Sibuna Discussions SID 0009

Chunked Request Bodies: Strict In-Place Decoding, Canonical Re-Framing, and Inspection Equivalence

STATE
committed
TYPE
Architectural Specification
CREATED
2026-10-02
STATUS
Committed
AUTHORS
Sibuna Contributors <team@sibuna.local>
UPDATED
2026-10-16

SUMMARY
Specifies how the reverse proxy accepts chunked request bodies without a heap allocation or a second buffer: a strict RFC 9112 chunk grammar that rejects every known terminator and extension ambiguity, an in-place decoder whose output never overtakes its input, Content-Length forwarding for bodies that complete within the connection buffer and canonical re-chunking for the rest, and proofs that inspection sees exactly the bytes a Content-Length request would show and that the origin cannot observe the client's framing.

DISCUSSION
No external discussion URL assigned

LABELS
http, proxy, security, request-smuggling, performance

Record status

The catalog defines the review process and the meaning of each state. This numbered record forms part of the catalog.

Contents

1. Revision note 2
2. Abstract 3
3. Introduction 3
4. Terminology and Scope 3
5. Problem Statement 4
6. Goals and Non-Goals 4
7. State of the Art 4
8. Protocol Rules 5
9. Detailed Design 6
10. Verification and Testing 9
11. Security Considerations 9
12. Rollout and Operational Plan 9
13. References 10

1. Revision note

2026-10-15: Ported from the Typst original in the Zig repository. The record describes the Zig implementation as it was decided; SID 0010 maps its packages to the Rust crates of the port.

2026-10-15 (Rust port): the framing rules R1 to R10, the decoder of Definition 1 and the admission phase are unchanged in `sibuna-http` and the daemon: admission still decodes in place in the connection buffer, and every accepted and refused input is the same. The streaming phase of Forwarding differs. Once the decoded prefix has been sent as the first chunk, the proxy (`sibuna-net`) reads further client bytes into a 16 KiB upload buffer, decodes them in place there, and sends each read as one canonical chunk; Zig decoded later reads inside the connection reader's buffer. Bytes still pending in the connection buffer when streaming begins are copied once into the upload buffer. The reason is ownership: the proxy borrows the parsed request, whose slices point into the connection buffer, so it must not refill that buffer, and the compiler now proves that the request is never read after a refill. The upload buffer is allocated once per connection, together with the origin read and write buffers, so a proxied request in the steady state still allocates nothing (the `steady_state` test of `sibuna-net` counts the allocations of 500 keep-alive requests). Goal 4's "no second per-connection buffer" no longer holds literally, because the upload buffer is such a second buffer. Lemma 2 applies to the upload buffer as it did to the connection buffer. Corollary 7 is unchanged, because the origin framing still depends only on B and on Sibuna's reads, but an origin chunk of the streaming phase now carries at most 16 KiB.

2026-10-15 (security review, Rust port): an origin connection that carried a request body, `Content-Length` framed or re-chunked, is closed after its exchange and never returns to the pool. Sibuna cannot observe whether the origin consumed the body as framed. An origin that ignores the body reads it as a request of its own and answers it later on the same socket, and a pooled socket would deliver that answer to another client (the CL.0 desynchronisation). Corollary 7 makes Sibuna's framing unambiguous, but it cannot make an origin read it. The cost is one origin connection per request with a body. The slow-body defence now has a counterpart on the response side: a client must accept at least one byte per idle period (a socket write timeout), so a stalled reader cannot hold a connection that a stalled writer could not.

2026-10-15 (Rust port, security fix): the request-target joins the protocol rules. A target that Sibuna and the origin read differently is a parser differential of the same kind as ambiguous framing. The Zig parser accepted every target form, the engine matched the path as sent, and the proxy forwarded it verbatim. Origins route on a decoded, merged and dot-resolved path, so a denied path reached the origin under another spelling (SID 0003, revision of 2026-10-15).

Sibuna now accepts only origin-form (`/` followed by a path and an optional query) and the asterisk-form `*` in an `OPTIONS` request (RFC 9112 section 3.2.4). Absolute-form (`http://host/path`) and authority-form (`host:port`) are answered 400 and the connection is closed. The check runs after every other head rule, so a head the Zig parser refused keeps its error. Sibuna is a reverse proxy, not a forward proxy. RFC 9112 section 3.2.2 asks a server to accept absolute-form, but clients send it only to forward proxies, and accepting it would force a choice between the target's authority and `Host`.

The path is then put in a normal form, once, by `sibuna_http::normalize_path`, which writes into a caller buffer without allocating: (1) every byte is visible ASCII; a control byte, space, DEL, a byte above 126, `#` or `?` is refused; (2) a `%` must start two hexadecimal digits; `%2F`, `%5C` (any case) and `%00` are refused; an escape of an unreserved character (ALPHA, DIGIT, `-`, `.`, `_`, `~`) is decoded; every other escape is kept, with upper-case digits; (3) a raw `\` is refused; (4) a `;` is refused anywhere in the path; (5) empty segments are dropped, so repeated slashes merge, and a trailing slash is kept; (6) dot segments are removed as RFC 3986 section 5.2.4 does, except that a `..` above the root is refused instead of dropped. The result is never longer than the input, and normalising it again changes nothing. A path without a normal form is answered 400 and the connection is closed. The normal path is the path policy evaluates and the path forwarded to the origin. The query is forwarded unchanged. `sibuna-net` refuses to write a request head whose path is not in normal form. The original URI of forward authorization (`X-Forwarded-Uri`, `X-Original-URI`) and the URL the interstitial reports are normalised the same way; a path without a normal form is answered 400.

`;` is refused rather than stripped because origins disagree on it: Tomcat, Jetty and Spring read it as the start of path parameters (`/admin;x/y` routes to `/admin/y`, and `..;` climbs), while nginx, Express, Go, Django and Rails read it as a literal byte. Stripping would forward a path the origin did not ask for, or judge one the origin does not route on. Refusal is right for every origin; Spring Security's `StrictHttpFirewall` refuses `;`

by default, and its main producer, `;jsessionid` URL rewriting, serves only clients without cookies, which cannot hold Sibuna's session cookie anyway. An origin that decodes twice (`%2561` read as `a`) is outside this model. The differential vectors of the Zig parser keep their verdicts except 75 accepted vectors, which are now refused and listed with their reasons in the test: 64 absolute-form, 4 asterisk-form with a method other than `OPTIONS`, and 7 other targets.

2. Abstract

Specifies how the reverse proxy accepts chunked request bodies without a heap allocation or a second buffer: a strict RFC 9112 chunk grammar that rejects every known terminator and extension ambiguity, an in-place decoder whose output never overtakes its input, Content-Length forwarding for bodies that complete within the connection buffer and canonical re-chunking for the rest, and proofs that inspection sees exactly the bytes a Content-Length request would show and that the origin cannot observe the client's framing.

Sibuna's reverse proxy answers every request that carries `Transfer-Encoding` with 400. The request parser returns `UnsupportedTransferEncoding` (`libs/net/src/http.zig`), so a valid HTTP/1.1 upload of unknown length never reaches the origin. Clients that stream a body produce exactly such requests: `curl -T -`, Go's `net/http` with an unsized reader, Java's chunked streaming mode, `fetch` with a stream body, and git, Docker registry and S3-compatible clients. For a proxy that claims transparency this is a compatibility gap, and forward-auth mode is not a substitute because the ingress never shows Sibuna the body.

This record specifies the fix. A strict chunk grammar rejects every terminator and extension ambiguity behind the request-smuggling disclosures of 2024 to 2025. An in-place decoder turns the chunked coding back into the body inside the connection's existing 64 KiB buffer, without a second buffer or a heap allocation. A body that completes within the buffer reaches the origin with `Content-Length`. A longer body is re-chunked canonically, one chunk per read. The record proves three properties. Decoding is safe in place. The policy engine and the WAF inspect exactly the bytes they would inspect for the same body sent with `Content-Length`. The origin cannot observe the client's chunk sizes, extensions or trailers, so no parser differential between Sibuna and the origin can exist. Requests without a body, and requests with `Content-Length`, keep their current code path.

3. Introduction

HTTP/1.1 delimits a request body in one of two ways: a `Content-Length`, or the chunked transfer coding when the sender does not know the length in advance (RFC 9112 Section 6.1, Section 7.1). Every HTTP/1.1 recipient must be able to parse chunked coding (RFC 9112 Section 7.1). Sibuna does not parse it today.

The rejection was deliberate. Chunked coding is where request smuggling lives. Watchfire's original 2005 paper named the CL.TE and TE.CL desynchronisations. Kettle's desync research (2019, 2021, 2022, 2025) turned them into a methodology. Weikop's "Funky chunks" (2025) showed that a lone LF or CR in a chunk line, or a chunk extension parsed differently by two hops, splits one request into two. Those reports led to fixes in Apache Traffic Server (CVE-2024-53868), aiohttp (CVE-2024-52304), Go `net/http` (CVE-2025-22871), h11 (CVE-2025-43859), ASP.NET Core Kestrel (CVE-2025-55315), Netty and Jetty. Refusing the coding was safe but incomplete. This record replaces the refusal with a decoder whose safety is argued, not assumed.

4. Terminology and Scope

Table 18 Terms used in this record

Term	Meaning in this record
Raw stream x	The bytes the client sends after the request head.
Body B	dechunk (x): the concatenated chunk data, as defined by the decoding algorithm of RFC 9112 Section 7.1.3.
Chunk line	A chunk-size line (size and optional extensions) or a trailer field line, each terminated by CRLF.

Term	Meaning in this record
C, H, L, P	Connection buffer capacity (65,536 bytes), length of the request head ($\leq$ 16,384), maximum chunk line length (4,096), inspected body prefix (`body_fields.max_prefix`, 8,192).
Admission	Everything before the policy decision: head parse, body prefix, rules and WAF.
Canonical chunked	Lower-case hexadecimal size without leading zeros, CRLF, data, CRLF; no extensions; terminated by `0` CRLF CRLF with no trailer fields.

In scope: request bodies in reverse-proxy mode, and the same admission decoding in forward-auth mode and for Sibuna's internal endpoints. **Out of scope:** transfer codings other than `chunked`, compressed content inspection (unchanged: compressed payloads stay opaque), the response direction (origin chunked responses are already relayed by `relayChunked`), and HTTP/2.

5. Problem Statement

`parseRequest` records whether `Transfer-Encoding` is present and returns an error whenever it is. The error becomes a 400 "Malformed HTTP request". The surrounding code also assumes that a body is described by `Content-Length` alone:

- `serveOne` buffers `min(declared, C - H)` body bytes;
- `RequestContext.declared_body` drives keep-alive and the verify endpoint's size check;
- `proxy.writeHead` re-emits `Content-Length` from the request header;
- `proxy.exchange` relays `declared - buffered` further bytes;
- `streamProxy` decides retry safety from the same two numbers.

A correct fix must therefore give the body an explicit framing that every one of these sites reads, rather than add a special case to one of them.

6. Goals and Non-Goals

6.1. Goals

1. Accept every chunked request that RFC 9112 permits, except the ones that rules R1 to R7 below reject.
2. Reject every input on which two RFC-adjacent parsers are known to disagree: lone CR or LF, malformed extensions, whitespace around sizes, size overflow, and malformed trailers.
3. Ensure the policy engine and the WAF cannot be evaded by splitting a payload across chunks.
4. Keep the request path free of heap allocation and of a second per-connection buffer.
5. Leave requests without a body, and `Content-Length` requests, on their current code path.
6. Bound decoding work and memory per connection, and keep the existing slow-body defence.

6.2. Non-Goals

- Spooling bodies to disk to give every upload a `Content-Length` (nginx's `proxy_request_buffering`). Sibuna has no disk on the request path.
- Forwarding request trailers. RFC 9110 Section 6.5.1 lets a recipient that removes the chunked coding discard them. Discarding also keeps uninspected fields away from the origin.
- Stricter validation of origin responses, which would risk breaking existing origins.

7. State of the Art

Table 19 Request-body handling in comparable systems

System	Request-body behaviour
nginx	Decodes chunked input. With `proxy_request_buffering on` (the default) it spools the whole body to memory or disk and forwards `Content-Length`; with it off, it re-chunks to an HTTP/1.1 upstream.
HAProxy	Parses and normalises chunked coding. `option http-buffer-request` waits for the body up to one buffer before the request is processed, so ACLs see a decoded prefix.

System	Request-body behaviour
Envoy, Pingora	Decode in the codec and re-encode for the upstream. The upstream framing is generated, never copied.
ModSecurity, Coraza	Inspect the de-chunked body up to `SecRequestBodyLimit`, either rejecting or processing partially beyond it. Rules that inspect raw chunked bytes are a well-known evasion path.
Hardened parsers after the 2025 disclosures	Accept CRLF only, reject lone CR/LF anywhere in a chunk line, parse extensions strictly (including quoted strings), bound line length and size digits.

The common lesson is that a proxy must **re-frame**: it decodes with a strict parser and emits framing it generated itself. A proxy that copies the client's framing verbatim lets the origin's parser make a second, possibly different, decision. Sibuna combines HAProxy's bounded buffering of the prefix that admission needs, nginx's `Content-Length` hand-off when the body is complete, and Envoy's generated upstream framing. It does this inside the buffer that already holds the request head.

8. Protocol Rules

Rule R1

`Transfer-Encoding` is accepted only in an HTTP/1.1 request, in exactly one field line whose value, trimmed of optional whitespace, is `chunked` (case-insensitive).

Rule R2

A request with both `Transfer-Encoding` and `Content-Length` is answered 400 and the connection closed. RFC 9112 Section 6.3 permits this, and the parser already does it.

Rule R3

A coding list whose final coding is not `chunked` is answered 400 and closed (RFC 9112 Section 6.3, a MUST). A list that ends in `chunked` after another coding, such as `gzip, chunked`, is answered 501 and closed (RFC 9112 Section 6.1, a SHOULD). The latter would need decompression before inspection. More than one `Transfer-Encoding` line is answered 400.

Rule R4

An HTTP/1.0 request with `Transfer-Encoding` is answered 400 and closed: RFC 9112 Section 6.1 requires faulty-framing handling for it.

Rule R5

A chunk line ends at its first LF. The byte before that LF must be CR. No other byte of the line may be CR, LF, or any control byte except HTAB. The line, including CRLF, may be at most $L = 4096$ bytes. A size is 1 to 16 hexadecimal digits with no sign, prefix or surrounding whitespace. Extensions follow the RFC 9112 Section 7.1.1 grammar exactly: optional whitespace, `;`, a token name, and optionally `=` with a token or a quoted string whose quoted pairs escape only visible characters, space or HTAB.

Rule R6

Chunk data must be followed by exactly CRLF.

Rule R7
Trailer field lines must satisfy the header grammar (token name, no whitespace before the colon, no control bytes in the value). The trailer section may total at most 16 KiB. Trailers are discarded.

Rule R8
A body that completes within the connection buffer is forwarded with `Content-Length: |B|`. Otherwise the decoded prefix and every later read are forwarded as canonical chunks.

Rule R9
A framing error found after forwarding began aborts the origin exchange without the terminal chunk. The origin sees a truncated body, never a complete altered one. The client receives 400 and the connection is closed.

Rule R10
An upgrade request with a body coding is refused, as one with a non-zero length already is. `Expect: 100-continue` is answered for chunked requests. The client connection is reused after a local response only when the body was fully consumed.

Rule R11 (2026-10-15, Rust port)
The request-target is origin-form, or `*` in an `OPTIONS` request; any other form is answered 400 and the connection closed. The path is put in the normal form of the revision note before any decision; a path without one is answered 400 and the connection closed. Policy evaluates the normal path and the origin receives it, with the query unchanged.

9. Detailed Design

9.1. The decoder

`libs/net/src/chunked.zig` defines a decoder state machine:

```
pub const Decoder = struct {
    state: enum { size, data, data_end, trailer, done } = .size,
    remaining: u64 = 0,     // data bytes left in the current chunk
    trailer_bytes: u32 = 0, // bytes of trailer section consumed so far
    pub fn decode(self: *Decoder, bytes: []u8) Error!Step   // Step{ output, consumed }
};
```

`decode` reads `bytes` from index r and writes decoded data back into the same slice at index w. It returns the output length w and the consumed length r. It consumes only complete syntactic units: a whole line, a run of data bytes, or the two bytes after data. It stops when the next unit is incomplete. The caller keeps the unconsumed tail and calls again after the next read.

Definition 1 (Configuration)
A decoder configuration is a pair (s, k): a state $s \in \{\text{size}, \text{data}, \text{data_end}, \text{trailer}, \text{done}\}$ and the count k of data bytes remaining when $s = \text{data}$. One call maps a configuration and a byte string b to a new configuration, an output length w, and a consumed length r, with $w \le r \le \mid b \mid$.

Lemma 2 (In-place safety)
Throughout a call, the write index never exceeds the read index: $w \le r$. Each data byte moves from $b[r]$ to $b[w]$ with $w \le r$, so a forward copy never overwrites a byte that has not yet been read.

Proof. Both indices start at 0. A line or the CRLF after data advances r alone. A run of n data bytes copies $b[r \,..\, r+n]$ to $b[w \,..\, w+n]$ and advances both indices by n. Each step preserves $w \le r$, and the base case satisfies it. `std.mem.copyForwards` is correct for overlapping ranges whenever the destination does not start after the source. □

Lemma 3 (Progress)
If the unconsumed tail is at least L bytes long and the state is not `done`, a call consumes at least one byte or returns an error.

Proof. In `data`, any available byte is consumed. In `data_end`, two available bytes are either CRLF and consumed, or rejected. In `size` and `trailer`, the decoder looks for an LF within the first L bytes. If it finds one, it consumes the line or rejects it. If it does not, the line exceeds L and R5 rejects it. □

Theorem 4 (Partition invariance)
Let $x = x_1 x_2 ... x_m$ be any split of the raw stream into the bytes delivered by successive reads, with each call's unconsumed tail prefixed to the next delivery. The concatenated outputs equal $\mathrm{dechunk}\ (x)$ exactly when x satisfies R5 to R7. Otherwise some call returns an error, and no output past the faulty unit is produced. The verdict and the output do not depend on the split.

Proof. By induction on consumed units. The decoder consumes a unit only when all of its bytes are present, and whether a unit is accepted depends only on those bytes and on the configuration. The configuration after a unit depends only on the configuration before it and on the unit. A split changes when a unit becomes complete, but not its bytes, so the sequence of units, configurations and outputs is the same for every split. A data run may be cut by a split, but it is consumed byte for byte, and k records exactly the bytes still owed. This matches the RFC 9112 Section 7.1.3 algorithm, which is defined on whole units. □
The theorem is what the property test checks. Random valid and invalid streams are decoded with random split points, and every split must produce the same output and verdict as a single call.

9.2. Admission: decoding into the head buffer

The connection reader holds the head in its first H bytes. Admission calls `decode` on everything after the head and the body decoded so far. It then **excises** the consumed framing by moving the unconsumed tail down to the end of the output, which shortens the reader's buffered region. Finally it reads more, unless the body is complete or the buffer is full. The head is never moved by excision. A reader rebase moves the head and body together, so positions are kept relative to the reader's seek and the head is re-parsed after the loop, as the `Content-Length` path already does.

Lemma 5 (Prefix bound)
When admission stops, the decoded prefix is either the whole body B, or longer than $C - H - L$ bytes.

Proof. Admission stops on `done` or when the buffered region fills the buffer. When it is full, the region holds H head bytes, d decoded bytes, and an unconsumed tail t. Every call consumes all complete units, so the tail is an incomplete line, with $t < L$ by Lemma 3, or at most one byte of a pending CRLF. Therefore $d = C - H - t > C - H - L$. □

Theorem 6 (Inspection equivalence)
The policy engine and the WAF see the first $\min(|\ B\ |, P)$ bytes of B, the same bytes they see when B is sent with `Content-Length`. Splitting a payload across chunks therefore cannot change a verdict.

Proof. The WAF inspects `body[0 .. min(len, P)]` (`body_fields.Fields.init`). By Lemma 5 the decoded prefix is B, or longer than $C - H - L = 65{,}536 - 16{,}384 - 4{,}096 = 45{,}056 > P$. Both cases contain $B[0\,..\,\min(|\ B\ |, P)]$. Theorem 4 makes that prefix independent of chunk sizes and arrival timing. Rules other than the WAF see the head, which is unchanged. □
Admission waits for the body to complete or the buffer to fill. A `Content-Length` request waits for $\min(\text{declared}, C - H)$ bytes, so this is the same bound, and it is HAProxy's `http-buffer-request` behaviour. The existing head deadline applies unchanged.

9.3. Forwarding

The proxy's request body becomes an explicit union read by `writeHead`, `exchange` and `streamProxy`:

```
pub const Upload = union(enum) {
    none,                       // no body framing; nothing follows the head
    length: u64,                // Content-Length; `length - body.len` bytes still follow
    chunked: *chunked.Decoder,  // the rest is chunked: decode in place, re-chunk per read
};
```

A complete chunked body becomes `.length = |B|`. From there it takes exactly the `Content-Length` path, including the retry rule for safe methods with a fully buffered body. An incomplete one becomes `.chunked`. Its head carries `Transfer-Encoding: chunked`; its decoded prefix is sent as the first chunk; then each read is decoded in place and sent as one chunk. The terminal `0` CRLF CRLF is written only once the decoder reaches `done`.

Rust port note (2026-10-15)
In the Rust proxy each later read goes into a 16 KiB upload buffer of the connection, not the connection buffer, and is decoded in place there. See the revision note.

Corollary 7 (Origin framing independence)
The bytes the origin receives depend only on B and on how Sibuna's reads segment the stream. They do not depend on the client's chunk sizes, extensions, trailers or letter case. Every message is either `Content-Length` framed or canonical chunked.

Canonical chunked is a subset of every grammar that strict and lenient parsers accept. A parser differential between Sibuna and the origin would need two parsers that disagree on some input. The origin only ever receives that subset, so the TE.TE, lone-terminator and extension classes cannot occur behind Sibuna.

9.4. Cost

Lemma 8 (Linear work, constant memory)
Admission moves each body byte at most once (Lemma 2) plus an unconsumed tail shorter than L per read, and moves any pipelined bytes once when the body completes. Streaming touches each byte once in place and adds at most 20 framing bytes per origin chunk (16 hexadecimal digits and two CRLFs). The decoder state is 24 bytes on the stack. Neither phase allocates.

Rust port note (2026-10-15)
In the Rust port the streaming phase works in a per-connection upload buffer allocated once with the connection, so the request path still allocates nothing, but that buffer is a second per-connection buffer. Bytes pending in the connection buffer when streaming begins are copied into it once.

The `Content-Length` and body-less paths gain one branch each in `serveOne` and in `exchange`. Chunked uploads pay one in-L1 `memmove` per byte, which is cheap next to the socket read that delivered the byte, plus a chunk-size line per read. Streaming keeps the relay discipline of `proxy.step`: forward what is buffered, flush the origin before waiting, read once. The slow-body defence counts raw bytes received, so a chunked upload must sustain the same 16 KiB per idle period as a `Content-Length` upload.

10. Verification and Testing

Table 20 Tests and their acceptance criteria

Test	Acceptance
`chunked.zig` rejection table	Every R5 to R7 violation is rejected, including the "Funky chunks" terminator and extension vectors, a lone CR inside a quoted string, 17-digit and signed sizes, whitespace before CRLF, and trailers with whitespace before the colon or above 16 KiB.
`chunked.zig` partition property	Random streams decoded with random split points produce the same output and verdict as a single call (Theorem 4), including streams with extensions and trailers.
`http.zig` framing rules	R1 to R4: one exact `chunked` field accepted; duplicate fields, `chunked, gzip`, HTTP/1.0 with a coding and TE with CL rejected with the specified errors.
Daemon end-to-end	A small chunked POST pipelined with a GET reaches the origin with `Content-Length` and the GET is answered on the same connection; a 256 KiB body in irregular chunks reaches the origin as chunked with an identical digest; an SQL-injection payload split across chunks is denied; a malformed chunk is answered 400 and the origin never sees the request.

11. Security Considerations

- **Smuggling.** R1 to R7 reject every chunked input that two HTTP/1.1 parsers are known to disagree on. Corollary 7 removes the second parser's decision altogether. Rule R9 ensures that an attacker who breaks framing mid-stream leaves the origin with a truncated body.
- **Evasion.** Theorem 6 means chunk boundaries are invisible to inspection.
- **Resource use.** Lines are bounded by L and trailers by 16 KiB. Sizes are bounded by 16 digits and counted in a `u64`. The minimum upload rate is unchanged. A client that sends one data byte per chunk pays six framing bytes per decoded byte. Sibuna's work per byte stays constant, and no state grows with the client's chunk count.
- **Trailers.** They are discarded, so fields the head inspection never saw cannot reach the origin.

12. Rollout and Operational Plan

No configuration is added. Operators who sent `Content-Length` because chunked uploads were refused need not change anything. The book's operations chapter, the README and SID 0002 drop the "chunked request bodies are rejected" limitation and state that trailers are discarded and other transfer codings receive 501. Origins that cannot parse chunked requests (HTTP/1.0-era servers) still receive `Content-Length` for any body under about 44 KiB. Larger streamed bodies require an HTTP/1.1 origin, which RFC 9112 Section 7.1 already requires.

13. References

1. R. Fielding, M. Nottingham, J. Reschke. RFC 9112, *HTTP/1.1*, Section 6 (message body), Section 7.1 (chunked transfer coding), Section 11.2 (request smuggling). IETF, 2022.
2. R. Fielding, M. Nottingham, J. Reschke. RFC 9110, *HTTP Semantics*, Section 6.5 (trailer fields). IETF, 2022.
3. C. Linhart, A. Klein, R. Heled, S. Orrin. *HTTP Request Smuggling*. Watchfire, 2005.
4. J. Kettle. *HTTP Desync Attacks: Request Smuggling Reborn*. Black Hat USA, 2019.
5. J. Kettle. *HTTP/1.1 Must Die: The Desync Endgame*. Black Hat USA and DEF CON 33, 2025. https://portswigger.net/research/http1-must-die
6. J. B. Weikop. *Funky chunks: abusing ambiguous chunk line terminators for request smuggling*, 2025. https://w4ke.info/2025/06/18/funky-chunks.html; follow-up on chunk extensions, https://w4ke.info/2025/10/29/funky-chunks-2.html
7. GitHub advisories GHSA-vqfr-h8mv-ghfj (h11, CVE-2025-43859), GHSA-fghv-69vj-qj49 (Netty chunk extensions), GHSA-355h-qmc2-wpwf (Jetty quoted-string extensions); Microsoft CVE-2025-55315 (Kestrel chunk extensions).
8. HAProxy configuration manual, `option http-buffer-request`; nginx `proxy_request_buffering`; OWASP Coraza `SecRequestBodyLimit`.

Lifecycle event

```
2026-10-15: prediscussion -> discussion. Ported from the Zig repository's
Typst record
```

Lifecycle event

```
2026-10-16: discussion -> accepted. Implemented and released in Sibuna
1.0.0-beta.1; reviewed by the maintainer on 2026-10-16.
```

Lifecycle event

```
2026-10-16: accepted -> committed. Implemented and released in Sibuna
1.0.0-beta.1; reviewed by the maintainer on 2026-10-16.
crates/http/src/chunked.rs and its tests (the R1 to R10 rejection table, the
partition property), crates/http/src/request.rs framing tests,
cmd/sibuna/tests/e2e.rs chunked uploads; release v1.0.0-beta.1
```