Dealing with H2/H3 abuses

Amaury Denoyelle – Willy Tarreau (HAProxy)

HTTP Workshop 2024

Background 1/2

- Historically haproxy has been dealing with floods using counters stored in tables (typically per-src, but not only)
- Rules applied at various levels allow to consider the counters to decide to reject the traffic / mark the source etc.
- Easily deal with connection rate, TLS handshake rate, request rate, concurrency etc.

Background 2/2

- H2 prone to various “work amplification attacks”, not as severe as DoS but still annoying
- No third-party involved, cost ratio between frame emission and processing quite high (typically CPU)
- An H2 frame is 9 bytes minimum and 16kB max by default. Let’s see how much work this can induce.

Classics

Request flood

- `:method`, `:scheme`, `:path`, `:authority` are mandatory. Down to 4 bytes with HPACK.
  => 13 bytes per request
  >100 requests per TCP segment (@1448)
- Haproxy parses ~1.3M req/s/core => ~150 Mbps sufficient to saturate one core
- “easy” to deal with using request-level rules

Classics

Request parallelism

- Open as many concurrent streams as permitted on a connection (typically 100)
- Very cheap (1 TCP segment, 1 source port)
- GET, POST, partial POST, 100-continue
- Mostly RAM usage (esp. with WAF), can be huge (>1 MB)
- Not easy to distinguish valid from abuse
  => reduce the concurrent streams limit

Classics

Request+RST flood

- Same as request flood except that client sends RST (aka “rapid reset”)
- Demuxing is paused until application-layer streams are released

=> same impact and handling as classic flood

Classics

Invalid Request flood

- Comparable to first one, but with parsing errors (e.g. missing :authority or invalid chars)
- Respond with RST_STREAM (PROTOCOL_ERROR)
- Request is not instantiated, no actionable ruleset

=> May consume quite a bit of CPU (typ. 1 core per ~150 Mbps) for as long as the attacker wants
=> **moderate impact**

Classics

PING

- Request is not instantiated, no actionable ruleset

=> essentially CPU usage (parse 17 bytes + respond), ~5 Gbps per core (40M frames/s) for as long as the attacker wants
=> **low impact**

Classics

HEADERS + empty/short CONTINUATION

- Request is never terminated
- Request is not instantiated, no actionable ruleset
- Variant: HPACK DTSU opcodes (0x20 to 0x3F)

=> essentially CPU usage (parse 9 bytes, possibly try to parse again), ~2 Gbps per core (26M frames/s) for as long as the attacker wants
=> **low to moderate impact**

Classics

WINDOW_UPDATE (1)

- Stream or connection window grows by 1-byte
- Causes stream lookups
- May cause processing wakeups
- No actionable ruleset

=> essentially CPU usage (parse 13 bytes), may cause multiple wakeups
=> **low to moderate impact**

Classics

Zero-length / small DATA frames

- Frames containing no (or very few) data
- May contain padding
- May cause memcpy() / reallocations
- No actionable ruleset

=> essentially CPU usage (parsing and possibly copies), may cause stream wakeups
=> **low to moderate impact**

Subtle

SETTINGS_INITIAL_WINDOW_SIZE

- Parameter of a SETTINGS frame (9+2+4 bytes)
- Affects **ALL** streams => create many before attacking
- May cause many iterations / wakeups
- No actionable ruleset

=> possibly important CPU usage (loops, many wakeups)
=> **moderate to high impact**

Subtle

PRIORITY

- May be sent in any state for any stream
- May cause stream lookups and/or updates to the dependencies tree
- Not implemented in haproxy but implementations may differ

=> possibly important CPU usage for implementers

Subtle

Abuse of the log system

- Anything that can be cheap to produce and will result in a log being emitted (e.g. invalid request)
- Often encountered and causing victims to disable logs and become less aware of what’s happening
- Addressed using log sampling

=> challenging to figure how to defend

Challenges

- Many frames not subject to rulesets
- No intent to implement per-frame rulesets
- Some special cases are expensive yet valid
=> not possible to break the connection
- Not possible to forbid these cases, despite super rare

=> **their occurrence must remain low**

Solution

- Let’s count the occurrences!

  => introduction of “**glitches**” counters

- Per connection
- Per table
- Accessible from rules (count and rates)

Principle

- Suspicious events increment the glitch counter of the connection
- A soft limit on the connection triggers a soft GOAWAY to renew the connection
- A hard limit forces a connection closure
- Connection updates entry in table if tracked
- New connection can be rejected based on past counter

Tuning

- No good threshold. Some normal connections will show a few units over their life time
  => always log the values
- CPU-intensive abusers will show tens to hundreds of thousands
- Limit just tells how fast to react

Currently monitored

- H2: all protocol violations, new stream reaching limits, window size reduction, ignorance of GOAWAY, frames triggering RST_STREAM, truncated frames, HPACK decompression errors, too large headers, every CONTINUATION frame < 1kB after the 4th one
- H3: all protocol violations, QPACK decompression errors, too large headers

Observations

- Extremely effective, has totally stopped H2 attacks on some large sites
- Attack scripts tend not to respect protocols well, and ignore GOAWAY
- Greasing may increase the counter, just like stop/reload.

Next steps

- Implement positive and negative scores to correct false positives (e.g. large WU may cancel a small WU), not done yet due to effectiveness of the current solution.
- +/- will allow to count ratio of bad to good frames.
- Generalize to other subsystems (H1, TLS)