers, cookies, URL parameters, and body content. Figure 1 illustrates this process. Numerous commercial and open-source WAFs implement these mechanisms, including Cloudflare, Google Cloud Armor, Microsoft Azure WAF, Amazon AWS WAF, and ModSecurity. WAFs typically rely on one or more detection strategies:

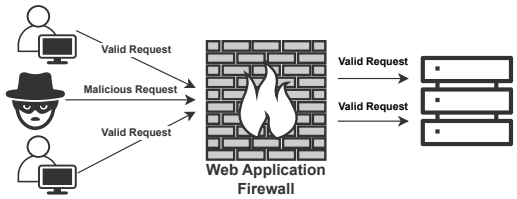

Figure 1: WAF filtering malicious requests.

Managed Rulesets. WAFs use predefined sets of rules that define patterns associated with common attacks. These rulesets are frequently updated to respond to new threats. Most WAFs use their own custom rulesets. But one of the famous rulesets that is widely used among almost all WAFs and most custom rulesets use it as a base, is OWASP CRS (Core Rule Set) [14]. This ruleset is a set of generic attack detection rules for use with compatible WAFs. It aims to protect web applications from a wide range of attacks, including the OWASP Top 10. CRS provides protection against many common attack categories, including SQL Injection, Cross Site Scripting, Local File Inclusion, etc.

Signature-Based Detection. This method involves identifying known attack vectors based on specific signatures or patterns in the data. It is effective against known threats, but can be limited in identifying novel or modified attacks.

Machine Learning-Based Detections. Advanced WAFs incorporate Machine Learning models that analyze traffic behavior and adapt over time, improving their ability to detect previously unseen threats.

To accommodate complex attack patterns, WAF rules often rely on regular expressions (regex). For instance, detecting SQL injection attempts may involve scanning URL parameters or request bodies for SQL-related keywords. When a request arrives, the WAF extracts relevant fields, evaluates them against the defined rules, and executes the appropriate action. If a match is found, the WAF may block the request, log the event, or challenge the user with additional security measures such as a CAPTCHA. In cases where no match occurs, the request is forwarded to the application without interference. By using well-defined rules and regular expressions, a WAF effectively protects web applications from a wide range of attacks.

Each request is parsed to extract these fields, allowing the WAF to evaluate them against a predefined set of rules. These rules follow a structured format that are displayed in listing 1 and are as follows:

```
1 if <field><operator><value>
2 then <action>
```

Listing 1: WAF Rule Format.

- **Field**: This could be any part of the HTTP request such as a header, cookie, MIME type, or URL parameter.
- **Operator**: Common operators include *equals*, *contains*, *not equal*, etc.
- **Value**: The specific value to match against.
- **Action**: The action to be taken if the rule matches, such as *block*, *skip*, or *send security challenge*.

Example of a WAF in action. Listing 2 illustrates a simple WAF rule definition and how it is applied during request inspection.

```
1 if url_parameter "user_input" contains "DROP
    TABLE"
2 then block
```

Listing 2: Example WAF Rule.

In this case, when a request with the URL parameter `user_input=DROP TABLE users` arrives, the WAF parses the request and extracts relevant fields. Then, it evaluates the value of `user_input` against the defined rule. The pattern `"DROP TABLE"` is matched using a regular expression, triggering the `block` action. As a result, the WAF denies the request and logs the incident for further analysis.

While such rule-based filtering mechanisms are effective in many scenarios, a critical but often overlooked aspect of WAF behavior is how the system parses and interprets incoming data. Different WAFs may handle HTTP request parsing in different ways, particularly when dealing with outdated RFCs, nested data structures, or uncommon content types. These *parsing discrepancies*, create opportunities for attackers to bypass protections by crafting inputs that are marked as valid by the WAF but would exploit the the target application. This work focuses on uncovering and categorizing such evasions by analyzing how parsing discrepancies arise and demonstrating practical techniques that abuse these mismatches to bypass modern WAFs.

3. Related Work

At the architectural level, bypasses exploit vulnerabilities within the web application's infrastructure. Techniques such as IP spoofing and server-side request forgery (SSRF) allow attackers to access the origin server directly, circumventing WAF protection. Cache poisoning attacks, as discussed by Meiners et al. [15], exploit CDN vulnerabilities to disrupt access to web applications. Additionally, file processing exploits, such as those highlighted by Freiling et al. [16], offer insights into similar architectural vulnerabilities.

Bypassing WAFs at the protocol level typically involves manipulating communication protocols. HTTP request smuggling, where a malicious request is embedded within a legitimate one, is a key example, with T-Reqs [17]

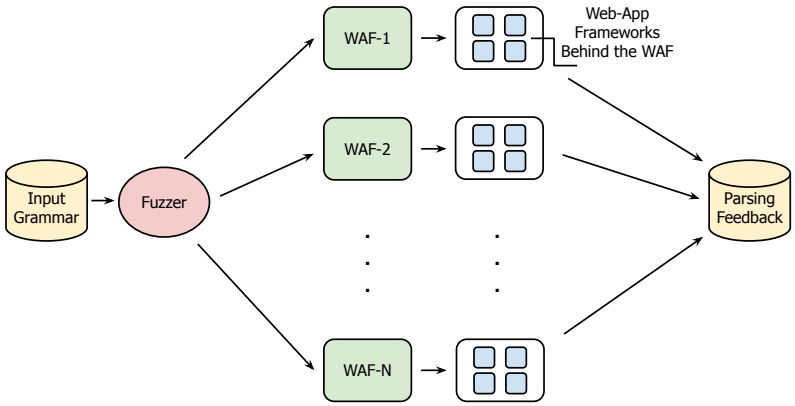

Figure 2: WAFFLED Overview.

providing methods to identify such vulnerabilities. Similarly, features such as chunked transfer encoding, which involves fragmenting HTTP data into smaller chunks, can be exploited to evade WAF detection. Dalili's work on IP fragmentation [18] explore these protocol-level evasion strategies. Also, SYMTCP [19] focuses on evading deep packet inspection. Furthermore, Handley et al. [20] inspects the evasion of network intrusion detection systems, a concept closely related to bypassing WAFs at the protocol level.

Recent work by Wang et al. [21] explores WAF bypasses through protocol-level evasion, whereas our study takes a distinct approach by targeting content-type-specific parsing discrepancies across headers and bodies. While both use fuzzing, our methodology systematically covers a wider range of content types, leading to the discovery of new bypass classes. We also propose a more detailed classification system with 24 attack types (versus 3 in prior work), validate our techniques on 100 real-world websites beyond controlled environments, and introduce a proof-of-concept normalizer to mitigate these issues. None of which are addressed in Wang et al.'s work.

Payload level bypassing techniques involve transforming the original payload to render it undetectable by the WAF. Obfuscation techniques, such as encoding or encryption, are frequently employed to conceal malicious code from WAF scrutiny. Additionally, evasion techniques, such as altering the order of characters or incorporating whitespace, can be used to confuse the WAF, and facilitate bypassing. Payload size manipulation, such as oversized POST requests that exploited a vulnerability in Google Cloud Armor [22], illustrates how payload modifications can defeat WAF protections. Not all bypasses are feasible using by targeting a single level approach above. Some works involve combining multiple techniques to achieve a bypass. AutoSpear [23] exemplifies a tool that automates WAF bypassing using a combination of architectural, protocol, and payload-level techniques.

The academic works discussed here significantly enhance our knowledge of bypass techniques and WAF behavior, providing a platform for further exploration in WAF bypassing methods. Our work builds upon this foundation by introducing a novel approach that exploits content-parsing discrepancies, pushing the boundaries of existing methodologies. Unlike prior research, which often focuses on obfuscating payloads or manipulating protocol features, our approach targets fundamental weaknesses in how WAFs parse HTTP requests.

4. WAFFLED Design and Methodology

Figure 2 provides an overview of the WAFFLED approach for finding discrepancy-based bypass vectors. The methodology involves generating, mutating, and testing HTTP requests across the WAFs and web application frameworks to identify successful bypasses.

4.1. Input Generation and Mutation

The fuzzer generates valid HTTP requests using a predefined grammar that contains a web attack payload (e.g., SQL injection, Cross-Site Scripting). Mutations are applied everywhere except the attack payload itself, ensuring that bypass attempts focus on discrepancies rather than payload obfuscation. The highlighted part in Listing 3 shows an XSS attack payload, which remains unmodified during mutations. These mutations are expected to give us new discrepancy-based bypass instances. The goal is to confuse WAFs so they fail to properly parse the mutated requests and allow them through, while the web application frameworks, despite the mutations, correctly interpret and execute the attack payload.

4.2. Identifying Successful Bypass Instances

The target infrastructure consists of all tested WAFs where each WAF is able to forward requests to all web application framework instances. Each instance is a tiny web application that parses received requests by using the parser library of the web application framework (e.g., Flask) for the tested content type (e.g., `multipart/form-data`). The results of parsing operation are logged for a later analysis. If a log shows that the request is parsed successfully and

the request body contains the attack payload, then the corresponding input request can be used to successfully bypass the relevant WAF.

4.3. Motivating Example

Listing 4 presents an example of how parsing discrepancies can lead to WAF circumvention. This example has two boundaries defined: `fake-boundary` and `real-boundary`. When this request arrives at the WAF, it picks the first one (i.e., `fake-boundary`) as the boundary value and ignores the remaining boundaries, thereby missing the attack payload. When it is forwarded to the web application framework running behind the WAF, as it supports the boundary continuation mechanism – which was standardized in RFC 2231 and allows the use of multiple parameters to contain a single parameter value – it takes the boundary value to be the concatenation of those two values (i.e., `real-boundary`), thereby parsing the XSS payload encapsulated between the `real-boundary` boundaries.

```
POST / HTTP/1.1
Host: victim.com
Content-Length: 114
Content-Type: multipart/form-data; boundary=1234

--1234
Content-Disposition: form-data; name="field1"

<script>alert(document.cookie)</script>
--1234--
```

Listing 3: A malicious HTTP request using multipart/form-data with an XSS attack payload.

```
POST / HTTP/1.1
Host: victim.com
Content-Length: 230
Content-Type: multipart/form-data;
boundary=fake-boundary;boundary*0=real-;boundary*1=
boundary

--fake-boundary
Content-Disposition: form-data; name="field1"

value1
--fake-boundary--
--real-boundary
Content-Disposition: form-data: name="id"

<script>alert(document.cookie)</script>
--real-boundary--
```

Listing 4: A malicious `multipart/form-data` request that uses parameter continuation.

5. Experiment Setup

5.1. Tools and Infrastructure for Experiments

We used a modified version of the T-Reqs [17] fuzzer for the experiments. The modifications include adding support for encrypted HTTP requests, and setting `content-length` dynamically based on the length of the body. We also modified the T-Reqs source code to avoid using < and > characters to parse the grammar, since these characters were also used in the XSS attack payload that was present in generated requests.

TABLE 1: List of Tested WAFs and Their Supported Request Content-Types (As of July 2024).

WAF	Content-Type	Bypassed
Cloudflare WAF	multipart, json, xml	Yes
Microsoft Azure WAF	multipart, json	Yes
Google Cloud Armor	multipart, json	Yes
Amazon AWS WAF	multipart, json	No
ModSecurity on NGINX	multipart, json, xml	Yes

Input grammars were created in accordance with RFC specifications, encompassing all standard components of the content type and their possible values. The fuzzer was executed using these specific grammars[2] for several hours, generating and testing a total of 373,670 requests across 5 WAFs and 6 web application frameworks.

Our server infrastructure was deployed across multiple cloud platforms, including Amazon AWS, Google Cloud, and Microsoft Azure. This configuration was utilized to host our web applications and conduct the experiments.

5.2. WAF Configuration and Setup

To comprehensively assess the resilience of WAFs against mutated HTTP/1.1 web requests, we conducted an extensive evaluation of several WAFs. The selection criteria focused on solutions with significant adoption across both commercial and open-source domains. The WAFs included in this study were AWS WAF, Cloudflare WAF, Google Cloud Armor, Microsoft Azure WAF, and ModSecurity on NGINX. This selection enabled us to evaluate the effectiveness of both widely-used community-driven tools and leading commercial solutions in the industry.

For all tested WAFs, default settings were utilized without any modifications to preserve the integrity of the testing environment and ensure the results accurately reflected typical user experiences. The consistent configuration, especially with the widespread adoption of the OWASP CRS, enabled a fair comparison across WAFs despite their differing underlying technologies and implementation strategies. This standardized methodology established a clear baseline to evaluate each WAF's capability to process mutated HTTP requests, effectively highlighting their individual strengths and potential vulnerabilities.

Table 1 lists the WAFs and their supported content-types in our study.

AWS WAF. AWS WAF was configured using Elastic Load Balancing (ELB). We employed the AWS Managed Ruleset in conjunction with OWASP CRS version 3.0, using default settings to accurately represent typical user configurations.

Cloudflare WAF. Cloudflare WAF was set up as the DNS provider for our domain, applying its WAF to all incoming traffic via its global CDN. We used the Pro version, enabling both Cloudflare's Managed Ruleset and the OWASP CRS to ensure comprehensive threat coverage.

2. Available in the public repository

Google Cloud Armor. Google Cloud Armor was deployed on a virtual machine behind Google Cloud's Load Balancer. We configured it with rulesets for SQL injection and XSS protection (`sqli-v33-stable` and `xss-v33-stable`) at Sensitivity Level 1.

Microsoft Azure WAF. Microsoft Azure WAF was set up through Azure's Application Gateway using the WAF V2 tier, with OWASP CRS version 3.0 enabled by default. This configuration aligns with common deployment practices.

ModSecurity. ModSecurity was deployed as a plugin for NGINX, configured with OWASP CRS version 3.0 at Paranoia Level 1, reflecting the default settings commonly used in many environments. The setup adhered to the official documentation guidelines, ensuring an accurate evaluation of its effectiveness in scenarios typical of real-world deployments.

The influence of rulesets on our findings is minimal for several key reasons. First, the attack payloads used in this study are intentionally straightforward, designed to be blocked by all standard rulesets across the tested WAFs, as confirmed during preliminary testing. Second, our bypass techniques do not depend on obfuscating the payload itself, but rather exploit parsing inconsistencies, focusing on how WAFs interpret HTTP requests before applying their rulesets. This approach highlights that the bypasses stem primarily from the WAF's inability to correctly parse the request or its decision to bypass parsing due to the complexity of the input, irrespective of the specific ruleset in use.

5.3. Web Application Frameworks Setup

In this work, we evaluated seven widely-used web application frameworks: Express, Node.js-HTTP, Flask, FastAPI, Gin, Laravel, and Spring Boot. To comprehensively analyze their parsing capabilities, we employed a combination of the frameworks' default parsers and HTTP parser packages sourced from reputable platforms. The selection of parser packages was informed by their prevalence and popularity within the developer community, as indicated by metrics from resources such as npm and PyPi Stats. This approach ensured that our study included modules that are both widely adopted and representative of real-world usage.

To accurately test how each framework and parser handles the request body, we implemented the following approach: we attempted to access the parsed content of the request body through framework-specific methods such as `request.body.parse()`, etc. The goal was to retrieve the value of a specific field (`field`), and have the framework and parser process the entire request body. After parsing, we searched for the sent attack payload within the parsed content. If any of the parsed fields contained the attack payload, it indicated that the request successfully bypassed the WAF, and was correctly parsed by the framework and parser. We then sent a success flag along with instance name and WAF information back to our client to mark the successful attacks.

For parsing request bodies in our study, each framework's default parser was utilized. In cases where the default parser did not support a specific content-type, we employed the most popular third-party parsers for that content-type. This approach of using both default and third-party parsers provided an understanding of how web application frameworks handle and interpret HTTP requests under different parsing scenarios.

For Node.js, the chosen parsers included `Busboy` and `Formidable` for parsing multipart content types, and `fast-xml-parser` for handling `application/xml` requests. Since Express and the HTTP module do not support parsing of multipart and xml content-types. For Python frameworks, we integrated `xmltodict` and `xmlminidom` for XML parsing. Multipart content types in Python were processed using `python-multipart` parsers.

In all our tests, we strictly adhered to the official documentation and examples of each framework, avoiding any custom development. This ensured that our evaluation covered the most common and correct usage patterns of these frameworks and parsers.

5.4. Validation of Bypass Effectiveness

To validate WAF bypasses, we dispatched malicious requests through the WAF to target frameworks, which attempted to parse and store the body content on the server. We then compared the original payload with the stored result to determine if the bypass succeeded. A script automated this comparison, confirming whether the payload had both circumvented the WAF and been correctly parsed by the framework. This process was necessary because defining a successful bypass solely based on the ability of a request to evade the WAF would lead to misclassification. Such a definition could include malformed requests that are not conforming to HTTP standards or are unprocessable by any framework, as successful bypasses. This would lead to inflating false positives and obscuring real-world risks because of the following reasons:

Unrealistic Attack Scenarios. Most WAFs allow malformed requests to pass because they cannot parse them. If we were to consider these instances as bypasses, more than 50% of mutated requests could be misclassified as successful.

Practical Mitigation Challenges. Malformed requests are frequently rejected at the framework level. Reporting these would inflate false positives and obscure actionable findings.

A true attack occurs when a malicious request bypasses the WAF, is parsed by the framework, and the payload is executed. Malformed requests that frameworks cannot parse pose lower security risks and are lower priority. All bypasses in our work meet this criteria, enabling execution of the malicious payload (e.g., XSS or SQL injection).

5.5. Experiment Results and Analysis

Table 2 provides the list of frameworks and Table 3 provides the list of parsers that were inspected in our work. The experiments revealed that all of the frameworks and parsers were vulnerable to at least one bypass technique, highlighting the effectiveness and practical implications of our findings.

TABLE 2: List of tested web frameworks and whether they are vulnerable to at least one bypass in our study or not.

Framework	Version	Language	Vulnerable
Laravel	10.48.16	PHP	Yes
Spring Boot	3.2.2	Java	Yes
Gin	v1.9.1	Go	Yes
Express	4.18.2	Node.js	Yes
Fastapi	0.109.2	Python	Yes
Flask	3.0.2	Python	Yes
Node.js-HTTP	18.16.1	Node.js	Yes

TABLE 3: List of tested parsers, their weekly downloads as of July 2024 collected from NPM and PyPI Stats [3]website, and whether they are vulnerable to at least one bypass in our study or not.

Parser	Version	Vulnerable	Downloads
Node.js			
Busboy	1.6.0	Yes	8,639,607
Formidable	3.5.1	Yes	6,998,735
fast-xml-parser	4.3.3	Yes	14,254,527
Python			
xmltodict	0.13.0	Yes	10,374,671
xmlminidom	3.10.12	Yes	n/a
python-multipart	0.0.9	Yes	4,288,034

6. Findings

Before diving into analyzing these bypasses and reporting them, we first, analyze all results to retain only those bypassed requests that are achieved with the minimum number of mutations. Then, we classify bypasses based on their mutations. This allows us to:

- Identify the minimum mutations needed for a request to bypass a WAF uniquely.
- Ensure our results are realistic, and our reports are minimal and accurate by focusing solely on unique bypasses.
- Reporting and documenting bypass techniques and ensuring that mitigation strategies can address entire classes of bypasses rather than individual instances.

6.1. Identification of Unique Bypass Requests

In the process of testing and validating bypass techniques against WAFs, identifying unique bypass requests is crucial for accurately assessing the effectiveness of the discovered vulnerabilities. During our experiments, we discovered 1207 mutated requests that were successful bypasses. The identification of unique bypass requests involves rigorous analysis and filtering of the mutated HTTP requests generated during the testing phase.

Not all successfully bypassed requests are unique in terms of the underlying techniques used. For instance, consider a scenario where two mutations contribute to the bypass success:

- **Mutation I**: Introduces a `\x00` character into the boundary value of a multipart request, transforming `--boundary` to `--\x00boundary`.

3. https://www.npmjs.com and https://pypistats.org

- **Mutation II**: Changes the capitalization of characters in the `charset` parameter of the `Content-Type` header from `utf-8` to `Utf-8`.

While Mutation I, alone, constitutes a successful bypass strategy, Mutation II does not contribute to bypassing the WAF independently. However, when both mutations are combined into a single request, the request successfully evades WAF detection due to the presence of Mutation A. In this scenario, the unique bypass is attributed to Mutation A, whereas Mutation B does not qualify as an independent bypass strategy. And the request that is formed by both of these mutations is not a unique bypass.

To maintain the integrity and accuracy of the findings, redundant bypass requests that do not introduce unique evasion techniques are filtered out during the analysis phase.

After applying this minimization technique, among all of the generated requests, 1207 of them were unique bypasses found for (WAF, Framework) pairs.

6.2. Classification of Bypasses

After extracting the bypasses with the minimum number of mutations, we proceed with the classification of these bypasses. This classification process is essential as it allows us to systematically analyze and categorize the bypasses, facilitating the development of effective defense mechanisms.

To achieve this, each bypass is examined based on the unique mutation strategy and the mutated element used rather than the specific mutation. For instance, consider a JSON request with the body {`"field1": "value1"`}. If two distinct bypasses are achieved by inserting a `\x00` or `\x02` after the second double quote, both are classified under the same category: *manipulating the field name wrapper*. Although different characters are used, the fundamental concept remains consistent. This categorization simplifies the study of bypass techniques by grouping similar strategies, thereby enabling a more structured approach to understanding and mitigating these vulnerabilities.

6.3. Bypassed Requests Analysis

We now present the classification results of the bypassed requests for each content-type in our work. Some of the mutation classes are common among all tested content-types, they are defined once but mentioned for each content-type that they were leading to a successful bypass.

6.3.1. Multipart Bypass Classes

Our examination identified a total of 351 unique bypasses related to multipart content-type parsing. Figure 3 demonstrates valid bypass classes that we found in our work for multipart content-type. Table 4 contains examples for each discussed bypass category.

Boundary Delimiter Manipulation. This category involves the removal or alteration of boundary delimiters within multipart content. For instance, removing the `\r\n` sequence before the boundary string can confuse the parsing logic, leading to successful bypasses.

TABLE 4: Classification of Multipart Bypass Categories with Examples. Removals are displayed with a strike-through text with a gray background, while additions and replacements are shown with only a gray background.

Category Name	Request Example
Boundary Delimiter Manipulation	~~\r\n~~---boundary
Content-Disposition Disruption	content-disposition: form-da \x00 a;
Distorted Header Injection to Body	conten\x00-extra: something
Content-Type Tweak in Body	Content-Type: text/plain \x00 ; charset=UTF-8
Charset Value Alteration in Body	charset= \x00 UTF-8
Header Separator Manipulation in Body	content-disposition: form-data; name="f1" \x00
Content-Type Parameter Tweak	C ~~o~~ ntent-Type: multipart/form-data;
Boundary Delimiter Removal	~~---boundary~~
Linefeed Removal	Content-Type: multipart/form-data; boundary=real\r ~~\n~~ \r\n
Whitespace Alteration	Content-Type: \t multipart/form-data; boundary*0=re;boundary*1=al
Disrupted Body Field	content-disposition: form-data; name="field1 \x00 "
Boundary Header Tampering	---boundary=value ;

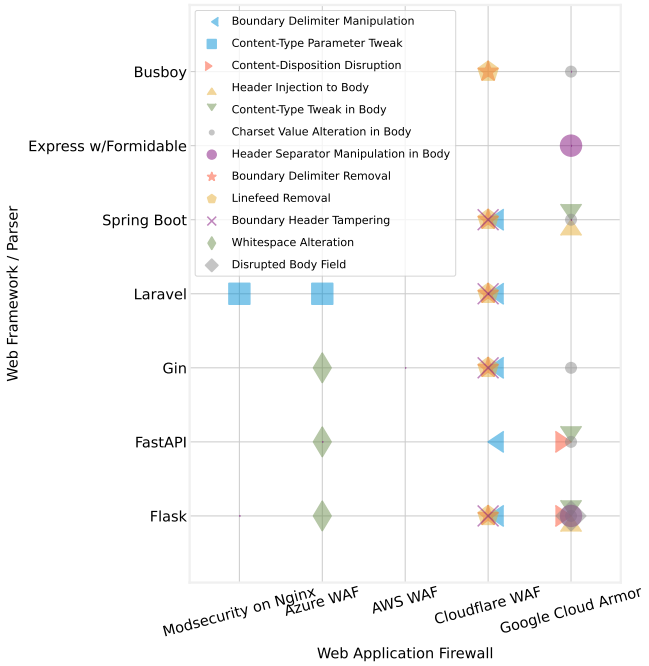

Figure 3: Occurrence of Successful Bypasses For Multipart Content-Types in Tested Frameworks

Content-Type Parameter Tweak. This technique includes modifications to the global Content-Type header's name, such as removing or inserting characters.

Content-Disposition Disruption. This technique targets the Content-Disposition header within the multipart content, altering its structure to evade detection.

Disrupted Header Injection to Body. This category includes adding redundant headers with disrupted header name directly into the multipart body content, which can mislead the WAF into processing the content incorrectly.

Content-Type Tweak in Body. Here, manipulations are performed on the Content-Type value within the body and not in the global header, such as inserting characters.

Charset Value Alteration in Body. This involves altering the charset value within the body content, affecting how the WAF interprets the encoding of the payload.

Header Separator Manipulation in Body. This technique modifies the separator between multiple header lines in the multipart body content, such as replacing newlines with other characters.

Boundary Delimiter Removal. This category involves the complete removal of boundary delimiters, affecting the WAF's ability to correctly parse multipart sections.

Linefeed Removal. This technique removes newline and carriage return characters. For example, removing the linefeed after the request headers and before beginning the boundary delimiter for the multipart request body. This approach can cause the WAF to misinterpret the structure of the multipart content.

Boundary Header Tampering. In this category, manipulations involve changes to the boundary value in the header, such as appending a semicolon to the boundary parameter in the content-type header line.

Whitespace Alteration. In this category, the whitespace character is replaced with `\t` in the content-type header. The request header must use the parameter value continuation feature of HTTP to define the boundary for this bypass to be effective. `boundary*0=re;boundary*1=al`

Disrupted Body Field. In this category, manipulations involve inserting invalid characters in a field name. The request header must use the parameter value continuation feature of HTTP to define the boundary for this bypass to be effective.

6.3.2. XML Bypass Classes

Our analysis revealed a total of 299 unique bypasses for XML content types. Figure 4 demonstrates valid bypass classes for application/xml content-type. Table 5 contains examples for each discussed bypass category.

Figure 4: Occurrence of Successful Bypasses For application/xml Content-Types in Tested Frameworks

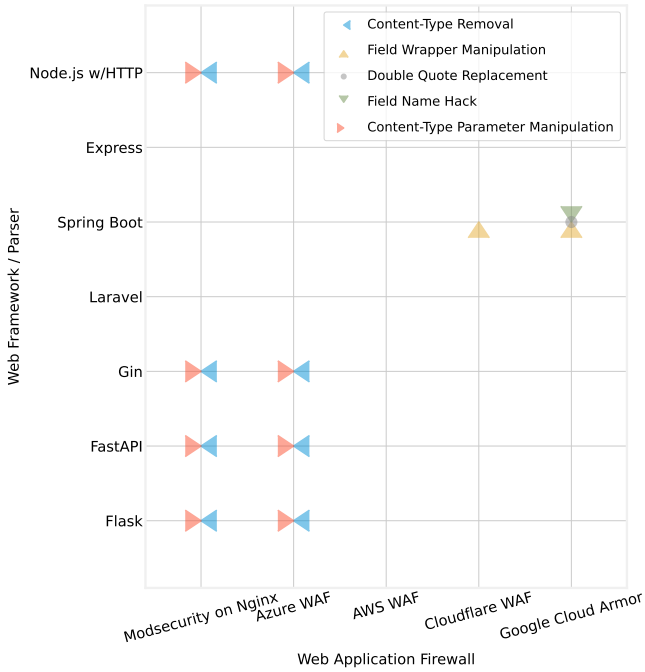

Figure 5: Occurrence of Successful Bypasses For application/json Content-Types in Tested Frameworks

Extra Field Addition. Adding an extra field outside the defined XML schema. For example, inserting a new field such as `<field2>value2</field2>` exploits schema validation weaknesses, allowing malicious payloads to bypass security checks.

DOCTYPE Closure Confusion. Placing an extra character at the end of the xml body confuses the WAF in parsing the DOCTYPE entity of the XML.

Schema Closure Manipulation. This technique manipulates the closure of XML schemas. By inserting characters, new elements, or duplicated field names and values at specific positions within the schema, the structure of the XML document is altered in a way that evades detection.

Newline Abuse. This method achieves the bypass by placing an extra new-line before the content-type header.

Content-Type Header Parameter Removal. Bypasses achieved by removing the Content-Type header's name. Such manipulations exploit the dependency of some WAFs on specific header configurations to enforce security rules.

Content-Type Header Replacement. The parameter name within the Content-Type header is replaced with its value. This manipulation can confuse WAFs that rely on precise header structures for detection.

Misplaced Field. Placing field values outside their corresponding XML tags. By altering the position of values within the XML document, the attack payload can bypass WAF rules that expect a specific structure.

6.3.3. JSON Bypass Classes

In our work, we identified a total of 557 unique bypasses for JSON content types. Figure 5 demonstrates valid bypass classes for application/json content-type. Table 6 contains examples for each discussed bypass category.

Content-Type Removal. This category involves bypassing the WAF by removing the `Content-Type: application/json` header from the global header. This manipulation exploits the reliance of some WAFs on specific content-type headers to enforce security rules.

Field Wrapper Manipulation. In this category, the attack is achieved by adding characters such as `\x00` between the field name and the colon in the JSON object. This manipulation disrupts the field parsing.

Double Quote Replacement. This method replaces the double quotes surrounding the field names and values with other characters.

Field Name Hack. This technique involves altering characters within the field names of the JSON object. For example, replacing a character in the field name with `"\x00"` can bypass WAFs that do not properly handle such anomalies in field names.

Content-Type Parameter Manipulation. Here, the manipulation targets the Content-Type header name.

6.4. Practicality of Findings

The bypass findings of this paper for three content types are practical and useful only if these content types are popular amongst the web applications in the real world. We performed a study on a sample collected from real-world applications to gauge the popularity of these content types. In addition, we performed another study to check interchangeability between the content-types which can also contribute to the practicality of our findings.

TABLE 5: Classification of XML Bypass Categories with Examples

Category Name	Request Example
Extra Field Addition	<field1>value1</field1> <field2 attr="history">hi</field2>
DOCTYPE Closure Confusion	<!DOCTYPE BOOK [...]><field1>value1</field1>] </BOOK>
Schema Closure Manipulation	<genre:schema><field1>value1</field1> j </genre:schema>
Newline Abuse	\r\n Content-Type: application/xml
Content-Type Header Parameter Removal	~~Content-Type:~~ application/xml
Content-Type Header Replacement	application/xml application/xml
Misplaced Field	... value1 <field1>value1</field1>...

TABLE 6: Classification of JSON Bypass Categories with Examples.

Category Name	Bypassed Request Example
Content-Type Removal	~~Content-Type: application/json~~
Field Wrapper Manipulation	{ "field1" \x00 : "<script>alert(document.cookie)</script>" }
Double Quote Replacement	{ "field1 \x00 : "<script>alert(document.cookie)</script>" }
Field Name Hack	{ "f \x00 eld1": "<script>alert(document.cookie)</script>" }
Content-Type Parameter Manipulation	Content - Type: application/json

We collected 100 high-ranking websites from PublicWWW [4], which is a search engine for website source codes, by using the query below which was tailored for the needs of the study.

```
type="email" "forgot password" -reCAPTCHA
```

This search query allowed us to find "forgot-password" web pages which contain an HTML form with an email-type input field and does not contain a CAPTCHA. The presence of CAPTCHA does not allow replaying the "forgot-password" requests, which makes it easier and faster to check the interchangeability of content types. We chose the first 100 search results, which are the "forgot-password" pages of popular websites ranking between 3K-50K. About half of these webpages were excluded from the study for reasons such as "non-english language", "obscenity" and "an emerging captcha".

On each "forgot-password" page, we submitted the form after typing in a non-existent email address and intercepted the generated HTTP request on the Burp Suite tool. We examined the `Content-Type` header field and the request body format to decide the content-type for each request. We found that more than two-thirds of the websites use the `application/x-www-form-urlencoded`, while one-fourth of them use the `application/json` and its variation. Only two websites use `multipart/form-data` and one uses URL parameters. For each intercepted "forgot-password" request, to check the interchangeability between content types, we used the "Change body encoding" feature of Burp Suite (in the "Repeater" section) to convert the content type to `multipart/form-data` in `application/x-www-form-urlencoded` requests. We then examined the response status code and response body to check the sameness of responses, which indicates that the application does not differentiate those two content types. We found that more than ninety percent of websites accept them both.

These results suggest that our bypass findings are widely applicable and practical across web applications. The largest portion (i.e., more than two-thirds), which uses `application/x-www-form-urlencoded`, are affected by our `multipart/form-data` findings, since an attacker can easily switch the content type and apply the bypass techniques. The second largest portion, which uses `application/json`, is affected by all the relevant bypass findings we reveal in this paper. The fact that these two content types together make up more than 90% of content types used across web applications, clearly shows the extent of the practicality and impact of these bypass techniques.

7. Protecting WAFs and Frameworks by Normalizing HTTP Request Bodies

In this section, we discuss our approach to mitigate the identified vulnerabilities by introducing an HTTP multipart body normalizer that protects web application firewalls (WAFs) and web application frameworks from parsing-related attacks. The normalizer acts as a gateway for the WAF that strictly enforces RFC grammar rules related to multipart form data, and rejects requests that are not compliant. For those requests that are compliant, optional message body components and fields that must be ignored are removed. This ensures that problematic message constructs are removed before they can induce differential parsing behavior between the WAF and the web application framework. The `HTTP-Normalizer`'s functionality could be expanded for each content-type to provide protection against other popular content types as well. We are introducing

4. https://publicwww.com/

```
POST / HTTP/1.1
Host: target.com
Content-Type: multipart/FoRm-dAtA; boundary="1234"
Content-Length: 90

--1234
Content-DISPOSITION:\tform-data;name="files";\t
    filename="ab.txt"

Foo
--1234--
```

Listing 5: A request before normalization.

```
POST / HTTP/1.1
Host: target.com
Content-Type: multipart/form-data; boundary=1234
Content-Length: 90
Accept: */*
Accept-Encoding: gzip, deflate
User-Agent: Python/3.12 aiohttp/3.9.5

--1234
Content-Type: text/plain
Content-Disposition: form-data; name="files";
    filename="ab.txt"

Foo
--1234--
```

Listing 6: The same request after normalization. Note the standardized capitalization and spacing, removed trailing line ending, and inserted Content-Type MIME header.

a proof of concept description in this section, and use `multipart/form-data` as demonstration.

Overview of HTTP Normalizer. The HTTP multipart body normalizer serves two primary functions: normalizing multipart request bodies, and rejecting non-compliant requests. Upon receiving an HTTP request, the normalizer separates the request's headers and body using the AIOHTTP Python library. It then parses and validates the request's `Content-Type` header, and if it indicates that the message body uses the MIME multipart encoding, it performs one of the following actions:

- **Normalization.** The normalizer begins by parsing requests using a strict multipart MIME parser generated from the ABNF in the RFCs. The output data structure of the parser is capable of representing only the necessary components of a multipart message body. Deprecated and optional portions of the multipart message are not representable in this structure. The structure is then re-serialized and forwarded to its destination. Crucially, the normalized request is deserialized from a data structure in which invalid state is not representable, so the output of the normalizer is never malformed.
- **Rejection.** If the normalization process cannot be completed because necessary components of the request body uses a deprecated RFC feature or is malformed, the request is rejected.

Listing 5 demonstrates a sample malformed request before being passed to the normalizer. Listing 6 demonstrates how the sample request would look like after being normalized.

Normalizing Request Body. Currently, the normalizer applies only to requests that use a multipart transfer encoding. Future enhancements will extend this normalization capability to other content-types, aiming to demonstrate the viability of this approach in enhancing WAF security by rejecting malicious HTTP requests and reducing false positives through normalization. This methodology can be uniformly applied to all content types to achieve comprehensive normalization.

Evaluation of Normalizer. We evaluated the normalizer first by verifying that it does not reject any of a set of valid multipart message bodies. We obtained these from the test suites of the Tornado web server, RStudio, and the multiparty multipart message parser.

TABLE 7: Evaluation Metrics for HTTP Normalizer.

	Bypassable Requests
Normalized	8
Rejected	55
Total Attempts	63

We sampled bypasses from all 12 discovered bypass classes that we mentioned in subsection 6.3.1 and took 63 bypassable requests in total to test against the normalizer, and recorded whether the request was accepted, normalized, or rejected. Note that while we could mutate HTTP requests using our fuzzer and send all of them to the normalizer, it was unnecessary because: 1) even if a request bypassed the normalizer, it would have been blocked by the WAF since it was not among the found bypasses, and 2) the normalizer will ultimately be merged into existing WAF rulesets and does not need to be fuzzed separately because it is not a WAF itself, but a ruleset enforcer.

Table 7 demonstrates these evaluation metrics. Out of the 8 bypassable requests that are not rejected by the normalizer, all are blocked by Cloudflare's WAF, resulting in a 100% success rate.

Performance Cost. The `HTTP-Normalizer` was developed to demonstrate that all identified bypass techniques can be prevented if WAFs adhere to proper parsing standards. Our findings, including the lack of successful bypasses against AWS WAF, validate this hypothesis. The intent of the project was not to create a fully optimized, production-grade tool with minimal overhead and broad content-type support, but rather to prove that an effective solution is feasible and implementable for a specific content type. By strictly adhering to RFC grammar rules, the HTTP-Normalizer effectively eliminates all bypass methods. While it currently functions as an additional evaluation layer, its rules can be seamlessly integrated into existing WAF rulesets with minimal performance impact.

8. Discussion

Usability vs. Security Debate. While WAFs can prevent these bypasses by following RFC standards, real-world deployments may face compatibility or customer-specific constraints. Thus, we do not blame vendors, as they often balance security with operational needs. However, our proposed defense mechanism, the `HTTP-Normalizer`, shows that all bypasses in this work are theoretically preventable without significantly impacting usability. Also, not all WAFs are vulnerable to every bypass in this research, indicating that a well-designed RFC-compliant parser can effectively mitigate such issues.

Practical Considerations. We focused on prominent web frameworks and their default or most popular parser packages. While multipart forms are common, not all frameworks have dedicated parsers for them, often leading to custom implementations from developers. These custom parsers can increase the risk of developer errors, potentially allowing bypassed requests to be parsed successfully server-side. However, our work did not examine custom parsers, leaving their implications outside the scope of our findings.

Limitations and Future Work. This work primarily focuses on HTTP/1.1 web requests, and future research could investigate bypass behaviors in HTTP/2. Additionally, while we covered major frameworks and WAFs, expanding the range of frameworks and content-types tested could provide broader insights. Our findings also highlight that parsing discrepancies can reveal fingerprinting opportunities for both WAFs and frameworks. Prior research has shown that HTTP parsing behavior can be used to fingerprint web servers [24], and our results show similar potential for WAFs. For example, sending a JSON bypass request under the Field Name Manipulation category can help identify Google Cloud Armor with the Spring Boot framework, as only that combination would parse and accept the payload. Future work could develop a systematic fingerprinting methodology based on these discrepancies.

9. Ethics and Disclosure Information

All attacks and bypasses in this paper were conducted in a controlled environment using our infrastructure, ensuring no external impact. No malicious payloads were sent to the tested websites, and each site received no more than 10 requests. All WAF tests adhered to bug bounty protocols, and bypasses were disclosed to the affected vendors, who subsequently acknowledged the existence of these vulnerabilities.

- **Google Cloud Armor** classified our report as a Tier 1, Priority 1, Severity 2 vulnerability under the "Insecure by Default" category and rewarded us with a bug bounty.
- **ModSecurity** acknowledged all bypasses we reported in CRS 3.3.
- **Cloudflare** confirmed the reported bypasses and stated that they are working on a fix.
- **Microsoft Azure** acknowledged the bypasses in CRS 3.0, which was the default ruleset at the time of our study. However, they are retiring CRS 3.0 and transitioning to DRS 2.1, an enhanced ruleset based on CRS 3.2, which addresses these vulnerabilities.

10. Conclusion

This work has highlighted the significant impact of parsing discrepancies on the effectiveness of Web Application Firewalls (WAFs) in protecting against cyber threats. Our experiments across major WAFs, particularly with `multipart/form-data`, `application/json`, and `application/xml` content-types, resulted in 1207 bypasses. These bypasses were successfully parsed by our target web application frameworks employing default or popular request parsers. This finding indicates a critical vulnerability in WAFs' ability to uniformly interpret and filter web requests.

This paper's investigation into popular WAFs, including Cloudflare, Cloud Armor, AWS WAF, Azure WAF, and ModSecurity, highlights significant concerns regarding parsing discrepancies in their request analysis. These discrepancies pose a substantial risk, as WAF users may falsely believe they are protected against common attack payloads. As attackers employ increasingly sophisticated methods, the development of dynamic, intelligent WAFs becomes important. Our research not only contributes to the understanding of current WAF limitations, but also proposes effective solutions to defend against the identified threats. This work emphasizes the critical need for improved parsing consistency in WAFs to ensure robust protection against these attacks. Our proposed `HTTP-Normalizer` offers a promising solution, balancing security and usability by enforcing strict compliance with RFC standards.

Acknowledgment

This work was supported by the National Science Foundation under grant 2329540.

References

[1] M. Grenfeldt, A. Olofsson, V. Engström, and R. Lagerström, "Attacking websites using http request smuggling: empirical testing of servers and proxies," in *2021 IEEE 25th International Enterprise Distributed Object Computing Conference (EDOC)*, pp. 173–181, IEEE, 2021.

[2] J. Kettle, "Http desync attacks: Request smuggling reborn." https://portswigger.net/research/http-desync-attacks-request-smuggling-reborn, 2020. Accessed: 2025-01-07.

[3] A. Klein, "Http request smuggling in 2020: New variants, new defenses, and new challenges," in *Black Hat USA*, 2020. Accessed: 2025-01-07.

[4] A. Klein, "Http request smuggling in 2020: New variants, new defenses, and new challenges." https://i.blackhat.com/USA-20/Wednesday/us-20-Klein-HTTP-Request-Smuggling-In-2020-New-Variants-New-Defenses-And-New-Challenges-wp.pdf, 2020. Accessed: 2025-01-07.

[5] Internet Engineering Task Force (IETF), "RFC Process," 2025. Accessed: 2025-01-07.

[6] D. Crocker and P. Overell, "Augmented BNF for Syntax Specifications: ABNF." Request for Comments: 5234, 2008. Accessed: 2025-01-07.

[7] L. M. Masinter, "Returning Values from Forms: multipart/form-data," Request for Comments RFC 7578, Internet Engineering Task Force, July 2015. Num Pages: 15.

[8] E. Levinson, "The MIME Multipart/Related Content-type," Request for Comments RFC 2387, Internet Engineering Task Force, Aug. 1998. Num Pages: 10.

[9] N. Freed and N. S. Borenstein, "Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies," Request for Comments RFC 2045, Internet Engineering Task Force, Nov. 1996. Num Pages: 31.

[10] N. Freed and D. N. S. Borenstein, "Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types," Request for Comments RFC 2046, Internet Engineering Task Force, Nov. 1996. Num Pages: 44.

[11] K. Moore, "MIME (Multipurpose Internet Mail Extensions) Part Three: Message Header Extensions for Non-ASCII Text," Request for Comments RFC 2047, Internet Engineering Task Force, Nov. 1996. Num Pages: 15.

[12] H. S. Thompson and C. Lilley, "XML Media Types," Request for Comments RFC 7303, Internet Engineering Task Force, July 2014. Num Pages: 35.

[13] T. Bray, "The JavaScript Object Notation (JSON) Data Interchange Format," Request for Comments RFC 8259, Internet Engineering Task Force, Dec. 2017. Num Pages: 16.

[14] "OWASP CRS | OWASP Foundation," 2024. [Online; accessed 3. Sep. 2024].

[15] H. V. Nguyen, L. L. Iacono, and H. Federrath, "Your Cache Has Fallen: Cache-Poisoned Denial-of-Service Attack," in *ACM Conference on Computer and Communications Security*, 2019.

[16] S. Jana and V. Shmatikov, "Abusing File Processing in Malware Detectors for Fun and Profit," in *IEEE Symposium on Security and Privacy*, 2012.

[17] B. Jabiyev, S. Sprecher, K. Onarlioglu, and E. Kirda, "T-Reqs: HTTP Request Smuggling with Differential Fuzzing," in *Conference on Computer and Communications Security (ACM CCS)*, Nov. 2021.

[18] "A forgotten invisibility cloak," June 2021. [Online; accessed 16. Nov. 2023].

[19] Z. Wang, S. Zhu, Y. Cao, Z. Qian, C. Song, S. Krishnamurthy, K. Chan, and T. Braun, "SymTCP: Eluding Stateful Deep Packet Inspection with Automated Discrepancy Discovery," in *Network and Distributed Systems Security Symposium (NDSS)*, 2020.

[20] M. Handley, V. Paxson, and C. Kreibich, "Network Intrusion Detection: Evasion, Traffic Normalization, and End-to-End Protocol Semantics," in *USENIX Security Symposium*, 2001.

[21] Q. Wang, J. Chen, Z. Jiang, R. Guo, X. Liu, C. Zhang, and H. Duan, "Break the wall from bottom: Automated discovery of protocol-level evasion vulnerabilities in web application firewalls," in *2024 IEEE Symposium on Security and Privacy (SP)*, pp. 129–129, IEEE Computer Society, 2024.

[22] "Google WAF bypassed via oversized POST requests," Mar. 2022.

[23] Z. Qu, X. Ling, and C. Wu, "AutoSpear : Towards Automatically Bypassing and Inspecting Web Application Firewalls,"

[24] C. Topcuoglu, K. Onarlioglu, B. Jabiyev, and E. Kirda, "Untangle: Multi-Layer Web Server Fingerprinting," in *Network and Distributed Systems Security Symposium (NDSS)*, Feb. 2024.

Appendix

1. Web Framework Parsing Methods

Table 8 summarizes the methods used for parsing request bodies in our work in each framework. If a content-type is not supported by the default parser of a framework, popular third-party parsers are used instead.

TABLE 8: Request Body Parsing Methods in Tested Frameworks

Framework	**Content-Type**	**Parsing Method**
Express (Node.js)	application/json	`request.body`
FastAPI (Python)	application/json	`request.json()`
Laravel (PHP)	application/json	`$request->collect()`
Spring Boot (Java)	application/json	`Defined a class and used @RequestBody`
Laravel (PHP)	multipart/form-data	`$request->collect()`
Express (Busboy)	multipart/form-data	`form.parse(request)`
FastAPI (Python)	multipart/form-data	`field1: str = Form("none")`
Flask (Python)	multipart/form-data	`request.form`
Spring Boot (Java)	multipart/form-data	`@RequestParam(value = "field1") String field1`
Laravel (PHP)	application/xml	`$request->getContent()`
Spring Boot (Java)	application/xml	`Defined a class and used @RequestBody`
Flask (xmltodict)	application/xml	`parse(request.data)`
Gin (Go)	application/xml	Defined an `xmlform` with `field1` and `field2`