Header Accordions: QPACK amplification attacks against HTTP/3 servers

Sébastien Féry

Oct 20, 2026

TLDR

A remote, unauthenticated client can force an HTTP/3 server to allocate tens to thousands of times more memory than it sends. One connection is enough to OOM or pin a default-configured server. Seven of the ten implementations tested are vulnerable; several are already fixed.

QPACK compresses HTTP/3 headers to minimize traffic sent over the wire. Across the affected implementations, decoded-header limits are missing, incomplete, or applied too late. A small HEADERS frame can therefore expand into hundreds of thousands of header objects in memory. Preventing that expansion from exhausting resources requires enforcing a cumulative budget as fields are decoded and retained.

Patched as of October 20, 2026

Library	Fixed version	Advisory
quic-go	0.57.0	CVE-2025-64702
Cloudflare quiche	0.29.3	CVE-2026-12523
lsquic	4.9.0	no CVE
Wireshark	4.6.2 / 4.4.12	CVE-2025-13945

Unpatched as of October 20, 2026

Library

Bun 1.4.3
h3 (hyperium)
aioquic / mitmproxy

Mitigations

Bun: Don't expose HTTP/3 servers

mitmproxy: disable HTTP/3 (pass *--set http3=false*)

h3 crate: set a reasonable *max_field_section_size*

Not affected: IIS, Envoy, ngtcp2/nghttp3

Background

During recent research into the different QUIC stacks for our active TLS scanner, JA4Scan, I found the same class of vulnerability across multiple HTTP/3 implementations. A remote, unauthenticated client can send HTTP/3 traffic that forces the server to allocate far more memory than the encoded request occupies on the wire.

This investigation started after reading Gal Bar Nahum's MadeYouReset series on HTTP/2 request-cancellation DoS. That work got me looking at whether HTTP/3's QPACK header compression had similar amplification issues.

HTTP/3 compresses headers using QPACK, so repeated fields like `user-agent` or `content-type` can be sent as a small index instead of the full string. It uses two lookup tables: a static table, a hardcoded set of common fields defined by the specification, and a dynamic table, built up at runtime between a client and server as new headers are seen.

For example, a 71-byte header like `strict-transport-security: max-age=31536000; includesubdomains; preload` becomes a single byte: `0xFA`, a static-table reference to index 58.

RFC 9114 Section 4.2.2 defines how servers should limit decoded header size:

The size of a field list is calculated based on the uncompressed size of fields, including the length of the name and value in bytes plus an overhead of 32 bytes for each field.

Some implementations tested got at least one of these wrong:

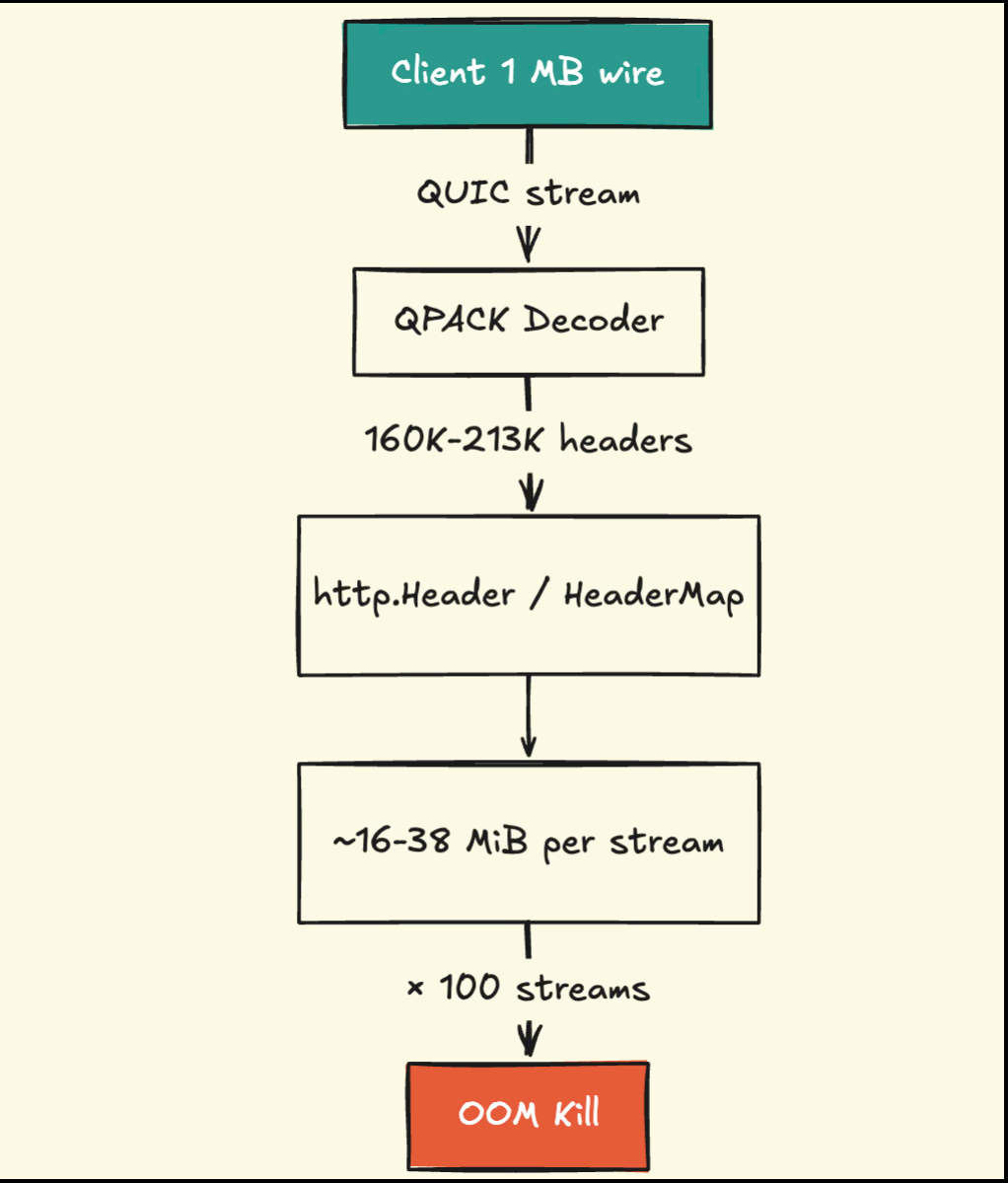
1. They check the compressed frame size, not the decoded size
2. They decompress everything before checking
3. They omit the 32-byte overhead from their size calculation
4. They default to unlimited when the setting is unconfigured
5. They never enforce the setting at all, even when it is configured

How does it work?

The goal is to maximize the number of headers inside a single HEADERS frame. For this variant, I use unique names to create distinct entries in the receiving header map: `a` through `z`, `aa` through `zz`, then three and four char names until the encoded-size budget is filled. There are 676 possible two-letter names but we need to skip `te` as that header has special value requirements..

A 1,000,000-byte HEADERS frame holds about 160,000 unique headers this way.

What the server does with those 160,000+ headers is where it breaks.



Example request flow on quic-go

The accordion also works in the other direction. Instead of many small headers, the dynamic table variant inserts one large entry (~4 KiB) via the encoder stream, then references it thousands of times with 1-byte indexed references. Each reference

forces a fresh allocation and copy of the full entry. This is how the lsquic and Bun attacks work: each 1-byte reference materializes the full entry, a ratio of 4064:1.

Implementations

Implementation	Vulnerable	CVE	Wire-to-memory	Root cause
quic-go	Yes	CVE-2025-64702	~15x retained heap	Checks compressed size only, no decoded limit
Wireshark	Yes	CVE-2025-13945	crash	QPACK dissector unbounded decode
Cloudflare quiche	Yes	CVE-2026-12523	~20x	Missing 32-byte overhead + frame prealloc bomb
lsquic	Yes	Fixed, no CVE	~600-1800x	No cumulative decoded limit, unlimited HEADERS blocks
aioquic / mitmproxy	Yes	-	~20-30x	MAX_FIELD_SECTION_SIZE parsed but never enforced
Bun	Yes	-	~4000x	Replaces lsquic's 64K cap with no cap
h3 (hyperium)	Yes	Reported, not fixed	Thread panic	HeaderMap::append panics at 32,768 entries
Google quiche	No	-	-	60K cap enforced during decode, stream RESET on violation
IIS (http.sys)	No	-	-	Incremental enforcement, 4K QPACK table, H3_EXCESSIVE_LOAD
ngtcp2 / nghttp3	No	Reported, documentation updated	-	Dynamic table default 0, immediate decref; max_field_section_size is signaling-only

quic-go

quic-go is the Go library behind Caddy, Traefik, and cloudflared. Its HTTP/3 server limits the HEADERS *frame* to 1 MiB (`DefaultMaxHeaderBytes`), but that limit applies to the compressed QPACK payload, not the decoded output. `DecodeFull` in `qpack/decoder.go` decompresses the entire block in one pass and returns a `[]HeaderField` slice, which is then converted into an `http.Header` map.

The `http.Header` map is a `map[string][]string`, whose keys, value slices and internal storage add overhead beyond the encoded headers. The 1,000,000-byte QPACK block decodes into 160,075 fields and retained approximately 15 MiB in the resulting request after gc, about 15 times the encoded size.

quic-go limited the compressed HEADERS frame without enforcing a corresponding decoded-header budget.

Fix: quic-go now sends `SETTINGS_MAX_FIELD_SECTION_SIZE`, uses incremental QPACK decoding that checks size after each entry, and aborts with HTTP 431 on the server side.

Cloudflare quiche

quiche had two issues rolled into one CVE.

First, `decode()` in `qpack/decoder.rs` uses `name.len() + value.len()` for size accounting, omitting the RFC-mandated 32-byte overhead per field. This lets an attacker fit up to 33x more headers than the limit intends:

With `max_field_section_size = 1 MiB`, the RFC allows ~32,000 headers (each costing at least 33 bytes). quiche's calculation allows ~1,000,000 one-byte headers to pass.

Second, quiche's HTTP/3 frame parser pre-allocates buffers based on the declared frame length before receiving the payload. An attacker declares a 16 MiB SETTINGS or GOAWAY frame, triggering immediate memory allocation, then never sends the data.

Fix: quiche 0.29.3 adds the 32-byte overhead and fixes frame pre-allocation.

lsquic

lsquic's vulnerability is the deepest I found. Three issues compound:

1. *header_out_dynamic_entry* allocates and copies per reference with no cumulative decoded-size limit
2. *es_max_header_list_size* defaults to 0 (unlimited) and is never passed to the QPACK decoder
3. The 64 KiB header cap resets per HEADERS **block**, not per stream, and lsquic accepts unlimited HEADERS blocks before DATA on a single stream

Dynamic-table references produce greater amplification in this implementation:

1. Insert one ~4 KiB entry via the encoder stream.
2. On each request stream, send a chain of HEADERS frames, each containing 15 one-byte indexed references.

Less than 10 MiB of wire traffic and 50 connections manage to pin the resident memory at the 5 GiB cap within a second, with cgroup `memory.events` recording 900+ limit hits. The PoC defaults (10 connections, 256 MiB container) show the same effect at a smaller scale.

Fix: fixed in lsquic 4.9.0 (July 15, 2026), which adds `es_max_header_sets` (server default 1) to cap the decoded header sets buffered per stream and suspend HTTP/3 parsing until the application drains them.

During testing, I also observed production servers, including `www.litespeedtech.com`, advertising `QPACK_MAX_TABLE_CAPACITY=4096` and `MAX_HEADER_LIST_SIZE=65536`.

Bun

Bun's HTTP/3 support uses lsquic under the hood but replaces lsquic's built-in header-set callbacks (which cap at 64 KiB cumulative) with its own `us_quic_hsi_prepare/process` in `quic.c`, which check per-header size (64 KiB) but never cumulative. The `es_max_header_list_size=16384` is advertised in SETTINGS but never enforced.

Same dynamic table trick: 1 connection, 100 streams, up to 500K one-byte references per stream to a 4 KiB entry. Only 60K references per stream (5.8 MiB of wire in total) are enough to OOM-kill a 2 GiB container.

Bun has a second bug in the same function that turns the amplification into a permanent denial of service. `us_quic_hsi_prepare` tracks buffer capacity in an `unsigned int` and grows it by doubling:

Once the decoded header set passes 2 GiB, `need` exceeds 2^31 and `ncap` doubles through 2^31 to 0. From zero, `ncap *= 2` stays zero forever and `0 < need` never becomes false: an infinite loop inside `lsquic_engine_process_conns`.

A single stream carrying ~516 KiB of wire (528,500 indexed references) leaves the server pinned at 100% CPU and it never recovers, blocking out new incoming requests.

The missing cumulative limit is what makes the overflow reachable; lsquic's stock 64 KiB cap prevents it.

The fix for both issues is the same check (`if (h->len + space > 16 * 1024) return NULL;`)

aioquic / mitmproxy

aioquic is the Python HTTP/3 library used by mitmproxy and hypercorn. It defines `MAX_FIELD_SECTION_SIZE` as a setting constant, parses it from the peer, stores it yet never checks it during decoding. `_decode_headers()` returns all decoded headers unconditionally.

aioquic's flow control also auto-doubles with no ceiling (`limit.value *= 2` when >50% consumed), so a single connection can deliver unlimited data.

The library itself only holds the decoded list transiently, so a minimal aioquic server that drops the headers after handling them survives, mitmproxy doesn't: it builds an `http.Request` per stream and keeps it while the stream is open.

h3 crate (hyperium)

h3 is the Rust HTTP/3 library from the hyperium project. It can be used with quinn through the h3-quinn adapter and is also used by tokio-quiche. It decodes all QPACK headers into a `Vec<HeaderField>` and then converts them into an `http::HeaderMap` via `TryFrom<Vec<HeaderField>>` in `h3/src/proto/headers.rs`:

`HeaderMap::append` calls `try_append(...).expect("size overflows MAX_SIZE")`, causing a panic when the map's capacity is exceeded. Under h3's default configuration, *max_field_section_size* is effectively unlimited thus allowing an oversized decoded header set to reach this path. Configuring a sufficiently small decoded-size limit prevents this reproducer from reaching the map's capacity limit. The resulting panic is shown below:

Wireshark

Funnily enough, this one was inadvertently found while testing payloads for quic-go. The issue was lying in an uncontrolled number of allowed decoded headers and a slow `tvb_composite`, which froze and finally crashed the HTTP/3 dissector.

Why does it happen?

The harder question is who owns the limit. Decoding and storing headers can involve several components:

- **QUIC** accounts for bytes received on the stream. It has no visibility into the header objects those bytes become.

- **The QPACK decoder** reconstructs the fields. Depending on its API, it may enforce a decoded-size budget or leave that responsibility to its caller.
- **The HTTP/3 layer and application** turn those fields into request objects and retain them. Their containers have their own memory costs and capacity limits.

The nghttp3 case makes this boundary explicit: its setting advertises a limit, but the application must enforce it. h3 illustrates a different boundary: a decoded field can pass through the decoder and later trigger a panic during insertion into HeaderMap.

The question is simple: which component actually rejects the oversized headers? If that responsibility belongs to the application, it needs to know and enforce the limit before retaining them.

Conclusion

A small HEADERS frame can represent a lot of work. Across the implementations tested, that work turned into large allocations, retained header sets, or a panic when a downstream container reached its limit. The protocol already provides a way to account for decoded header size, but the receiving code has to enforce it.

Checking compressed frame size is merely the starting point. The decoded budget needs to include the per-field overhead, apply while headers are being produced, and cover what the application retains. Several projects have now fixed these paths. The remaining findings show how an advertised limit can look reassuring while providing no protection against the input it was meant to bound.

Finding affected servers

To find impacted publicly facing servers in your fleet, here are the JA4Scan-QUIC fingerprints that map to the different servers:

Library	**JA4Scan-QUIC**
quic-go v0.56.0	031g0hlss000_5973261134f6_6d6a17008abe_sr10sc7b63be
quiche 0.29.2	03100hlss200_8960528bd01f_6d6a17d04082_sr20ss7fb1bf
OLS 1.9.1 (lsquic 4.8.2)	032g0h0ss200_24e1d2569964_d1082305b91e_0r100s0b3f78
Bun 1.4.3	032g0h0ss200_756c1a5a0530_67f99a05b91e_s010s0fecef0
aioquic 1.7.0	0310000ss000_2a673bc4abf8_db4d1fe4882a_sr1s0seb0750
mitmproxy 12.2.3	0310000ss000_2a673bc4abf8_aee409e4882a_sr2s0seb0750
h3 0.0.8	031g0hlsr200_8960528bd01f_6d6a17129383_se10s0fecef0

Read the full blog post about JA4Scan here: https://foxio.io/blog/introducing-ja4scan-active-server-fingerprinting-for-tls-and-quic

POC

Reproduction code is available at https://github.com/FoxIO-LLC/blog_poc/tree/main/headeraccordions

Timeline

- Nov 9 2025 - quic-go report
- Nov 14 2025 - Wireshark report
- Nov 15 2025 - quiche report
- Nov 21 2025 - quic-go 0.57.0 released
- Dec 3 2025 - Wireshark 4.6.2 release
- Dec 4 2025 - h3 report
- Dec 5 2025 - h3 maintainer closes issue - alpha library + default setting to be changed
- Dec 17 2025 - nghttp3 documents *max_field_section_size* as signaling-only
- May 24 2026 - Bun report
- July 13 2026 - aioquic report
- July 13 2026 - lsquic report
- July 14 2026 - quiche 0.29.3 released
- Oct 20 2026 - blog published

References

- RFC 9114 - HTTP/3, Section 4.2.2 (Header Size Constraints)
- RFC 9204 - QPACK: Field Compression for HTTP/3
- nghttp3 - Clarify field size limits: https://github.com/ngtcp2/nghttp3/pull/445
- CVE-2025-64702 (quic-go): https://github.com/quic-go/quic-go/security/advisories/GHSA-g754-hx8w-x2g6
- CVE-2025-13945 (Wireshark): https://www.wireshark.org/security/wnpa-sec-2025-07.html
- CVE-2026-12523 (Cloudflare quiche): https://github.com/cloudflare/quiche/security/advisories/GHSA-4fgf-9xrr-88gf
- Gal Bar Nahum - MadeYouReset HTTP/2: https://galbarnahum.com/made-you-reset

Previous Next