Gudifu: Guided Differential Fuzzing for HTTP Request Parsing Discrepancies

Bahruz Jabiyev
Dartmouth College
Hanover, NH, USA

Anthony Gavazzi
Northeastern University
Boston, MA, USA

Kaan Onarlioglu
Akamai Technologies
Cambridge, MA, USA

Engin Kirda
Northeastern University
Boston, MA, USA

ABSTRACT

Modern web applications involve multiple HTTP processors on the traffic path, each acting as a reverse proxy and processing client requests. Even when such proxies are secure in isolation, when combined into complex systems, minor HTTP parsing discrepancies between them can lead to various severe attacks such as cache poisoning and HTTP request smuggling attacks.

We propose Gudifu, a new approach that improves the state-of-the-art HTTP differential fuzzing approaches in two main ways: 1) taking a graybox fuzzing approach to probe the parsing behavior of HTTP proxies and 2) using a new algorithm which is capable of searching for discrepancies in the entire HTTP request. These improvements lead to the discovery of significantly more parsing discrepancies and discrepancy-based attack vectors which were previously unknown.

CCS CONCEPTS

• **Security and privacy → Web application security**.

KEYWORDS

HTTP Parsing Discrepancies, Guided Differential Fuzzing, HTTP Server Chain Attacks

1 INTRODUCTION

The classic client-server model no longer accurately represents practical web application deployments. Today, origin servers are supported by many intermediate services such as web caches, load balancers, security products, and API gateways, which all act as reverse proxies and process traffic at the application layer. In other words, a single request generated by a user agent is often parsed, processed, and transformed by a plethora of HTTP proxies.

This design is key to highly distributed and scalable Internet infrastructures. However, it also adds complexity, and complex systems are inherently hazardous [10]. Although this is a well-understood concern in the safety engineering literature [30], the security community is only recently flooded with new classes of attacks exacerbated by this complexity. These attacks are not caused by faulty components in the system, but they arise due to *hazardous interactions* between multiple components that may otherwise perform to specification.

In particular, discrepancies in how different HTTP proxies on the traffic path process a given request are shown to be a major factor, leading to a steady stream of novel *web cache poisoning* and *request smuggling* attacks (e.g., [8, 12, 25, 28, 29]). We collectively refer to these as discrepancy attacks. Discrepancy attacks are practical, and were weaponized for exploiting high-profile targets; e.g., to hijack HTTP requests on Slack and steal cookies [11], to poison PayPal's web cache and serve malicious JavaScript [26], and to steal HTTP responses destined for arbitrary users of Atlassian Jira [27].

While academic research into this domain is sparse, two recent works explored methods for identifying novel discrepancy attack vectors through differential analysis of HTTP processors [24, 38]. Both works involve a similar fuzzing approach: 1) fuzzing multiple proxies with mutated HTTP requests, 2) capturing the requests processed and forwarded by each proxy, and 3) comparing the forwarded requests to identify discrepancies.

The fuzzing methods of past research are performed in a blackbox manner, relying on static grammars for input generation. In other words, they have no visibility into how each input performs, and therefore, their ability to exercise the target HTTP processors and induce unusual behavior is fundamentally limited.

In addition, past works tailor their discrepancy detection strategies to a narrowly defined set of attack types, targeting specific parts of HTTP requests. For example, as an indicator of HTTP request smuggling vulnerabilities, prior work only searches the request body in the forwarded requests for discrepancies. That leads to missing out on opportunities to detect new attack vectors which may arise from discrepancies in request components that have received less attention from the security community.

In light of these limitations of prior work, we formulate the following research questions:

- (Q1) Does a guided, graybox fuzzing approach enable more effective discovery of HTTP parsing discrepancies compared to a blackbox approach?
- (Q2) Can we search for discrepancies holistically, regardless of what part of the request they arise at?
- (Q3) Does a graybox and holistic approach help identify novel and exploitable discrepancy attacks that impact ubiquitous proxy technologies?

To investigate these questions, we propose Gudifu, a graybox differential fuzzing approach, which uses a code coverage metric to guide input selection and mutation. We present an implementation of Gudifu, and use it to test for novel discrepancy attacks between six popular reverse proxies: Apache httpd, NGINX, H2O, ATS, HAProxy, and Envoy.

We empirically evaluate our approach against the aforementioned T-Reqs fuzzer of prior work [24], and show that Gudifu finds significantly more discrepancies in our experiment setup.

Finally, we craft practical attacks based on Gudifu's novel findings. We demonstrate that these discrepancies have real-world impact via access control bypasses, cache poisoning and HTTP request smuggling.

We summarize again the contributions of our work below.

- We develop Gudifu for graybox differential fuzzing of HTTP proxies.
- We propose a novel holistic search method for discrepancies in requests, which identifies attack vectors missed by prior work.
- We demonstrate that our novel approach is more effective in finding discrepancies than existing blackbox approaches.
- We demonstrate with concrete exploits that discrepancy attacks have dire security implications.

Availability. Gudifu is open source. The code can be viewed at `https://github.com/bahruzjabiyev/gudifu-fuzzer`.

Coordinated Disclosure. We notified the tested proxy vendors by providing them a copy of the paper. The discussion of vendor responses is at the end of the paper.

2 BACKGROUND AND RELATED WORK

Here, we give an overview of the relevant fuzzing techniques and the past research on HTTP discrepancy attacks – attacks which exploit the discrepancies in servers' parsing behavior.

Fuzzing is the process of generating numerous inputs with various forms and contents in order to exercise as much of a target program as possible. Fuzzing techniques are typically categorized as either blackbox or graybox. In blackbox fuzzing, inputs are generated independent from their impact on the target program, and are often generated from a static corpus or grammar. In graybox fuzzing, each executed input is assigned a success value, where the higher the success value of an input is, the higher the chance that input has to be used again for fuzzing. By giving priority to successful inputs, graybox fuzzing guides the fuzzing process toward more successes. The success metric can be, for example, the code coverage [31, 39] (e.g., the number of code blocks visited by the input) or the coverage of the program's internal state space [35, 40] (e.g., the number of program states exercised by the input).

A relatively recent research trend has been to use graybox fuzzing to find vulnerabilities in network services. Pham et al. developed AFLNet [35], a fuzzer which is guided by both code coverage and the state space of the target network service. Nyx-Net [37] and SnapFuzz [3] aimed to address the limitations of the AFLNet, such as the low throughput, the need for the hard-coded timeout values and cleanup scripts for resetting the fuzzing environment.

Differential fuzzing techniques have also seen an increase in their adoption by security researchers. The main idea of differential fuzzing is to find bugs through comparing the behavior of programs of the same type. In other words, a large amount of inputs are fed into similar programs, say, two different C code compilers, and their reaction to the same inputs are examined for a difference which might signal a bug.

Bernhard et al. [6] use a differential fuzzing technique to find logic bugs in the Javascript engines, while the Reen and Rossow [36] developed DPIFuzz to find techniques for evading deep packet inspection. Petsios et al.. [34] developed a generic differential testing tool called NEZHA, and used it to find semantic bugs and discrepancies – some with critical security implications – across a wide variety of applications.

Past research has also used differential fuzzing to find HTTP parsing discrepancies. Jabiyev et al. [24] developed T-Reqs, and used it to search for parsing discrepancies in HTTP request bodies which can lead to request smuggling. Shen et al. [38] took a broader look at parsing discrepancies and developed HDiff to search for discrepancies which can lead to `Host` header confusion and cache poisoning in addition to request smuggling.

HTTP parsing discrepancies have also been examined by Chen et al. [8] and Nguyen et al. [32] in the context of cache poisoning and `Host` confusion attacks. While they do not develop an automated technique to find these discrepancies, they demonstrate that the few discrepancies they find can have serious security implications.

3 APPROACH

In this section, we describe our approach for the guided differential fuzzing of HTTP servers, which we call the Gudifu approach. Figure 1 shows the data flow diagram of this approach.

The data flow starts with a single input corpus populated with a set of HTTP requests. A number of fuzzer instances, one for each target server, share this input corpus and read inputs from it before mutating them and delivering them to their respective target servers.

The target servers receive the inputs and process them, possibly returning an error message to the fuzzer instance, and possibly forwarding a message to their respective echo server. Regardless, the target servers are instrumented to report the code coverage achieved by processing the input back to the fuzzer instance in order to influence future input selection and mutation.

The echo servers receive the forwarded requests from the target servers and log them to a single shared database for offline processing. They then send a response back to the target server, which sends it back to the fuzzer instance, which then can decide whether to add the mutated test case back to the input corpus for other fuzzer instances to draw from and mutate in future fuzzing iterations.

3.1 Shared Input Corpus

The input corpus is a collection of HTTP requests which serve as the pool from which fuzzer instances continuously read and write inputs. Prior to fuzzing, it is populated with a large set of distinct and valid HTTP requests. This set contains all possible combinations of standard HTTP methods (e.g., "GET", "OPTIONS") and standard HTTP headers (e.g., "Expect", "Referer") with valid values compiled from HTTP RFC specifications.

Every fuzzer instance loads the entire input corpus when fuzzing first starts and then again every second to check for new inputs. As such, the input corpus serves as the source of input sharing among fuzzer instances to ensure that the same inputs are delivered to every target server.

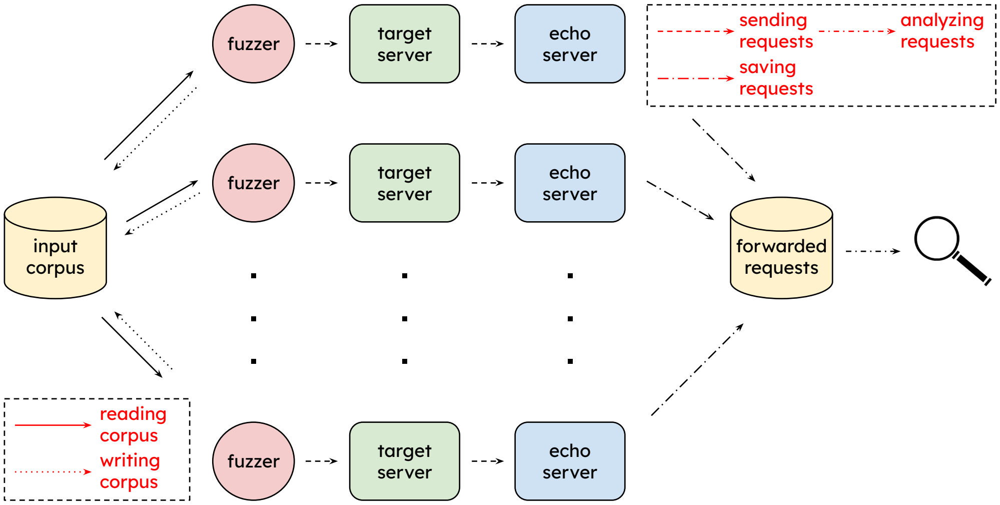

Figure 1: Dataflow diagram of GUDIFU

The number and the validity of the requests in the corpus is important, as we need a rich body of common inputs which will be forwarded by the target servers based on which we can detect parsing discrepancies. In the following subsections, we explain the steps we take to achieve the desired qualities of the input corpus.

3.2 Fuzzer Instances

A number of fuzzer instances share the single input corpus. At a high level, the responsibility of each fuzzer instance is to drive the fuzzing process for a single target server. Each fuzzing iteration, the first action a fuzzer instance takes is to select one input from the input corpus. This decision is not random, and inputs are both chosen and processed based on an internal *search strategy* that will prioritize inputs based on certain key factors.

The search strategy works as follows. First, it will select any inputs in the corpus that have not yet been delivered to the target server. These inputs are not mutated, and are sent *as is* to the target server. Thus, when the fuzzer first starts up, it loads the initial shared input corpus and delivers each input directly to the target server. This ensures that every target server receives the same inputs from the corpus, and enables later analysis of how different servers processed the request. Additionally, the fuzzer instance reloads the shared corpus once every second, and as other fuzzer instances add inputs to the corpus, these inputs will be prioritized and sent as is to the target server.

If every input in the corpus has been sent as is to the target server, then the search strategy selects an input such that a higher probability is assigned to inputs that exercised new code paths in the target server (or a random input is selected if such information is not yet available). The fuzzer instance then mutates the input according to a set of predefined mutation operators. These operators include transformations such as flipping the value of a single bit, inserting random bytes, or inserting a value pulled from a dictionary of meaningful HTTP keywords.

The fuzzer instance then delivers the input to the target server by acting as a network client to it, connecting over a socket to the port the target server is listening on and sending the input in full. While the target server is started only once, the fuzzer instance starts a new connection to deliver an input in each iteration. It then waits for a response from the target server. After receiving a response, the fuzzer instance uses two key pieces of information to guide future input selection.

First, the fuzzer instance checks the HTTP status code of the response from the server, and will consider adding the mutated input to the input corpus only if the server returned a `200 OK` (Because it was by far the most common non-error status code. But one can also use redirection status codes (e.g., 301, 302) in addition to 200). Without this check, we found that inputs which exercised new code paths in the error-handling portions of each target server were being prioritized, leading to fewer and fewer requests being forwarded by the target servers over time. Without the servers forwarding requests, there is less data to detect parsing discrepancies in, and thus we only add requests back to the corpus if they return a `200 OK` in order to keep the fuzzing process productive.

The second piece of information used by the fuzzer instance to guide future input selection is the code coverage achieved on the target server when it processed the current input. If the current input exercised code that had not yet been exercised, and if the target server returned a `200 OK`, then the mutated request is added back to the input corpus and assigned a higher priority in the search strategy in future iterations. By prioritizing these inputs, we drive the fuzzer to exercise more and more of the target server's request-parsing code, and hope that by doing so, we will exercise portions of the code that blackbox fuzzing approaches struggle to reach, and can effect more parsing discrepancies as a result.

3.3 Target Servers

The principal systems being fuzzed are the target servers. Each target server's responsibility is to listen on a specific port, forward requests to its own echo server, return the echo server's response back to the fuzzer instance, and report the code coverage achieved from processing the request to the fuzzer instance. The target servers are built from source and instrumented at compile time to report code coverage to a shared memory buffer for the fuzzer instance to read from.

3.4 Echo Servers

Upstream from the target servers are a set of echo servers, one for each target server, whose responsibilities are to listen on a specific port, log every request from the target server to a single shared database of all forwarded requests, and respond with a `200 OK` to the target server.

3.5 Request Database and Search Method

The request database is a database of forwarded requests shared by all echo servers. As soon as a new forwarded request is captured by each echo server, it instantly writes it to this common database.

To search for parsing discrepancies, we analyze the forwarded requests saved to the database. Prior research compared only those parts of forwarded requests which they thought were the most relevant for specific attack types. Whereas, we use a holistic search methodology to capture parsing discrepancies regardless of which part of the request they arise at.

Our method comprises three stages. In the first stage, for every captured forwarded request, we get the list of changes made by the server on the input request. For example, if an input request contains an extra space before the method name, and the forwarded request does not have that extra space, then the removal of the space is a change made by the server. We developed an algorithm to obtain this list of changes, Python code for which is shown in Listing 1.

In the second stage, we look at the changes made by all servers for a given input request, and discard any changes which every server makes, as they are not suitable for comparison. We repeat this for every input request and obtain 1) the list of changes made to it, and 2) for each change, the list of the servers which made them.

In the third stage, we bucketize the changes based on the set of servers which made them. For example, if changes C1 and C3 are made by only servers S1 and S3, then C1 and C3 go into the bucket of [S1, S3]. Once we populate each bucket with the changes belonging to it, we manually examine starting from the least populated bucket.

In the manual examination phase, we pay little or no attention to those which are trivial changes (e.g., a space is added after colon in `host:example.com` and it becomes `host: example.com`). When we see the changes which are unusual and non-trivial, we confirm them by checking the behavior one more time by sending the relevant requests to the servers and take a note of them once they are confirmed.

4 EXPERIMENTATION

In the experiment with the Gudifu approach, we focus on the HTTP/1 parsing discrepancies of the servers. The general information about the servers which were instrumented and fuzzed is listed in Table 1. We use libFuzzer [31] as the fuzzing engine.

```python
# Changes to input by the forwarded request
def changes(r, f): # r: input, f: forwarded

  # addition, modification, deletion lists
  adds, mods, dels = [], [], []

  for header in headers(f):
    if header in headers(r):
      continue
    if lineno(header) == 1: # request line
      mods.add(headers(r)[0], header)
    if ':' not in line:
      adds.add(header)
    else:
      name, value = header.split(':')
      # get input headers with name 'name'
      named_headers = headersnamed(r, name)
      if len(named_headers) == 0:
        adds.add(header)
      if len(named_headers) == 1:
        mods.add(named_headers[0], header)
      else:
        mods.add(join(named_headers), header)

  for header in headers(r):
    if header in headers(f):
      continue
    if header not in mods:
      dels.add(header)

  if input_body != forwarded_body:
    mods.add(input_body, forwarded_body)

  return adds, mods, dels
```

Listing 1: Algorithm for getting the changes on the input request (in Python).

Table 1: Names, versions, and source languages of tested proxies.

Server Name	Version	Source
Apache httpd	2.4.54	C
NGINX	1.22.1	C
H2O	2.2.6	C
Apache Traffic Server (ATS)	10.0.0	C++
HAProxy	2.7.1	C
Envoy	1.24.1	C++

Table 2 gives details about the data generated during the experiment. "Valid" requests are those which were forwarded by at least one server. The number of requests forwarded by each server is also listed. The number of commonly forwarded requests (i.e., the input requests which are forwarded by at least two different servers) is relatively small. The reason is, the inputs are shared through the input corpus and only successful inputs (i.e., the ones which cover new code blocks) are added to the corpus.

4.1 Configuring Servers

When we configure the target servers for the experiment, we have the least possible number of directives which allow the servers to receive requests at a certain port and forward them to where their own echo server is running. Only for NGINX and H2O, we enabled the preservation of the `Host` header value in order to get visibility into how this header is parsed by these servers (by default these servers overwrite the incoming `Host` header with the IP and port of the host they forward to).

Usually, servers do not have configuration directives to let users control their parsing behavior (with few exceptions like Envoy allowing users to merge slashes in the request URI [14]). Because the parsing behavior of servers is usually shaped by the way the developers implement them under the guidance of the specifications and it is not high-level enough for letting users change it.

We expect that the target servers' default parsing behavior is the most secure parsing behavior as they would not allow an insecure behavior to remain as the default. Also, given that testing all possible configurations for each target is a hard task if not impossible, we run each server with its default configuration.

4.2 Capturing Cacheable Responses

In addition to saving the input requests and forwarded requests to the filesystem for a later analysis, we also saved requests which generate a cacheable error status code. We compile the list of those status codes by combining the unsuccessful cacheable codes defined by the HTTP specification and those which are commonly cached by CDN servers in practice. The list consists of the following status codes: `300 Multiple Choices`, `301 Moved Permanently`, `302 Moved Temporarily`, `404 Not Found`, `405 Method Not Allowed`, `410 Gone`, `414 URI Too Long`, `501 Not Implemented`.

Past research [32] has shown the security implications of parsing discrepancy between two servers where one of them forwards a request and the other one responds with a cacheable error status code. Therefore, in this work, we also want to look at this type of parsing discrepancy.

5 PARSING DISCREPANCIES

We use the search methodology (described in Section 3) on the forwarded requests collected from the experiment to find the parsing discrepancies. We discuss the parsing discrepancies under four categories in the given order: request line discrepancies, request headers discrepancies, request body discrepancies and behavior discrepancies related to cacheable responses.

Table 2: Experiment overview.

Experiment duration	12 hours
# of Valid requests	6,737,538
# of Apache forwarded requests	1,062,694
# of NGINX forwarded requests	939,348
# of H2O forwarded requests	739,600
# of ATS forwarded requests	939,348
# of HAProxy forwarded requests	856,792
# of Envoy forwarded requests	903,666
# of Commonly forwarded requests	45,633

5.1 Request Line Discrepancies

We further divide request line discrepancies into three classes. The first one is about the parsing of absolute URIs. The second one looks at the treatment of reserved characters. Finally, the third one consists of the discrepancies caused by normalization. The examples for them can be found in Table 3. Note that the `HTTP/1.1` version token is shortened to `H/1` in the table for the sake of brevity.

5.1.1 Absolute URI. As shown in the first row of Table 3, servers exhibit discrepancies in their parsing of URI when it is in the absolute form. HTTP RFC 2616 [16] states that if the URI is in the absolute form, then the host is what is given in the URI and the `Host` header should be ignored. Apache httpd, NGINX, ATS and Envoy convert the URI into the relative form and add the uri-host in the `Host` header. Whereas, H2O adds a slash in front of the URI which changes the semantics, and HAProxy throws an error as it requires the `Host` header value and the uri-host in the URI to be the same.

When the absolute URI has an empty path (example in the second row) and the `Host` header is missing, servers parse it differently. According to the URI RFC 3986 [5], when the authority component is present in the URI, then the path can be empty. As for the `Host` header, RFC 7230 [17] states that it is required, and a request that lacks the header must be responded with `400 Bad Request`. As shown in the second row, Apache httpd and NGINX throw an error, H2O adds a slash in front and make the `Host` header value `default`, ATS and Envoy generate `Host` header value based on the uri-path and HAProxy keep the same URI and does not add a `Host` header.

As can be seen in the third row, when the URI consists of just the scheme name, most servers see it as an error while few of them see it valid. NGINX, HAProxy and Envoy respond with `400 Bad Request` and ATS closes the connection as a result of the error. Whereas, Apache httpd converts it into a slash and H2O adds a slash before the URI while they retain the `Host` value.

5.1.2 Reserved Characters. Reserved characters mainly serve as delimiters for the URI sections (e.g., ? signals the start of the query part). As we see in Table 3, they can easily trigger parsing discrepancies in the parsing behavior of servers.

For instance, as shown in the fourth row, when the URI is just a question mark (which is for starting the URI query section), while NGINX, HAProxy and Envoy return an error, Apache httpd, H2O and ATS forwards the request. Apache httpd converts into a slash, H2O adds a slash before the question mark and finally ATS removes the whole URI in the requests they forward.

Whereas, as shown in the fifth row, when the URI is just an at-sign (which is for separating the user information from the host), Apache httpd, NGINX and Envoy throw an error while the rest forward the request. H2O and ATS replaces the URI with a slash, whereas HAProxy retains the at-sign as the whole URI.

Semicolon (";") is reserved because it can be used to separate the URI parameter names and values as stated in Section 3.3 of RFC 3986 [5]. As seen in the fifth row when the path is just a

Table 3: Examples of parsing discrepancies in request lines.

Input	Apache httpd	NGINX	H2O	ATS	HAProxy	Envoy

`GET http://a/b H/1`
`Host: c`
	`GET /b H/1`
	`Host: a`
		Same as
		Apache httpd
			`GET /http://a/b H/1`
			`Host: c`
				Same as
				Apache httpd
					Bad Request
					Error
						Same as
						Apache httpd

`GET http://a H/1`
~~`Host: c`~~
	Bad Request
	Error
		Bad Request
		Error
			`GET /http://a H/1`
			`Host: default`
				`GET / H/1`
				`Host: a`
					`GET http://a H/1`
					~~`Host: c`~~
						Same as
						ATS

`GET http: H/1`
`Host: c`
	`GET / H/1`
	`Host: c`
		Bad Request
		Error
			`GET /http:`
			`Host: c`
				Connection
				Reset
					Bad Request
					Error
						Bad Request
						Error

`GET ? H/1`	`GET / H/1`	Bad Request Error	`GET /? H/1`	`GET` ~~`?`~~ `H/1`	Bad Request Error	Bad Request Error
`GET @ H/1`	Bad Request Error	Bad Request Error	`GET / H/1`	`GET / H/1`	`GET @ H/1`	Bad Request Error
`GET /b; H/1`	`GET /b; H/1`	`GET /b; H/1`	`GET /b; H/1`	`GET /b`~~`;`~~ `H/1`	`GET /b; H/1`	`GET /b; H/1`
`GET /b#c H/1`	Bad Request Error	`GET /b`~~`#c`~~ `H/1`	`GET /b#c H/1`	`GET /b`~~`#c`~~ `H/1`	`GET /b#c H/1`	Bad Request Error
`GET /%61 H/1`	`GET /a H/1`	`GET /a H/1`	`GET /%61 H/1`	`GET /%61 H/1`	`GET /%61 H/1`	`GET /%61 H/1`
`GET /// H/1`	`GET /`~~`//`~~ `H/1`	`GET /`~~`//`~~ `H/1`	`GET /// H/1`	`GET //`~~`/`~~ `H/1`	`GET /// H/1`	`GET /// H/1`
`GET /a/.. H/1`	`GET / H/1`	`GET / H/1`	`GET /a/.. H/1`	`GET /a/.. H/1`	`GET /a/.. H/1`	`GET /a/.. H/1`

semicolon, it is also treated differently. ATS removes it in the URI before forwarding. Whereas, all other servers keep it.

Number sign ("#") is also a special character as it serves as the start of the fragment section. When servers receive a URI with a fragment, they respond differently. Apache httpd and Envoy respond with an error. NGINX and ATS drops the fragment before forwarding the request. Finally, H2O and HAProxy preserve the fragment section.

5.1.3 Normalization. Finally, servers have discrepancies in the way they normalize request URIs. Also, while one server normalizes the URI before forwarding, the other one might prefer not to normalize.

For instance, as shown in the eighth row, when URI has the non-reserved characters percent-encoded (%61 is percent-encoded version of the ASCII character "a"), the servers act differently. Percent-encoding is a means to safely transfer reserved characters if they are part of the URI content. While Apache httpd and NGINX decode them before they forwarded the request, all other servers forward it without decoding.

The URI path is not allowed by RFC 3986 to start with multiple slashes as it is stated in the "Path" section of the document. Again, servers parse request URIs which start with multiple slashes differently. Apache httpd and NGINX trim all the extra slashes and just keep one. ATS removes only one slash and leaves the rest. Finally, H2O, HAProxy and Envoy forward them as is.

According to RFC 3986, the ".." and "." characters have a similar role to their role within operating systems and they are intended for use at the beginning of relative paths. When servers receive a request URI containing these characters, some of them normalize the URI, some not. Apache httpd and NGINX normalize the request URI. Whereas, all other servers forward them with no change.

5.2 Request Header Discrepancies

When it comes to the parsing of request headers, the parsing of servers differ widely again. For example, as seen in the first row of Table 4, when a header with an empty name is present in a request, while most servers report an error, some accept it. Apache httpd, NGINX, H2O and Envoy return the `400 Bad Request` response. ATS still forwards the request after dropping the header with the empty name. Whereas, HAProxy drops the header which comes after the empty-named header even if the following header is `Content-Length` and the request has a body. As a result, a request which has a body, yet no body length header, is still forwarded.

According to RFC 7230, each header field contains a colon (":") as a means to separate the header name and header value. When the headers block of a request has a line without a colon, some servers respond with an error, some accept it. While Apache httpd, H2O, HAProxy and Envoy respond with `400 Bad Request`, ATS forwards the request after dropping the line which does not have a colon. Whereas, NGINX adds a colon (and makes it a header with empty value) to this line and forwards to the upstream.

Another parsing discrepancy is observed when a request contains multiple `Host` header values. As stated in section 4.4 of RFC 7230, a server must respond with `400 Bad Request` status code to any request that contains more than one `Host` header. While most servers respond with the `400 Bad Request`, some forward the request. Apache httpd, NGINX and HAProxy send the `400 Bad Request` status code, while ATS closes the connection to report the error. Whereas, H2O ignores the first `Host` header and forwards the second one, while Envoy merges the two `Host` header values into one with a comma when forwarding the request.

The `Expect` header has only one value defined by the specification and it is `100-continue`. When a client has a request with a

Table 4: Examples of parsing discrepancies in request headers.

Input	Apache httpd	NGINX	H2O	ATS	HAProxy	Envoy
POST / H/0 :b Content-Length:4	Bad Request Error	Bad Request Error	Bad Request Error	POST / H/1 Content-Length:4	POST / H/0 ~~Content-Length:4~~	Bad Request Error
GET / H/1 Host: a b	Bad Request Error	GET / H/1 Host: a b :	Bad Request Error	GET / H/1 Host: a ~~b~~	Bad Request Error	Bad Request Error
GET / H/1 Host: a Host: b	Bad Request Error	Bad Request Error	GET / H/1 Host: b	Connection Reset	Bad Request Error	GET / H/1 Host: a,b
POST / H/1 Host: a Expect: b	Expectation Failed Error	POST / H/1 Host: a ~~Expect: b~~	POST / H/1 Host: a ~~Expect: b~~	Connection Reset	POST / H/1 Host: a Expect: b	POST / H/1 Host: a Expect: b
POST / H/1 Referer:h://a Referer:h://b	POST / H/1 Referer:h://a,\ h://b	POST / H/1 Referer:h://a Referer:h://b	POST / H/1 Referer:h://a Referer:h://b	Connection Reset	POST / H/1 Referer:h://a Referer:h://b	POST / H/1 ~~Referer:h://a~~ ~~Referer:h://b~~
POST / H/1 a: b \tc	POST / H/1 a: b c	Bad Request Error	POST / H/1 a: b : \tc	POST / H/1 a: b \tc	POST / H/1 a: b \tc	POST / H/1 a: b \tc

huge body, it sends the request line and headers first adding Expect header to make sure that the server is ready to receive the body and this is for improving the efficiency. When this header is sent with an invalid value, the responses of the servers vary. Apache httpd responds with the Expectation Failed Error and ATS closes the connection. Whereas, NGINX and H2O forward the request after dropping the header, while HAProxy and Envoy keep the header.

The referer header allows clients to specify the address of the page from which the request is made. When a request contains double referer header, NGINX, H2O and HAProxy keeps both of them as they are and forward them along. Apache httpd merges two headers into one header with a new value where the two values are combined with a comma. Envoy drops both of them if they lack a path. Finally, ATS reports an error.

Line folding is a means for headers to have a multiline value (e.g., user-agent: mozilla\r\nfirefox) and as seen in the last row of the Table 4, it triggers parsing discrepancies. When a line-folded request is sent to servers, Apache httpd, ATS, HAProxy and Envoy seem to support the line folding as they append the following line to the current before forwarding to the upstream. NGINX respond with the 400 Bad Request. Whereas, H2O seems not to support the line folding as it makes a new header from the following line by adding a colon before it.

5.3 Request Body Discrepancies

Similar to request line and header parsing, the parsing of request body is implemented differently by servers as shown in Table 4 (Transfer-Encoding: chunked and Content-Length have been abbreviated as Transfer-Enc: and Content-Len respectively for the brevity). For example, Content-Length is for indicating the size of the request body and is expected to have a numeric value. However, when this value is very big, then servers might respond differently. In fact, when its value is 10^{19} (a value between 2^{63} and 2^{64}), ATS does not forward the body while keeping the Content-Length as is. Whereas, HAProxy and Envoy forward the body and the Content-Length as is.

Trailer headers, as explained in RFC 7230, allow senders to send metadata at the end of the chunked body (e.g., to allow the recipient to check the integrity). When a chunked body with a trailer header is sent to servers, Apache httpd, H2O, NGINX and Envoy do not forward it to the upstream. Whereas, ATS and HAProxy respect the trailer header and keep it when forwarding.

When the request body does not follow the chunked body format properly, some of them choose to normalize it, while some leave as is. In the example shown in the third row (the backslash followed by the space is not a part of the request contents, it is for line wrapping), the second chunk size (i.e., 1b) is bigger than the chunk data (i.e., 0\r\n\r\n). While ATS forwards the body as is, HAProxy and Envoy reconstructs the chunked body by merging two chunk data sections and sizing them properly.

Similarly, when the chunk size contains invalid characters (as the one shown in the fourth row, \xff, not all servers act the same way. Apache httpd, NGINX, H2O and HAProxy respond with the 400 Bad Request. ATS forwards the request, but does not retain the request body. Whereas, Envoy reconstructs a new chunked body where the chunk data is wrapped under a valid chunk size.

If a client wishes to send a huge request body, it is recommended to send Expect header alongside with the request line and headers to see if the recipient is ready to receive the body. When an Expect request with a request body sent to servers, NGINX, H2O and Envoy drop the Expect header and keep the body, while HAProxy keeps

Table 5: Examples of parsing discrepancies in request bodies.

Input	Apache httpd	NGINX	H2O	ATS	HAProxy	Envoy
Content-Len:10^{19} bbbb	Bad Request Error	Bad Request Error	Bad Request Error	Content-Len:10^{19} ~~bbbb~~	Content-Len:10^{19} bbbb	Content-Len:10^{19} bbbb
Transfer-Enc: 0\r\na:b\r\n\r\n	Content-Len:0	Content-Len:0	Content-Len:0	Transfer-Enc: 0\r\n a:b \r\n\r\n	Transfer-Enc: 0\r\n a:b \r\n\r\n	Transfer-Enc: 0\r\n\r\n
Transfer-Enc: 1\r\nb\r\n1b \ 0\r\n\r\n	Request Timeout	Request Timeout	Request Timeout	Transfer-Enc: 1\r\nb\r\n1b \ 0\r\n\r\n	Transfer-Enc: 6\r\nb0 \ \r\n\r\n\r\n	Transfer-Enc: 6\r\nb0 \ \r\n\r\n\r\n
Transfer-Enc: \xff20\r\n\r\n	Bad Request Error	Bad Request Error	Bad Request Error	Transfer-Enc: ~~\xff20\r\n\r\n~~	Bad Request Error	Transfer-Enc: 2\r\n\r\n\r\n
Expect:100-cont Content-Len:4 bbbb	Content-Len:4 ~~bbbb~~	Content-Len:4 bbbb	Content-Len:4 bbbb	Expect:100-cont Content-Len:4 ~~bbbb~~	Expect:100-cont Content-Len:4 bbbb	Content-Len:4 bbbb
Transfer-Enc: dddd0001\r\n	Bad Request Error	Bad Request Error	Request Timeout	Transfer-Enc: ~~dddd0001\r\n~~	Transfer-Enc: 2 \r\n\r\n\r\n	Bad Request Error

Table 6: Examples of discrepancies where servers respond with a cacheable error code.

Input	Apache httpd	NGINX	H2O	ATS	HAProxy	Envoy
GET /%2f H/1	Not Found Error	GET / H/1	GET /%2f H/1	GET /%2f H/1	GET /%2f H/1	GET /%2f H/1
GET h://a?/ H/1	GET /?/ H/1	GET /?/ H/1	GET / h://a?/ H/1	GET ?/ H/1	Bad Request Error	Not Found Error
OPTIONS *h://a/ H/1	Bad Request Error	Bad Request Error	/ OPTIONS *h://a/ H/1	Connection Reset	OPTIONS *h://a/ H/1	Not Found Error

both. Whereas, Apache httpd drops both the header and the body, and ATS keeps the header while dropping the body.

Finally, a chunk size with a large value such as the one in the last row of Table 5 triggers a body parsing difference. The hexadecimal value of 0xdddd0001 is equivalent to a decimal value between 2^{31} and 2^{32}. While Apache httpd, NGINX, Envoy reports an error for this request, H2O waits for more data to be sent. HAProxy reconstructs the body and sends two bytes of new chunk data (i.e., \r\n). Whereas, ATS ignores the body in the income request and does not forward it to the upstream.

5.4 Cacheable Responses

As shown in Table 6, for some requests at least one server responded with an error status code which is cacheable. For instance, when the URI contains a percent-encoded version of a reserved character ("/" in the example), then H2O, ATS, HAProxy and Envoy keep the URI as is. NGINX decodes it and makes the URI a single slash. Whereas, Apache httpd responds with 404 Not Found status code which is defined as cacheable by RFC 7231 [18] and is commonly cached by cache servers in practice.

When a question mark, which signals the start of the query fragment, comes before the path, it triggers parsing discrepancies in servers. As seen in the second row, Apache httpd and NGINX convert the URI into the origin form and make the question mark a part of the path. H2O adds a slash before the URI as it is an absolute URI. ATS also converts the URI to the origin form, but it leaves the question mark before the path. HAProxy reports an error for this request. Finally, Envoy responds with 404 Not Found which is a cacheable status code.

The asterisk-form URI ("*") is only used for server-wide OPTIONS requests, where the client wishes to make the OPTIONS request for the server as a whole, not for a specific resource [17]. When both the asterisk and absolute URI forms come together in a request URI, servers again act differently. As seen in the third row, Apache httpd, NGINX and ATS report error, while H2O adds a slash before the URI and HAProxy keeps as is. Finally, Envoy responds with a cacheable status code, 404 Not Found.

6 ATTACKS

To demonstrate the security implications of these parsing discrepancies, we devise a number of attack scenarios and test them in a lab setup. As our attacks exploit the parsing discrepancies between servers, our targets are server pairs deployed in a proxy-origin fashion.

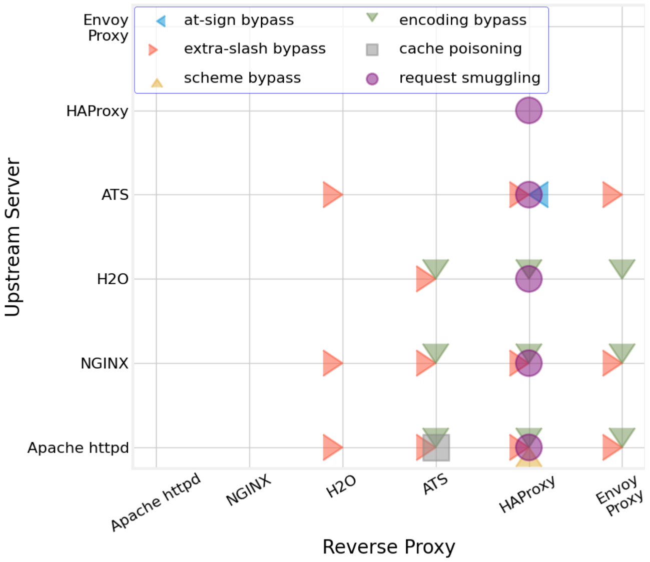

Figure 2: Different types of attacks targeting each server pair.

6.1 Access Control Bypass

Access control is a common security mechanism used on the Internet today, typically to restrict public access to pages which are not for public use. For instance, one might want to block public access to the `/admin` page on a web server to make sure that only those who are authorized to use that page can access it. This type of access control is usually configured as a rule on the reverse proxy, which ensures that requests which match this rule do not get forwarded to the origin.

To test for this attack, we add a rule on the reverse proxy of every pair to block all requests to `/admin` by following the instructions given in their documentation for page access control. The origin server of every pair serves certain content when it receives a request for `/admin`. If the origin server of a pair is not a web server (i.e., the origin is ATS, HAProxy, or Envoy), then we have it forward to an additional upstream server serving the same specific content at `/admin`.

We then designed a set of mutated `GET` requests for `/admin` based on the requests which trigger request line parsing discrepancies and sent them to the reverse proxy of each pair. If the content at `/admin` was served, then we confirm the existence of an access control bypass attack. As seen in Figure 2, we find many server pairs affected by this attack due to four main types of request line parsing discrepancies.

The most common discrepancies arises when the request URI is `//admin` (see the extra-slash category in Figure 2). When the affected reverse proxies receive such a request, their rules fail to match it to `/admin` and as most of them forward the request as is, the origin server ignores or trims the extra slash, and the content is served. Though most proxies forward the request as is, ATS actually trims one of the slashes before forwarding, yet still fails to match the rule, likely because trimming happens after the access control check.

The next type of discrepancy happens when the URI has an at-sign before the path (i.e., `@/admin`). Only one server pair was affected for this case: HAProxy-ATS. HAProxy does not interpret the path as `/admin` and forwards the URI as is. When ATS receives it, it trims the at-sign, converting the URI to `/admin` and therefore serves the content.

Another type of discrepancy leading to access control bypass is caused by the presence of a scheme name right before the path (i.e., `http:/admin`. This attack affects one server pair: HAProxy-Apache httpd. Like the previous attack, HAProxy cannot match this URI to the rule it has and therefore forwards it to Apache httpd. When Apache httpd receives it, it trims the scheme part and processes the path successfully.

Finally, we see several pairs affected by bypass attacks caused by the encoding of the path, for example, where the first letter of the page name is percent-encoded (i.e., `/%61dmin`. When the affected servers receive this URI they do not interpret the page as `/admin`, and therefore forward it to the upstream where this is interpreted as an access to the admin page.

6.2 Cache Poisoning Denial-of-Service

The next type of attack which happens due to a parsing discrepancy is a cache poisoning denial-of-service (CPDoS) attack. In this attack, the attacker constructs an HTTP request for a legitimate resource which would be accepted and forwarded by the cache server on the request path, but would trigger a cacheable error status code (e.g., `HTTP 404 Not Found`) on the origin server. As a result, the attacker manages to poison the cache for the legitimate resource and every other user trying to reach that resource will receive an error code for their request.

To test this attack scenario, we first configure the reverse proxy to cache responses. ATS does not cache error response codes by default, so we enable this on ATS in order to simulate the common real-world scenario where error responses *are* cached [1, 9, 15]. On the origin side, we create a legitimate resource for which we want to have victims receive a poisoned response.

As with testing for access control bypass, we created a set of mutated requests for the legitimate resource based on the requests which trigger request line parsing discrepancies, and sent them to the reverse proxy of each pair. We follow this with a valid, unmutated request for the same resource, and if we receive an error response code, then we confirm the existence of a CPDoS attack.

We find that due to a parsing discrepancy of percent-encoded characters, a CPDoS attack exists for one server pair: ATS-Apache httpd. When an attacker sends a request with a URI of `/foo%2fbar`, where the `%2f` is the percent-encoded version of the slash character, ATS interprets the URI as `/foo/bar` for caching purposes, but forwards the URI as is to Apache httpd. When Apache httpd receives this request URI, it responds with `HTTP 404 Not Found` which is cached by ATS. As a result, ATS returns this error code to subsequent requests made for `/foo/bar`.

6.3 HTTP Request Smuggling

HTTP Request Smuggling (HRS) occurs when the attacker is able to smuggle an additional request through the reverse proxy into the connection between the reverse proxy and the upstream server.

As the reverse proxy is not aware of that additional request, this can be exploited for many serious attacks from response queue poisoning to request hijacking.

To test for this attack, we first configure each reverse proxy to reuse connections to the origin server. Then, we send requests with apparent request smuggling potential (i.e., where two different servers forward requests with different actual body lengths and/or `Content-Length` header values) to the reverse proxy. We then use a popular technique adopted by prior work [24, 25] to confirm the existence of the HRS vulnerability, in which we send the potential smuggler request followed by a normal request, and check to see if the normal request receives an error response from the origin.

We find an HRS attack affecting multiple server pairs with HAProxy as the reverse proxy due to the parsing behavior of a header with an empty name, which is the first of its type, to the best of our knowledge. As shown in the first row of Table 4, when HAProxy receives a request with an empty header name, it removes the header coming after this header. As a result, if the input request has a request body and has a `Content-Length` header after the empty-named header, HAProxy forwards this request without the `Content-Length` header while leaving the request body intact. We see that when we put another request in the body of this request, it is forwarded by HAProxy and treated as the next request by the origin server.

We can attribute finding this attack in large part to our holistic search methodology. If, like prior work, we were to try to find HRS attack vectors by looking just for differences in the forwarded requests' body sizes, we would see that ATS and HAProxy forward a request with the same body length in response to the request with the empty-named header, and would therefore miss this finding. However, as we searched for the discrepancies in the header parsing as well, we noticed that HAProxy drops the `Content-Length` header, unlike ATS.

7 COMPARISON WITH OTHER TOOLS

The other tools which were developed by the past research to find HTTP parsing discrepancies are T-Reqs [24] and HDiff [38], both of which use the blackbox fuzzing technique. Source code for both tools are publicly available on Github. However, due to the combination of missing code and unclear usage instructions we were not able to run HDiff.

In order to compare the abilities of Gudifu and T-Reqs in finding parsing discrepancies, we run an additional experiment with T-Reqs. We take several steps in order to run the T-Reqs under the same conditions that applied to the Gudifu experiment. First, we write a new grammar to enable T-Reqs to generate the requests which were given to Gudifu in the initial corpus. Second, we use the same number of processes, and the same timeout durations for T-Reqs. Finally, we let the T-Reqs experiment run for twelve hours.

In order to compare the results, we use our own search methodology on the forwarded requests captured in the T-Reqs experiment. We find two discrepancy types in the body parsing. One of them is that ATS and Apache httpd do not forward the request body to the echo server when the `expect: 100-continue` header is present in the request, while other servers still forward the body. The other is that some servers (e.g., HAProxy) forward the trailer headers in the chunked body, while some (e.g., Envoy) ignore them.

As Table 5 shows, Gudifu also finds both of these discrepancy types. In addition to them, Gudifu finds four additional discrepancy types in the body parsing, making it six in total. The difference in the number of findings show that the Gudifu approach is more effective than the T-Reqs approach in finding body parsing discrepancies.

8 DISCUSSION

In this section, we discuss the limitations of our work, possible ways of addressing the discrepancy attacks and finally the vendor responses.

8.1 Limitations

When we search for the discrepancies in the servers' parsing behavior, we only look at the forwarded requests. However, forwarded requests might not always accurately reflect the parsing behavior of the server. For example, when ATS receives a request for `//admin`, it trims the first slash and forwards a request for `/admin`. At a glance, this would seem to suggest that when ATS is configured to block access to `/admin`, it would block a request for `//admin`, but in reality it does not, because the trimming occurs after access control checks.

One possible way to increase our visibility into the parsing behavior is to have the servers log different fields of the request after they parse it. However, this feature is not supported by all servers and is also limited to recording only the components of an HTTP request that a server exposes for logging.

Also, the number of commonly forwarded requests in our experiment is small, given the duration of the experiment. Just 45,633 requests out of 6,737,538 total valid requests are common to all servers. This limitation originates from the design choice of only adding forwarded requests back to the input corpus if they both exercise new code in the target server and receive a `200 OK` from the echo server. However, we believe that this results in a higher quality of inputs in the corpus, which compensates for the small number of common requests.

While not a limitation of Gudifu per se, an obvious drawback of our research and presentation is that we are unable to experiment with proxies where no source code is available. In particular, Content Delivery Networks (CDNs)–prime targets for both researchers and attackers due to their critical place in the Internet ecosystem–are necessarily left out of scope. Nevertheless, Gudifu has been publicly released, and this allows CDNs and other proprietary technology owners to test their systems in house and avail from our research contributions. We acknowledge that prior work that utilized strictly blackbox approaches does not have this limitation.

8.2 Potential Improvements

The Gudifu framework was designed to identify discrepancies in HTTP/1 parsing. Discrepancies in HTTP/2 and HTTP/3 parsing can also be identified with Gudifu with a few adjustments. These adjustments would include the support for a structure-aware fuzzing due to the highly-structured binary input formats of HTTP/2 and HTTP/3. For instance, the "Protocol Buffers" input format [7] with its mutator mechanism [21] can be used to enable structure-aware fuzzing in LibFuzzer [22]. Another adjustment would be configuring HTTP/2 and HTTP/3 targets to forward requests in HTTP/1 format in order to support the use of Gudifu's discrepancy search method. The protocol downgrade is a commonly supported feature in HTTP servers [23].

Also, Gudifu relies on the source code availability for the instrumentation of the target programs. There are mainly two approaches for the instrumentation when the compile-time instrumentation is not possible: 1) dynamic binary translation, which performs the instrumentation at runtime and therefore has inherent performance overhead, and 2) static binary translation, which statically rewrites the target program binary to add the instrumentation code, with recent works such as rev.ng [19], RWFuzz [33] and Retrowrite [13] proposing new tools and techniques in this direction. In particular, rev.ng [19] can be used to translate a target program binary into LLVM IR and make it suitable for fuzzing with LibFuzzer [20].

Another improvement that can be made to the Gudifu framework is to increase the robustness of fuzzing experiments. Currently, if any input causes its target to stop running (i.e., exit) , then the fuzzer instance also stops running. This resets all the coverage achieved until that point in the experiment. This issue can be resolved with extra user-implemented measures. One such measure could be identifying the inputs that cause the targets to exit and preventing their delivery to the relevant target.

Finally, the number of commonly forwarded requests, which determines the size of the search space for parsing discrepancies, can be increased in order to broaden the search space. Currently, this number is equal to the total number of inputs that achieve a new coverage in any target (i.e., the input corpus size). This amount can be increased by having each fuzzer instance deliver its generated input to all targets in each iteration, while using its own target's coverage feedback to guide the input selection. This would significantly increase the size of commonly forwarded requests, since every target receives and parses each request generated by all fuzzer instances.

8.3 Addressing Discrepancy Attacks

Recall that discrepancy attacks are an outcome of systems-centric interaction hazards. Even in an idealized system where every component is perfectly secure in isolation, discrepancy attacks that undermine the entire system's security goals can crop up when the right (or perhaps wrong) components are allowed to interact without an external enforcement mechanism for security constraints.

This is a threat that cannot easily be modeled, let alone effectively addressed, by traditional security thought or the tools available to us. Standard approaches to developing secure code (e.g., code reviews, static and dynamic program analysis), defense (e.g., firewalls, anomaly detection systems, system segmentation), and security management (e.g., asset discovery, CVE monitoring, automatic patching) are all primarily designed to secure an environment under the full control of its operator.

In contrast, discrepancies impact highly distributed systems developed, owned, and operated by distinct entities, without any interaction between them. Regardless of how rigorously they are tested in isolation, system interaction flaws can remain hidden. When an issue is discovered, since there is no single component or root cause to blame, assigning responsibility may not be possible, and a fix infeasible. Attempts to eliminate hazardous interactions could result in changes that introduce further unexpected interactions with other technologies elsewhere. This is an emergent, hard problem in computer science due to increasing system complexity; there is no known solution.

As the pace of research in this domain increases, one immediately actionable recommendation for all proxy and server developers is to strictly follow RFC guidance in their HTTP implementations. As evident in our findings, many discrepancies are due to liberties taken when implementing behavior explicitly defined in protocol specifications. To reiterate, in isolation, these may be harmless and even meaningful optimizations, but the impact of seemingly inconsequential deviations from the specification could be exploitable in unpredictable ways, as repeatedly demonstrated with the steady stream of new attacks.

The same recommendation applies to the working groups in charge of designing and standardizing protocol specifications. In the light of emergent discrepancy attacks and ever-increasing complexity of networked systems, specifications should consider moving away from the traditional *MAY, SHOULD, MUST* framework for requirements, and instead be more prescriptive. Such prescriptive specifications are still helpful for security even when there is no clear winning approach among the design alternatives under consideration, as the end goal is consistent behavior across different implementations.

An immediate action that can be taken by network operators is to implement the whitelisting approach to prevent discrepancy-based attacks. More specifically, they can analyze their own network traffic and identify the legitimate request patterns. This can enable them to implement rules for allowing only those requests that follow those patterns and blocking the rest. However, the rest can contain unusual legitimate requests, leading to false positive scenarios and usability issues. In fact, this is precisely why major Internet companies cannot comply with all RFC guidelines and prefer the satisfaction and the needs of their customers by allowing them to use non-compliant behavior unless they choose otherwise [2].

Alternatively, network operators can take the blacklisting approach to prevent discrepancy-based attacks. While it has limited ability in blocking previously-unseen attacks, it can be very effective in preventing subsequent attempts of the same attack by learning its patterns and implementing preventive rules based on those patterns. In fact, some CDN companies take this approach against the HTTP Request Smuggling attacks and search incoming requests for attack patterns such as "an HTTP request in `POST` body" and "duplicate `Content-Length` headers" [4].

8.4 Vendor Responses

Vendors of three out of six products we tested in this paper have responded to our report. ATS developers made a release with fixes for access control and cache poisoning attacks and assigned a CVE, CVE-2023-33934, with a critical severity. NGINX developers did not take an immediate action, but they said NGINX parsing can be improved in some of the request mutations we reported. The HAProxy team urgently made an emergency release and assigned a CVE , CVE-2023-25725, with a critical severity. They also thanked us for making the Internet a safer place.

9 CONCLUSION

We presented Gudifu, a guided differential fuzzer for efficient discovery of novel discrepancy attacks targeting HTTP servers. Our approach differentiates itself from the existing work in this domain through our graybox testing approach and novel discrepancy search methodology, evidently achieving better attack discovery performance. This affirmatively answers our research questions (Q1) and (Q2) we laid out in Section 1.

Through our extensive experiments with six prominent server technologies, detailed findings, and concrete exploits crafted from these results in Sections 5 & 6, we demonstrated the feasibility and severity of discrepancy attacks, therefore answering our remaining research question (Q3).

ACKNOWLEDGMENTS

The initial version of the Gudifu framework was developed with the author's collaborative work with the Envoy Platform Team at Google. We thank Adi Peleg and Harvey Tuch of Envoy Platform Team for their feedback and contributions, which played a very important role in the success of this research. We also thank the anonymous reviewers and our shepherd for their suggestions and directions for the improvement of the paper. This project was partially supported by National Science Foundation grants CNS-2031390 and CNS-2329540.

REFERENCES

[1] Akamai. [n.d.]. Caching. Akamai Techdocs. https://techdocs.akamai.com/api-definitions/docs/caching.

[2] Akamai. [n.d.]. Strict Header Parsing. Akamai Techdocs. https://techdocs.akamai.com/property-mgr/docs/strict-header-parsing.

[3] Anastasios Andronidis and Cristian Cadar. 2022. SnapFuzz: High-Throughput Fuzzing of Network Applications. In *ACM SIGSOFT International Symposium on Software Testing and Analysis*.

[4] Ryan Barnett. 2021. HTTP/2 Request Smuggling. Akamai Blog. https://www.akamai.com/blog/security/http-2-request-smulggling.

[5] Tim Berners-Lee, Roy T. Fielding, and Larry Masinter. 2005. Uniform Resource Identifier (URI): Generic Syntax. https://datatracker.ietf.org/doc/html/rfc3986.

[6] Lukas Bernhard, Tobias Scharnowski, Moritz Schloegel, Tim Blazytko, and Thorsten Holz. 2022. JIT-Picking: Differential Fuzzing of JavaScript Engines. In *ACM Conference on Computer and Communications Security*.

[7] Protocol Buffers. [n.d.]. Protocol Buffers - Google's data interchange format. Github Repository. https://github.com/protocolbuffers/protobuf.

[8] Jianjun Chen, Jian Jiang, Haixin Duan, Nicholas Weaver, Tao Wan, and Vern Paxson. 2016. Host of Troubles: Multiple Host Ambiguities in HTTP Implementations. In *ACM Conference on Computer and Communications Security*.

[9] Cloudflare. [n.d.]. Configure cache by status code. Cloudflare Docs. https://developers.cloudflare.com/cache/how-to/configure-cache-status-code.

[10] Richard I. Cook. 1998. How Complex Systems Fail. https://how.complexsystems.fail/.

[11] Evan Custodio. 2019. Mass account takeovers using HTTP Request Smuggling on https://slackb.com/ to steal session cookies. https://hackerone.com/reports/737140.

[12] Evan Custodio. 2020. Practical Attacks Using HTTP Request Smuggling by @defparam. NahamCon. https://www.youtube.com/watch?v=3tpnuzFLU8g.

[13] Sushant Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. 2020. Retrowrite: Statically instrumenting cots binaries for fuzzing and sanitization. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1497–1511.

[14] Envoy. 2023. HTTP connection manager (proto). envoyproxy.io. https://www.envoyproxy.io/docs/envoy/latest/api-v3/extensions/filters/network/http_connection_manager/v3/http_connection_manager.proto#envoy-v3-api-field-extensions-filters-network-http-connection-manager-v3-httpconnectionmanager-merge-slashes.

[15] Fastly. 2022. Caching configuration best practices. Fastly Documentation. https://docs.fastly.com/en/guides/caching-best-practices.

[16] Roy T. Fielding, Jim Gettys, Jeffrey C. Mogul, Henrik Frystyk, Larry Masinter, Paul Leach, and Tim Berners-Lee. 1997. Hypertext Transfer Protocol – HTTP/1.1. https://datatracker.ietf.org/doc/html/rfc2616.

[17] Roy T. Fielding and Julian F. Reschke. 2014. Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing. https://datatracker.ietf.org/doc/html/rfc7230.

[18] Roy T. Fielding and Julian F. Reschke. 2014. Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content. https://datatracker.ietf.org/doc/html/rfc7231.

[19] Antonio Frighetto. 2019. Coverage-guided binary fuzzing with REVNG and LLVM libfuzzer. (2019).

[20] Antonio Frighetto. 2020. Fuzzing binaries with LLVM's libFuzzer and rev.ng. REVNG Blog. https://rev.ng/blog/fuzzing-binaries.

[21] Google. [n.d.]. libprotobuf-mutator. Github Repository. https://github.com/google/libprotobuf-mutator.

[22] Google. [n.d.]. Structure-Aware Fuzzing with libFuzzer. Github Repository. https://github.com/google/fuzzing/blob/master/docs/structure-aware-fuzzing.md.

[23] Bahruz Jabiyev, Steven Sprecher, Anthony Gavazzi, Tommaso Innocenti, Kaan Onarlioglu, and Engin Kirda. 2022. {FRAMESHIFTER}: Security implications of {HTTP/2-to-HTTP/1} conversion anomalies. In *31st USENIX Security Symposium (USENIX Security 22)*. 1061–1075.

[24] Bahruz Jabiyev, Steven Sprecher, Kaan Onarlioglu, and Engin Kirda. 2021. T-Reqs: HTTP Request Smuggling with Differential Fuzzing. In *ACM Conference on Computer and Communications Security*.

[25] James Kettle. 2019. HTTP Desync Attacks: Request Smuggling Reborn. PortSwigger Web Security Blog. https://portswigger.net/blog/http-desync-attacks-request-smuggling-reborn.

[26] James Kettle. 2019. Stored XSS on https://paypal.com/signin via cache poisoning. HackerOne. https://hackerone.com/reports/488147.

[27] James Kettle. 2021. HTTP/2: The Sequel is Always Worse. PortSwigger Web Security Blog. https://portswigger.net/research/http2.

[28] Iustin Ladunca. 2020. Cache Key Normalization DoS. https://youst.in/posts/cache-key-normalization-denial-of-service/.

[29] Iustin Ladunca. 2021. Cache Poisoning at Scale. https://youst.in/posts/cache-poisoning-at-scale/.

[30] Nancy G. Leveson. 2011. *Engineering a Safer World*. The MIT Press, Cambridge, MA, USA.

[31] libFuzzer. 2023. libFuzzer – a library for coverage-guided fuzz testing. LLVM.org. https://llvm.org/docs/LibFuzzer.html.

[32] Hoai Viet Nguyen, Luigi Lo Iacono, and Hannes Federrath. 2019. Your Cache Has Fallen: Cache-Poisoned Denial-of-Service Attack. In *ACM Conference on Computer and Communications Security*.

[33] Eric Pauley, Gang Tan, Danfeng Zhang, and Patrick McDaniel. 2022. Performant binary fuzzing without source code using static instrumentation. In *2022 IEEE Conference on Communications and Network Security (CNS)*. IEEE, 226–235.

[34] Theofilos Petsios, Adrian Tang, Salvatore Stolfo, Angelos D. Keromytis, and Suman Jana. 2017. Nezha: Efficient Domain-Independent Differential Testing. In *IEEE Symposium on Security and Privacy*.

[35] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. 2020. AFLNET: A Greybox Fuzzer for Network Protocols. In *IEEE International Conference on Software Testing, Validation and Verification*.

[36] Gaganjeet Singh Reen and Christian Rossow. 2020. DPIFuzz: A Differential Fuzzing Framework to Detect DPI Elusion Strategies For QUIC. In *Annual Computer Security Applications Conference*.

[37] Sergej Schumilo, Cornelius Aschermann, Andrea Jemmett, Ali Abbasi, and Thorsten Holz. 2022. Nyx-Net: Network Fuzzing with Incremental Snapshots. In *European Conference on Computer Systems*.

[38] Kaiwen Shen, Jianyu Lu, Yaru Yang, Jianjun Chen, Mingming Zhang, Haixin Duan, Jia Zhang, and Xiaofeng Zheng. 2022. HDiff: A Semi-automatic Framework for Discovering Semantic Gap Attack in HTTP Implementations. In *IEEE/IFIP International Conference on Dependable Systems and Networks*.

[39] Michał Zalewski. 2023. american fuzzy lop. lcamtuf.coredump.cx website. https://lcamtuf.coredump.cx/afl/.

[40] Yong-Hao Zou, Jia-Ju Bai, Jielong Zhou, Jianfeng Tan, Chenggang Qin, and Shi-Min Hu. 2021. TCP-Fuzz: Detecting Memory and Semantic Bugs in TCP Stacks with Fuzzing. In *USENIX Annual Technical Conference*.