Funky chunks – addendum: a few more dirty tricks

Funky chunks – addendum: a few more dirty tricks - Jeppe Bonde Weikop, w4ke.info.

- Published: 2025-10-29
- Original: <https://w4ke.info/2025/10/29/funky-chunks-2.html>
- Preserved from: https://w4ke.info/2025/10/29/funky-chunks-2.html (live) 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.

After revisiting my own recent article introducing a small family of request smuggling techniques, I was struck by the realization that I had not quite drawn the family tree to completion. There are still a few branches left to trace – close relatives that until now have escaped our attention. To remedy this oversight, I have put together this short addendum in which we will finally make the proper introductions and welcome these neglected smuggling techniques into the family.

In the interest of brevity, I will not include an introduction here. To understand the context of this article, you will therefore need to read the original one first.

The curious case of the two-byte terminator

In *Funky chunks: abusing ambiguous chunk line terminators for request smuggling*, we surveyed a series of HTTP/1.1 chunked-body parsing leniencies. One of them, mentioned only briefly, now turns out to be of a fundamentally different nature than the others. In fact, as we will soon see, this particular leniency unlocks an entirely new subclass of chunk-based request smuggling techniques.

The leniency in question is the following: *accepting any two bytes as the line terminator of a chunk body*. A parser affected by such a leniency would interpret the highlighted `XX` sequence as a line terminator in the example chunked body below.

```
d\r\n
Hello, world!XX
0\r\n
\r\n
```

This is a fairly common quirk, presumably because *only* the sequence `\r\n` is valid in this location. Many parsers simply skip two bytes, not bothering to confirm that the skipped sequence is in fact

a CRLF. This behavior is (or rather, *was*) exhibited by parsers such as h11, uHTTPd, and even older versions of llhttp.

Now, recall another common leniency in chunk body parsing: *accepting a lone* `\n` *as a line terminator*, a technically incorrect yet highly prevalent behavior. Perhaps you already see where this is going.

The vulnerability

If either the front-end proxy or the back-end server assumes a two-byte CRLF without checking it, and the other accepts `\n` (or any other one-byte or zero-byte sequence) as a line terminator, **chunk boundaries begin to blur**. To see this, consider what happens when a chunk body with a one-byte line terminator is processed by a parser that carelessly advances two bytes after each chunk body. The parser will inadvertently consume a byte from the subsequent chunk header, effectively corrupting the chunk size. This causes the front-end and back-end parsers to disagree on the size of the next chunk, thereby enabling – *you guessed it* – HTTP request smuggling.

I see two variants of this new length-based technique:

- **Front-end overread**: The proxy interprets any two-byte sequence as a line terminator, and the server accepts either some one-byte line terminator (e.g. `\n`) or no line terminator at all.
- **Back-end overread**: The proxy accepts either some one-byte line terminator (e.g. `\n`) or no line terminator at all, and the server interprets any two-byte sequence as a line terminator.

It is worth noting that the parsing leniencies we are exploiting here are not actually any different from the ones described in my original blog post – these are just additional ways of combining leniencies to obtain a request smuggling primitive.

Example: 1-byte front-end overread

To keep things short, we'll discuss only one of these variants in depth. I trust that you, dear reader, will be able to construct an equivalent attack for other variants, should the need arise.

Let us then consider the arguably most plausible scenario: a front-end accepting `\n` as a line terminator and a back-end accepting any two-byte sequence.

```
GET /one HTTP/1.1\r\n**request #1**
Host: localhost\r\n
Transfer-Encoding: chunked\r\n
\r\n
2;\r\nchunk header
xx\nchunk body
50\r\nchunk header
\r\nchunk body
GET /two HTTP/1.1\r\n
Host: localhost\r\n
X-Pad: AAAAA\r\n
Transfer-Encoding: chunked\r\n
\r\n
0\r\nlast chunk
\r\n
```

Proxy interpretation

```
GET /one HTTP/1.1\r\n**request #1**
Host: localhost\r\n
Transfer-Encoding: chunked\r\n
\r\n
2;\r\nchunk header
xx\n5chunk body
0\r\nlast chunk
\r\n
GET /two HTTP/1.1\r\n**request #2**
Host: localhost\r\n
X-Pad: AAAAA\r\n
Transfer-Encoding: chunked\r\n
\r\n
0\r\nlast chunk
\r\n
```

Server interpretation

The front-end interprets the `\n` as a line terminator and `50` as the size of the second chunk. On the back-end, the first byte of the second chunk size is consumed by the server, assuming it to be part of the line terminator. This changes the perceived size of the second chunk from `50` to `0`, causing the server to interpret it as the end of the request. What the front-end considers the content of the second chunk is therefore interpreted as a second pipelined request on the back-end.

Funky trailers

We now move on from chunks and chunk sizes and instead turn our attention to another notable feature of HTTP/1.1 chunked encoding, a feature that we foolishly ignored in the original article despite its clear applicability to our request smuggling endeavors: the chunked trailer section.

The trailer section is essentially an optional header section following the last chunk of an HTTP message using chunked encoding. Let's get familiar with the syntax by taking a look at an example.

```
POST /some/path HTTP/1.1\r\n
Host: example.com\r\n
Content-Type: text/plain\r\n
Transfer-Encoding: chunked\r\n
\r\n
d\r\nchunk header
hello, world!\r\nchunk body
0\r\nlast chunk
Trailer-One: value-one\r\ntrailer section
Trailer-Two: value-two\r\n
\r\n
```

In parsing the trailer section, I've noticed two common approaches.

The first approach is to reuse the parsing logic for the header section. From a programmer's perspective, this is a sensible choice – surely, one should not implement the same parsing logic twice! Unfortunately, there is a subtle but important difference between the headers and the trailers: *a lone newline is **not** an acceptable line terminator in the chunked trailer section.* As you may imagine, many parsers ignore this nuance and interpret a single `\n` as a line terminator in the trailer section anyway.

The second approach is to treat the trailer section much like the chunk extensions: consume it with no regard for its contents. This might seem like odd behavior, but it is a perfectly valid choice; the trailer section is optional metadata and recipients are allowed to discard it. Parsers employing this approach often look only for the `\r\n\r\n` sequence that marks the end of the trailer section, effectively (and erroneously) allowing any byte – including lone `\n` and `\r` characters – within the section.

These observations lead us to a brand-new set of exploitable parsing leniencies: by placing what one parser interprets as two consecutive line terminators in what another parser interprets as the trailer section, we have once again stumbled upon a new flavor of chunk-based request smuggling.

TRAIL.TERM

Consider first the scenario in which the front-end proxy ignores lone `\n` characters in the chunked trailer section, but the back-end web server interprets them as line terminators. In such a scenario, we can smuggle a request past the front-end using the surprisingly simple payload below.

```
GET /one HTTP/1.1\r\n**request #1**
Host: localhost\r\n
Transfer-Encoding: chunked\r\n
\r\n
2\r\nchunk header
xx\r\nchunk body
0\r\nlast chunk
\ntrailer section
GET /two HTTP/1.1\r\n
Host: localhost\r\n
\r\n
```

TRAIL interpretation (proxy)

```
GET /one HTTP/1.1\r\n**request #1**
Host: localhost\r\n
Transfer-Encoding: chunked\r\n
\r\n
2\r\nchunk header
xx\r\nchunk body
0\r\nlast chunk
\n
GET /two HTTP/1.1\r\n**request #2**
Host: localhost\r\n
\r\n
```

TERM interpretation (server)

The proxy ignores the lone newline following the last chunk, interpreting it as part of the trailer section. It perceives the remaining data – which the back-end interprets as a second request – as a chunked trailer section. Conveniently, the last two consecutive CRLF sequences serve both as the termination of the trailer section and the second pipelined request.

***Note**: The first chunk is not strictly needed, but experience has taught me that some proxies rewrite requests with an empty body. The first chunk serves to prevent this rewriting behavior.*

TERM.TRAIL

As it turns out, the TERM.TRAIL scenario is quite a bit more complicated. Before we deep-dive into why, let's first think about how we may construct an equivalent to the TRAIL.TERM payload above. In doing so, we quickly realize that we cannot *'split'* a request as we usually would, because the *'split'* can only occur on the front-end; once we add the ambiguous line terminators, the front-end interprets them as the end of the request. We have no way of splitting the request on the back-end instead.

There is a workaround, though: we can use *two* requests. This would perhaps more accurately be described as *'request joining'* rather than *'request splitting'*, because what the front-end perceives as two separate requests is squashed into a single request on the back-end – not the other way around.

```
GET /one HTTP/1.1\r\n**request #1**
Host: localhost\r\n
Transfer-Encoding: chunked\r\n
\r\n
2\r\nchunk header
xx\r\nchunk body
0\r\nlast chunk
\n
GET /two HTTP/1.1\r\n**request #2**
Host: localhost\r\n
Content-Length: 40\r\n
\r\n
GET /three HTTP/1.1\r\nrequest body
Host: localhost\r\n
\r\n
```

TERM interpretation (proxy)

```
GET /one HTTP/1.1\r\n**request #1**
Host: localhost\r\n
Transfer-Encoding: chunked\r\n
\r\n
2\r\nchunk header
xx\r\nchunk body
0\r\nlast chunk
\ntrailer section
GET /two HTTP/1.1\r\n
Host: localhost\r\n
Content-Length: 40\r\n
\r\n
GET /three HTTP/1.1\r\n**request #2**
Host: localhost\r\n
\r\n
```

TRAIL interpretation (server)

Using our two-request technique, it seems we yet again have managed to hide a request from the front-end parser. The back-end sees a trailer section where the front-end sees a second request,

and as a result, the `Content-Length` header is ignored and the body of the second request is interpreted as a separate request on the back-end.

There is one major problem, however.

The early-response problem

Consider what happens when a proxy receives these two pipelined requests. It will initially only forward what it interprets as the first request, which in turn is interpreted as an incomplete request on the back-end. Since the request is incomplete, the back-end will not return a response, and the proxy will eventually time out and therefore never forward the second request – the attack fails.

Until recently, I had dismissed TERM.TRAIL as unexploitable due to this inevitable upstream connection timeout. I later discovered that it *is* in fact exploitable against a small subset of web servers like AIOHTTP, Koa, and Actix Web, which respond **before receiving the request body** (unless the body is explicitly read by the application). Shortly after this realization, James Kettle introduced the concept of an *early-response gadget* in his 2025 HTTP desync research, proving that even servers like nginx and IIS exhibit early-response behavior when rubbed the right way. We may therefore conclude that TERM.TRAIL *is* exploitable – with the added caveat that an early-response gadget is required.

Although Kettle's work on early-response focused on 0.CL vulnerabilities, the idea is equally applicable to our TERM.TRAIL case; if the back-end responds early, the proxy will forward the second request, allowing the smuggled request to be delivered. It's worth noting that unlike in 0.CL exploitation, here we do not have to worry about the lengths of any request headers added by the front-end.

Any more bounties...?

Armed with our newfound knowledge, it is only natural to wonder whether any more bounties or CVEs might be unearthed using these techniques. Unfortunately, the yields have been rather underwhelming.

Since the length-based techniques are only exploitable against parsing behaviors that I have already demonstrated to be dangerous, there are no additional CVEs to be issued. Scanning for these vulnerabilities across a range of bug bounty targets sadly met with little success, perhaps partly as a result of my having reported these vulnerabilities to a dozen projects months ago.

Regarding trailer-based techniques, TRAIL.TERM remains a theoretical vulnerability, as none of the proxies I've tested exhibited the required parsing behavior. I did identify multiple TERM.TRAIL-vulnerable setups, including even a couple of real-world instances in bug bounty targets, but I was unable to find the necessary early-response gadgets in most cases. The only exploitable setup I did find was AIOHTTP behind Akamai, Imperva, or Google Classic Application LB, which has now been fixed. Google Cloud even awarded a generous $13,337 bounty for the parsing flaw in their load balancer.

Safe to say, these vulnerabilities are by no means as prevalent as the ones discussed in *Funky chunks*. Nonetheless, I found them interesting to include in this addendum. If you wish to go

looking for these vulnerabilities in the wild yourself, I've updated smugchunks with more blind detection payloads.