The Silent Danger in HTTP: Identifying HTTP Desync Vulnerabilities with Gray-box Testing

Keran Mu[1], Jianjun Chen[1,2,✉], Jianwei Zhuge[1,2], Qi Li[1,2], Haixin Duan[1,2], Nick Feamster[3]
[1]*Tsinghua University*
[2]*Zhongguancun Laboratory*
[3]*University of Chicago*

Abstract

HTTP Desync is a high-risk threat in today's decentralized Internet, stemming from discrepancies among HTTP implementations. Current automatic detection tools, primarily dictionary-based scanners and black-box fuzzers, lack insights into internal states of implementations, leading to ineffective testing. Moreover, they focus on the request-side Desync, overlooking vulnerabilities in HTTP responses.

In this paper, we present HDHUNTER, a novel automatic HTTP discrepancy detection framework using the gray-box coverage-directed differential testing technique. HDHUNTER can discover discrepancies in not only HTTP requests but also HTTP responses and CGI responses. We evaluated our HDHUNTER prototype against 19 state-of-the-art HTTP implementations and identified 17 new HTTP Desync vulnerabilities. We have disclosed all identified vulnerabilities to corresponding vendors and received acknowledgments and bug bounty rewards, including 9 CVEs from well-known HTTP software, including Apache, Tomcat, Squid, etc.

1 Introduction

Today's Internet has largely deviated from the end-to-end principle of its original design. Numerous middleboxes such as proxies, firewalls, and content delivery networks (CDN) are commonly deployed in the network. As a result, a typical end-to-end HTTP request may traverse multiple middleboxes before reaching its final destination.

However, this layered architecture introduces a potential threat: *HTTP Desync*. The vulnerability arises when different HTTP implementations in the chain may have processing discrepancies on the same message. Attackers can leverage the discrepancies to desynchronize the HTTP message queues, resulting in message smuggling and manipulation. Such attacks could lead to severe security consequences, such as cache poisoning, session hijacking, account takeovers, and security policy bypass. Numerous such vulnerabilities [27,29] and attacks [11] towards famous Web services have recently emerged, illustrating HTTP Desync has become a serious threat to the Internet.

Previous work has developed various detection tools for HTTP Desync vulnerabilities [13,24,45]. Smuggler [13] employs a series of predefined payloads to test the exploitability of websites, but this method lacks the capability to uncover new variants of attacks. On the other hand, T-Reqs [24] and HDiff [45] introduce black-box fuzzing techniques to generate test cases based on HTTP grammar or RFC documents to identify HTTP request smuggling attacks. However, they suffer from two limitations: First, the 'blind' nature of black-box testing makes their testing ineffective, because they lack insights into the internal states of the targets, leading to ineffectiveness and incompleteness in testing. Second, they only focus on the request side testing, overlooking the potential HTTP Desync vulnerabilities in HTTP responses.

Gray-box coverage-directed testing has proven to be highly effective in uncovering vulnerabilities in various targets, including TLS [9] and JVM [8]. This technique leverages execution coverage to guide the mutation of inputs, thereby enabling a more thorough exploration of program branches. However, the state-of-the-art gray-box coverage-directed tools like AFL [51] are not readily suitable for identifying HTTP Desync due to three challenges. First, existing tools like AFL are adept at identifying memory-related bugs within a single target, but HTTP Desync involves discrepancies between multiple HTTP implementations. Second, there is a lack of effective vulnerability detectors to detect Desync vulnerabilities in HTTP requests and responses. Third, HTTP Desync's potential to disrupt the HTTP message queue and TCP state can significantly destabilize the fuzzing process, resulting in numerous false positives and negatives.

In this paper, we propose HDHUNTER, a novel HTTP Desync vulnerability discovery framework to address the above issues. We address the first challenge by utilizing the execution coverage information from multiple implementations to optimize the test case generation. For the second challenge, we developed a new HTTP Desync detector to extract internal

✉ Corresponding author: jianjun@tsinghua.edu.cn

states from inside of the implementations to identify all forms of HTTP Desync vulnerabilities. For the third challenge, we implemented a snapshot-based execution framework that utilizes the snapshot to reset the network states effectively.

We implemented a HDHUNTER prototype and evaluated it against 19 state-of-the-art open-source HTTP implementations. We highlight five primary types of found discrepancies: 1) Non-standard numbers; 2) Inconsistent trailer section acceptance; 3) Non-standard line separator; 4) Different TE.CL attack handling strategies; 5) Incomplete response sanitization. We identified 17 HTTP Desync vulnerabilities in those implementations, affecting famous implementations like Apache, Tomcat, Squid, etc. We disclosed all of the found vulnerabilities to corresponding maintainers and received positive feedback. A total of 9 CVE IDs have been assigned. $4660 bounty has been granted by Internet Bug Bounty for the vulnerability discovered in Tomcat.

Contributions. In summary, we make the following contributions:

- We proposed a novel approach, HDHUNTER, that utilizes the gray-box coverage-directed differential testing framework to identify HTTP Desync vulnerabilities automatically.
- We developed and open-sourced a prototype of HDHUNTER[1], and evaluated it on 19 state-of-the-art HTTP implementations. We found 17 new HTTP Desync vulnerabilities in Apache, Tomcat, Squid, and other HTTP implementations.
- We conducted the first study to automatically identify Desync vulnerabilities in HTTP responses and CGI responses. We responsibly disclosed all of them to the corresponding vendors and received positive feedback.

2 Background

2.1 HTTP, CGI, and Request Smuggling

Hypertext Transfer Protocol (HTTP) is the foundation of web applications, functioning as a request-response protocol between clients and servers. HTTP has evolved significantly, with HTTP/1.1 and earlier versions using a text-based format, while HTTP/2 and later versions adopted a binary format. Despite its age, HTTP/1.1 remains widely supported due to its established infrastructure and ease of debugging, even though it presents challenges in message parsing. In this paper, 'HTTP' specifically refers to HTTP/1.1.

HTTP has two key features that enhance web communication: persistent connections and HTTP pipelining. Persistent connections (keep-alive) allow multiple HTTP requests and responses to use the same TCP connection, reducing the need for new connections. HTTP pipelining extends this by enabling multiple requests to be sent through the same connection at once without waiting for each response, though responses must be delivered in order. Together, these features reduce network and processing overhead, improving efficiency.

Common Gateway Interface (CGI) is a protocol for generating dynamic web content, serving as an intermediary between a web server and external applications. CGI scripts, written in various programming languages, enable interactive elements in web applications. Building on CGI, advancements like WSGI [15, 16] used by Python applications and FastCGI [34] used by PHP have emerged, improving performance and expanding CGI techniques. Other related technologies include SCGI [42], uwsgi [48], Rack [41], and AJP [3], all extending the original CGI concept.

HTTP Request Smuggling (HRS) is a technique that exploits inconsistencies in the interpretation of request boundaries between two web servers, typically a front-end proxy and a back-end server. By crafting ambiguous HTTP requests, an attacker can *smuggle* a request within another, leading to various security issues such as security controls bypassing, or request manipulating.

A common method for exploiting HRS involves leveraging two distinct HTTP encoding methods: no encoding and chunked encoding, regulated by two headers: `Content-Length` (CL) and `Transfer-Encoding` (TE). No encoding imposes no restrictions on the message body format and the value of CL denotes the message body length. Chunked encoding is intended for use when the message's overall length cannot be determined by the time the headers are dispatched. It adheres to a structured format where the content is divided into segments, each labeled with its own size indicator. Although, it is mentioned in the HTTP RFC [18] that "a sender MUST NOT send a Content-Length header field in any message that contains a Transfer-Encoding header field", HTTP servers exhibit diverse levels of support for chunked encoding and may not adopt the best practice to process messages when both TE and CL headers are present. This can lead to differences in how the message is recognized and handled by different servers.

Figure 1 demonstrates that when both TE and CL are present, the front-end proxy accepts the TE header and forwards the whole message without discarding the CL header. In the meanwhile, the back-end server accepts the CL header, leading to the splitting of one request into multiple requests.

2.2 Fuzzing Techniques

Fuzzing is an automated software testing technique, that provides a large quantity of unexpected or random input data to the test target, in order to identify potential vulnerabilities.

Categories of Fuzzing Techniques. Fuzzing techniques are categorized into black-box, white-box, and gray-box based

[1] Link to the repository of HDHUNTER: https://github.com/mukeran/HDHunter

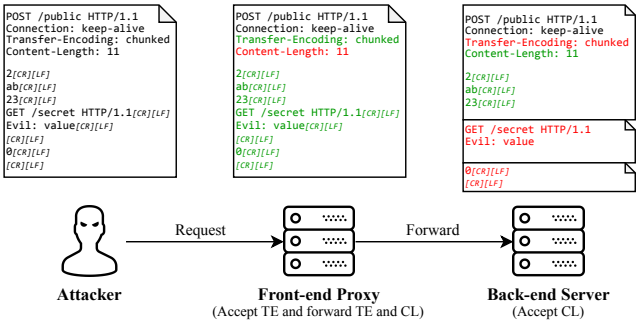


Figure 1: Example of HTTP Request Smuggling

on the fuzzer's knowledge of the software. Black-box fuzzing operates without internal knowledge, treating the software as opaque and testing it externally. White-box fuzzing requires a comprehensive understanding of the software's internals, enabling precise testing. Gray-box fuzzing occupies a middle ground, leveraging a partial understanding of the software's internal running states to enhance testing efficiency. It offers greater insight than black-box fuzzing yet does not require the exhaustive detail of white-box fuzzing.

Coverage-directed Fuzzing. In gray-box fuzzing, the most commonly used internal state to direct the fuzzing process is code coverage. Coverage in this context refers to the extent to which the code of the program is executed when subjected to test cases. By monitoring which parts of the code are being exercised by the inputs, gray-box fuzzing efficiently identifies untested portions of the program, directing the input mutation or generation towards these areas to enhance the thoroughness of the testing.

Differential Testing. Normally, the focus of fuzzing is on a single instance and is intended to detect memory-related vulnerabilities like buffer overflows, memory leaks, and null pointer dereferences. It uses various memory sanitizers as the oracle. In contrast, differential testing is particularly effective in discovering logic issues. This method entails analyzing the same inputs on multiple implementations or software versions to highlight differences in how each system handles the data by comparing their output. Such comparative analysis is essential to detect logical errors that may not be detected during a single-instance test.

3 Overview

3.1 Threat Model

The core of HTTP Desync lies in the discrepancies in how different implementations interpret the HTTP protocol, allowing attackers to perform a variety of malicious attacks. The aforementioned HTTP Request Smuggling is one common form of HTTP Desync.

In this paper, we extend the scope of traditional HTTP Request Smuggling to encompass six forms of desynchronization in HTTP transactions. First, we have expanded and categorized the form of HTTP Request-side Desync as follows.

- **Inconsistent number of messages.** The borders between HTTP messages are determined by the semantics of the message. The HTTP protocol includes multiple optional fields that define these boundaries, and different HTTP parsers may interpret these fields differently. Therefore, different HTTP parsers may recognize one payload into different numbers of HTTP messages. This is the traditional form of the HTTP Desync, as shown in Figure 2(a).

- **Inconsistent content of messages.** In some cases, the handling discrepancies did not bring more or less HTTP messages but resulted in different contents, as demonstrated in Figure 2(b). This threat model exists in protocol conversion between HTTP and other protocols like CGI, while the HTTP and CGI parsers may adopt different strategies to recognize the payload.

- **Inconsistent order of messages.** Modern HTTP servers use techniques like HTTP pipelining and persistent connections to improve performance. Those features allow the client to send multiple requests simultaneously and the server is supposed to process and respond to these requests in the order they were received. However, if the implementation sends the response in a wrong or unexpected order, it could result in HTTP Desync. Figure 2(c) provides an illustrative example of a proxy that handles messages in a reversed order, resulting in HTTP Desync.

Second, we extend the threat model to cover HTTP Response Desync, because issues prevalent in request handling can similarly affect response processing. In the multifaceted internet infrastructure of today, characterized by Content Delivery Networks (CDNs) and microservices, proxies often serve multiple endpoints, each managed by distinct entities. If an attacker gains control over any of these endpoints and discrepancies in response parsing exist among proxies or between proxies and clients, they can potentially infiltrate the response queue, leading to the manipulation of responses or the exposure of sensitive information. For example, services like AWS API Gateway and projects like Kubernetes Ingress support configuring different paths that point to different upstream URLs. If one of the upstreams is compromised and the response parser is vulnerable to the TE.CL attack, attackers can smuggle a fake response with both TE and CL headers into the victim's response queue, resulting in response manipulation. Figure 2(d),2(e) and 2(f) are 3 examples when inconsistent number, content, and order of messages take place in HTTP responses. It is important to note that endpoints may have not only an HTTP upstream but also an application up-

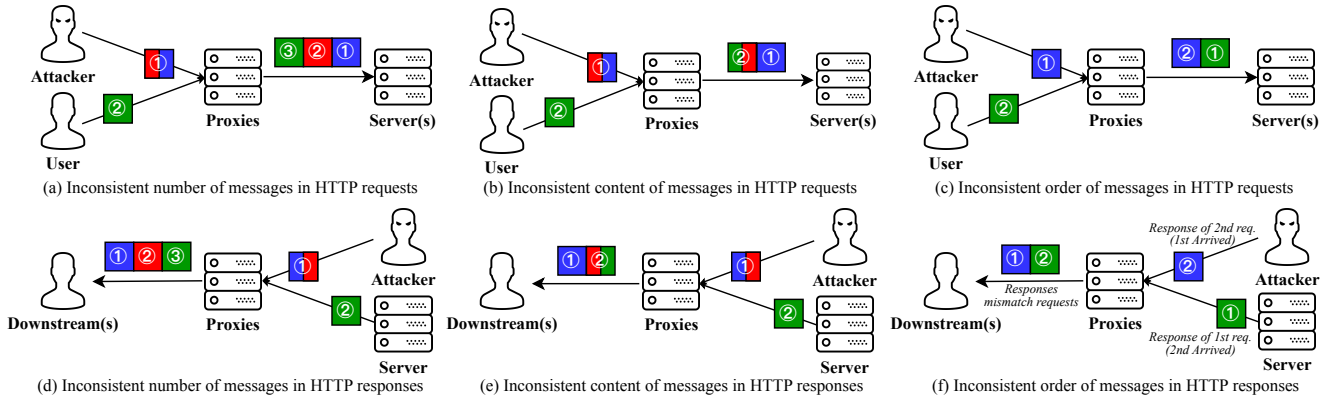


Figure 2: Examples of HTTP Desync: Inconsistent number, content, and order of messages

stream capable of delivering HTTP responses derived from CGI responses.

To the best of our knowledge, this represents the most comprehensive taxonomy of HTTP Desync attacks. As we will describe later, those insights enable us to detect and discover novel attacks missed by previous research.

3.2 Challenges

Typically, fuzzing involves three key components: (1) generating effective test cases; (2) executing the targets with each test input; (3) detecting whether vulnerabilities are triggered. Our research aims to advance existing detection tools in all three components to effectively identify HTTP Desync vulnerabilities. We have identified the following research challenges that need to be addressed to accomplish our objective.

C1. How do we generate effective test cases to maximize the possibility of HTTP desynchronization? HTTP desynchronization is caused by parsing and processing discrepancies between multiple implementations. The generation and mutation strategies should be designed based on the objectives.

Previous studies have presented several approaches to address this. T-Reqs [24] proposes CFG-based input generation and tree-level mutation to enhance its methodology. HDiff [45] utilizes similar methods while incorporating natural language processing techniques in its input generation. However, they are both black-box fuzzing techniques that lack internal insights from the target system. Their "blind" nature makes their testing ineffective in uncovering deep corner cases in implementations. Moreover, these tools do not generate test cases targeting HTTP responses and CGI responses.

On the other hand, current gray-box coverage-guided fuzzing tools like AFL excel in identifying memory-related bugs in a single target. In contrast, HTTP Desync involves discrepancies between multiple HTTP implementations. Presently, there are no off-the-shelf gray-box fuzzing tools for identifying HTTP Desync vulnerabilities, which require more effort in strategy design.

C2. How do we identify a potential HTTP desynchronization? Given a test input, we need an oracle to decide whether the HTTP desynchronization occurs, similar to sanitizers in memory-related bug identification. Previous methods, such as error-based and timeout-based detection, are inefficient and have a high false positive rate. Meanwhile, these methods target existing attacks and are not designed to find new types of HTTP Desync attacks based on the principle and overlook the detection of response-side Desync vulnerabilities. A new identification method is required to cover all potential HTTP desynchronization scenarios.

C3. How to efficiently restore states of HTTP implementations? HTTP desynchronization can disrupt both the HTTP message queue and the TCP state. The remaining state from each test can significantly influence the stability of the fuzzing process, leading to numerous false positives and negatives. To ensure effective fuzzing, it is crucial to start each test with a clear state. The straightforward solution is to restart the target for each test input to reset the state. However, the initialization process of large HTTP implementations like Tomcat can take a considerable amount of time. Therefore, developing an efficient mechanism for state recovery is essential to optimize the testing process.

4 Design

We propose HDHUNTER, a gray-box coverage-directed differential fuzzing technique targeting discrepancies between open-source HTTP implementations. Unlike existing black-box solutions, HDHUNTER can explore more execution paths in test targets thanks to the coverage information collected during processing. Besides, the combination of coverage from two implementations can be used as an indicator to help induce more discrepancies. HDHUNTER not only targets HTTP requests, but also HTTP and CGI responses, as illustrated in Figure 2.

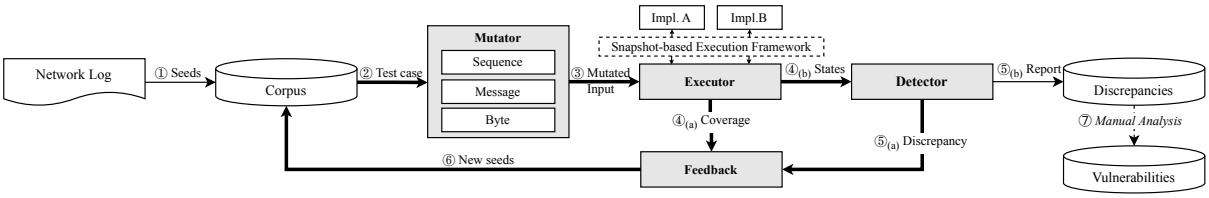


Figure 3: Architecture and Workflow of HDHUNTER

4.1 Workflow

We adopt state-of-the-art AFL-like fuzzing framework based on Genetic Algorithm, while adding targeted strategies and mechanisms for HTTP Desync.

The workflow of HDHUNTER is illustrated in Figure 3. It can be broken down into seven steps: 1) We manually extract HTTP messages from the network traffic as initial seeds and add them to the corpus; 2) The main fuzzing loop starts, where the fuzzer selects a test case from the corpus; 3) The mutators mutate the selected test case; 4) The executor runs the mutated input on two implementations while collecting their coverage and states; 5) The detector determines if there is any discrepancy between the outputs of these two implementations, and if so, generates reports; 6) The feedback mechanisms determine if the mutated input has value in triggering more discrepancies, if so, add it to the corpus, in any case loop steps 2–6; 7) Manually analyze discrepancies to identify HTTP Desync vulnerabilities.

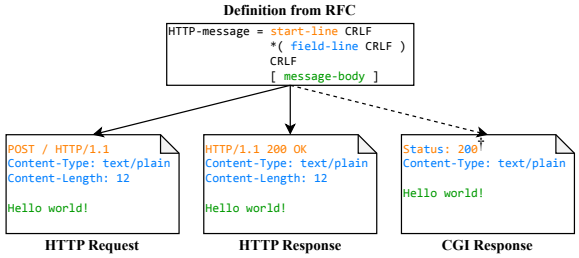
4.2 Test Case Generation

We create a new structure for test cases and designated mutation strategies to induce more discrepancies within the mutator, and use coverage-directed feedback to direct new test case generation.

4.2.1 Structure of Test Case

HTTP is a text-based network protocol. Unstructured test cases only support byte-level mutation strategies that cannot satisfy HTTP's semantic rules. It will take too much time for the fuzzer to generate a legitimate input. Therefore, HTTP syntax should be considered in the test case's structure to pave the way for high-level mutators.

As illustrated in Figure 4, HTTP messages, including requests and responses, are composed of three parts: the start line, the field lines (i.e.: headers), and the message body, according to RFC. They differ in the start line but share the same format in the subsequent parts. The start line and field lines have fixed formats, while the message body is either a raw binary stream controlled by `Content-Length` or a structured chunked encoding enabled by `Transfer-Encoding`. Besides, the HTTP pipelining and long connection allow



† The *Status* header in a CGI response is one of the field lines but acts as the start line.

Figure 4: Comparison Between HTTP Request, Response, and CGI Response: Three different colors represent three different parts of HTTP.

multiple HTTP messages in one single connection. Moreover, CGI responses are similar in format to HTTP responses, with a notable distinction in their method of conveying the status code. Unlike HTTP responses, CGI responses employ a specialized *Status* header for transmitting the status code. Consequently, it is feasible to design a universal test case structure that encompasses all three message types.

We restructure the test cases, which contain an array of HTTP messages. Each message is composed of three parts, start line, field lines, and message body, reflecting the nature of HTTP. The message body supports the raw and chunked encoding. Messages are stored using a dynamic tree structure, reflecting the ABNF definition listed in Appendix A, which is extracted from RFC. Each field is labeled with its datatype corresponding to its actual function: string, number, symbol, to facilitate subsequent mutation operations. However, the labels do not restrict the datatypes of the fields. The test case can be serialized into HTTP requests, HTTP responses, or CGI responses, adapting to their respective structural requirements as needed.

4.2.2 Mutation Strategies

To explore larger input space and introduce more discrepancies, we optimize the mutators with three levels of mutation strategies: 1) Sequence: Based on the fact that HTTP supports long connection and pipelining, sequence-level strategies can help explore potential discrepancies introduced by multiple messages; 2) Message: Message-level strategies reflect the

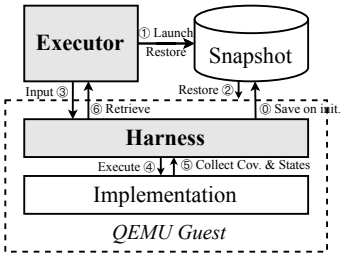


Figure 5: Workflow of *Snapshot-based Execution Framework*

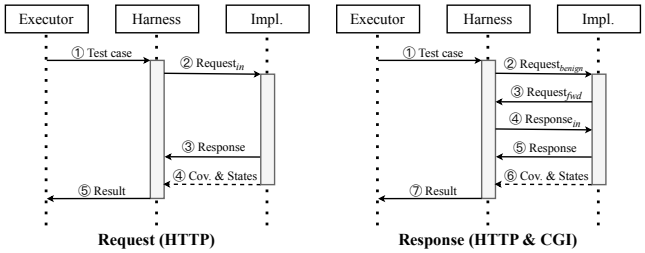


Figure 6: Execution Procedure

HTTP syntax, while blurring the formatting requirements; 3) Byte: Introduce randomness to the input to cover the discrepancies that caused by Robustness Principle. Detailed mutation strategies are present in Table 3. One test case can undergo multiple mutations using multiple mutation strategies in a single run.

4.2.3 Coverage-directed Feedback

The feedback of the seed selection process plays a crucial role in deciding whether a mutated input should be added to the corpus. In this process, AFL utilizes an accumulated edge hit count map, a mechanism designed to assess if the current input introduces new execution paths or activates existing ones with greater frequency, which is the core of the coverage-directed fuzzing. This approach is adaptable to various implementations, effectively steering the fuzzing process toward generating seeds that are more likely to activate a broader range of edges within each implementation. HDHUNTER utilizes this mechanism by combining two edge hit count maps from two implementations into one double-sized edge map, to reflect coverage information from both targets.

The seeds that trigger both new edges and discrepancies between two implementations will not be added to the corpus during the fuzzing process, because the inputs mutated from those seeds are more likely to trigger previously found discrepancies, which will affect the fuzzing efficiency.

4.3 Snapshot-based Executor

The executor runs the mutated input on both targeted implementations under the *snapshot-based execution framework*. The workflow of the framework is illustrated in Figure 5. The framework utilizes the snapshot technique to clear the network states efficiently. It also contains a test harness that supports testing not only HTTP requests but also HTTP responses and CGI responses. To support implementations written in interpreted programming languages, we adopt the solution proposed by Witcher [47] to collect coverage from interpreted languages, which modifies the bytecode interpreters of the programming languages to update the coverage information using the line number and opcode of the current and prior instructions. For implementations written in compiled languages, such as C and C++, we use SanitizerCoverage [46] provided by Clang to collect coverage. The coverage information is situated within two distinct processes within two QEMU guests. The generated coverage map is collected from these processes via shared memory and guest memory access.

4.3.1 Snapshot-based State Recovery

HTTP is running over TCP/IP protocol. The remaining states of the network layer can affect HTTP processing. Besides, each implementation has coded its own buffer mechanism, which may have flaws. Thus, it is necessary to clear its states.

Snapshot is commonly used in file systems and virtual machines. It saves states of a specific point and is designed to restore them efficiently. Previous research, including Nyx [43] and its extension Nyx-Net [44], has proposed a fast snapshot restoration mechanism for hypervisors. However, Nyx utilizes hardware-level processing tracing to collect the code coverage, which is good for kernels and drivers but brings more noise for userspace applications. Nyx-Net supports AFL-like coverage collection, but it implements a custom hook-based network stack to communicate with test targets, which has limited support for multi-processing programs and has compatibility issues when testing complex network applications such as HTTP servers.

To enable highly efficient state recovery for HTTP implementations, we reused Nyx's fast snapshot reloading mechanism, while implementing AFL-like coverage collection. We built a dedicated harness to communicate with the test targets using Linux's original socket API. The harness uses Nyx's API to take a snapshot right after the target is ready and collect its edge map through shared memory. The states are restored after executing each test case, and the coverage is retrieved using the guest memory access functionality.

The harness checks if the target is ready by repeatedly executing a benign probing input. Once the input is accepted without error, we assume that the target is in the ready state.

4.3.2 Support for HTTP Requests, Responses, and CGI Responses

The test harness has two modes of execution, one for HTTP requests and another for HTTP and CGI responses. Figure 6 demonstrates the procedures for execution.

For HTTP requests, the test harness follows a standard procedure to send the request. It obtains the test case from the executor, transforms it into an HTTP request ($Request_{in}$), and passes it to the implementation.

For HTTP and CGI responses, the implementation is configured to forward the HTTP and CGI requests to the harness. After obtaining the test case, the harness initially sends a benign request ($Request_{benign}$) to the implementation. After receiving the forwarded request ($Request_{fwd}$), the harness responds with the formatted test case ($Response_{in}$).

4.4 HTTP Desync Detector

The detector is responsible for identifying discrepancies between implementations. It follows a pre-defined set of rules to identify discrepancies using the internal states collected during the execution process. We will begin by detailing how we collect the internal states from implementations via code insertion, and then introduce our detection rule.

4.4.1 Internal States Extraction

Information about the handling process can be partially acquired by referring to the forwarded requests and responses. However, information collected through this method can be inaccurate due to the sanitization and post-processing. For example, NGINX will convert chunk-encoded requests to raw encoding during the forwarding process so that we cannot know what kind of encoding NGINX has used to process the request from the forwarded message. Therefore, we take measures to extract internal states from the inside of the implementations.

We collect seven types of states that form the State Tuple:

$$(Count, Consumed, Body, Encoding, CL, Order, Status)$$

1) *Count*: the number of requests or responses that the implementation recognized; 2) *Consumed*: the actual length of messages that the implementation consumed during the parsing process; 3) *Body*: the content of the parsed message body. Since it is hard to extract the message body in some implementations, it is downgraded to the length of the message body; 4) *Encoding*: the encoding used in the parsing process: raw or chunked; 5) *CL*: the value of the `Content-Length` header; 6) *Order*: the parameter that describes the order of messages; 7) *Status*: the HTTP response status code.

We collect the first five types of states by manually inserting customized code into the HTTP handling functions of the implementations. The inserted code utilizes the internal API of the corresponding implementation to extract headers, copies the internal buffer to retrieve the body content and length, and records the message count before dispatching. The consumed length is tracked by incrementing counters each time the implementation reads data from the buffer. The detailed example of the code we inserted into Apache is demonstrated in Appendix D.2.

For *Order* and *Status*, we collect them externally and automatically. Although both disordered requests and responses can cause inconsistent order of messages, the most convenient way to check if the disorder occurs is to check if the order of the final responses matches. Therefore, we employ an external method to determine *Order* by embedding a header called *X-Desync-Id* with a distinct UUID string during execution. The implementations are configured to forward the original *X-Desync-Id* header. *Order* will be set to the collection of *X-Desync-Id* values. The status code is a useful and stable external flag to identify if a request is accepted or denied by the implementation. We collect the status code by reading the first lines of produced or forwarded responses in the test harness.

4.4.2 Detection Rule

The detailed detection rules are illustrated in Appendix C. A discrepancy is considered to exist if any of the rules are not satisfied. For *Count*, *Body* and *Order*, the states from two implementations are simply compared to ascertain whether they match exactly. For *Encoding*, *CL* and *Consumed*, the *Status* code is first consulted. In the event that both implementations generate status codes between 400 and 599, which are treated as the same error state, it is not meaningful to compare their encodings, content lengths, and consumed lengths, as they end up with the result — to reject the current message and possibly stop parsing the subsequent input.

5 Evaluation and Findings

5.1 Experiment Setup

Evaluation Targets Selection. HDHUNTER is a gray-box fuzzer designed to find HTTP discrepancies in open-source projects, so we focus on open-source HTTP implementations. We consider both the GitHub stars and deployment popularity since we find that some HTTP implementations, such as Apache HTTP Server, receive fewer stars than their popularity on GitHub.

At last, we selected 19 state-of-the-art HTTP implementations written in different languages as our evaluation targets, including integrated servers, cache servers, network frameworks, and application servers, detailed in Table 1. We test these implementations against their latest version at the time of the experiment.

Table 1: Evaluation Targets and New Vulnerabilities Found

Category	Name	Version	ME[1]	CGI	Star[2]	Host[2]	Vuln.	Status	Inconsistency	Attack	CVE[3]	Severity
Integrated Servers	NGINX	1.25.2	✓	✓	19.3K	20.1M	-	-	-	-	-	-
	Apache	2.4.57	✓	✓	3.3K	13.7M	4	Fixed	Resp. TE.CL	Resp. Forgery	✓	Low
	Lighttpd	1.4.70	✓	✓	547	3.6M	-	-	-	-	-	-
	H2O	2.2.6	✓	✓	10.6K	2.1K	1	Fixed	Number Parsing	Req. Smuggling	-	N/A
	HAProxy	2.8.2	✓	✓	4.1K	30.0K	-	-	-	-	-	-
Cache Servers	Squid	6.1	✓	✗	1.8K	5.1M	1	Fixed	Number Parsing	Req. Smuggling	✓	Critical
	Varnish	7.3.0	✓	✗	3.4K	1.7M	-	-	-	-	-	-
	ATS	9.2.0	✓	✗	1.7K	126.2K	1	Fixed	Trailer Section	Req. Smuggling	✓	Critical
Network Frameworks	Twisted	23.8.0	✓	✗	5.3K	78.2K	2	Fixed	Resp. Order	Resp. Stealing	✓	High
								Confirmed	Resp. TE.CL	Resp. Forgery	✓	N/A
	Tornado	6.3.2	✗	✓	21.3K	144.3K	-	-	-	-	-	-
	gevent	23.7.0	✗	✓	6.1K	N/A	1	Fixed	Trailer Section	Req. Smuggling	✓	Critical
	Eventlet	0.33.3	✗	✓	1.2K	N/A	3	Confirmed	Number Parsing	Req. Smuggling	-	N/A
									Trailer Section	Req. Smuggling	-	N/A
									Req. TE.CL	Req. Confusing	-	N/A
Application Servers	Tomcat	10.1.9	✗	✗	7.0K	690.5K	1	Fixed[4]	Trailer Section	Req. Smuggling	✓	High
	Jetty	12.0.0	✗	✗	3.7K	367.5K	1	Fixed	Number Parsing	Req. Smuggling	✓	Moderate
	Puma	6.3.0	✗	✓	7.5K	N/A	-	-	-	-	-	-
	Falcon	0.42.3	✗	✓	2.4K	2.2K	1	Fixed	Number Parsing	Req. Smuggling	✓	Moderate
	uWSGI	2.0.21	✗	✓	3.4K	N/A	-	-	-	-	-	-
	Waitress	2.1.2	✗	✓	1.3K	20.4K	-	-	-	-	-	-
	Gunicorn	21.2.0	✗	✓	9.2K	175.1K	1	Reported	Req. TE.CL	Req. Confusing	-	N/A

[1] ME: Support for hosting multiple endpoints pointing to different upstreams.
[2] Data crawled from GitHub and Censys on Nov. 16th, 2023.
[3] Every CVE except Twisted's is assigned by its project maintainers. Twisted's is assigned by MITRE with the maintainers' permission.
[4] $4660 bounty has been granted by Internet Bug Bounty to acknowledge our contribution.

HDHUNTER extracts the State Tuple by inserting code into targets to identify handling discrepancies. The statistics of the inserted lines, functions, and files are attached to Appendix D.1. An example of code insertion for Apache HTTPd is attached to Appendix D.2

In addition to identifying discrepancies related to HTTP requests, HDHUNTER has the exceptional ability to identify discrepancies in HTTP and CGI responses. Among 19 selected implementations, 9 support hosting multiple endpoints, 5 of which support CGI proxy. Since the threat model of HTTP Response Desync we previously defined can only apply to HTTP implementations with multiple endpoints, we only set up response testing configurations for these implementations.

All implementations are configured to the single-thread mode, in order to minimize the influence of multi-threading on coverage and path collection during execution.

Ultimately, we established a total of 171 testing pairs for HTTP requests, 36 for HTTP responses, and 10 for CGI responses. We run our tool against each test pair for 12 hours and repeat for 5 times. For initial seed selection, we use tcpdump [21] to capture real-world traffic on our router and select 20 raw HTTP messages covering different HTTP specifications for each run. Our seed selection criteria is to collect initial seeds that encompass a wide variety of HTTP methods and headers. This includes diversity in values, such as identity/no encoding and chunked encoding. Additionally, we aim to include seeds with varied body content types, such as URL encoding, form data, JSON, and binary format.

Experiment Platform Setup. We conduct our experiment on a machine with Intel Core i7-1260P (up to 4.70GHz) CPU, 64GB RAM, and Ubuntu 22.04.4 LTS (6.5.0-41-generic x86_64) operating system.

5.2 Findings

We ran our HDHUNTER prototype on each pair within each configuration and removed the duplicates by utilizing the coverage. We then manually analyzed each report by referring to the State Tuple. We highlight 5 primary types of discrepancies found by HDHUNTER, as detailed in Appendix E. The *Trailer Section* and *Response TE.CL* are 2 novel discrepancies we identified. Besides, we found new variants in other 3 types of discrepancies. In total, we identified 17 new HTTP Desync vulnerabilities, affecting well-known HTTP servers such as Apache, Tomcat, Squid, etc. We have disclosed these

vulnerabilities to their maintainers and received 9 CVE IDs, as shown in Table 1.

5.2.1 Non-standard number parsing

Number fields are common in HTTP protocol. Two important number fields control the boundary of the message body: the value of the `Content-Length` header and the chunk sizes. HTTP RFCs allow decimal digits in CL and hexadecimal digits in the chunk sizes. However, we found 8 tested HTTP implementations that allow other characters in these fields, including *0x prefix*, *+ prefix*, *_ between*, and *any suffixes*. We demonstrate the detailed acceptance of these characters in Appendix F. This behavior results in inconsistent number and content of messages.

For example, let us consider how Squid and H2O handle a chunk with a 0x-prefixed size respectively. Squid processes the chunk normally, but H2O interprets it as the last chunk, which indicates the end of the request, leading to a significant difference in their understanding of the message's length and boundary. As a result, an intentionally crafted message may cause H2O to misinterpret it as two separate requests.

5.2.2 Inconsistent trailer section acceptance

Trailer sections enable the ability to send additional metadata after sending contents in chunked messages. Despite being an RFC standard, state-of-the-art HTTP implementations show no interest in supporting it while handling it diversely. We will discuss the inconsistencies in two stages: parsing and forwarding.

In the parsing stage, some implementations, such as Apache HTTP Server and HAProxy, will accept trailer sections and validate the format. They throw 400 when sending malformed trailer sections. In contrast, implementations like NGINX accept requests with trailer sections even if they are malformed. Other implementations do not support trailer sections at all.

In the forwarding stage, most tested implementations ignore trailer sections because of the request sanitization. However, Apache Traffic Server does not sanitize the request and forwards the original raw request with trailer sections.

For implementations that do not support trailer sections, they may misinterpret the trailer section as other HTTP components, leading to an inconsistent number of messages. We found that gevent, Eventlet, and Puma exhibit this issue. When an HTTP request with a trailer section is sent, these implementations respond with two HTTP responses: the first is normal, and the second has a 400 status. Analysis shows that gevent and Eventlet drop the first line of the trailer section and treat the rest payload as a new request. Puma skips two characters after the last chunk, leading to similar misinterpretation.

Tomcat supports the trailer section but has problems parsing the trailer section if there's no colon in the line. It will skip lines until one line has a colon. Therefore, when processing the first payload in Appendix G.2, Tomcat would interpret it as two requests while other implementations interpret it as three.

5.2.3 Non-standard line separator

The standard line separator for HTTP is *CRLF*: a Carriage Return (`\r`) together with a Line Feed (`\n`). We found several HTTP implementations that allow the use of a single LF as the line separator. This tolerance may result in inconsistent number of messages between two implementations if one only allows CRLF and accepts LF in the chunk extension, while the other allows both LF and CRLF. Please refer to Appendix G.2 for a detailed payload and description. We have verified the payload on NGINX and Gunicorn.

5.2.4 Different request TE.CL handling strategies

Sending both TE and CL headers at the same time is a classic way to introduce desynchronization. Most state-of-the-art HTTP implementations have taken measures to mitigate this issue. We found no implementation forwards these two headers without sanitization. Some of the implementations reject the request and respond with status 400. Others accept the request, but they handle persistent connections with different behaviors. Some implementations abort persistent connections upon receiving the request, while others maintain them. It should be noted that uWSGI does not support chunked encoding by default. It would read the request body according to the CL header.

Eventlet and Gunicorn are two implementations that support Python's WSGI interface. They support the chunk-encoded message body, but they also accept the CL header and pass the value to the application through the `CONTENT_LENGTH` environment variable. According to most CGI standards, applications should obey the value of the CL environment variable when processing the request body. However, developers of the applications may choose to read the whole HTTP body without referring to it, resulting in inconsistent content of messages.

5.2.5 Different response TE.CL handling strategies

HTTP requests and responses differ only in the first line, so the classic threat model that confuses the request's boundary using both TE and CL headers should also work for responses. We found plenty of discrepancies in their handling of HTTP and CGI responses.

When handling HTTP responses, the majority of implementations are consistent with the behavior of handling requests. They accept the TE header and keep TE or CL when forwarding. Some of them abort the persistent connections with the upstream server on receiving responses with both TE and CL headers. Varnish throws *503 Backend Fetch Failed* when receiving such responses. H2O and Twisted unexpectedly

accept the CL header, which is inconsistent with their behavior when processing requests. Twisted even forwards both TE and CL headers to the downstream client, which leads to inconsistent number of messages.

As for CGI responses, five tested CGI proxies demonstrate three different behaviors. Lighttpd and HAProxy accept the TE header and chunk-encoded message body. NGINX and H2O accept the CL header. Apache does not refer to either CL or TE but forwards both TE and CL and the whole body to the downstream client. By specifying a smaller CL or constructing a chunked message body, inconsistent number of messages occurs between Apache and the downstream client. We found that four of Apache's *mod_proxy* CGI modules, including FastCGI, SCGI, uWSGI, and AJP, are affected.

5.2.6 Other notable discrepancies

We found that when sending two pipelined requests to Twisted, it will handle them simultaneously without waiting for the response of the first request. If the first request takes longer to process than the second request, the responses are disordered, resulting in inconsistent order of messages.

5.3 Attacks

This section delves into the exploitation of target implementations by capitalizing on previously identified inconsistencies. We outline four distinct attack techniques, visually represented in Figure 7.

5.3.1 Request Smuggling

By leveraging inconsistent number of messages, we can perform the classic request smuggling attack to bypass the authentication mechanism and execute any request.

Reverse proxies and cache servers can apply restrictions and require authentication on certain request paths to protect sensitive APIs, such as the back-end management and command execution interfaces. Normally, the proxy will block a direct request towards these paths.

However, when Tomcat, gevent, Eventlet, and Puma are deployed as the upstream server of the reverse proxy that forwards the raw request without sanitization, an attacker can leverage their misbehavior of handling trailer sections to bypass the access control mechanism.

We have practiced this attack on Apache Traffic Server and gevent. ATS does not sanitize the request and forwards the raw request with trailer sections. We configured ATS to forward requests that target at */path1* to the upstream gevent application. By sending payloads in Figure 7(a), an attacker can access */path2* on the gevent application, which cannot be directly requested through ATS.

Non-standard number parsing and line separators can also be leveraged to facilitate request smuggling, as they result in inconsistent number of messages.

5.3.2 Request Confusing

Sometimes, boundary confusion cannot smuggle a new request but can interfere with the application to retrieve the proper content, resulting in condition bypass. We refer to this attack as Request Confusing.

We can leverage the defeats, that accept chunked encoded body while setting the value of the `CONTENT_LENGTH` environment variable according to the CL header in Gunicorn and Eventlet, to perform Request Confusing. According to the standard of WSGI [15, 16], applications should not attempt to read more data than is specified by the `CONTENT_LENGTH` value, which seems to make this value the authoritative request length. However, developers of the applications may choose to read the whole input without referring to that value. We find a possible combination that can lead to Request Confusing in Gunicorn and Flask, where Flask is deployed as a WSGI application.

Gunicorn accepts *Chunked* with a capitalized C as a valid chunked encoding indicator. Flask has a function to get the content length of the request body. If the value of the `HTTP_TRANSFER_ENCODING` environment variable is not exactly *chunked*, it will return the value of `CONTENT_LENGTH` environment variable. Meanwhile, Flask will parse the whole as form data input without referring to `CONTENT_LENGTH`.

Figure 7(b) demonstrates the attack scenario. After sending the payload that contains a malformed *Chunked* and a CL header with value 0, developers who use Flask can successfully get the form-encoded data through `request.form`, but the value of `request.content_length` is 0. Any code that uses that value will incorrectly assume that the length of the body is 0, resulting in potential condition bypasses.

5.3.3 Response Stealing

Response Stealing refers to an attack method where attackers exploit discrepancies in response processing across implementations to illicitly acquire a victim's response. This can lead to significant issues such as privacy breaches or cookie hijacking.

The disordered responses issue found in Twisted can be leveraged to perform Response Stealing. As illustrated in Figure 7(c), we assume there is a proxy that handles incoming requests and forwards them in pipelined form to a Twisted server through a persistent connection. For instance, proxies employing the `Net::HTTP::NB` library from Perl to dispatch requests are aligned with this requirement. The Twisted server serves as a shared web-hosting gateway that allows tenants to configure an endpoint pointed to their servers or applications. By holding the request sent by himself, the attacker can steal the first response that should have been sent to the victim.

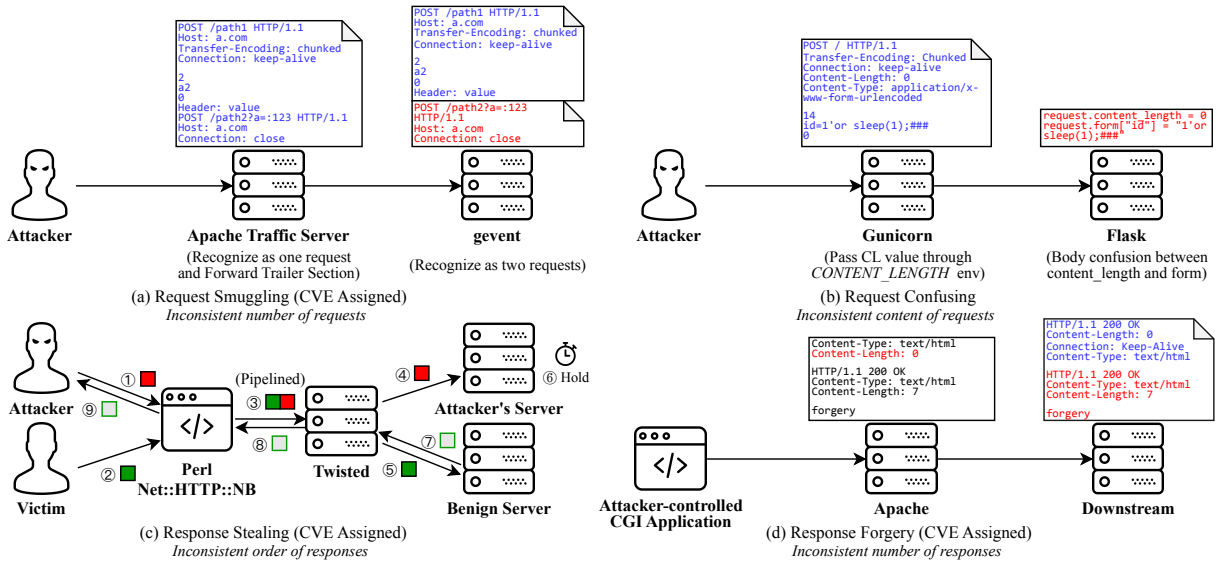


Figure 7: Examples of HTTP Desync Attacks We Discovered

5.3.4 Response Forgery

Victims' responses can not only be stolen but also manipulated. Response forgery is a type of attack in which an attacker creates a deceptive response to replace the benign one.

The improper handling of TE and CL headers in CGI responses by Apache can also be exploited to execute a Response Forgery attack. As shown in Figure 7(d), when the attacker-controlled application sends a CGI response, which contains a misleading CL header whose value is 0 as the header and a deceptive response as the message body, Apache does not refer to the CL, accepts the whole message body, converts it into an HTTP response without sanitization, and forwards them back to the downstream. If the downstream and Apache hold a persistent connection and enable HTTP pipelining, the encapsulated response will poison the response queue and be forged as the second response, resulting in Response Forgery. In fact, the HTTP pipelining is not required since the downstream may not clear the receiving buffer after processing a response. Consequently, after the second request is forwarded, any residual payload in the buffer is mistakenly interpreted as its response.

5.4 Comparison

Comparing to Existing Tools. Before evaluating our tool, we used existing tools, including T-Reqs [24] and HDiff [45], to test our evaluation targets. T-Reqs by default uses three distinct configurations to test the first line, headers, and bodies respectively. We manually merged three configurations into one to cover all three parts. We ran both T-Reqs and HDiff for 12 hours and 5 times. The evaluation results indicate that they indeed found some discrepancies. However, these discrepancies can cause at least one implementation of the pair denying an input and are difficult to exploit for HTTP Desync attacks. Table 2 shows the comparison of discovered discrepancies with these two tools. They are capable of identifying discrepancies caused by Line Separator and Request TE.CL, but missing the remaining discrepancies, including those pertaining to Number Parsing and Trailer Section. To find out the underlying cause, we investigated their runtime logs. Our findings indicate that their generators are unable to generate a valid payload capable of triggering the discrepancies due to a lack of coverage guidance. Additionally, their detectors are unable to identify some types of discrepancies due to the absence of internal states.

Table 2: Comparison Between Types of Discovered Discrepancies

	Number Parsing	Trailer Section	Line Sep.	Req. TE.CL	Resp. TE.CL
T-Reqs	○	○	●	◑	○
HDiff	◑	○	●	◑	○
HDHUNTER	●	●	●	●	●

○ : Not capable; ◑ : Capable to find some types; ● : Capable.

Furthermore, we performed a comparison experiment to find out how code coverage contributes to the discrepancy exploration process. We kept running T-Reqs, HDiff, and HDHUNTER and collecting the respective increment in the number of covered edges after processing a baseline request. Figure 8 shows an example of coverage growth for the Apache–NGINX and Apache–Tomcat combination. The evaluation shows that HDHUNTER covered more edges than

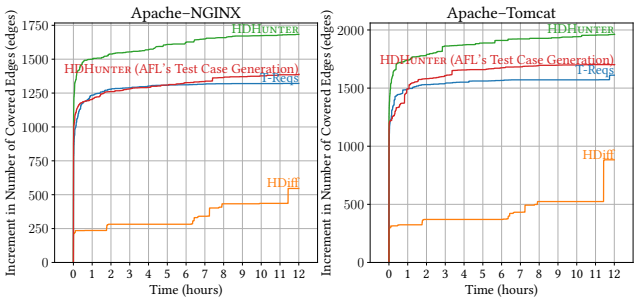


Figure 8: Comparison of Covered Edges over Time when Testing Apache–NGINX and Apache–Tomcat Combinations

T-Reqs and HDiff under the same circumstances, thereby increasing the likelihood of uncovering deeper vulnerabilities.

Ablation Study. To evaluate the contributions of our design, we conducted an ablation study by replacing two of the key components of our system — the test case generation and snapshot-based executor — with existing implementations.

To assess the efficacy of HDHunter's test case generation, we replaced it with AFL's original generation strategies. We used the same set of seeds to run the experiment. As illustrated in Figure 8, HDHunter with AFL's test case generation covers more edges than T-Reqs and HDiff, but fewer edges compared with our HDHunter's test case generation. Additionally, the execution speed significantly decreased when using AFL's original generation, dropping from 120 exe/s to 13 exe/s during the Apache–NGINX evaluation. We discovered that the AFL's byte-level mutations generated a substantial number of incomplete test cases, which caused timeouts and significantly slowed down the fuzzing process.

We then assessed the performance of our snapshot-based executor in comparison to the traditional full-restart approach by measuring the executions per second (exe/s) across various test setups. In scenarios where full-restart operates efficiently, such as the Apache–Squid combination, our framework increased execution speeds significantly — boosting performance from 1.2 exe/s to 62 exe/s, achieving an acceleration factor of up to 52 times. For more time-intensive restart scenarios, such as the Apache–Tomcat combination, the snapshot framework improved performance from 0.24 exe/s to 21 exe/s, corresponding to an even higher acceleration ratio of 88. These results underscore the significant efficiency gains possible with our snapshot-based executor.

6 Discussion

6.1 Contribution of Fuzzing

In addition to enhancing the automation of the vulnerability detection process, fuzzing can also identify previously unknown variants of vulnerabilities that developers may not be aware of. Furthermore, with coverage guidance, the traversal depth is greater than that of black-box approaches.

Taking number parsing for example, our fuzzer discovered vulnerabilities rooted in programming languages that many developers had overlooked. Another example is the trailer section issue newly discovered by HDHunter, where our tool identified a novel payload attached in Appendix G.2 that lacked a colon in a trailer field, which goes beyond the traditional TE.CL threat model.

6.2 Insights

We dive into the insights behind the vulnerabilities, which can be classified into three categories.

Number Parsing. We discovered that five of the implementations we tested do not adhere to the HTTP RFCs regarding number parsing, despite clear restrictions on permissible characters outlined in the RFCs. We identified that the primary cause of this discrepancy lies in developers frequently relying on built-in number parsers of programming languages, such as `int()` in Python. However, these parsers vary across different programming languages, each adhering to their unique standards for number parsing. This includes differing levels of support for various formats and base systems, like underscore separators and hexadecimal notation, as shown in Appendix F. Discrepancies among HTTP RFCs and the standards of different programming languages create unintentional semantic gap variations.

Trailer Sections. We observed that many implementations have different support for trailer sections, leading to HTTP Desync attacks. These features, being rarely used in HTTP and relatively more complex to test, often escape thorough scrutiny. However, with the help of coverage information guiding the generation of test cases, we can produce test cases capable of activating the erroneous branches associated with these least-used components or hidden in deep program logic, thus uncovering vulnerabilities missed by previous research.

CGI Converting Issues. Another type of new vulnerability we found stems from the protocol conversion process between CGI and HTTP. CGI protocols, which are binary-based or reliant on in-process communication, define message boundaries using their own distinct mechanisms instead of relying on HTTP's TE or CL headers. While certain CGI protocols mandate that the gateway should not read data beyond what is specified by the CL if present, not all implementations follow this rule. We found that headers declared in CGI responses are often carried over into HTTP responses, leading to new types of HTTP Response Desync.

6.3 Responsible Disclosure

We responsibly disclosed the vulnerabilities in Table 1 to the respective vendors. Falcon, Jetty, Squid, H2O, gevent, Tomcat, ATS, Apache, and Twisted have confirmed and

patched the vulnerabilities, with 9 CVE IDs assigned. We received a 4660 USD bounty from Internet Bug Bounty [23] for the Tomcat vulnerability. Eventlet's maintainers confirmed the issues and plan to fix them in future versions. Gunicorn's developers received our report, but we have not received further updates from them.

6.4 Mitigation

HTTP Desync is caused by the handling discrepancies between implementations, which is a long-addressing problem. To mitigate HTTP Desync, we propose the following three solutions:

Deny Malformed Messages and Support Required Specifications. A major cause of HTTP Desync vulnerabilities is deviations from the standards outlined in RFC documents, such as supporting non-standard number format and lacking support for trailer sections. Developers of HTTP handlers should thoroughly understand the relevant RFCs and ensure their implementations conform to these specifications by denying malformed messages and supporting required specifications. It is important to note, however, that some RFC guidelines are advisory rather than mandatory. While strict adherence to RFC specifications can significantly reduce issues, it may not completely eliminate them.

Sanitize the Message Before Forwarding. Most HTTP Desync attack scenarios involve interaction between a proxy and a server. If the proxy forwards an unambiguous message to the server, it eliminates the potential for a discrepancy. Hence, it is crucial for the proxy to sanitize the message before forwarding it. This process can be done by first parsing the original message and then reconstructing it anew for forwarding. A significant number of state-of-the-art HTTP proxies have already adopted this practice and have lowered the occurrence of HTTP Desync, demonstrating its efficacy.

Integrate Differential Fuzzing into the Development Workflow. Incorporating differential fuzzing into the implementation's validation routine can greatly help locate and resolve handling discrepancies. This method should be used alongside unit testing in the development stage to ensure a more secure implementation.

6.5 Limitation

Our methodology is not applicable to closed-source HTTP services, as it is not feasible to gather coverage and internal state information externally. Nonetheless, new findings from open-source implementations can serve as valuable references for testing whether similar issues exist in closed-source services.

Moreover, our work applies only to HTTP/1.1 due to its specific input structure, mutation strategies, and detector. HTTP/2 and HTTP/3 have transformed into binary protocols, eliminating the text-based parsing differences. However, to support legacy HTTP/1.1 clients, HTTP proxies have implemented protocol downgrade features that can convert messages to HTTP/1.1 format, potentially leading to HTTP Desync by exploiting their boundary understanding gap. Previous work [28] has systematically summarized HTTP/2 Desync attacks. In addition, the complexity and flexibility of HTTP/2's features, such as multiplexing, stream prioritization, etc., introduce expanded attack surfaces that can be exploited for DoS attacks. To detect such threats, the test case generation and the detector need to be refactored, but other components and the workflow can be reused.

Additionally, we acknowledge that developing HDHUNTER requires considerable manual effort during code insertion to extract internal states and discrepancy analysis. However, the code insertion is one-time work, and the tool can be run periodically in case software updates may introduce new vulnerabilities. So in the long term, we believe the approach will have a good return on investment. Since the Large Language Models (LLM) can read, understand, and modify the source code, further research can utilize LLMs, like GPT [35], Gemini [12], LLaMA [1], etc., to reduce human labor. Previous research [52] has leveraged LLM to generate fuzz drivers. In our case, LLM can help locate the entry function of the HTTP handler by explaining the functions within the project and subsequently filtering out those that are associated with HTTP handling. For discrepancy analysis, targeted validation environments can be set up to validate the presence of existing vulnerability types to reduce human labor.

7 Related Work

HTTP Desync. The concept of HTTP Desync was first raised by Kettle [26]. The article extended the scope of traditional HTTP Request Smuggling [32], the confusion between `Content-Length` and `Transfer-Encoding`, to the parsing discrepancy of the TE header. There are more studies [27, 29] involving new variants of HTTP Desync, but they still focus on the inconsistent number of HTTP requests interpreted between HTTP proxies and servers. In contrast, the Request Confusing raised in our paper focuses on the inconsistent content of messages interpreted between CGI gateways and applications. Doyhenard [14] first raised HTTP Response Desync. By controlling the responses, the attacker can manipulate the HTTP response queue to inject crafted messages into the HTTP pipeline. However, these studies mainly focus on manually discovering HTTP Desync. Our work discovered that Desync can further arise in CGI responses.

Another response-related vulnerability, HTTP Response Splitting [2], is a type of Web application-level vulnerability distinct from HTTP Desync. It involves the injection of headers and bodies into a response by leveraging bugs in the Web application. In contrast, Response Forgery and Stealing are interpreter-level vulnerabilities, caused by the response

forwarding misbehavior of HTTP servers and CGI gateways.

Previous research proposed different techniques to find HTTP Request Smuggling vulnerabilities. T-Reqs [24] proposed a grammar-based fuzzer to identify discrepancies and HRS vulnerabilities. HDiff [45] utilized the natural language processing techniques to extract rules from the RFC documents to generate semantically diverse inputs in differential testing. Both of them are black-box fuzzing techniques without obtaining information from the test target, limiting their effectiveness. HTTP Garden [25] leveraged the path information to guide test case generation to discover HRS vulnerabilities. In this paper, we employ the gray-box fuzzing approaches to discover the HTTP Desync vulnerabilities. With the help of coverage, we are able to find deeper vulnerabilities. Moreover, we systematically summarized the categories of HTTP Desync, broadened the HTTP Desync to HTTP responses, and introduced a snapshot-based execution framework to mitigate the impact of network state on results.

Broadly speaking, HTTP Desync belongs to a family of "semantic gap" attacks. Similar inconsistency problems also exist in other systems, such as email systems [6, 53], Web application firewalls [50], cache systems [5, 30], CDN systems [7, 22, 31, 54], and Web applications [49].

Gray-box Fuzzing. Gray-box Fuzzing employs a genetic algorithm to guide the input generation and mutation based on the internal states in order to enhance the overall effectiveness. The most famous solutions are AFL [51] and its enhanced version AFL++ [19], which leverage the branch coverage collected during execution to guide the test process. LibAFL [20] was developed by the maintainers of AFL++, where developers can easily reuse state-of-the-art fuzzing components to increase overall efficiency and only need to implement the necessary ones that are required to achieve their objectives. We implemented HDHUNTER based on LibAFL, making it easy to extend and co-operate with existing fuzzing solutions. AFLNet [38] utilized the server's response code as feedback to guide the fuzzing process for interactive network protocols. NSFuzz [40] integrated static analysis to extract internal state variables as feedback. The remaining states of the network stack can interfere with the testing process. Coverage-directed differential testing has been widely applied to detecting logical defects in various protocols, such as mucert [9] for SSL/TLS and classfuzz [8] for JVM. Our work does not adopt their Markov chain Monte Carlo (MCMC) algorithm, but to reuse the mutual AFL framework with augmentations for HTTP Desync. NEZHA [37] optimized the coverage feedback process of differential testing by introducing δ-diversity that synthesized the path information to guide the seed selection process. However, it does not apply to network protocols, because the unstable cycle times and execution paths introduced by the network's asynchronous waiting and handling can produce different δ-diversity when executing the same input. The combined edge map used by HDHUNTER, with AFL's hit-count bucket division and accumulated edge coverage information, can provide immunity to such scenarios. In summary, the above work is targeting one single programming language. Witcher [47] first applied gray-box fuzzing to SQL and command injection vulnerabilities and introduced the coverage collection for interpreted languages by inserting code into the bytecode interpreter. By combining and enhancing Witcher and SanitizerCoverage [46], we broaden our test scope to HTTP implementations in more languages. However, existing tools are not tailored to identify discrepancies across two or more HTTP implementations. Consequently, their effectiveness in detecting HTTP Desync vulnerabilities, which are primarily caused by such discrepancies, is significantly limited.

8 Conclusion

In this paper, we expanded the scope of HTTP Desync to both HTTP requests and responses, and proposed a novel coverage-directed differential testing framework HDHUNTER, capable of automatically identifying HTTP discrepancies. We tested 19 state-of-the-art HTTP implementations using our HDHUNTER prototype. We highlighted 5 types of discovered discrepancies and validated their ability to trigger HTTP Desync, resulting in 4 types of attacks including Request Smuggling, Request Confusing, Response Stealing, and Response Forgery. We responsibly disclosed these vulnerabilities to relevant vendors. A total of 9 CVE IDs have been assigned.

Acknowledgement

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

Ethical Considerations

HDHUNTER targets open-source HTTP implementations. We set up all the testing environments on our local machine without interfering with the real-world servers and networks. Additionally, any identified vulnerabilities were privately reported through the project's security email or the GitHub Security Advisory in the official repository. No details regarding the vulnerabilities were disclosed to the public before the vulnerabilities were patched.

Open Science

We have opened the source of HDHUNTER on GitHub (https://github.com/mukeran/HDHunter) and Zenodo (https://zenodo.org/records/14557763).

References

[1] Meta AI. Introducing llama: A foundational, 65-billion-parameter large language model. https://ai.meta.com/blog/large-language-model-llama-meta-ai/, 2023.

[2] Director of Security Amit Klein and Inc. Research, Sanctum. Divide and conquer - http response splitting, web cache poisoning attacks, and related topics. https://repository.root-me.org/Exploitation%20-%20Web/EN%20-%20HTTP%20Response%20Splitting%20-%20Divide%20and%20Conquer.pdf, 2005.

[3] Apache. The apache tomcat connectors - ajp protocol reference. https://tomcat.apache.org/connectors-doc/ajp/ajpv13a.html, 2023.

[4] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. NAUTILUS: fishing for deep bugs with grammars. In *26th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, California, USA, 2019. The Internet Society.

[5] Jianjun Chen, Jian Jiang, Haixin Duan, Nicholas Weaver, Tao Wan, and Vern Paxson. Host of troubles: Multiple host ambiguities in http implementations. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1516–1527, 2016.

[6] Jianjun Chen, Vern Paxson, and Jian Jiang. Composition kills: A case study of email sender authentication. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2183–2199, 2020.

[7] Jianjun Chen, Xiaofeng Zheng, Hai-Xin Duan, Jinjin Liang, Jian Jiang, Kang Li, Tao Wan, and Vern Paxson. Forwarding-loop attacks in content delivery networks. In *NDSS*, 2016.

[8] Yuting Chen, Ting Su, Chengnian Sun, Zhendong Su, and Jianjun Zhao. Coverage-directed differential testing of JVM implementations. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2016)*, pages 85–99, Santa Barbara, CA, USA, 2016. ACM.

[9] Yuting Chen and Zhendong Su. Guided differential testing of certificate validation in SSL/TLS implementations. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*, pages 793–804, Bergamo, Italy, 2015. ACM.

[10] The MITRE Corporation. Cve - cve. https://cve.mitre.org/, 1999.

[11] d3d. From akamai to f5 to ntlm... with love. https://blog.malicious.group/from-akamai-to-f5-to-ntlm/, 2023.

[12] Google DeepMind. Gemini - google deepmind. https://deepmind.google/technologies/gemini, 2024.

[13] defparam. Smuggler. https://github.com/defparam/smuggler, 2020.

[14] Martin Doyhenard. Response smuggling: Exploiting http/1.1 connections. https://media.defcon.org/DEF%20CON%2029/DEF%20CON%2029%20presentations/Martin%20Doyhenard%20-%20Response%20Smuggling-%20Pwning%20HTTP-1.1%20Connections.pdf, 2021.

[15] Phillip J. Eby. Pep 333 – python web server gateway interface v1.0. https://peps.python.org/pep-0333/, 2003.

[16] Phillip J. Eby. Pep 3333 – python web server gateway interface v1.0.1. https://peps.python.org/pep-3333/, 2010.

[17] Roy T. Fielding, Mark Nottingham, and Julian Reschke. HTTP Semantics. RFC 9110, jun 2022.

[18] Roy T. Fielding, Mark Nottingham, and Julian Reschke. HTTP/1.1. RFC 9112, jun 2022.

[19] Andrea Fioraldi, Dominik Christian Maier, Heiko Eißfeldt, and Marc Heuse. AFL++ : Combining incremental steps of fuzzing research. In *14th USENIX Workshop on Offensive Technologies, (WOOT 2020)*. USENIX Association, 2020.

[20] Andrea Fioraldi, Dominik Christian Maier, Dongjia Zhang, and Davide Balzarotti. Libafl: A framework to build modular and reusable fuzzers. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS 2022)*, pages 1051–1065, Los Angeles, CA, USA, 2022. ACM.

[21] The Tcpdump Group. Tcpdump. https://www.tcpdump.org/, 1999.

[22] Run Guo, Jianjun Chen, Yihang Wang, Keran Mu, Baojun Liu, Xiang Li, Chao Zhang, Haixin Duan, and Jianping Wu. Temporal {CDN-Convex} lens: A {CDN-Assisted} practical pulsing {DDoS} attack. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 6185–6202, 2023.

[23] HackerOne. Internet bug bounty. https://hackerone.com/ibb, 2021.

[24] Bahruz Jabiyev, Steven Sprecher, Kaan Onarlioglu, and Engin Kirda. T-reqs: HTTP request smuggling with differential fuzzing. In *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 1805–1820, Virtual Event, Republic of Korea, 2021. ACM.

[25] Ben Kallus, Prashant Anantharaman, Michael Locasto, and Sean W. Smith. The http garden: Discovering parsing vulnerabilities in http/1.1 implementations by differential fuzzing of request streams, 2024.

[26] James Kettle. Http desync attacks: Request smuggling reborn. https://portswigger.net/research/http-desync-attacks-request-smuggling-reborn, 2019.

[27] James Kettle. Http desync attacks: what happened next. https://portswigger.net/research/http-desync-attacks-what-happened-next, 2019.

[28] James Kettle. Http/2: The sequel is always worse. https://portswigger.net/research/http2, 2021.

[29] Amit Klein. Http request smuggling in 2020–new variants, new defenses and new challenges, 2020.

[30] Yuejia Liang, Jianjun Chen, Run Guo, Kaiwen Shen, Hui Jiang, Man Hou, Yue Yu, and Haixin Duan. Internet's invisible enemy: Detecting and measuring web cache poisoning in the wild. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, pages 452–466, 2024.

[31] Ziyu Lin, Zhiwei Lin, Ximeng Liu, Jianjun Chen, Run Guo, Cheng Chen, and Shaodong Xiao. {CDN} cannon: Exploiting {CDN}{Back-to-Origin} strategies for amplification attacks. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 5717–5734, 2024.

[32] Chaim Linhart, Amit Klein, Ronen Heled, and Steve Orrin. Http request smuggling. https://www.cgisecurity.com/lib/HTTP-Request-Smuggling.pdf, 2005.

[33] LLVM. libfuzzer – a library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html, 2015.

[34] Inc. Open Market. Fastcgi a high-performance web server interface. https://fastcgi-archives.github.io/FastCGI_A_High-Performance_Web_Server_Interface_FastCGI.html, 1996.

[35] OpenAI. Chatgpt. https://chat.openai.com, 2022.

[36] Anshuman Pattnaik. Http request smuggling detection tool. https://github.com/anshumanpattnaik/http-request-smuggling, 2020.

[37] Theofilos Petsios, Adrian Tang, Salvatore J. Stolfo, Angelos D. Keromytis, and Suman Jana. NEZHA: efficient domain-independent differential testing. In *2017 IEEE Symposium on Security and Privacy (SP 2017)*, pages 615–632, San Jose, CA, USA, 2017. IEEE Computer Society.

[38] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. AFLNET: A greybox fuzzer for network protocols. In *13th IEEE International Conference on Software Testing, Validation and Verification (ICST 2020)*, pages 460–465, Porto, Portugal, 2020. IEEE.

[39] PortSwigger. Http request smuggler. https://github.com/PortSwigger/http-request-smuggler, 2018.

[40] Shisong Qin, Fan Hu, Zheyu Ma, Bodong Zhao, Tingting Yin, and Chao Zhang. Nsfuzz: Towards efficient and state-aware network service fuzzing. *ACM Trans. Softw. Eng. Methodol.*, 32(6):160:1–160:26, 2023.

[41] Rack. Rack specification. https://github.com/rack/rack/blob/main/SPEC.rdoc, 2008.

[42] Neil Schemenauer. scgi. https://github.com/nascheme/scgi, 2002.

[43] Sergej Schumilo, Cornelius Aschermann, Ali Abbasi, Simon Wörner, and Thorsten Holz. Nyx: Greybox hypervisor fuzzing using fast snapshots and affine types. In *30th USENIX Security Symposium (USENIX Security 2021)*, pages 2597–2614. USENIX Association, 2021.

[44] Sergej Schumilo, Cornelius Aschermann, Andrea Jemmett, Ali Abbasi, and Thorsten Holz. Nyx-net: network fuzzing with incremental snapshots. In *EuroSys '22: Seventeenth European Conference on Computer Systems*, pages 166–180, Rennes, France, 2022. ACM.

[45] Kaiwen Shen, Jianyu Lu, Yaru Yang, Jianjun Chen, Mingming Zhang, Haixin Duan, Jia Zhang, and Xiaofeng Zheng. Hdiff: A semi-automatic framework for discovering semantic gap attack in HTTP implementations. In *52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2022)*, pages 1–13, Baltimore, MD, USA, 2022. IEEE.

[46] The Clang Team. Sanitizercoverage. https://clang.llvm.org/docs/SanitizerCoverage.html, 2014.

[47] Erik Trickel, Fabio Pagani, Chang Zhu, Lukas Dresel, Giovanni Vigna, Christopher Kruegel, Ruoyu Wang, Tiffany Bao, Yan Shoshitaishvili, and Adam Doupé. Toss a fault to your witcher: Applying grey-box coverage-guided mutational fuzzing to detect SQL and command injection vulnerabilities. In *44th IEEE Symposium on Security and Privacy (SP 2023)*, pages 2658–2675, San Francisco, CA, USA, 2023. IEEE.

[48] uWSGI. The uwsgi protocol. https://uwsgi-docs.readthedocs.io/en/latest/Protocol.html, 2012.

[49] Enze Wang, Jianjun Chen, Wei Xie, Chuhan Wang, Yifei Gao, Zhenhua Wang, Haixin Duan, Yang Liu, and Baosheng Wang. Where urls become weapons: Automated discovery of ssrf vulnerabilities in web applications. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 216–216. IEEE Computer Society, 2024.

[50] Qi Wang, Jianjun Chen, Zheyu Jiang, Run Guo, Ximeng Liu, Chao Zhang, and Haixin 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)*, pages 132–132, Los Alamitos, CA, USA, may 2024. IEEE Computer Society.

[51] Michal Zalewski. American fuzzy lop, 2017.

[52] Cen Zhang, Yaowen Zheng, Mingqiang Bai, Yeting Li, Wei Ma, Xiaofei Xie, Yuekang Li, Limin Sun, and Yang Liu. How effective are they? exploring large language model based fuzz driver generation. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1223–1235, 2024.

[53] Jiahe Zhang, Jianjun Chen, Qi Wang, Hangyu Zhang, Chuhan Wang, Jianwei Zhuge, and Haixin Duan. Inbox invasion: Exploiting mime ambiguities to evade email attachment detectors. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, pages 467–481, 2024.

[54] Linkai Zheng, Xiang Li, Chuhan Wang, Run Guo, Haixin Duan, Jianjun Chen, Chao Zhang, and Kaiwen Shen. Reqsminer: Automated discovery of cdn forwarding request inconsistencies with differential fuzzing. In *NDSS*, 2024.

A HTTP Grammar

HDHUNTER uses the ABNF rules in Listing 1, which are manually extracted from RFC, to perform input mutations. The non-terminals without a rule are regarded as data fields, i.e. `*OCTET`.

Listing 1: ABNF Rules Used to Build the Structure of HTTP Messages

```
HTTP-message     = start-line *( field-line CRLF )
    ↪ CRLF [message-body]
start-line       = request-line / status-line
request-line     = method SP request-target SP HTTP-
    ↪ version CRLF
status-line      = HTTP-version SP status-code SP [
    ↪ reason-phrase] CRLF
field-line       = field-name ":" OWS field-value OWS
    ↪ CRLF
message-body     = chunked-body / *OCTET
chunked-body     = *chunk last-chunk trailer-section
    ↪ CRLF
chunk            = chunk-size [ chunk-ext ] CRLF chunk
    ↪ -data CRLF
chunk-size       = 1*HEXDIG
last-chunk       = 1*("0") [ chunk-ext ] CRLF
trailer-section  = *field-line

SP               = %x20
HTAB             = %x09
OWS              = *( SP / HTAB )
CRLF             = %x0D %x0A
HEXDIG           = DIGIT / "A" / "B" / "C" / "D" / "E"
    ↪ / "F"
```

B Detailed Mutation Strategies

Table 3: Mutation Strategies of HDHUNTER

Level	Strategy
Sequence*	1) Randomly select a message from corpus and add to the sequence 2) Delete a message from the sequence
Message*	3) Duplicate a field line 4) Delete a field line 5) Randomly select field lines from corpus and insert them at a random position 6) Randomly swap two values of the same type 7) Randomly replace a value with one random preset token of the same type 8) Randomly select field lines from corpus and replace the trailer section (chunked only)
Byte	9) Randomly insert bytes 10) Randomly remove bytes 11) Randomly duplicate bytes 12) Randomly select bytes from corpus and insert them at a random position

* New mutation strategies introduced in HDHUNTER.

The detailed mutation strategies used by HDHUNTER are presented in Table 3.

C Detection Rules

Algorithm 1 is the logic of our HTTP Desync Detector.

D Code Insertion

D.1 Statistics

Table 4 shows the number of lines, functions, and files we modified in 5 representative targets' request handling procedures, covering 4 categories and 5 different programming languages.

Algorithm 1 HTTP Desync Detection Process: Determine whether there is an HTTP Desync, using State Tuples S_A, S_B of implementation A and B

```
procedure DESYNCDETECTION(S_A,S_B)
    if COUNT(S_A) ≠ COUNT(S_B) then
        return true
    for i ← 0 to COUNT(S_A) do
        if ORDER(S_A,i) ≠ ORDER(S_B,i) then
            return true
        if BODY(S_A,i) ≠ BODY(S_B,i) then
            return true
        if ISERROR(STATUS(S_A,i)) and ISERROR(STATUS(S_B,i)) then
            continue
        if ENCODING(S_A,i) ≠ ENCODING(S_B,i) then
            return true
        if CL(S_A,i) ≠ CL(S_B,i) then
            return true
        if CONSUMED(S_A,i) ≠ CONSUMED(S_B,i) then
            return true
    return false
```

Table 4: Statistic of Inserted Code in 5 Representative Targets

Target	Line†	Func	File	Hour	Category	Lang.
Apache	19	4	4	2–3	Intergrated Server	C
Squid	17	7	3	2–3	Cache Server	C++
gevent	25	5	1	1–2	Network Framework	Python
Tomcat	11	6	5	1–2	Application Server	Java
Falcon	64	6	5	1–2	Application Server	Ruby

† Comments and empty lines are not included.

D.2 Example

The *apache.diff* file in the repository refers to the diff of the code we insert into Apache. This code extracts the first five types of states for the HTTP requests handling process. `ap_process_http_async_connection` is Apache's HTTP handling function. After calling `ap_read_request`, Apache has parsed the request, and we can read and set *Encoding* and *CL*. *Count* is incremented after the processing of the request. `ap_http_filter` is responsible for parsing the HTTP body. We locate the positions where Apache reads the body, including the chunk size, content, and body content, and insert code to maintain *Consumed* and *Body*. Apache reuses the code to read raw body and chunked body. `ap_rgetline_core` is called when reading the start line and field lines.

E Detailed Discrepancies

A substantial number of discrepancies were identified during the experiment. For instance, in the Apache–Tomcat pair, approximately 13500 discrepancies were reported during each run. After the preliminary deduplication, around 500 discrepancies remained.

Figure 9 details 5 primary types of discrepancies found by HDHUNTER.

Table 5: Different Types of Non-standard Number Parsing

Type	Example	Affected
0x prefix	`Content-Length: 0x8`	Falcon

	`Transfer-Encoding: chunked`
	`0x8`
	`abcdefgh`
		Falcon
		Squid
		Eventlet
		Tornado

+ prefix	`Content-Length: +8`	Jetty Eventlet

	`Transfer-Encoding: chunked`
	`+8`
	`abcdefgh`
		Falcon
		Eventlet

_ between	`Content-Length: 1_0`	Falcon Tornado

	`Transfer-Encoding: chunked`
	`1_0`
	`abcdefgh`	

Any suffixes	`Transfer-Encoding: chunked`
	`8irrelavent_characters`
	`abcdefgh`
		H2O
		ATS

F Different Types of Non-standard Number Parsing

Non-standard number parsing is one of the primary discrepancies between HTTP implementations discovered by HDHUNTER. We summarize their different behaviors in Table 5.

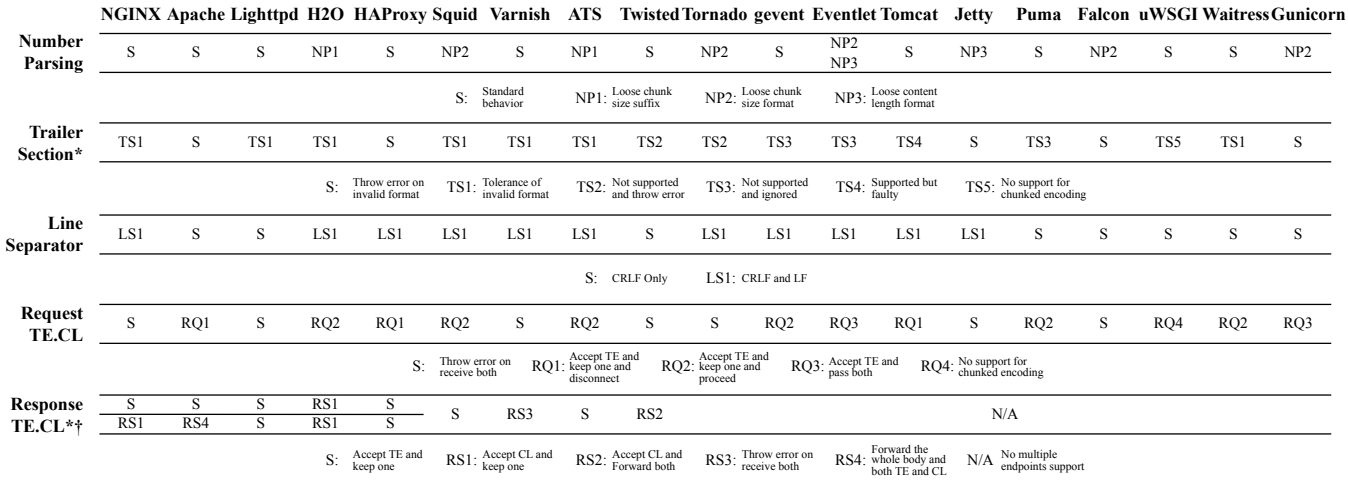
G HTTP Payloads

G.1 Payloads as Artifact

To facilitate future research, the collection of the vulnerable payloads has been uploaded to the repository.

G.2 Interesting Payloads

In this section, we share two interesting payloads found by HDHUNTER that can be leveraged to perform HTTP Desync Attacks.



* Novel types of discrepancies identified by HDHUNTER.
† The top and bottom halves of the five left-hand tiles of Response TE.CL represent discrepancies in HTTP responses and CGI responses respectively.

Figure 9: Five Primary Types of Discrepancies Between Implementations We Discovered

The first one (Listing 2) is an exploitable HTTP request smuggling payload in the tested version of Tomcat. Tomcat will interpret this payload as two requests — with lines 1–15 as the first and lines 17–19 as the second. The other HTTP implementations, such as ATS, will interpret this payload differently — with lines 1–10 as the first and lines 12–19 as the second. If a setup uses ATS as the proxy and Tomcat as the server, the malicious payload embedded in the second request can be executed on Tomcat.

Listing 2: Payload that Leads to Discrepancy in Tomcat

```
1  POST /benign_path HTTP/1.1
2  Host: a.com
3  Connection: keep-alive
4  Transfer-Encoding: chunked
5
6  5
7  12345
8  0
9  Content: hello
10 a
11
12 POST /benign_path HTTP/1.1
13 Host: a.com
14 Connection: keep-alive
15 Content-Length: 37
16
17 GET /evil_path HTTP/1.1
18 Any: any
19 Host: b.com
```

The second one (Listing 3) is a payload that can lead to discrepancy by non-standard line separators. Assume implementation A only allows CRLF and accepts LF in the chunk extension, and implementation B allows both LF and CRLF. A will interpret lines 6–7 as the first chunk, lines 8–9 as the second chunk, and lines 12–15 as the second request. B will interpret line 6 with the left 8 characters as the first chunk, line 8 as the trailer section, lines 9–11 as the second request, and drop lines 12–15 since the connection is set to *close*. The discrepancy is caused by the confused boundary of the first chunk.

Listing 3: Payload that Leads to Discrepancy by Non-standard Line Separator

```
1  POST /proxy HTTP/1.1[CR][LF]
2  Host: a.com[CR][LF]
3  Transfer-Encoding: chunked[CR][LF]
4  Connection: keep-alive[CR][LF]
5  [CR][LF]
6  0a;[LF][CR][LF]
7  12345678[LF]0[CR][LF]
8  4b;:123[CR][LF]
9  [CR][LF]POST /proxy HTTP/1.1[CR][LF]Host: b.com[CR]
   [LF]Connection: close[CR][LF]Content-Length: 5
   [CR][LF][CR][LF]
10 0[CR][LF]
11 [CR][LF]
12 GET /proxy HTTP/1.1[CR][LF]
13 Host: a.com[CR][LF]
14 Connection: close[CR][LF]
15 [CR][LF]
```

H Impact of Initial Seeds

Initial seeds play a vital role in the fuzzing process. In order to evaluate the impact of the initial seeds, a series of experiments were conducted against the Apache–NGINX combination. The experiments involved the use of 20 selected, 10 selected, 20 random, and 20 selected seeds, respectively, with the last environment using AFL's test case generation,

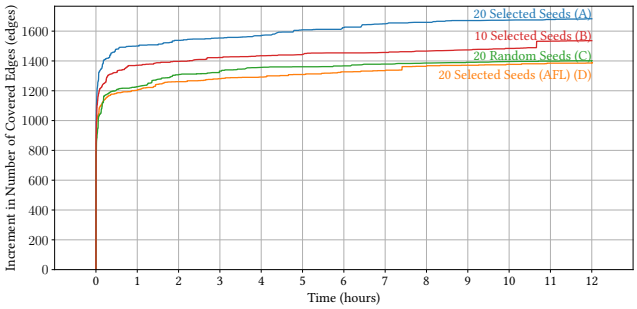


Figure 10: Comparison of Covered Edges over Time when Testing Apache–NGINX Using Four Different Sets of Initial Seeds

denoted as A, B, C, D. To be specific, D shares the same set of initial seeds with A. B is a subset of A which has lower diversity. C is selected randomly from the network flow. The coverage growth of these setups is illustrated in Figure 10. The experiment results demonstrate that the size of the initial seeds is not a contributing factor to the outcome. Instead, a positive correlation exists between the diversity of the initial seeds and coverage.