HTTP/2 Bomb: Anatomy of an HPACK Indexed-Reference Amplification Chain Attack

A Protocol-Level Analysis of Vulnerability Class, Cross-Implementation Impact, and Remediation Patterns

Zhicheng Wu

Xinhui No.1 Middle School, Class of 2028

Based on coordinated vulnerability disclosure by

Quang Luong, Jun Rong, Duc Phan — Calif Security Research

Synthesis & Technical Deep-Dive

June 13, 2026

Abstract

On June 2, 2026, security researchers at Calif disclosed a novel class of remote denial-of-service vulnerability affecting default HTTP/2 configurations across all five major web server implementations: nginx, Apache httpd, Microsoft IIS, Envoy Proxy, and Cloudflare Pingora. The attack chains two decade-old protocol abuse primitives—an HPACK indexed-reference compression bomb and an HTTP/2 flow-control window stall—to achieve memory amplification ratios ranging from 68:1 to 5,700:1, enabling a single 100 Mbps client to exhaust 32 GB of server memory in as few as 10 seconds. This paper presents a comprehensive technical analysis of the attack mechanism at the protocol and implementation level, documents the specific code-level fixes applied by each vendor, traces the vulnerability's lineage through five predecessor CVEs spanning a decade, and identifies the root specification defect in RFC 7541 §7.3. We further present a general defensive model based on the principle that *maximum decoded header size* and *maximum header field count* are two independent limits, both of which any HTTP/2 termination point must enforce.

Keywords: HTTP/2, HPACK, denial-of-service, amplification attack, flow control, header compression, CVE-2026-49975, CVE-2026-47774

1 Introduction

HTTP/2, standardized in RFC 7540 [1] and subsequently revised in RFC 9113 [3], introduced binary framing, multiplexed streams, and header compression through HPACK (RFC 7541 [2]) as a replacement for HTTP/1.1's plaintext header model. These efficiency features have driven HTTP/2 adoption to over 70% of web traffic [4]. However, the same mechanisms that improve performance—stateful header compression and per-stream flow control—also create new classes of resource exhaustion attack surface.

On June 2, 2026, Calif Security Research published a coordinated disclosure [5, 6] of a remote denial-of-service vulnerability dubbed *HTTP/2 Bomb* that chains two decade-old HTTP/2 abuse primitives into a single, high-amplification attack chain. The vulnerability was discovered by Quang Luong working with OpenAI's Codex model, which analyzed the codebases of five major HTTP/2 server implementations and recognized that an HPACK compression amplification technique (CVE-2016-6581, documented by Cory Benfield in 2016 [7]) could be composed with an HTTP/2 flow-

control window stall (building on CVE-2016-8740 [8] and CVE-2016-1546 [9]) to create an attack more severe than either primitive alone.

The first author encountered this vulnerability while browsing Bilibili one morning, where a video covering the disclosure caught his attention. Struck by the elegance of the attack chain and the scale of its impact—affecting every major HTTP/2 implementation simultaneously—he set out to conduct a systematic technical deep-dive, which ultimately became this paper.

This paper provides a comprehensive technical analysis of the HTTP/2 Bomb vulnerability class. Our contributions are:

1. A protocol-level analysis of the attack mechanism, including the precise HPACK encoding that achieves the amplification and the flow-control sequence that pins memory.
2. A code-level analysis of the fixes applied by each of the five affected implementations.
3. A historical CVE lineage tracing the evolution of this vulnerability class from 2016 to 2026.
4. An identification of the root specification defect in RFC 7541 §7.3.
5. A general defensive model applicable to HTTP/2 termination points and future protocol designs.

2 Background: HPACK and HTTP/2 Flow Control

2.1 HPACK Header Compression

HPACK (RFC 7541) is a stateful header compression scheme that uses two lookup tables: a *static table* (indices 1–61) containing common header fields predefined in the specification, and a *dynamic table* (starting at index 62) that accumulates header name-value pairs over the lifetime of an HTTP/2 connection. Both tables are combined into a single address space for index values [2].

A sender can use one of four header field representations:

1. **Indexed Header Field** (`0x80 | index`): A single byte referencing a pre-existing entry in the static or dynamic table. The most compact representation.
2. **Literal with Incremental Indexing** (`0x40 | ...`): The header name and value are sent literally, then added to the dynamic table for future reference.
3. **Literal without Indexing** (`0x00 | ...`): Sent literally, not added to the table.
4. **Literal Never Indexed** (`0x10 | ...`): Sent literally, explicitly excluded from indexing (e.g., for sensitive headers).

The dynamic table maintains entries in first-in, first-out order. When a new entry is added that would exceed the table's maximum size (negotiated via `SETTINGS_HEADER_TABLE_SIZE`, default 4,096 bytes), the oldest entries are evicted. The encoder has full control over which entries to index and when to evict [2].

2.2 HTTP/2 Flow Control

HTTP/2 implements a credit-based flow control scheme at two levels: per-stream and per-connection (RFC 9113 §5.2). Each receiver advertises a window size indicating how many octets it is willing to receive. A sender must not transmit `DATA` frames beyond the receiver's advertised window. The initial window size for new streams defaults to 65,535 octets and can be modified via the `SETTINGS_INITIAL_WINDOW_SIZE` parameter (0x04) [3].

Crucially, the client and server independently control their respective receive windows. The client advertises the window within which the server may send response data. By setting `INITIAL_WINDOW_SIZE` to 0, the client effectively tells the server that it cannot receive any response data whatsoever.

Window replenishment occurs through `WINDOW_UPDATE` frames (type 0x8), whose payload is a 31-bit unsigned integer indicating the number of additional octets the sender may transmit above the existing window [3].

3 Attack Mechanism

The HTTP/2 Bomb chains two primitives on a single multiplexed HTTP/2 connection:

Phase 1: HPACK Indexed-Reference Bomb — Force the server to allocate large amounts of memory by sending thousands of single-byte HPACK indexed references, each triggering a full per-entry allocation on the server.

Phase 2: Flow-Control Window Stall — Prevent the server from ever completing its response (and thus freeing the allocated memory) by advertising a zero-byte flow-control window and periodically dripping 1-byte `WINDOW_UPDATE` frames to reset the send timeout.

3.1 Phase 1: HPACK Indexed-Reference Bomb

The attacker constructs a header block with the following HPACK encoding:

Step A: Seed the Dynamic Table

```
0x40 0x06 "x-bomb" 0x00
```

This is a Literal with Incremental Indexing representation: the header name `"x-bomb"` (6 bytes) with an empty value (0 bytes). The entry is added to the dynamic table at index 62. Total wire cost: 9 bytes. Server-side cost: one dynamic table entry, approximately 38–59 bytes depending on implementation details.

Step B: Emit Thousands of Indexed References

```
0xBE  x N     -- where 0xBE = 0x80 | 62
```

Each `0xBE` byte is an Indexed Header Field reference to dynamic table entry 62. On the wire: 1 byte. On the server: the implementation must look up the entry and materialize a complete header structure. Measurements by Calif [5] show that this costs approximately 59 bytes of pool memory per reference on nginx, broken down as 3 bytes in `state.pool` plus 56 bytes in the `ngx_table_elt_t` structure.

3.2 Why Existing Defenses Fail

The critical insight that distinguishes HTTP/2 Bomb from its predecessors is *where the amplification originates*. The classic HPACK Bomb (CVE-2016-6581 [7]) fills the dynamic table with a header value equal to the maximum table size, then references it repeatedly, achieving compression ratios of 4,096:1 or higher. Servers learned to defend against this by capping the *total decoded header size*.

The HTTP/2 Bomb inverts this: the header value is nearly empty (`""`). The decoded-size limit never fires because there is almost nothing to decode. The amplification comes from the *per-entry bookkeeping* the server allocates around each reference—the pool blocks, struct overhead, and internal metadata that surround each header entry regardless of its value size [5].

3.3 The Cookie Crumb Bypass (Apache and Envoy)

For servers that enforce a header *field count* limit rather than (or in addition to) a decoded size limit, the attacker employs a secondary bypass. RFC 9113 §8.2.3 explicitly permits splitting the `Cookie` header into multiple individual fields (termed "crumbs") to improve HPACK compression efficiency. The server is expected to reassemble these crumbs into a single HTTP/1.1-compatible `Cookie` header before forwarding to backend services.

The vulnerability arises because Apache's `mod_http2` and Envoy's HTTP/2 codec did not count cookie crumbs against their respective field-count limits. In Apache, each crumb was excluded from the `LimitRequestFields` tally; in Envoy, the `max_request_headers_kb` check was applied before cookie reassembly. The attacker can therefore send thousands of individual cookie crumbs, each an HPACK indexed reference, and slip under both the size-based and count-based limits simultaneously.

3.3.1 Envoy Amplification Path

Envoy appends each arriving cookie crumb into a growing buffer. The allocator overhead compounds with each append, producing a measured amplification of approximately 5,700:1 on a single stream [5, 20]. Critically, Envoy validates `max_request_headers_kb` *before* merging cookie fragments. The merged cookie total—which can reach 126.9 MiB from 32,768 crumbs—completely bypasses the size check [10].

3.3.2 Apache httpd Amplification Path

Apache's `mod_http2` exhibits even more aggressive memory consumption due to its cookie merging implementation. The function `h2_req_add_header()` calls `apr_psprintf(pool, "%s; %.*s", existing, ...)` for each arriving crumb, allocating a new APR pool string that concatenates the previous result. Critically, older intermediate strings survive until stream cleanup due to APR's pool-based memory management design [14]. This produces *quadratic memory growth*:

$$\text{Memory} = \sum_{i=1}^{N} (2i + 1) \approx \mathcal{O}(N^2) \tag{1}$$

With 4,091 empty cookie crumbs, the *final* merged cookie value is only 8,182 bytes (well within the default `LimitRequestFieldSize` of 8,190 bytes), but the accumulated intermediate pool allocations consume approximately 16 MB per stream. At 100 concurrent streams per connection, this exceeds 1.5 GB [5].

Table 1 summarizes the amplification characteristics across implementations.

Table 1: Per-implementation amplification characteristics

Implementation	**Amp.**	**32 GB**	**Attack Vector**
Envoy 1.37.2	∼5,700:1	∼10 s	Cookie crumb + buffer append
Apache httpd 2.4.67	∼4,000:1	∼18 s	Cookie crumb + quadratic rebuild
nginx 1.29.7	∼70:1	∼45 s	Bookkeeping (`ngx_table_elt_t`)
Microsoft IIS	∼68:1	∼45 s	Bookkeeping overhead
Cloudflare Pingora	∼68:1	—	Bookkeeping overhead

3.4 Phase 2: Flow-Control Window Stall

Phase 1 achieves large memory allocation. Phase 2 prevents the memory from being freed. The precise HTTP/2 frame sequence is:

HTTP/2 Frame Sequence

```
Client -> Server:
  PRI * HTTP/2.0\r\n\r\nSM\r\n\r\n          (connection preface)
  SETTINGS [INITIAL_WINDOW_SIZE=0]           (stall: no response
     allowed)
  SETTINGS ACK                               (ack server's SETTINGS
     )
  HEADERS [stream 1] <- bomb payload         (pseudo-headers +
     HPACK bomb)
  HEADERS [stream 3] <- bomb payload
  HEADERS [stream 5] <- bomb payload
  ...
  WINDOW_UPDATE(1) per stream / 1 second     (keep-alive: reset
     send_timeout)
```

The key elements are:

- `SETTINGS_INITIAL_WINDOW_SIZE = 0`: The server processes the `HEADERS` frames, allocating all bomb-triggered memory, but cannot send any `DATA` in response.
- The server's send timeout would normally close the connection after a period of inactivity, freeing the memory. The attacker defeats this by sending a 1-byte `WINDOW_UPDATE` approximately once per second, resetting the server's internal timer without meaningfully advancing the flow-control window.
- The attack is invisible in HTTP access logs because the server never completes processing the request; no log entry is written [15].

3.5 Why the Combination is Synergistic

Neither primitive is devastating in isolation. HPACK amplification creates large allocations, but those allocations would normally be freed within milliseconds. A flow-control stall can keep resources occupied, but the base memory per stalled stream is typically small (kilobytes). The combination creates a multiplicative effect: high-amplitude memory inflation pinned for attacker-controlled durations [5, 16].

4 The Specification Defect

RFC 7541 §7.3, titled "Memory Consumption", opens with the warning that "an attacker can try to cause an endpoint to exhaust its memory" and proceeds to explain that HPACK bounds the dynamic table via `SETTINGS_HEADER_TABLE_SIZE`. The section then concludes the matter is handled.

Five independent implementation teams read this section and shipped the same class of vulnerability. We identify two distinct specification defects:

4.1 Defect 1: Amplification Ratio as Insufficient Model

The specification frames memory risk purely in terms of *compression ratio*—the relationship between wire bytes and decompressed output bytes. But as the HTTP/2 Bomb demonstrates, ratio is only half the equation. A 70:1 amplifier is harmless if memory is freed when the request completes; it becomes a devastating attack when the protocol permits the client to hold the connection open almost for free, pinning every allocated byte for attacker-controlled durations [5].

4.2 Defect 2: Per-Entry Allocator Overhead is Invisible

The specification treats `SETTINGS_HEADER_TABLE_SIZE` as a sufficient bound on HPACK-related memory consumption. However, this setting controls only the *compression state*—the dynamic table itself. It does not bound the temporary allocations that a server creates while processing decoded header lists. The per-entry bookkeeping overhead—pool blocks, `ngx_table_elt_t` structures, zero-termination copies—is entirely outside the specification's memory model [5, 24].

5 Code-Level Fix Analysis

5.1 nginx — Version 1.29.8

Commit: `3656941` (Maxim Dounin, April 7, 2026)
Origin: Imported from freenginx changeset `199dc0d6b05b` (May 24, 2024)
No independent CVE assigned.

The fix introduces a new `max_headers` configuration directive with a default limit of 1,000 headers per request. The implementation spans three source files [21]:

- `src/http/ngx_http_core_module.c`: Declares the `max_headers` parameter, initializes with `NGX_CONF_UNSET_UINT`, merges with default value `1000`.
- `src/http/ngx_http_request.c`: In the HTTP/1 request parser, increments a counter after each header line and rejects with `NGX_HTTP_REQUEST_HEADER_TOO_LARGE` (HTTP 431) when the limit is exceeded.
- `src/http/v2/ngx_http_v2.c`: Applies the same counter check in the HTTP/2 header handling path before pushing a parsed header into `r->headers_in.headers`.

The critical enforcement code [21]:

```
if (r->headers_in.count++ >= cscf->max_headers) {
    r->close = 1;
    ngx_log_error(NGX_LOG_INFO, c->log, ...);
    ngx_http_finalize_request(r, NGX_HTTP_REQUEST_HEADER_TOO_LARGE);
}
```

The fix applies uniformly to both HTTP/1 and HTTP/2 code paths. The counter is incremented *before* the `ngx_list_push()` call, preventing the allocation from occurring once the limit is reached.

Notable: Ubuntu Security Team [23] reported that this fix "breaks ABI and introduced a regression causing nginx to crash when being used with external modules. The CVE fix was reverted in 8398-2 pending further investigation."

5.2 Apache httpd — CVE-2026-49975

Commit: `47d3100b` (Stefan Eissing, May 27, 2026)
Affected versions: 2.4.17 through 2.4.67
Fix: mod_http2 v2.0.41, Apache 2.4.68
CVSSv4: 8.7 (HIGH) — `AV:N/AC:L/AT:N/PR:N/UI:N/VC:N/VI:N/VA:H`

Apache's fix consists of three concurrent changes [14, 11]:

1. **Repeated headers now counted against the `LimitRequestFields` directive.** Previously, `apr_table_mergen` collapsed duplicate empty headers into a single logical entry for counting purposes while the memory pool allocated storage for each occurrence. The fix makes the count reflect the actual number of allocations.
2. **Zero-terminated copies moved to a connection-level scratch buffer.** Instead of allocating a new pool string for each header name/value copy, the code now uses a single reusable

buffer held at the connection level, eliminating the per-allocation overhead that was the primary amplification vector.

3. **HTTP/2 request pool system memory returned at request end.** Previously, the APR pool's design—where `apr_pool_clear()` retains memory for reuse—allowed allocations to persist across requests on the same HTTP/2 connection. The fix forces the return of system memory when a request completes.

Partial mitigation note: Lowering `LimitRequestFieldSize` reduces the per-stream blast radius (it caps the merged cookie, and thus the crumb count), but an attacker can still multiply the effect across streams and connections. Lowering `LimitRequestFields` alone is ineffective because the duplicate cookie crumbs were not counted against it [5].

5.3 Envoy — CVE-2026-47774 (GHSA-22m2-hvr2-xqc8)

Patched versions: 1.35.11, 1.36.7, 1.37.3, 1.38.1 (June 3, 2026)
Affected: All versions prior to 1.39
CVSSv4: 7.5 (HIGH)

The Envoy advisory [10] identifies two interacting weaknesses that must both be addressed for a complete fix:

Fix 1 — Cookie byte accounting in header size validation. During HTTP/2 request processing, cookie header fragments were buffered separately and merged only *after* request header size validation completed. The merged cookie total was not subject to `max_request_headers_kb` enforcement. The fix ensures buffered cookie bytes are included in request header size accounting before request acceptance.

Fix 2 — Decoded header size limits. The internal `oghttp2/quiche` codec enforced header block limits on *encoded* HPACK bytes only, without a corresponding limit on total *decoded* header size. An attacker could use dynamic table references to keep the encoded representation small (36,844 bytes) while the decoded cookie value expanded to 133 MB. The fix enforces limits on decoded header size in addition to encoded block size.

The advisory explicitly states: "A complete fix requires addressing both contributing issues. Fixing only one side may reduce exploitability but does not fully address the underlying issue." [10]

Follow-up releases (v1.35.12, 1.36.8, 1.37.4, 1.38.2, ∼June 10, 2026): Added observability instrumentation—histograms for `header_count`, `header_list_size`, `cookie_count`, and `cookie_size`—plus a dedicated `http2_max_cookies_size_in_kb` runtime configuration to aid operators with legitimate high-cookie traffic [22].

5.4 Microsoft IIS — CVE-2026-49160

Microsoft addressed the vulnerability through its June 10, 2026 Patch Tuesday release. The specific fix mechanism has not been publicly documented in detail, but is understood to follow the general pattern of enforcing a header field count limit independent of decoded size.

5.5 H2O — GHSA-qcrr-wrhc-pgq9

Commit: `9265bdd` (Kazuho Oku, June 3, 2026)

H2O implements a dual-limit strategy [13]:

- `H2O_HPACK_MAX_HEADERS_HARD_LIMIT = 1000`: Exceeding this limit causes the connection to be closed immediately.
- `H2O_MAX_HEADERS = 100`: Exceeding this limit returns an HTTP 400 (Bad Request) error.

H2O already represents HTTP header names and values internally as references where possible, reducing HPACK state amplification by design. The new limits provide explicit bounds on decoded header state and prevent amplified state from being retained by stalled streams.

5.6 HAProxy — Architecturally Immune

HAProxy's HTTP/2 implementation is not vulnerable to the HTTP/2 Bomb exploit. Its core design treats HTTP/2 streams with strict, fixed-size memory constraints and processes frames at line-rate speeds without dynamic allocation accumulation. The memory footprint of individual connections and streams is bounded by design, preventing the unbounded consumption that characterizes this attack class [19].

6 Historical CVE Lineage

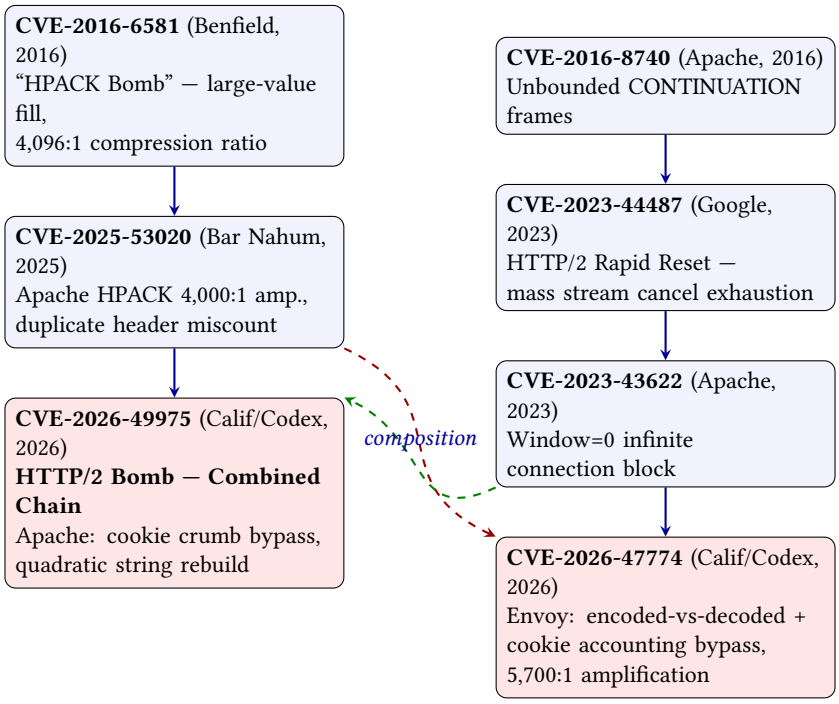

Figure 1: Historical lineage of the HTTP/2 amplification vulnerability class. Two independent strands—HPACK compression amplification (left column) and flow-control resource exhaustion (right column)—evolved in parallel from 2016 before being composed into a single attack chain in 2026. Dashed cross-arrows indicate that both strands contributed to both implementations. The two 2026 CVEs are parallel disclosures from the same coordinated release.

Figure 1 traces the evolution of this vulnerability class across two parallel strands. The HPACK compression amplification strand (left column) progressed through the original HPACK Bomb (CVE-2016-6581) to Gal Bar Nahum's independent rediscovery achieving 4,000:1 amplification against Apache (CVE-2025-53020). The flow-control and resource exhaustion strand (right column) progressed through unbounded CONTINUATION frames (CVE-2016-8740), the cross-implementation Rapid Reset attack (CVE-2023-44487), and explicit zero-window connection blocking (CVE-2023-43622). Both strands converge in the 2026 coordinated disclosure: CVE-2026-49975 (Apache) and CVE-2026-47774 (Envoy), which are parallel findings from the same research effort targeting different implementations.

What Codex identified was that these two strands could be *composed*—that the memory amplification of the first strand could be *pinned* by the stall mechanism of the second strand. As Calif's disclosure notes: "That combination is obvious once you see it, and yet as far as we can tell no human had put it together against these servers." [5]

7 General Defensive Model

From the analysis of the attack mechanism and the vendor fixes, we derive a general defensive model for HTTP/2 termination points. The core principle, articulated by Calif's research team, is:

> ***"Maximum decoded header size" and "maximum header count" are two different limits, and a server needs both.***

7.1 Defensive Controls Matrix

Table 2: Defensive controls and their coverage

Control	Defends Against	Sufficiency
Max decoded header size	Traditional HPACK bomb (CVE-2016-6581)	Insufficient alone — bypassed by near-empty headers
Max header field count (incl. cookie crumbs)	Indexed-reference bomb	**Required** — directly caps amplification window
Max reassembled cookie size	Cookie-specific amplification	Partial — does not address non-cookie per-entry overhead
Stalled stream lifetime bound (independent of `WINDOW_UPDATE`)	Flow-control memory pinning	**Required** — prevents indefinite retention
Per-worker memory cap (cgroups, `ulimit -v`)	Blast radius containment	Last line of defense — worker OOM-killed before host enters swap

7.2 Configuration Recommendations

For nginx 1.29.8+:

```
max_headers 1000;              # default; lower if application
   permits
http2_max_concurrent_streams 32;
http2_max_field_size 4k;
http2_max_header_size 16k;
send_timeout 10s;
```

For Apache httpd (mod_http2 $\geq$ 2.0.41):

```
H2MaxSessionStreams 32
LimitRequestFields 100
LimitRequestFieldSize 8190
H2MaxRequestsPerConn 100
```

For Envoy ($\geq$ 1.38.1 or equivalent):

```
http_connection_manager:
  max_request_headers_kb: 128
  common_http_protocol_options:
    max_headers_count: 1000
```

8 Discovery Methodology and AI Role

The HTTP/2 Bomb was discovered through an AI-assisted methodology that carries implications for vulnerability research practice. Quang Luong at Calif worked with OpenAI's Codex model, which was directed to read the codebases of five major HTTP/2 server implementations simultaneously [5, 17, 18].

Both components of the attack—the HPACK Bomb (CVE-2016-6581) and the HTTP/2 Slowloris variants (CVE-2016-8740, CVE-2016-1546)—had been publicly documented for nearly a decade. What Codex recognized was that the two techniques could be *composed* within a single HTTP/2 connection to achieve multiplicative effect.

This discovery pattern—an AI model with the capacity to read across multiple independent codebases simultaneously—reveals a structural vulnerability in the traditional model of per-project security review. The composition lived in the seams between teams, and, as Calif notes, "nobody owns the seams" [25]. The disclosure suggests that cross-codebase analysis, whether AI-assisted or otherwise, represents an increasingly important tool for identifying protocol-level vulnerability classes that manifest identically across independent implementations.

9 Conclusion

The HTTP/2 Bomb represents a significant vulnerability class at the intersection of protocol specification, implementation diversity, and the composition of independently-understood primitives. Five major implementations, studied independently by their respective teams, each read RFC 7541 §7.3 and arrived at the same incomplete model of memory risk. The specification's framing of risk purely as compression ratio, combined with its failure to account for per-entry allocator overhead and flow-control-mediated memory pinning, created a class of vulnerability that lay dormant for a decade before being identified through cross-codebase AI analysis.

The fix pattern is consistent across implementations: enforce both a decoded header size limit *and* a header field count limit, independently; include cookie crumbs in the field count; and bound stalled stream lifetimes regardless of `WINDOW_UPDATE` activity. These principles should inform future protocol designs, particularly as HTTP/3 and QUIC become more widely deployed with their own compression and flow-control semantics.

The coordinated disclosure process functioned effectively: nginx shipped a fix the day after notification (April 2026); Apache committed a fix the same day of disclosure (May 27, 2026); Envoy

released patches within 24 hours of public disclosure (June 3, 2026); and Microsoft followed through its regular Patch Tuesday cycle (June 10, 2026). The speed of response underscores both the severity of the vulnerability and the simplicity of the fix once the defect is correctly identified.

Acknowledgments

The authors acknowledge Quang Luong, Jun Rong, and Duc Phan at Calif Security Research for the original discovery and coordinated disclosure; Cory Benfield for the original HPACK Bomb research in 2016; Gal Bar Nahum for the independent Apache HPACK amplification research in 2025; Stefan Eissing for Apache httpd's same-day fix; the nginx, Envoy, H2O, and HAProxy maintainers for their rapid response; and the oss-security community for facilitating coordinated disclosure.

Special thanks to **Laodeng** for generously providing the Ark Coding Plan that supported this work.

References

[1] M. Belshe, R. Peon, and M. Thomson, Eds., "Hypertext Transfer Protocol Version 2 (HTTP/2)," RFC 7540, IETF, May 2015. https://www.rfc-editor.org/rfc/rfc7540

[2] R. Peon and H. Ruellan, "HPACK: Header Compression for HTTP/2," RFC 7541, IETF, May 2015. https://www.rfc-editor.org/rfc/rfc7541

[3] M. Thomson and C. Benfield, Eds., "HTTP/2," RFC 9113, IETF, June 2022. https://www.rfc-editor.org/rfc/rfc9113

[4] Cloudflare, "Cloudflare Radar: Internet Traffic Trends," https://radar.cloudflare.com

[5] Q. Luong, J. Rong, and D. Phan, "Codex Discovered a Hidden HTTP/2 Bomb," Calif Security Research Blog, June 2, 2026. https://blog.calif.io/p/codex-discovered-a-hidden-http2-bomb

[6] Q. Luong et al., "HTTP/2 Bomb affects Apache httpd, nginx, envoy, & pingora," oss-security mailing list, June 2, 2026. https://seclists.org/oss-sec/2026/q2/790

[7] C. Benfield, "CVE-2016-6581: HPACK Bomb — Python HPACK library," oss-security mailing list, August 4, 2016. https://openwall.com/lists/oss-security/2016/08/04/3

[8] MITRE, "CVE-2016-8740: Apache httpd unbounded CONTINUATION frames," NVD, 2016.

[9] MITRE, "CVE-2016-1546: Apache httpd worker-thread starvation," NVD, 2016.

[10] Envoy Proxy Security, "GHSA-22m2-hvr2-xqc8: HTTP/2 memory exhaustion via cookie header size bypass and HPACK amplification (CVE-2026-47774)," GitHub Security Advisory, June 3, 2026. https://github.com/envoyproxy/envoy/security/advisories/GHSA-22m2-hvr2-xqc8

[11] S. Eissing, "mod_http2: Fix cookie header accounting against LimitRequestFields (CVE-2026-49975)," Apache httpd Git, commit 47d3100b, May 27, 2026. https://github.com/apache/httpd/commit/47d3100b252dc6668a9e46ae885242be9eeca9cd

[12] M. Dounin, "HTTP/2: added max_headers directive (imported from freenginx)," nginx Git, commit 3656941, April 7, 2026. https://github.com/nginx/nginx/commit/365694160a85229a7cb006738de9260d49ff5fa2

[13] K. Oku, "GHSA-qcrr-wrhc-pgq9: HTTP/2 state amplification," H2O Security Advisory, June 3, 2026. https://github.com/h2o/h2o/security/advisories/GHSA-qcrr-wrhc-pgq9

[14] S. Eissing, "HPACK Bombing Apache (CVE-2025-53020)," Personal Blog, 2025. https://github.com/icing/blog/blob/main/hpack-bombing-apache.md

[15] B. Nowotarski, "HTTP/2 CONTINUATION Flood," Personal Blog, April 3, 2024. https://nowotarski.info/http2-continuation-flood/

[16] Flowtriq Security, "HTTP/2 Bomb: How a Single Machine Can Exhaust 32 GB of Server RAM in Seconds," Flowtriq Blog, June 4, 2026. https://flowtriq.com/blog/http2-bomb-dos-attack-hpack-compression

[17] Cloud Security Alliance, "CSA Research Note: HTTP/2 Bomb — AI-Discovered DoS Hits Every Major Web Server," CSA Labs, June 4, 2026. https://labs.cloudsecurityalliance.org/research/csa-research-note-http2-bomb-ai-discovered-dos-20260604-csa/

[18] J. Lyons, "OpenAI's Codex chains decade-old DoS techniques into HTTP/2 Bomb," The Register, June 4, 2026. https://www.theregister.com/security/2026/06/04/openais-codex-chains-decade-old-dos-techniques-into-http2-bomb/

[19] HAProxy Technologies, "Protecting against HTTP/2 Bomb vulnerability (CVE-2026-49975) with HAProxy," HAProxy Blog, June 5, 2026. https://www.haproxy.com/blog/haproxy-cve-2026-49975-http2-bomb

[20] lilting.ch, "HTTP/2 Bomb: 5,700x Envoy, 4,000x Apache amplification via HPACK + flow control," June 4, 2026. https://lilting.ch/en/articles/http2-bomb-hpack-flow-control-dos

[21] PatchLeaks Security Research, "CVE-2026-49975: nginx exploit & PoC analysis," June 6, 2026. https://pwn.az/articles/nginx/CVE-2026-49975

[22] Y. Vlasov, "Mitigation Recommendation for CVE-2026-47774," Envoy GitHub Issue #45483, June 5, 2026. https://github.com/envoyproxy/envoy/issues/45483

[23] Ubuntu Security Team, "CVE-2026-49975 — HTTP/2 Bomb denial of service," Ubuntu CVE Database, June 3, 2026. https://ubuntu.com/security/CVE-2026-49975

[24] Penligent Security, "CVE-2026-49975, the HTTP/2 Bomb Behind Header Limits," June 5, 2026. https://www.penligent.ai/hackinglabs/cve-2026-49975/

[25] K. Kierii, "HTTP/2 Bomb (CVE-2026-49975): the HPACK + flow-control DoS, and how to patch it," DEV Community, June 4, 2026. https://dev.to/kkierii/http2-bomb-cve-2026-49975-26ba

[26] Red Hat, "RHSB-2026-007: HTTP/2 HPACK Denial of Service — httpd, nginx, Envoy," Red Hat Security Bulletin, June 6, 2026. https://access.redhat.com/security/vulnerabilities/RHSB-2026-007