FRAMESHIFTER: Security Implications of HTTP/2-to-HTTP/1 Conversion Anomalies

Bahruz Jabiyev, Steven Sprecher, Anthony Gavazzi, Tommaso Innocenti, Kaan Onarlioglu[†], Engin Kirda
Northeastern University, [†]Akamai Technologies

Abstract

HTTP/2 adoption is rapidly climbing. However, in practice, Internet communications still rarely happen over *end-to-end* HTTP/2 channels. This is due to Content Delivery Networks and other reverse proxies, ubiquitous and necessary components of the Internet ecosystem, which only support HTTP/2 on the client's end, but not the forward connection to the origin server. Instead, proxy technologies predominantly rely on HTTP/2-to-HTTP/1 protocol conversion between the two legs of the connection.

We present the first systematic exploration of HTTP/2-to-HTTP/1 protocol conversion anomalies and their security implications. We develop a novel grammar-based fuzzer for HTTP/2, experiment with 12 popular reverse proxy technologies & CDNs through HTTP/2 frame sequence and content manipulation, and discover a plethora of novel web application attack vectors that lead to Request Blackholing, Denial-of-Service, Query-of-Death, and Request Smuggling attacks.

1 Introduction

HTTP/2 has seen quick and massive adoption since its introduction in 2015. A 2020 measurement by HTTP Archive showed that 64% of HTTP requests were served using HTTP/2 [10]. However, these measurements come with a subtle yet critical caveat: In practice, clients and origin servers rarely communicate over *end-to-end* HTTP/2 channels, but instead use a mix of HTTP/2 and HTTP/1.[1]

This situation is largely due to the widespread use of Content Delivery Networks (CDNs) and other stand-alone reverse proxies, which intercept and process the traffic exchanged between a client and origin server. Even though such proxy technologies support HTTP/2 on the client-facing leg of the connection, they *rarely* do so for proxy-to-origin connections, and instead fall back to using HTTP/1, regardless of the origin's support for HTTP/2. As a result, proxies need to dynamically translate between HTTP/2 and HTTP/1 as they forward packets in either direction.

There is no officially documented account of this need for the HTTP/2-to-HTTP/1 conversion, or a formal analysis of its implications, to the best of our knowledge. However, informal exchanges observed online (e.g., a post to the NGINX mailing list by an NGINX developer [6]) provide insights into potential reasons. For example, some proxy developers see no performance benefit to using HTTP/2 for proxy-to-origin connections, especially when the proxy is co-located with the origin. Other reasons include general technical debt concerns and the infeasibility of updating established man-in-the-middle technologies. For instance, web application firewalls and load balancers that run on proxies are only designed to process HTTP/1; an overhaul is made difficult by the fact that HTTP/2 is a binary protocol [2].

Regardless of the reasons, in an exploratory study, we found that out of the ten most popular reverse proxies, only **one** supported upstream HTTP/2 connections–and that support too was disabled by default. Given the solidifying position of proxies as critical infrastructure for a scalable Internet, and their ubiquitous use repeatedly demonstrated by public data and scholarly measurements (e.g., [4, 12, 21, 22]), HTTP/1 is poised to remain in heavy use.

Contemporary research on HTTP Request Smuggling, Web Cache Deception, and cache poisoning attacks have already shown that web security suffers from the complexity of the HTTP protocol and discrepancies between the behaviors of server technologies on the traffic path [13, 16, 21–23]. HTTP/2-to-HTTP/1 conversion adds more complexity to an already intricate web protocol, and opens up the possibility of introducing further flawed HTTP processing mechanisms and non-conformant behavior. In fact, researchers have already utilized this new attack surface to successfully mount Request Smuggling attacks against major proxy technologies and the origin servers they front [18, 19].

In this paper, we present the first analysis of HTTP/2-to-HTTP/1 conversion flaws within a scientific framework. The aforementioned prior HTTP/2 research is limited to investi-

[1]In this paper, we will refer to all HTTP/1.* protocol versions simply as HTTP/1 for brevity.

gating basic mangling of a single HTTP/2 *frame*–the smallest unit of communication encapsulated within a *stream*. In contrast, we systematically explore ways to manipulate both the *frame sequences* and the *content* therein.[2] Specifically, we aim to answer the following research questions.

(Q1) Do frame sequence and content manipulation cause HTTP/2-to-HTTP/1 conversion anomalies?

(Q2) What manipulation patterns cause conversion anomalies and why?

(Q3) What attacks are possible by exploiting conversion anomalies?

To answer these questions, we develop FRAMESHIFTER, a grammar-based fuzzer for HTTP/2. FRAMESHIFTER leverages an input grammar to generate valid HTTP/2 frame sequences, and then applies sequence and content mutations. We use FRAMESHIFTER to exercise 12 popular technologies including 8 stand-alone proxies and 4 CDNs. We then capture the resulting HTTP/1 requests forwarded by these proxies and check for anomalies.

Our experiment reveals a myriad conversion anomalies caused by these prominent technologies. We categorize our findings and test samples for real-world attacks in our experiment infrastructure. We successfully execute damaging attacks such as Request Blackholing, Denial-of-Service, Query-of-Death, and Request Smuggling.

We summarize our contributions below.

- We introduce FRAMESHIFTER, a grammar-based fuzzer for HTTP/2.
- We present a systematic and holistic approach to study HTTP protocol conversion anomalies.
- We discover novel attack vectors on HTTP/2 conversions and provide insights into why they happen.
- We demonstrate successful attacks and coordinate mitigations with the impacted technology vendors.

Availability. FRAMESHIFTER is open source and available online [15].

2 Background and Related Work

In this section, we give an overview of the HTTP/2 protocol, HTTP/2-to-HTTP/1 conversions, HTTP/1 chunked encoding, grammar-based fuzz testing, and notable related works.

2.1 HTTP/2 Protocol

HTTP/1 suffers from major performance issues such as head-of-line blocking and packet bloat due to having to repeat headers for each request-response exchange. HTTP/2 addresses these issues within the protocol. It uses request and response multiplexing, header compression, and has explicit support for request prioritization and server push. It does all this without altering the underlying semantics of HTTP, but instead by redesigning how the data is formatted and transferred.

The new protocol achieves these advantages by introducing new primitives. Request-response pairs are encapsulated within a *stream*. Each stream has a unique identifier, and packets from different streams can be interleaved, transferred over a single TCP connection.

Streams are made up of sequences of *frames*. A frame is the smallest unit in the protocol. To illustrate, Listing 1 shows a stream with three frames containing a `POST` request with the message `hello, world!` sent to the `/echo` endpoint on `echo.com`. In this example, the `HEADERS` frame carries a set of header fields which are later supplemented by the following `CONTINUATION` frame. The `CONTINUATION` frame has the `END_HEADERS` flag set, indicating no more headers will follow. The `DATA` frame at the end of the sequence contains the entire message body.

There are many other types of HTTP/2 frames. A short description for each per the HTTP/2 specification [3] is below.

`DATA`: Carries a request or a response body.
`HEADERS`: Carries header fields of a request or a response.
`PRIORITY`: Specifies the priority of a stream and its dependency on another stream.
`RST_STREAM`: Terminates the stream.
`SETTINGS`: Conveys information about preferences and constraints of the sender.
`PUSH_PROMISE`: Notifies the peer endpoint about streams it intends to initiate in the future.
`PING`: Measures round-trip time and checks if an idle connection is still functional.
`GOAWAY`: Shuts down a connection.
`WINDOW_UPDATE`: Implements flow control.
`CONTINUATION`: Continues a sequence of header fields.

2.2 HTTP/2-to-HTTP/1 Conversion

HTTP/2 is the most widely used HTTP version by clients today. A 7M-site measurement using the Chrome browser, done by the HTTP Archive in 2020, showed that 64% of requests use HTTP/2 [10].

```
HEADERS             CONTINUATION        DATA
:method = POST      + END_HEADERS       + END_STREAM
:path = /echo       :scheme = https     hello, world!
                    host = echo.com
```

Listing 1: `POST` request in HTTP/2.

[2] We provide a refresher on the HTTP/2 protocol in Section 2.

Yet, these requests mainly originate from end users. Reverse proxies almost always downgrade HTTP/2 to HTTP/1 when forwarding requests as shown in Figure 1. For instance, the HTTP/2 request in Listing 1 will be converted into a HTTP/1 request like those shown in Listing 2.

Reverse proxies perform this conversion for many reasons, reportedly to support legacy tools that only work on HTTP/1 and to make optimization decisions. Most notably, they see little to no performance benefit in using HTTP/2 for last-mile connections [6].

When investigating ten of the most popular reverse proxies, we found only one that supported HTTP/2 connections to origins, and not by default. Recent work corroborated that CDN servers only support HTTP/2 with connections to clients, and not to origins [13].

2.3 HTTP/1 Chunked Encoding

HTTP/1 supports various ways to encode a request body [7]. One of these is the *chunked encoding*. Chunked encoding is especially useful when the size of the data to be transferred is not known in advance.

Listing 2 shows the same request in two different body formats. The body of the request on the left is not encoded, whereas the one on the right is chunk encoded–the "hello, world!" message is sent in two chunks. Each chunk consists of a chunk-size (e.g., 7) and chunk-data (e.g., "hello, "). The final zero-sized chunk indicates the end of the chunked body.

2.4 Grammar-Based Fuzz Testing

Grammar-based fuzzing is commonly used for testing programs with a complex input structure.

One of the most popular choices for describing an input language is a context-free grammar (CFG) [25]. A CFG has four components: a start symbol, non-terminal symbols, terminal symbols, and production rules. The start symbol is where the expansion of a CFG begins. In Listing 3, the start symbol is denoted by `<start>`. Symbols surrounded by <> are non-terminals, meaning they are expanded before the input is fully generated. For example, `<sequence>` is expanded to a sequence of other non-terminal symbols, whereas, `<method>` can be expanded into multiple terminal strings. Finally, production rules define how symbols are expanded. Each line in Listing 3 is a production rule.

Grammar-based fuzzing has been widely used by researchers and industry to uncover bugs in all sorts of programs including language compilers and interpreters [14, 24], and even web browsers [9].

FRAMESHIFTER combines grammar-based fuzzing with mutation-based fuzzing to exercise HTTP/2 processors. Previous research has also adopted similar approaches, for example, Aschermann et al. to find bugs in interpreters [1], and Jabiyev et al. to discover discrepancies between HTTP processors [16].

2.5 Related Work

Even though HTTP/2 is commonplace, security research focusing on this protocol is still relatively limited.

The most closely related work to ours focuses on exploiting the HTTP/2-to-HTTP/1 request conversion for HTTP Request Smuggling (HRS) [18, 19]. The key insights researchers leveraged are that HTTP/2 does not require a `content-length` and forbids chunked `transfer-encoding`, and that header fields are not separated by a `CRLF` in HTTP/2. When these vectors are exploited attackers can smuggle a request following a doctored request.

Guo et al. found two Denial-of-Service attacks by abusing HTTP/2 conversion features on CDN servers [13]. The first attack relies on the HPACK mechanism of HTTP/2 where repeated header fields are saved in a table and transmitted as an index to save bandwidth. The second takes advantage of the fact that some CDN servers forward `POST` requests as soon as request headers are processed, without waiting for the

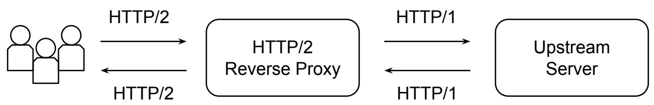


Figure 1: HTTP/2 is used only between end users and HTTP/2 servers. Usually, HTTP/1 is used when sending requests to the upstream servers.

```
POST /echo HTTP/1.1
Host: echo.com
content-length: 13

hello, world!
```

```
POST / HTTP/1.1
Host: echo.com
transfer-encoding:chunked

7
hello,
6
world!
0
```

Listing 2: Requests with a regular and chunked body.

```
<start> ::= <sequence>
<sequence> ::= <headers><data> | <headers>
<headers> ::= <method><path><host>
<method> ::= :method=GET | :method=POST
<path> ::= /echo
<host> ::= echo.com
<data> ::= hello,world! | bye,world!
```

Listing 3: Example CFG for an HTTP/2 frame sequence.

request body to arrive.

Other academic research has instead focused attacking HTTP/2 directly, and not on the conversion between different protocols. Notably, Goethem et al. studied HTTP/2 stream concurrency and potential timing side-channels [11].

3 Scope and Definitions

3.1 Investigation Scope

We study abnormal HTTP/2-to-HTTP/1 conversions with a focus on headers and frames that affect the request body (i.e., `content-length` and `transfer-encoding` headers). The relation between those headers and the request body are within our scope as well. As discussed earlier, there have been many attacks focused on these parts of HTTP requests, warranting their focus for our study [5, 17].

Additionally, we limit the frame sequences used in our experiments to a single stream in order to simplify analysis. We choose to only study HTTP/2 servers that have the capability to be run as a reverse proxy. Reverse proxies are the only servers that do the protocol conversion of interest to us.

3.2 Abnormal Conversions

We define the HTTP/2-to-HTTP/1 conversion as normal if it meets these conditions:

- One HTTP/1 request is generated from a stream.
- If the generated HTTP/1 request has a body, either a `content-length` header is present with a numeric value equal to the length of the body, or the request has a `transfer-encoding: chunked` header and the body follows the proper chunked format.

The failure to meet these conditions signals the presence of a *body-related anomaly* and makes the conversion an *abnormal conversion*.

4 FRAMESHIFTER

We develop a grammar-based HTTP/2 fuzzer called FRAMESHIFTER, named after a DNA mutation called "frameshift mutations." Our tool has two main capabilities: 1) generating inputs from a grammar, and 2) mutating the generated inputs.

4.1 Generating HTTP/2 Frame Sequences

FRAMESHIFTER uses an input grammar to generate HTTP/2 frame sequences. The input grammar defines the content of each frame type, as well as their combination.

For each production rule in the grammar, a list of options can be specified. To illustrate this, the example grammar shown in Listing 3 (line 2) can generate a sequence with a single `HEADERS` frame, or a sequence with one `HEADERS` and one `DATA` frame.

Because there are many options specified in the grammar, FRAMESHIFTER uses a random number generator to seed the sequence creation. For options that are more of interest, preferences can be codified into the grammar allowing for options to be selected on a user-specified probability distribution.

4.2 Mutating HTTP/2 Frame Sequences

After FRAMESHIFTER generates an input sequence from a grammar, it then makes mutations. FRAMESHIFTER supports two types of mutations: 1) *frame sequence mutations* and 2) *frame content mutations*.

4.2.1 Frame Sequence Mutations

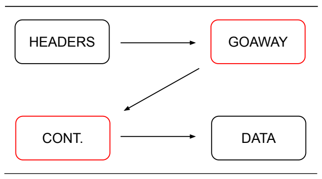
The tool can be configured to apply any number of mutations to a sequence by adding a grammar-defined frame to a grammar-built sequence at a random position (i.e., insertion or replacement), or by removing a frame at a random position from the sequence (i.e., deletion). For example, Listing 4 (left side) shows an example where `GOAWAY` and `CONTINUATION` frames are inserted at random positions into a sequence of a `HEADERS` and `DATA` stream.

FRAMESHIFTER allows for the specification of probability distributions for both types of mutation operators. For example, in Listing 5, line 13, "insert_symbol" has a 90% selection probability. Frame types can have their selection probabilities specified as well, for instance, as shown in Listing 5, line 5.

4.2.2 Frame Content Mutations

Frame sequence mutations can be accompanied by any number of frame content mutations. Figure 4 (right side) shows an example where the dash in `content-length` is replaced by an underscore, the last letter of the header name is removed, and a semicolon is added after the value.

FRAMESHIFTER mutates only those fields which are marked as mutable in the configuration file (see line 9 in Listing 5). For insertion and replacement operations, a character is chosen from a pool specified in the configuration, an example of which is on line 2 of Listing 5. Also, a probability distribution can be specified for both mutation operations



HEADERS
`:method = POST`
`:path = /echo`
`content_lengt`~~h~~`: 5;`

Listing 4: Example FRAMESHIFTER mutations.

```
1  # Character pool for insertion/replacement
2  config.char_pool = [(\x01, opts(prob=0.2)), \x02,
   ↪  \x03, \x04, \x05, \x06, \x07, \x08, \t, \n, ...]
3
4  # Symbol pool for insertion/replacement
5  config.symbol_pool = [(<headers-1>, opts(prob=0.25)),
   ↪  (<continuation-1>, opts(prob=0.25)), (<data-1>,
   ↪  opts(prob=0.25)), <goaway-1>, <settings-1>,
   ↪  <ping-1>, ...]
6
7  # List of mutable symbols and their allowed
8  # mutation types (sequence: 0, content: 1)
9  config.symbol_mutation_types = {<sequence>: 0,
   ↪  <headers-1-content-length-header-name>: 1,
   ↪  <headers-1-content-length-header-value>: 1,
   ↪  <headers-1-transfer-encoding-header-name>: 1,
   ↪  ...}
10
11 # Mutation operators
12 config.sequence_mutators =
13 [(insert_symbol, opts(prob=0.9)), remove_symbol]
14 config.content_mutators =
15 [(insert_char, opts(prob=0.9)), remove_char]
```

Listing 5: Excerpt from a configuration file showing a character pool, a symbol pool, and a list of mutable elements and mutators.

and characters in the character pool (see lines 2 and 15 of Listing 5).

5 Experiments

To understand abnormal HTTP/2-to-HTTP/1 conversions, we conduct two experiments, each with identical configurations, with differing mutations. Table 1 shows general information about both experiments.

5.1 Experimental Setup

Figure 2 shows an overview of the experiment setup. First, an input grammar is determined from which a random base frame sequence is generated. Then the base frame sequence is mutated randomly based on the seed number. Finally, the mutated frame sequence is sent to all HTTP/2 servers in our lab setup, which converts the sequence into an HTTP/1 request. This request is forwarded to a listener server, and ultimately saved to a log file for later analysis.

Table 1: Experiment overview.

Name	Duration	# Inputs	Mutation Types
ONLY-SEQ	15 hours	2,580,000	frame sequence
SEQ-CON	54 hours	6,690,000	frame sequence and content

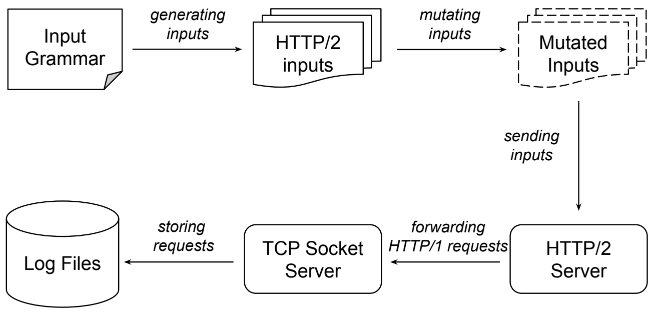


Figure 2: HTTP/2 frame sequences are generated from a grammar, mutated, and sent to the tested server. The TCP socket server receives forwarded requests by the tested server and saves them to log files for later analysis.

We test 12 popular HTTP/2 reverse proxies, including 4 CDNs, using the latest versions available at the time of our experiment. Table 2 details the servers and their versions.

5.2 ONLY-SEQ Experiment

In this experiment, only frame sequence mutations are applied on the HTTP/2 base frame sequences.

During the input generation phase, only semantically valid frame sequences are generated as Listing 6 describes. All sequences are equivalent to a simple HTTP/1 `POST` request with a body, made up of `HEADERS`, `CONTINUATION` and `DATA` frames (one or two from each) coming together to form the HTTP/2 base sequence.

We apply a random number of sequence mutations (in the range of 1 to 4) for each input. Deletion operations easily destroy valid base sequences resulting in more server errors; thus we weigh insertion operations at 90%.

The pool from which a new frame is chosen for insertion contains all ten types of frames defined by the HTTP/2 specification. We set the probability distribution to select one of the following frame types 75% of the time: 1) `HEADERS` 2) `CONTINUATION` and 3) `DATA`. These frames are the most

Table 2: Tested HTTP/2 servers and versions.

HTTP/2 Server	Tested Version
Apache	2.4.51
NGINX	1.21.3
Caddy	2.4.5
Apache Traffic Server (ATS)	9.1.0
HAProxy	2.5-dev10
Varnish	7.0.0
Traefik	2.5.3
Envoy	1.20.0
Akamai	N/A
Cloudflare	N/A
CloudFront	N/A
Fastly	N/A

```
<start> ::= <base-sequence>
<base-sequence> ::= <headers><data> |
 ↪  <headers><data><data> |
 ↪  <headers><cont><data> |
 ↪  <headers><cont><data><data> |
 ↪  <headers><cont><cont><data> |
 ↪  <headers><cont><cont><data><data>
```

Listing 6: Partial grammar showing the possible base sequences.

relevant when it comes to determining the request body.

Unlike the frames in the base sequence, `HEADERS` and `CONTINUATION` frames in the pool have either `content-length` or `transfer-encoding` as one of their headers, and multiple options for method names. This also applies to the `PUSH_PROMISE` frame as it can carry headers. The relevant part of the input grammar is shown in Listing 7. The reason for including additional headers and methods is that they usually have an impact on the request body of the HTTP/1 requests.

Finally, all frames in the pool have been made to support all flags (`END_HEADERS`, `END_STREAM`, `PADDED` and `PRIORITY`) by overwriting the underlying HTTP/2 code library. However, native flags have higher precedence during input generation. The point of building frames with different flag sets is to confuse the stream parsing of the target server.

While base sequences are semantically correct, mutated input sequences are usually not because of the `transfer-encoding` header and unsupported flags added to frames. However, they are still syntactically valid, and therefore they should not cause any frame parsing errors on reverse proxies.

5.3 SEQ-CON Experiment

In this experiment, in addition to frame sequence mutations, frame content mutations are also applied on individual frames. The maximum number for both sequence and content mutations is 2. Thus, the total maximum mutations are capped at 4, the same as the previous experiment.

Content mutations are defined by adding special characters–ASCII characters excluding alphanumeric characters–only at the beginning and end of the `content-length` and `transfer-encoding` header names and values, and the request method. These choices are based on the insights of past research that shows these mutations are critical in request body parsing [16, 17].

5.4 Input Coverage

We explore a random sample of 50,000 inputs for both ONLY-SEQ and SEQ-CON experiments, 100,000 in total, in order to illuminate the main characteristics of inputs that

```
<method-name> ::= POST | GET | HEAD | OPTIONS |
 ↪  TRACE | PUT | DELETE | CONNECT
(..truncated..)
<len-header> ::= <tenc-header> | <clen-header>
<tenc-header> ::= <tenc-name><tenc-value>
<tenc-name> ::= transfer-encoding
<tenc-value> ::= chunked | identity
<clen-header> ::= <clen-name><clen-value>
<clen-name> ::= content-length
<clen-value> ::= 5 | 10 | 15 | 20
```

Listing 7: Partial grammar for the added `HEADERS`-like frame types.

FRAMESHIFTER creates and tests. We analyze a random sample due to computational constraints, yet we argue that it still provides insight into the coverage of our inputs. Figure 3 depicts the distributions for the main characteristics of a request across the input sample. For instance, the "flags" distribution shows that roughly 80% of sample input sequences consist of combinations of only `END_STREAM` and `END_HEADERS`, while 15% contain other flag types (i.e., `PRIORITY`, `PADDED` and `ACK`). The rest contain either `END_STREAM` or `END_HEADERS` flags exclusively.

We also use this sample to shed light on the details of mutations done by the fuzzer in ONLY-SEQ and SEQ-CON experiments. Table 3 shows what mutation operators are applied on what elements with what frequency. For instance, while in 29.4% of sample input sequences a characer is inserted in the `content-length` header, in 95.8% of them a frame is inserted into the input sequence.

6 Findings

After completing the experiments as previously detailed, we remove normal conversions per our definition, and analyze all remaining requests in our log for anomalies. We additionally investigate the originating HTTP/2 input sequences responsible for said anomalies, and report that below. Since the observed anomalies for both the ONLY-SEQ and SEQ-CON experiments overlap considerably, we report them together.

6.1 Conversion Anomalies

We determine 10 types of conversion anomalies and describe them in detail below.

6.1.1 Incomplete Content-Length Without Body

In this category of abnormal conversions, we observe a `content-length` value in the generated HTTP/1 request that is larger than zero yet the request has no body. According to section 3.4 of RFC 7230, if the size of the request body is less than the value given by `content-length`, the request is incomplete [7]. An example is shown in Listing 8 (left side).

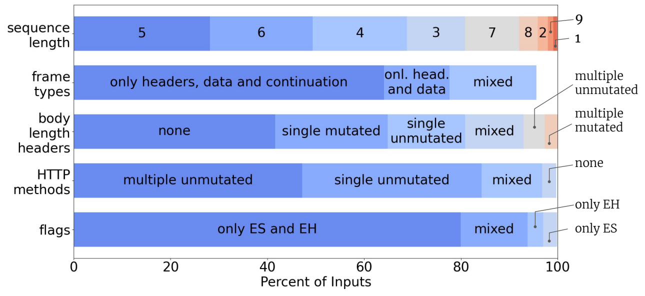


Figure 3: The distribution of sequence lengths (i.e., number of frames), frame types, body length headers (i.e., `content-length` and `transfer-encoding`), HTTP methods and flags (e.g., `END_STREAM`) across the input sample.

6.1.2 Incomplete Content-Length With Body

This anomaly category is very similar to the previous one. The only difference is that the generated HTTP/1 request has a body, but its length is less than the `content-length` value (Listing 8, right side). We separate this category from the previous to allow for the different applications of attacks described later. For example, controlling the request body is often vital for HRS attacks.

6.1.3 Missing Last Chunk

Just like the previous two categories, requests that fall into the "Missing Last Chunk" category are also incomplete. As shown in Listing 9 (left side), the generated request has a `transfer-encoding` and a chunked request body. Yet, it is missing the last chunk which signals the termination of chunked body. According to section 3.4 of RFC 7230, chunked request body is incomplete if the zero-sized chunk (i.e, last chunk) is missing [7].

6.1.4 Missing Chunk Data Termination

In this category, the generated HTTP/1 request lacks not just the last chunk, but also the terminating `CRLF` that signals the end of the chunk data. `transfer-encoding` is present in the request and the body is chunked. An example for this category is the same as the one shown in Listing 9 (left side), except that the `CRLF` in the very end of the body is missing.

```
POST / HTTP/1.1
content-length: 10
```

```
POST / HTTP/1.1
content-length: 10

BBBBB
```

Listing 8: Requests with incomplete bodies.

Table 3: Frequency of mutation operations (the third column does not sum to 100% as multiple mutation operations can be applied on a single input).

Mutation Operator	Mutated Element	% Inputs
insert_symbol	sequence	95.8
remove_symbol	sequence	18.5
insert_character	content-length	29.4
	transfer-encoding	10.1
	HTTP method	12.9

6.1.5 Missing Chunk Data

As the example in Listing 9 (right side) shows, even though chunk-size is present, chunk-data, chunk-data termination, and the last chunk are all missing. Similar to the previous two categories, this category can be classified under incomplete `transfer-encoding` requests. The reason for treating them separately, is again their significance from an attack perspective.

6.1.6 Invalid Header Value

This category refers to requests with an invalid `content-length`. As an example, Listing 10 (left side) shows a non-numeric value given by the `content-length`. In section 3.3.3 of RFC 7230, it is stated that a recipient must respond with a `400 (Bad Request)` status code to a request with a `content-length` header field having an invalid value [7], yet the reverse proxy performs the protocol downgrade anyway.

6.1.7 Invalid Header Termination

In this category, generated requests have a `content-length` header which is terminated by a single `LF` instead of `CRLF`. According to section 3.5 in RFC 7230, the terminator for header fields is the `CRLF` [7] sequence. Even though the same specification also states that a recipient may recognize a single `LF` as a terminator, some HTTP servers do not. For example, Apache HTTP server responds with a `400 (Bad Request)` status code to a request with a `content-length` terminated by a `LF`.

```
POST / HTTP/1.1
transfer-encoding:chunked

5\r\nBBBBB\r\n
```

```
POST / HTTP/1.1
transfer-encoding:chunked

5\r\n
```

Listing 9: Requests with incomplete chunked bodies.

```
POST / HTTP/1.1
content-length: 5&

BBBBB
```

```
POST / HTTP/1.1
content-length: 5
content-length: 10

BBBBB
```

Listing 10: Requests with `content-length` anomalies.

6.1.8 Repeating Header Name

In this category, generated requests have two `content-length` headers with different values. Listing 10 (right side) shows an example for this category. According to section 3.3.3 of RFC 7230, a request with multiple `content-length` header fields having differing values must be treated as an error and the recipient must respond with a `400 (Bad Request)` status code [7]. Yet, in our experiments we still observe reverse proxies forwarding these requests.

6.1.9 Repeating Header Value

"Header value" refers to the `chunked` value of `transfer-encoding`. The requests of this category have a `transfer-encoding` header with two or more `chunked` values (i.e., `transfer-encoding: chunked, chunked`). While RFC 7230 allows multiple transfer coding values in the `transfer-encoding` (for example, `transfer-encoding: gzip, chunked`, to signal that `chunked` and `gzip` encodings have been applied to the request body), section 3.3.1 of the same specification states that a sender must not apply chunked more than once to a request body [7].

6.1.10 Multiple Forwarded Requests

In this category of abnormal conversions, multiple HTTP/1 requests are generated as a result of the conversion. In our experiments, all the frames in the input frame sequence are contained within a single stream (i.e., the stream identifier is 1 for all frames), only a single HTTP/1 request should be generated. In fact, section 2 of RFC 7540 says that each HTTP request/response exchange is associated with its own stream [3].

6.2 Input Categories

We categorize all HTTP/2 inputs that cause the conversion anomalies discussed above in this section. All of these input categories along with the conversion anomalies they cause are shown in Figure 4.

6.2.1 Missing END_STREAM

This category of inputs creates an anomaly where the generated HTTP/1 request is incomplete. This category affects all servers except for Apache, NGINX, and Cloudflare.

For most of the affected servers the input sequence does not have an `END_STREAM` flag. CloudFront is the only server that has slightly different behavior. If the first frames are of `DATA` type and carry the `END_STREAM` flag, CloudFront ignores those frames and considers just the frames that follow.

When the `END_STREAM` is missing, the reverse proxy simply rushes to forward the request assuming that the stream is not finished yet and more is to come.

6.2.2 No Mismatch Check

Similar to the previous category, inputs in this group force Caddy and Traefik to forward an incomplete request.

For valid streams with an `END_STREAM` flag, these servers do not check for a match between a larger `content-length` and the smaller number of bytes it receives in `DATA` frames. As a result, they generate and forward a request where `content-length` value does not match the length of the body.

6.2.3 HEADERS After END_HEADERS

This input category also creates abnormal conversions where the generated request is incomplete. For Caddy, ATS, Varnish, Traefik, and Fastly, when a `HEADERS` frame follows another `HEADERS` frame bearing the `END_HEADERS` flag, this anomaly happens. The reverse proxy halts the stream processing once it encounters this pattern (i.e., a `HEADERS` frame after the `END_HEADERS`) and forwards the request as it stands.

6.2.4 Only First DATA

This is the last input category that generates incomplete requests and it affects ATS, Envoy Proxy, and Fastly. When the payload of a `DATA` frame creates a mismatch between the overall payload size and the `content-length` value, the proxy halts the stream processing and forwards the request as it stands (i.e., until the `DATA` frame which creates the mismatch).

6.2.5 No Mutation Filter

Inputs that fit into this category result in HTTP/1 requests where either the header value or the header termination is invalid. Unsurprisingly, the HTTP/2 input has some non-alphanumeric ASCII character added to the `content-length` value in a `HEADERS` frame.

ATS, Varnish, and Akamai all seem to have insufficient filtering for non-alphanumeric characters. For example, ATS

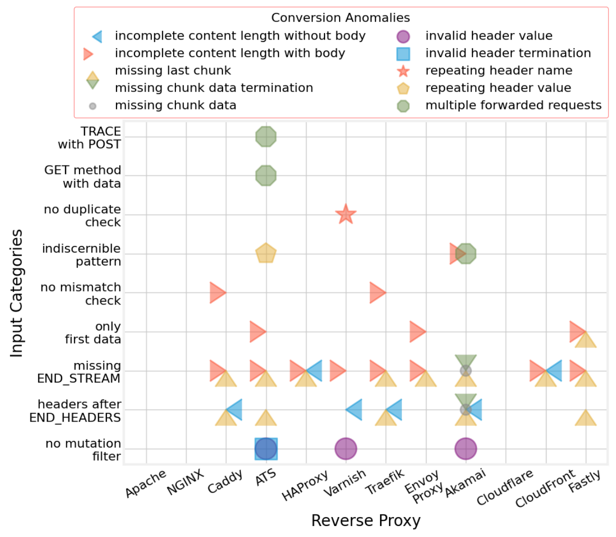


Figure 4: Input categories causing conversion anomalies.

does not filter \n from the input, but Varnish and Akamai do. The preservation of \n by ATS results in an HTTP/1 request having a `content-length` with no value or a `content-length` with an invalid termination.

6.2.6 No Duplicate Check

Requests resulting from inputs in this category contain more than one `content-length` header. The only server this affects is Varnish because they do not check for duplicate `content-length` headers and blindly add them to the generated request.

6.2.7 GET Method with DATA

This input category only contains the sequences with a `HEADERS` frame with the `GET` method followed by a `DATA` frame containing data. We see this pattern in input sequences that cause ATS to forward multiple requests. The same behavior is observed when the method is `HEAD` or `OPTIONS` instead of `GET`.

6.2.8 TRACE with POST

This is another category that makes ATS generate multiple forwarded requests. The input sequences in this category have two consecutive `HEADERS` frames. The first `HEADERS` frame has the `TRACE` method with the `END_HEADERS` flag set. The second `HEADERS` frame has the `POST` method with `END_HEADERS` set again. These two `HEADERS` frames are followed by a `DATA` frame.

6.3 Causes of Anomalies

In this section, we seek to clarify the causes of anomalies in light of direct correspondence with vendors.

6.3.1 Mode of Operation

Reverse proxies have two modes of operation: buffering and streaming. In buffering mode, a proxy waits for the entire client request to complete before forwarding it to the upstream service. In streaming mode, a proxy eagerly transmits requests without waiting for their completion for the sake of memory efficiency and speed.

We observe this in our experiments, particularly on inputs missing `END_STREAM` (i.e., Section 6.2.1). Conversion anomalies listed in Section 6.1.1-6.1.5 (i.e., those creating incomplete requests) can be partly attributed to the streaming mode of reverse proxies forwarding incomplete requests once they receive the `END_HEADERS` flag. Specifically, Akamai, CloudFront, Fastly, Caddy, ATS, HAProxy, Varnish, Traefik, and Envoy Proxy run in streaming mode by default, resulting in their prevalence in Figure 4.

6.3.2 Error Handling

In addition to mode of operation, the way in which reverse proxies handle errors in an input stream contribute to the anomalies discussed in Section 6.1.1-6.1.5. Input patterns discussed in Section 6.2.2-6.2.4 (i.e., "No Mismatch Check", "HEADERS After END_HEADERS" and "Only First DATA") typically trigger an error during stream processing and are handled one of two ways. Reverse proxies can choose to forward the request to the upstream server and close the connection shortly after to signal the error, or they can send an error response to the client and refrain from forwarding a request. When reverse proxies choose to forward the request followed by closing the connection, we find the aforementioned anomalies.

For instance, Caddy and Traefik react to inputs containing a mismatch between `content-length` value and data payload size with an error. As a result, shortly after those reverse proxies forward a request, they close the connection carrying that request by sending a FIN packet. Similarly, ATS raises an error when inputs have the "HEADERS After END_HEADERS" pattern and forwards the request. Fastly and ATS also raise an error for inputs in the "Only First DATA" category and forward requests.

6.3.3 Insufficient Validation

Conversion anomalies listed in Section 6.1.6-6.1.8 can be explained by insufficient validation. There are some cases where irrelevant characters are allowed to be added to sensitive parts of a request (e.g., `content-length` value or the end of a header field). In other cases, there is simply no check in place to prevent duplicate headers with different values. To be more specific, while Varnish does not prevent the presence of two `content-length` header fields with differing values in a forwarded request, Akamai, ATS, and Varnish allow irrelevant characters.

6.3.4 Faulty Retrying

Conversion anomalies listed in Section 6.1.9-6.1.10 can be attributed to faulty behavior of ATS. Specifically, when ATS encounters an input like those explained in Section 6.2.7-6.2.8 (i.e., "GET Method with DATA" and "TRACE with POST"), it triggers an error and forwards the request along but fails to close the connection due to a confirmed bug.

As a result, the connection is kept open and it does not receive a response to the forwarded incomplete request. ATS keeps retrying hoping for a response to send back to its client. It also adds "chunked" value to the `transfer-encoding` in the request in each retry because of another confirmed bug.

7 Attacks

To understand whether our identified HTTP/2-to-HTTP/1 conversion anomalies can be abused, we come up with a list of attacks that can possibly be created by each conversion anomaly. We then test each of these attacks on every possible reverse proxy and origin server pair in a lab environment. We exclude pairs where the reverse proxy is a non-CDN server (e.g., Apache) and a CDN server is upstream (e.g., Akamai), since they are not likely to be deployed in that order in practice.

During our tests, we run a web application on the origin to help us better understand the effects of each attack. We deploy the application directly to the upstream server unless we are testing a pair where the origin is unable to run as a web server.

7.1 Denial-of-Service

The Denial-of-Service (DoS) attack we test for is one in which a mutated frame sequence causes the reverse proxy to send an HTTP/1 request with an incomplete body. The origin server then waits, expecting the remaining data until a timeout occurs. When a new request then arrives at the reverse proxy, it cannot be forwarded to the origin because all persistent connections are exhausted, and so the request cannot be processed in a timely manner.

To test for DoS, we use the following configuration. For each reverse proxy, we compile every mutated frame sequence that resulted in any of the following anomalies: "incomplete content-length with body", "incomplete content-length without body", "missing last chunk," "missing chunk data termination," and "missing chunk data." We choose these anomalies because each of them results in HTTP/1 requests that are missing data. The intuition is that if an origin server is vulnerable to a DoS attack, it will hang while waiting for the rest of the data to arrive.

When the reverse proxy is not a CDN, we configure it to use just one persistent connection to the upstream server. This simplifies the attack detection process, as we do not have to consider the possibility of an attack not working just because a request was processed on a different connection. We later confirm that our detected attacks work with a larger number of persistent connections.

Then, we send the mutated frame sequences one at a time, and send a normal frame sequence like the one in Listing 1 between each mutated frame sequence. This way, if an error occurs in the handling of a normal sequence, then we know that the previous sequence interfered in some way. We wait for a response to arrive or time out after five seconds before sending the next sequence.

When the reverse proxy is a CDN, we do not have control over the number of persistent connections. In these cases, we send the mutated sequences in batches of 50 at a time to the CDN, and send 50 normal sequences in parallel after that. In doing this, we hope that if a mutated sequence would enable an attack, either one of the other mutated sequences or one of the normal sequences would be forwarded on the same port and would be interfered with, allowing us to detect the attack.

While sending the sequences, we collect all TCP traffic between the reverse proxy and the upstream server. Because CDNs may forward HTTP/1 requests strictly over HTTPS, rendering inspection of the traffic useless, we additionally collect the access logs on the upstream server to understand what requests arrived and how the server processed them.

For all reverse proxy and upstream server pairs, we note that normal frame sequences return very quickly, consistently within a fraction of a second. To detect a DoS attack, we flag any mutated frame sequences which take multiple seconds to receive a response or that time out.

To confirm the DoS, we send these flagged sequences and then immediately send a normal sequence without waiting for a response from the first. In the case of CDNs, we send a batch of 256 of the sequences in parallel followed by a single normal sequence. If the normal sequence also takes several seconds to return, then we say that DoS is possible.

As shown in Figure 5, we find that a DoS attack is possible on every upstream server when Caddy, HAProxy, or Envoy Proxy is the reverse proxy, and that the attack is created by anomalies "incomplete content-length with body," "incomplete content-length without body," and "missing last chunk." We additionally find that DoS is possible when Akamai is the reverse proxy and Apache is the upstream server, and is created by all five types of anomalies.

For all affected pairs, an attacker can repeatedly send the mutated frame sequences to make all persistent connections to the origin unresponsive until a timeout occurs between the reverse proxy and origin. Only when this timeout occurs will requests that arrived during this period be served. As the attacker cannot control this timeout value, an attacker likely cannot completely bring down the reverse proxy, but can drastically reduce its throughput depending on how long the timeout duration is.

7.2 Request Blackholing

Another attack type we test for is a Request Blackholing attack. In Request Blackholing, a mutated frame sequence causes the reverse proxy to send an HTTP/1 request with an incomplete body. Instead of the connection between the reverse proxy and origin hanging like in the DoS attack, subsequent forwarded requests here are interpreted as part of the body of the mutated sequence and are never processed correctly by the origin. These requests that are never processed are considered "blackholed."

To test for Request Blackholing, we use the same configuration and frame-sequence testing methodology as in testing for DoS attacks. To detect Request Blackholing, we look for any normal sequences that either never received a response or that received a 400 error code and note the mutated sequence that was sent directly before it. We confirm the attack by sending just that mutated sequence followed by the normal sequence. In the case of CDNs, we send a batch of 256 of the mutated sequences in parallel, followed by one normal sequence. If we see that the normal sequence again either receives a 400 response code or never receives a response, then we say that the attack is possible.

As shown in Figure 5, we find that Request Blackholing is possible when ATS is the reverse proxy and either NGINX or HAProxy is the origin server, and that only anomalies in the category "incomplete content-length without body" make the attack possible.

The fact that only anomalies of this type enabled the attack is significant as it reduces an attacker's capabilities. Depending on how much control an attacker has over the request that enables the attack, a Request Blackholing attack could be used as part of a powerful request hijacking attack.

For example, imagine a website where some page accepts POST requests and where some part of the body of the request is displayed on the page itself, such as the page to edit one's profile on a social media site. If one sends the mutated sequence that results in a Request Blackholing attack as a request to this page, then subsequent requests are interpreted as part of the body and are displayed in plain text on the target page, potentially allowing an attacker to steal cookies and passwords.

However, because the only anomaly type that enabled the attack does *not* have a body, an attacker does not have the ability to send any data that might be required of the request for it to be interpreted correctly by the target application. Thus, in the absence of an "echo" page that displays anything sent in the body, an attacker can only use the Request Blackholing attack to perform a DoS on the affected server pairs.

An additional caveat of the attack is that each mutated sequence allows for the blackholing of just one other request, making the attack symmetric. An attacker can repeatedly send out mutated frame sequences and blackhole other users' requests as fast as they can send them out.

7.3 Query-of-Death

We additionally discovered a Query-of-Death attack that works when Caddy is the reverse proxy. In this attack, the mutated sequence is sent once per persistent connection between Caddy and the origin. Requests are then forwarded until Caddy becomes unresponsive. The Caddy process does not crash, but becomes unresponsive even to control commands, so Caddy must be killed and manually restarted.

As shown in Figure 6, the attack is possible between Caddy and every origin server except for Varnish using anomalies in the categories "incomplete content-length with body," "incomplete content-length without body," and "missing last chunk." We only speculate, but we believe the attack does not work on Varnish because it quickly detects the anomalous HTTP/1 request and fails early, whereas some subsequent communication between the other origins and Caddy causes the attack.

7.4 CPDoS Attack

Cache-Poisoned Denial-of-Service (CPDoS) attacks aim to have a caching server store a negative response (i.e., error response) for a legitimate URI (e.g.,/home) [23]. An attacker must send a malicious request (with the victim URI) that gets forwarded by the caching server to the origin server. When the request reaches the origin server, it triggers an error and eventually the origin returns a negative response. This negative response is saved by the caching server which is now poisoned.

To test for this attack, we send every converted HTTP/1 request captured to each of our twelve servers. We look for request response pairs where the request method is a "cacheable method" and the response status code is "cacheable by default" as defined by RFC 7231 [8].

We find that the "repeating header value" conversion anomaly meets this criteria. Essentially, when ATS generates and sends a `GET` request with `transfer-encoding: chunked, chunked` header to NGINX, Caddy, Traefik or Envoy Proxy, the upstream server responds with `501 Not Implemented` status code.

By default, ATS does not cache negative responses. We enable negative response caching on ATS, put all susceptible servers one by one as the upstream server to the ATS and finally send HTTP/2 frame sequences which create the needed conversion anomaly.

Our attempts to poison the ATS cache all failed. We believe that it is because the abnormal request is not the first request forwarded by ATS. Essentially, ATS forwards multiple requests for a single HTTP/2 input sequence and the poisoning request is the second request forwarded to the upstream server and ATS does not cache the response for a request it does not forward first.

7.5 Response Queue Poisoning

Researchers have shown that it is possible to poison the response queue of a reverse proxy with an additional HTTP response of which the reverse proxy is unaware [5, 18]. This forces the reverse proxy to mix up the request response matching and eventually allows the attacker to retrieve responses for the requests of victim users. Attackers achieve this through smuggling an HTTP request into the request buffer of the upstream server and have it send a response back to the reverse proxy for the smuggled request. As a result, the reverse proxy sends that response for another request and from that point on the response queue is poisoned with an "off by one" error until the underlying connection is killed.

Reverse proxies that are affected by the "multiple forwarded requests" anomaly, send additional requests to upstream servers. None of those requests come from the downstream server (or client) showing a clear potential for Response Queue Poisoning.

In a test environment, we put susceptible reverse proxies before every server separately. We create two different pages on the origin, one for the victim and one for the attacker. The simulated victim continuously sends HTTP/2 requests to the target and the simulated attacker sends HTTP/2 frame sequences which create the "multiple forwarded requests" anomaly. Finally, we look for a case where the victim receives a response to the page requested by the attacker.

In the end, we did not observe the victim user receiving a response intended for the attacker. However, we believe that the outcome could be different in a real-world setup because the number of users and requests in real-world setups is much larger than what we have in this test setup.

7.6 HTTP Request Smuggling

Past research has shown that when the body parsing behavior of a reverse proxy and the upstream server differs in a way that they disagree about the message boundaries, bad things happen [5, 16, 17]. Ultimately one server sees a single request, whereas the other sees two. The "second request" (i.e., smuggled request) can be used for many type of severe attacks from cache poisoning to request hijacking.

Any body parsing difference between two servers in the request chain can be abused for HRS. Many conversion anomalies, especially "invalid header value" and "repeating header name", we document in this paper show a clear potential for causing a difference in body parsing behavior. For "invalid header value", as the example in Listing 10 (left side) shows,

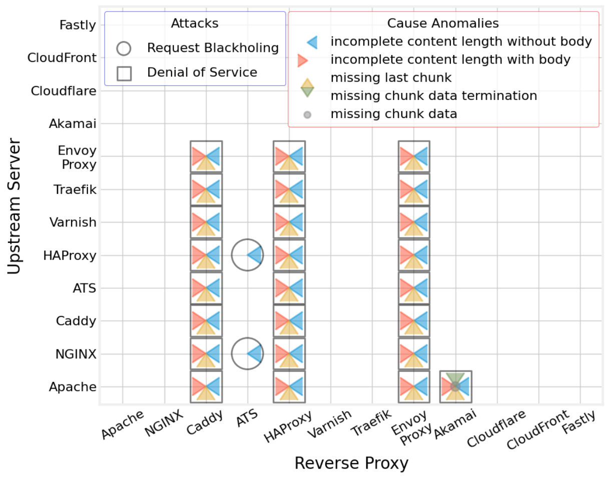


Figure 5: Attacks caused by abnormal conversions.

non-numeric values are forwarded to the upstream server. If an upstream server happens to trim anything not numeric to extract the header value or if it decides to ignore the body as the value is invalid, a real potential for HRS emerges. As an example, if a comma is added to the `content-length` value, previous research identified that some servers parse this differently [16]. The requests in the "repeating header name" (i.e., with two `content-length` headers with different values) category can cause a different body parsing behavior between servers, if one of them chooses the first `content-length` to decide the body size while the other chooses the second.

We take every request (including the ones from the susceptible categories) captured from the forwarding of each reverse proxy and send them to each server and examine the responses. We find that some responses include two response codes signaling that the server sees two requests in what was sent by another server as a single request. We then manually check them to confirm whether it can be used for HRS.

Interestingly, we find that none of the requests which we confirm to have the HRS ability is generated as a result of an abnormal conversion. They usually have a request method or an invalid header name which makes them useful for HRS. Affected pairs are shown in Figure 6. The reasons for each of them is summarized below:

- ATS, HAProxy, Envoy Proxy and Fastly forward a request with `HEAD` method, `transfer-encoding` header and a chunked body. Caddy and Traefik ignore the body in such requests.

- Cloudflare and Fastly forward a request with `GET` or `HEAD` method and a request body. Akamai ignores the body in such requests.

- Akamai forwards a request with `transfer-encoding: identity` having a vertical tab character or a new page character added before the header name, a

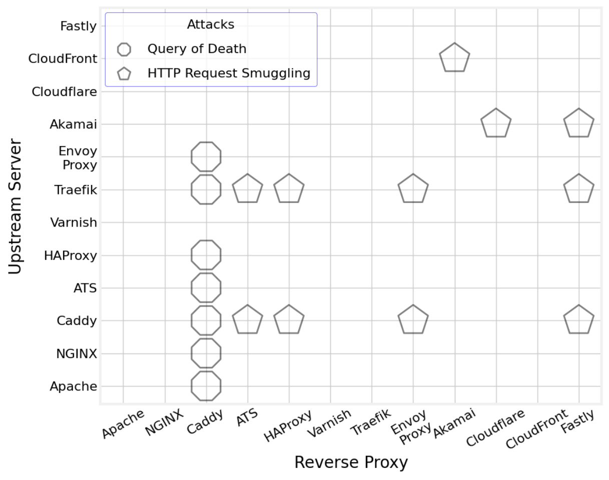


Figure 6: Attacks not caused by abnormal conversions.

`transfer-encoding: chunked` header and a request body. CloudFront ignores the body in such requests.

The HRS case affecting Akamai-CloudFront is interesting from the conversion anomaly perspective. We do not count it as an conversion anomaly, because essentially the reverse proxy can treat it as any header that it does not recognize and forward it to the upstream. It is interesting that CloudFront treats that as a valid header and chooses not to see the response body in a request with two separate `transfer-encoding` headers having different values.

Even though the HRS cases we find in this research are not caused by an abnormal conversion, we think that they are still valuable because they demonstrate one more use case of FRAMESHIFTER. To give an example, FRAMESHIFTER can be used to generate and send a large number of mutated HTTP/2 frame sequences to a server pair where the reverse proxy is an HTTP/2 server and the upstream server is an HTTP/1 server. If the upstream server responds with multiple status codes, it can be concluded that HRS is affecting the tested pair.

8 Discussion & Conclusion

In this paper we set out to explore the HTTP/2-to-HTTP/1 protocol conversion, a near-universal behavior observed with all major CDNs and reverse proxy technologies, from a security viewpoint. In doing so, we successfully met our goals and answered the research questions we laid out in Section 1. Specifically, we presented FRAMESHIFTER and an accompanying HTTP/2 frame manipulation methodology to test popular proxy technologies against conversion anomalies (Q1), systematically explored the causes and effects involved in Section 6 (Q2), and finally translated our findings into concrete web application attacks in Section 7 (Q3).

Before we wrap up, below we highlight the fundamental limitations of our work and provide a high-level analysis of the discovered issues from a systems safety engineering lens.

Limitations. The anomaly discovery phase of this work relies on fuzz testing. While fuzzing has evolved into a de facto method for security analysis, as the name implies, fuzz testing is primarily a *testing* tool. Consequently, the findings we present in this paper are the results of a systematic investigation, but not an *exhaustive* one. That is a fundamental limitation of all fuzzing-based approaches. We make FRAMESHIFTER publicly available in the hopes that the security community expands on it, and that these findings lead to more robust methodologies for analyzing protocol conversion anomalies.

We also point out that, while all attacks we present are practical, real-life exploitation will still be impacted by various factors including proxy configurations, security products deployed on path that can block anomalous requests, and other man-in-the-middle devices that can transform the traffic in unpredictable ways, rendering some attacks ineffective. This is not a limitation of our work per se.

A Systems Safety Problem...with a Different Spin. Recent trends in web application security signal that systems-level attacks are rapidly taking over the more traditional exploitation vectors. Attacks such as Web Cache Deception and HTTP Request Smuggling are harbingers of a new wave of web application security concerns affecting system interactions, rather than individual component resilience. In particular, both Mirheidari et al [21, 22] and Jabiyev et al. [16] explicitly call out that these attacks are a consequence of the increasingly complex *interactions* between Internet infrastructure components (i.e., clients, servers, and proxies), and that there is no particular failing component–this tracks the systems *safety* engineering literature [20].

The issues we present in this paper begin in a similar vein. Foremost, protocol conversion is necessitated due to the existence of competing HTTP versions and conflicting performance & business requirements between the entities involved in the communication. Furthermore, vulnerabilities depend on the exact technologies present on the traffic path, their designs, and their implementation specifics. Therefore, we reiterate the takeaways of prior work: Identifying protocol conversion vulnerabilities and their impact on web applications is not straightforward when systems are analyzed in a bubble. The methodologies, tools, technologies devised to analyze and address these concerns need to consider *all* HTTP processors on the traffic path and their complex interactions. Unfortunately, doing security at this scale still poses many open research questions.

Despite this general view of systems safety, we note that the specific categories of attacks we present in this paper have relatively straightforward mitigations that could directly be implemented on the proxies. In other words, every conversion anomaly we have identified is patchable by the respective vendor, without coordination with the client or origin technology vendors. Thus, our findings represent a more tractable subset of the aforementioned systems safety problem space. However, we point out once again that our fuzzing-based discovery scheme is not exhaustive, and that this observation is not generalizable to all protocol conversion anomalies.

The above observation also implies that conversion anomalies could be minimized with guidance from the relevant protocol specifications, in an effort to standardize this mechanism among different technologies and avoid the most common pitfalls. In fact, the HTTP/2 protocol specification RFC 7540, under the section "Security Considerations, Intermediary Encapsulation Attacks" briefly touches on similar attack vectors, but does not go into details [3]. While standardization is not the panacea for this issue, we believe there is significant room for improving the state of the art by providing formal HTTP/2-to-HTTP/1 conversion guidelines.

Ethical Considerations

We have conducted this study within a controlled experimental setup. We did not launch any attacks against external entities. We followed the established coordinated-disclosure best practices; we notified all tested technology vendors of our findings, provided them with a copy of this paper, and made our data and team available for further assistance.

The vendors have acknowledged the impact of our reported issues, and we have coordinated with them on implementing the appropriate mitigations. Apache Traffic Server confirmed the Request Blackholing issue and all the anomalies we reported; they are planning to assign the appropriate CVEs and have patches ready in an upcoming release. Envoy Proxy confirmed the DoS attack and discovered a gap in their DoS protection. Varnish also confirmed our finding and reported they would have a patch in their next release. Caddy requested us to report the findings to their underlying Go HTTP library, developed and maintained by Google. Google confirmed the Query-of-Death attack; they will have a patch in the next release and a CVE assigned for the issue. One of the authors of this work is affiliated with Akamai, and has coordinated the fixes internally with the vendor. The remaining vendors acknowledged receiving our report, but did not provide information about the remediation actions they took.

Acknowledgments

The authors would like to thank our anonymous reviewers and our shepherd Peter Snyder for their feedback and guidance. This work was partially-funded by the National Science Foundation grants CNS-1703454 and 2127200.

References

[1] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. NAUTILUS: Fishing for Deep Bugs with Grammars. In *The Network and Distributed System Security Symposium*, 2019.

[2] Yaron Azerual. HTTP/2 Will Break Your Security – Here's How to Fix it, 2015. https://blog.radware.com/security/2015/09/http2-security-fix/.

[3] Mike Belshe, Roberto Peon, and Martin Thomson. Hypertext Transfer Protocol Version 2 (HTTP/2), 2015. https://datatracker.ietf.org/doc/html/rfc7540.

[4] BuiltWith. BuiltWith Technology Lookup. https://trends.builtwith.com/CDN/Content-Delivery-Network.

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

[6] Maxim Dounin. HTTP/2 Gateway. NGINX Mailing List, 2015. https://mailman.nginx.org/pipermail/nginx/2015-December/049445.html.

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

[8] Roy T. Fielding and Julian F. Reschke. Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content, 2014. https://www.rfc-editor.org/rfc/rfc7231.

[9] Ivan Fratric. Domato. GitHub Repository, 2021. https://github.com/googleprojectzero/domato.

[10] Andrew Galloni, Robin Marx, and Mike Bishop. HTTP/2, 2020. https://almanac.httparchive.org/en/2020/http.

[11] Tom Van Goethem, Christina Pöpper, Wouter Joosen, and Mathy Vanhoef. Timeless Timing Attacks: Exploiting Concurrency to Leak Secrets over Remote Connections. In *USENIX Security Symposium*, 2020.

[12] Run Guo, Jianjun Chen, Baojun Liu, Jia Zhang, Chao Zhang, Haixin Duan, Tao Wan, Jian Jiang, Shuang Hao, and Yaoqi Jia. Abusing CDNs for Fun and Profit: Security Issues in CDNs' Origin Validation. In *IEEE International Symposium on Reliable Distributed Systems*, 2018.

[13] Run Guo, Weizhong Li, Baojun Liu, Shuang Hao, Jia Zhang, Haixin Duan, Kaiwen Sheng, Jianjun Chen, and Ying Liu. CDN Judo: Breaking the CDN DoS Protection with Itself. In *The Network and Distributed System Security Symposium*, 2020.

[14] Christian Holler, Kim Herzig, and Andreas Zeller. Fuzzing with Code Fragments. In *USENIX Security Symposium)*, 2012.

[15] Bahruz Jabiyev. Grammar-based HTTP/2 fuzzer with mutation ability, 2022. https://github.com/bahruzjabiyev/frameshifter.

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

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

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

[19] Emil Lerner. http2smugl: HTTP2 request smuggling security testing tool, 2021. https://lab.wallarm.com/http2smugl-http2-request-smuggling-security-testing-tool/.

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

[21] Seyed Ali Mirheidari, Sajjad Arshad, Kaan Onarlioglu, Bruno Crispo, Engin Kirda, and William Robertson. Cached and Confused: Web Cache Deception in the Wild. In *USENIX Security Symposium*, 2020.

[22] Seyed Ali Mirheidari, Matteo Golinelli, Kaan Onarlioglu, Engin Kirda, and Bruno Crispo. Web Cache Deception Escalates! In *USENIX Security Symposium*, 2022.

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

[24] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and Understanding Bugs in C Compilers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2011.

[25] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. Fuzzing with Grammars. The Fuzzing Book, 2022. https://www.fuzzingbook.org/html/Grammars.html.