

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

STATE

committed

CREATED

2026-10-02

LAST UPDATED

2026-10-02

INTENDED STATUS

Committed

AUTHORS

Sibuna Contributors <team@sibuna.local>

DISCUSSION

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.

LABELS

http, proxy, security, request-smuggling, performance

Status of This Memo

This document is an internal Sibuna discussion record authored in Typst and tracked in git. It intentionally follows RFC-style structure so the design scope, rationale, trade-offs, and operational constraints remain explicit. Documents that still use the placeholder number XXXXX are provisional drafts. Numbered SID documents are part of the permanent project record.

Contents

Abstract	1
Introduction	1
The SID Lifecycle	1
Tooling and Numbering Workflow	1
Creating a Draft	1
Promoting a Draft	1
Listing Discussions	1

Engineering Principles and Code Standards	1
Hard Structural Limits	1
Core TigerStyle Priorities	1
Elm-Style Diagnostic and Error Reporting	1
The Three Elements of an Elm-Style Error	1
Concrete Diagnostic Example	1
Implementation Architecture	1
Recoverable Errors vs Invariant Assertions	1
Compilation Targets and Verification	1
Conclusion	1
Decision	1
Request contract	1
Authentication and work	1
Policy and memory	1
Measurement gates	1
Release toolchain	1
Native platforms and release boundaries	1
Remaining limits	1
1. Abstract	9
2. Introduction	9
3. The SID Lifecycle	9
4. Tooling and Numbering Workflow	10
4.1. Creating a Draft	10
4.2. Promoting a Draft	10
4.3. Listing Discussions	10
5. Engineering Principles and Code Standards	10
5.1. Hard Structural Limits	10
5.2. Core TigerStyle Priorities	11
6. Elm-Style Diagnostic and Error Reporting	11
6.1. The Three Elements of an Elm-Style Error	11
6.2. Concrete Diagnostic Example	11
6.3. Implementation Architecture	12
6.4. Recoverable Errors vs Invariant Assertions	12
7. Compilation Targets and Verification	12
8. Conclusion	12
9. Decision	9
10. Request contract	10
11. Authentication and work	11
12. Policy and memory	11
13. Measurement gates	11
14. Release toolchain	12
15. Native platforms and release boundaries	12
16. Remaining limits	12
Decision summary	1
Data model	1
Evaluation order	1
Built-in policy	1
Policy file	1
Dynamic policies	1

Verification	1
Milestones	1
Review corrections (2026-09-07)	1
17. Decision summary	9
18. Data model	9
19. Evaluation order	10
20. Built-in policy	10
21. Policy file	11
22. Dynamic policies	12
23. Verification	12
24. Milestones	12
25. Review corrections (2026-09-07)	12
Context and problem statement	1
Engine choice: automata and tokenizers, not regular expressions	1
Design	1
Byte-class pre-scan	1
26. Context and problem statement	9
27. Engine choice: automata and tokenizers, not regular expressions	9
28. Design	10
28.1. Byte-class pre-scan	10
Context and motivation	1
Zaxonlite foundations	1
Build and configuration	1
Schema	1
Hot-path isolation	1
Read-copy-update engine slot	1
Incident ring	1
Storage thread tick	1
Campaign clustering	1
Forensics	1
Product boundary	1
Delivery record	1
Verification	1
Open items	1
Review corrections (2026-09-07)	1
29. Context and motivation	9
30. Zaxonlite foundations	10
31. Build and configuration	10
32. Schema	10
33. Hot-path isolation	11
33.1. Read-copy-update engine slot	11
33.2. Incident ring	11
33.3. Storage thread tick	11
34. Campaign clustering	12
35. Forensics	12
36. Product boundary	12
37. Delivery record	12
38. Verification	12
39. Open items	13

40. Review corrections (2026-09-07)	13
Abstract	1
Introduction and Adversary Model	1
Setting	1
Axioms	1
Cost calculus	1
Part A: The Hashcash Tier	1
Part B: Proof of Sequential Work	1
Why sequential work	1
The construction	1
Theorems	1
Why not the alternatives	1
Implementation and calibration	1
Part C: Stateless Challenges and Symmetric Tokens	1
Pseudorandom functions and message authentication	1
Stateless challenge identifiers	1
Session tokens	1
Part D: Rate Limiting by the Generic Cell Rate Algorithm	1
Part E: Robin Hood Hashing for the Spent Set	1
Part F: Linear-Time Inspection	1
Multi-pattern matching	1
The semantic layer	1
Vectorised matchers	1
Part G: Read-Copy-Update Engine Slots	1
Part H: Load-Adaptive Difficulty	1
Part I: Campaign Clustering by Feature Hashing	1
Verification and Benchmark Gates	1
Security Considerations	1
Open Problems	1
References	1
41. Abstract	9
42. Introduction and Adversary Model	10
42.1. Setting	10
42.2. Axioms	10
42.3. Cost calculus	11
43. Part A: The Hashcash Tier	11
44. Part B: Proof of Sequential Work	12
44.1. Why sequential work	12
44.2. The construction	12
44.3. Theorems	13
44.4. Why not the alternatives	14
44.5. Implementation and calibration	15
45. Part C: Stateless Challenges and Symmetric Tokens	15
45.1. Pseudorandom functions and message authentication	15
45.2. Stateless challenge identifiers	16
45.3. Session tokens	17
46. Part D: Rate Limiting by the Generic Cell Rate Algorithm	17
47. Part E: Robin Hood Hashing for the Spent Set	18
48. Part F: Linear-Time Inspection	19

48.1. Multi-pattern matching	19
48.2. The semantic layer	20
48.3. Vectorised matchers	20
49. Part G: Read-Copy-Update Engine Slots	21
50. Part H: Load-Adaptive Difficulty	21
51. Part I: Campaign Clustering by Feature Hashing	22
52. Verification and Benchmark Gates	22
53. Security Considerations	23
54. Open Problems	23
55. References	24
Abstract	1
Introduction and Motivation	1
Operator Workflows	1
Design Principles	1
Don't make me think	1
Fast judgement: System 1	1
Careful judgement: System 2	1
Elegance	1
Principle audit by page	1
Terminology and Scope	1
Goals and Non-Goals	1
Goals	1
Non-Goals	1
Architecture Overview	1
Modules and directories	1
Process model and the isolation contract	1
Startup from the command line	1
The Serve Kernel	1
Authentication and Authorization	1
The Telemetry Pipeline	1
The sampler	1
The incident tap	1
The challenge funnel	1
Traffic sampling and top-k rankings	1
Data-plane changes this record requests	1
Cluster aggregation	1
The Real-Time Protocol	1
Data Model	1
GeoIP	1
Live earth globe on the landing page	1
Cluster Management	1
The Interface Module	1
Pages	1
Wireframes	1
The Build Pipeline	1
Security Considerations	1
Performance Budget	1
Delivery Plan	1
Verification	1

Resolved Design Decisions	1
Operational Contracts and Trade-offs	1
Retained observations	1
Rule observations	1
Management consistency	1
Browser and peer state	1
Acceptance and Open Questions	1
Query and authentication observation bounds	1
Launch scope (2026-09-11)	1
Post-launch completion (2026-09-11)	1
Interface budget review (2026-09-11)	1
References	1
56. Abstract	9
57. Introduction and Motivation	10
58. Operator Workflows	10
59. Design Principles	11
59.1. Don't make me think	11
59.2. Fast judgement: System 1	12
59.3. Careful judgement: System 2	13
59.4. Elegance	13
59.5. Principle audit by page	13
60. Terminology and Scope	14
61. Goals and Non-Goals	15
61.1. Goals	15
61.2. Non-Goals	15
62. Architecture Overview	16
62.1. Modules and directories	16
62.2. Process model and the isolation contract	17
62.3. Startup from the command line	18
63. The Serve Kernel	19
64. Authentication and Authorization	20
65. The Telemetry Pipeline	21
65.1. The sampler	21
65.2. The incident tap	22
65.3. The challenge funnel	23
65.4. Traffic sampling and top-k rankings	23
65.5. Data-plane changes this record requests	24
65.6. Cluster aggregation	24
66. The Real-Time Protocol	24
67. Data Model	26
68. GeoIP	28
68.1. Live earth globe on the landing page	29
69. Cluster Management	30
70. The Interface Module	30
70.1. Pages	31
71. Wireframes	33
72. The Build Pipeline	43
73. Security Considerations	44
74. Performance Budget	45

75. Delivery Plan	46
76. Verification	47
77. Resolved Design Decisions	48
78. Operational Contracts and Trade-offs	49
78.1. Retained observations	49
78.2. Rule observations	49
78.3. Management consistency	50
78.4. Browser and peer state	50
79. Acceptance and Open Questions	51
79.1. Query and authentication observation bounds	51
79.2. Launch scope (2026-09-11)	51
79.3. Post-launch completion (2026-09-11)	52
79.4. Interface budget review (2026-09-11)	53
80. References	53
Abstract	1
Introduction and Motivation	1
Terminology and Scope	1
Problem Statement and Architectural Challenges	1
The Gap in the Current Codebase	1
The Asymmetric Challenges of AI Bot Identification	1
System Invariants and Design Principles	1
Detailed Design	1
1. The Multi-Tier Bot Classification Engine	1
1.1 Provider and Bot Taxonomy	1
1.2 Comprehensive AI Bot Signatures Database	1
1.3 Sub-Microsecond Radix CIDR Verification	1
1.4 Behavioral and Structural Heuristics	1
2. High-Efficiency Streaming Analytics: Algorithms, Mathematics, and Storage Decoupling	1
2.1 The Analytics Dilemma: Why Traditional OLAP Fails Edge WAFs	1
2.2 Axiomatic Foundations of Edge Analytics	1
2.3 Contention-Free Cache-Line Striped Accumulation	1
2.4 Bounded Stochastic Sampling and Poisson Ring Dynamics	1
2.5 Space-Saving Heavy-Hitters Algorithm for Path Scraping	1
2.6 Micro-RRD Pre-Aggregated Rollups and Zaxonlite Storage Decoupling	1
2.7 Long-Term Telemetry Egress Strategy	1
3. Operator Console Dashboard Interface	1
3.1 Operator Screen Wireframes	1
3.2 Visual Components	1
3.3 Granular Tabular Components	1
3.4 Actionable Policy Integration	1
4. Elm-Style Operator Error and Diagnostic Reporting	1
Verification and Testing Matrix	1
Security Considerations	1
Rollout and Operational Plan	1
References	1
81. Abstract	9
82. Introduction and Motivation	9
83. Terminology and Scope	10

84. Problem Statement and Architectural Challenges	11
84.1. The Gap in the Current Codebase	11
84.2. The Asymmetric Challenges of AI Bot Identification	12
85. System Invariants and Design Principles	12
86. Detailed Design	13
86.1. 1. The Multi-Tier Bot Classification Engine	13
86.1.1. 1.1 Provider and Bot Taxonomy	13
86.1.2. 1.2 Comprehensive AI Bot Signatures Database	14
86.1.3. 1.3 Sub-Microsecond Radix CIDR Verification	15
86.1.4. 1.4 Behavioral and Structural Heuristics	16
86.2. 2. High-Efficiency Streaming Analytics: Algorithms, Mathematics, and Storage Decoupling	16
86.2.1. 2.1 The Analytics Dilemma: Why Traditional OLAP Fails Edge WAFs	16
86.2.2. 2.2 Axiomatic Foundations of Edge Analytics	17
86.2.3. 2.3 Contention-Free Cache-Line Striped Accumulation	17
86.2.4. 2.4 Bounded Stochastic Sampling and Poisson Ring Dynamics	18
86.2.5. 2.5 Space-Saving Heavy-Hitters Algorithm for Path Scraping	19
86.2.6. 2.6 Micro-RRD Pre-Aggregated Rollups and Zaxonlite Storage Decoupling . . .	20
86.2.7. 2.7 Long-Term Telemetry Egress Strategy	21
86.3. 3. Operator Console Dashboard Interface	22
86.3.1. 3.1 Operator Screen Wireframes	22
86.3.2. 3.2 Visual Components	24
86.3.3. 3.3 Granular Tabular Components	25
86.3.4. 3.4 Actionable Policy Integration	25
86.4. 4. Elm-Style Operator Error and Diagnostic Reporting	26
87. Verification and Testing Matrix	26
88. Security Considerations	26
89. Rollout and Operational Plan	27
90. References	27
Abstract	1
Introduction	1
Terminology and Scope	1
Problem Statement	1
Goals and Non-Goals	1
Goals	1
Non-Goals	1
State of the Art	1
Protocol Rules	1
Detailed Design	1
The decoder	1
Admission: decoding into the head buffer	1
Forwarding	1
Cost	1
Verification and Testing	1
Security Considerations	1
Rollout and Operational Plan	1
Implementation Record (2026-10-02)	1
References	1
91. Abstract	9

92. Introduction	9
93. Terminology and Scope	10
94. Problem Statement	10
95. Goals and Non-Goals	10
95.1. Goals	10
95.2. Non-Goals	11
96. State of the Art	11
97. Protocol Rules	11
98. Detailed Design	12
98.1. The decoder	12
98.2. Admission: decoding into the head buffer	13
98.3. Forwarding	14
98.4. Cost	15
99. Verification and Testing	15
100. Security Considerations	15
101. Rollout and Operational Plan	16
102. Implementation Record (2026-10-02)	16
103. References	17

91. Abstract

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 2024–2025. 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.

92. 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 §6.1, §7.1). Every HTTP/1.1 recipient must be able to parse chunked coding (RFC 9112 §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.

93. Terminology and Scope

Term	Meaning in this record
Raw stream x	The bytes the client sends after the request head.
Body B	$\text{dechunk}(x)$: the concatenated chunk data, as defined by the decoding algorithm of RFC 9112 §7.1.3.
Chunk line	A chunk-size line (size and optional extensions) or a trailer field line, each terminated by CRLF.
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 ($\text{body_fields.max_prefix}$, 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 `relyChunked`), and HTTP/2.

94. 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 $\min(\text{declared}, C - H)$ 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.

95. Goals and Non-Goals

95.1. 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.

95.2. 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 §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.

96. State of the Art

System	Request-body behaviour
nginx	Decodes chunked input. With <code>proxy_request_buffering</code> 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 <code>http-buffer-request</code> waits for the body up to one buffer before the request is processed, so ACLs see a decoded prefix.
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 <code>SecRequestBodyLimit</code> , 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.

97. Protocol Rules

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).

R2 A request with both Transfer-Encoding and Content-Length is answered 400 and the connection closed. RFC 9112 §6.3 permits this, and the parser already does it.

R3 A coding list whose final coding is not chunked is answered 400 and closed (RFC 9112 §6.3, a MUST). A list that ends in chunked after another coding, such as `gzip, chunked`, is answered 501 and closed (RFC 9112 §6.1, a SHOULD). The latter would need decompression before inspection. More than one Transfer-Encoding line is answered 400.

R4 An HTTP/1.0 request with Transfer-Encoding is answered 400 and closed: RFC 9112 §6.1 requires faulty-framing handling for it.

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 §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.

R6 Chunk data must be followed by exactly CRLF.

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.

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.

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.

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.

98. Detailed Design

98.1. 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 { 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 111 (Configuration). A decoder configuration is a pair (s, k) : a state $s \in \{\text{size, data, data_end, trailer, done}\}$ 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 $w \leq r \leq |b|$.

Lemma 112 (In-place safety). Throughout a call, the write index never exceeds the read index: $w \leq r$. Each data byte moves from $b[r]$ to $b[w]$ with $w \leq r$, 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 $w \leq r$, and the base case satisfies it. `std.mem.copyForwards` is correct for overlapping ranges whenever the destination does not start after the source. $\square$

Lemma 113 (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. $\square$

Theorem 114 (Partition invariance). Let $x = x_1x_2\dots x_m$ 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 `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 §7.1.3 algorithm, which is defined on whole units. $\square$

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.

98.2. 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 115 (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 $d = C - H - t > C - H - L$. $\square$

Theorem 116 (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. $\square$

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.

98.3. 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 $\emptyset$ CRLF CRLF is written only once the decoder reaches done.

Corollary 117 (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.

98.4. Cost

Lemma 118 (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.

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.

99. Verification and Testing

Test	Acceptance
<code>chunked.zig</code> rejection table	Every R5–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.
<code>chunked.zig</code> 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.
<code>http.zig</code> framing rules	R1–R4: one exact chunked field accepted; duplicate fields, chunked, <code>gzip</code> , 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.

100. Security Considerations

- **Smuggling.** R1–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.

101. 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 §7.1 already requires.

102. Implementation Record (2026-10-02)

The design landed without changes to the rules. The decoder is `libs/net/src/chunked.zig` (Decoder.decode, line limit 4,096, trailer limit 16 KiB, at most 16 size digits). The parser rules R1–R4 are in `net.http.parseRequest`, which sets `Request.chunked` and distinguishes `InvalidTransferEncoding` (400) from `UnsupportedTransferEncoding` (501). Admission decoding is `bufferChunked` in `apps/sibuna/src/server.zig`. It cuts each call's framing out of the reader's window by moving the tail down and shortening end. Forwarding uses `net.proxy.Upload` (none, length, chunked), which `writeHead`, `exchange` and `streamProxy` all read; `relayChunkedBody` re-chunks per read, flushes before it waits, and writes the terminal chunk only at done. A body that continues past the buffer records no declared length in incident evidence: the evidence carries the decoded bytes seen, flagged incomplete.

Verification. Unit tests cover the grammar (six accepted forms, 24 rejected vectors and the trailer bound), the partition property (3,000 random streams, half of them mutated, each decoded whole and in random pieces), and R1–R4. Three daemon end-to-end tests check:

- a pipelined small upload that reaches the origin with Content-Length, followed by a GET on the same connection;
- a 256 KiB upload in irregular chunks with extensions that reaches the origin as canonical chunks with an identical digest;
- an injection payload split across chunks, which is denied;
- a lone LF, gzip, chunked, HTTP/1.0 with a coding, and a malformed chunk after 80 KiB had already streamed, each answered as R3–R9 require.

The origin stub in those tests parses only canonical framing, so any client framing leaking through fails the test. `zig build test` passed on Linux (paxos-zig).

Performance. These measurements were taken on paxos-zig, 2026-10-02, on loopback, with the product pinned to four CPUs. On the reverse-proxy GET path, the build with this change and the build before it were indistinguishable under the tools-comparison load: medians of 59.0k and 59.2k requests per second over five interleaved rounds. Uploads of 256 MiB reached 2.0–2.4 GiB/s chunked and 1.7–2.9 GiB/s with Content-Length. Both figures are bound by the Python origin that drains them, so they show that chunked uploads stream at the same order as sized ones. They do not measure the decoder.

Correction. A review found that a streamed body, chunked or sized, refilled the reader buffer that still held the request head. The telemetry sample and any audit findings held for the origin head read that head after the relay. With `--console-capture-heads`, the recorded path and request head then held upload bytes. This flaw predates chunked support; this record's tests did not combine streaming with head capture. `forward` now lends the relay only the buffer behind the head (`HeadPin`) and restores the reader when the relay returns. A body streams only after admission filled the whole buffer, so the head is at the start of the buffer and the relay keeps at least 48 KiB of it; nothing is copied. An end-to-end test sends 192 KiB uploads in both framings

through a fixture that captures heads and audits SQL injection. It asserts that the recorded path, client, agent and redacted request head match the request exactly.

103. References

1. R. Fielding, M. Nottingham, J. Reschke. RFC 9112, *HTTP/1.1*, §6 (message body), §7.1 (chunked transfer coding), §11.2 (request smuggling). IETF, 2022.
2. R. Fielding, M. Nottingham, J. Reschke. RFC 9110, *HTTP Semantics*, §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-wpww (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`.