Technical Report

OpenEXR

Security Assessment

Prepared for:
OSTIF

SHIELDER
WEB SECURITY

1. Document Details

Classification	**Public** - CC BY-SA 4.0
Last review	July 10, 2025
Author	Davide Silvetti, Pietro Tirenna, Nicolò Daprelà

1.1. Version

Identifier	Date	Author	Note
v1.0	March 7, 2025	Davide Silvetti, Pietro Tirenna, Nicolò Daprelà	First version
v1.1	March 12, 2025	Abdel Adim Oisfi	Peer review
v1.2	July 10, 2025	Davide Silvetti, Pietro Tirenna	Public release

1.2. Contacts Information

Company	Name	Position	Contact
Shielder	Abdel Adim Oisfi	CEO / CTO	abdeladim.oisfi@shielder.com
Shielder	Davide Silvetti	Consultant	davide.silvetti@shielder.com
Shielder	Pietro Tirenna	Consultant	pietro.tirenna@shielder.com
OSTIF	Derek Zimmer	Executive Director	derek@ostif.org
OSTIF	Amir Montazery	Managing Director	amir@ostif.org
OSTIF	Helen Woeste	Communications and Community Manager	helen@ostif.org
OSTIF	Tom Welter	Project Manager	tom@ostif.org
Academy Software Foundation	Cary Phillips	OpenEXR Core Developer	cary@ilm.com
Academy Software Foundation	Kimball Thurston	OpenEXR Core Developer	kdt3rd@gmail.com

1.3. About OSTIF

The **Open Source Technology Improvement Fund (OSTIF)** is dedicated to resourcing and managing security engagements for open source software projects through partnerships with corporate, government, and non-profit donors. We bridge the gap between resources and security outcomes, while supporting and championing the open source community whose efforts underpin our digital landscape.

Over the past ten years, OSTIF has been responsible for the discovery of over 800 vulnerabilities, (121 of those being Critical/High), over 13,000 hours of security work, and millions of dollars raised for open source security. Maximizing output and security outcomes while minimizing labor and cost for projects and funders has resulted in partnerships with multi-billion dollar companies, top open source foundations, government organizations, and respected individuals in the space. Most importantly, we've helped over 120 projects and counting improve their security posture.

Our directive is to support and enrich the open source community through providing public-facing security audits, educational resources, meetups, tooling, and advice. OSTIF's experience positions us to be able to share knowledge of auditing with maintainers, developers, foundations, and the community to further secure our infrastructure in a sustainable manner.

We are a small team working out of Chicago, Illinois. Our website is ostif.org. You can follow us on social media to keep up to date on audits, conferences, meetups, and opportunities with OSTIF, or feel free to reach out directly at contactus@ostif.org or our Github.

Derek Zimmer, Executive Director
Amir Montazery, Managing Director
Helen Woeste, Communications and Community Manager
Tom Welter, Project Manager

2. Summary

3. Executive Summary

The document aims to highlight the findings identified during the “Security Assessment” against the "OpenEXR" product described in section “3.2 Context and Scope”.

For each detected findings, the following information are provided:

- **Severity**: findings score ("3.9 Findings Severity Classification").
- **Affected resources**: vulnerable components.
- **Status**: remediation status ("3.10 Remediation Status Classification").
- **Description**: type and context of the detected finding.
- **Impact**: loss of confidentiality, data integrity and/or availability in case of a successful exploitation and conditions necessary for a successful attack.
- **Proof of Concept**: evidence and/or reproduction steps.
- **Suggested remediation**: configurations or actions needed to mitigate the finding.
- **References**: useful external resources.

3.1. Overview

In January 2025, **Shielder** was hired by the **Open Source Technology Improvement Fund (OSTIF)** to perform a *Preliminary Security Review and Threat Model Assessment* of **OpenEXR** (openexr.com), an open specification and reference implementation of the EXR file format, the professional-grade image storage format of the motion picture industry.

The **OpenEXR** file format is a custom binary file format that supports a wide variety of features, including but not limited to multi-part images, lossy and lossless compression, deep data images, etc.

The OpenEXR software is composed of the following components:

- **OpenEXR**, a library exposing C++ APIs to read and write EXR files.
- **OpenEXRCore**, a library implementing low-level functionalities and exposing C APIs.

A team of **3** (three) Shielder engineers worked on this project for a total of 12 (twelve) person-days of effort.

3.2. Context and Scope

The OpenEXR file-format is used by many enterprises and professional software in the VFX, animation, and film industries, namely Pixar RenderMan, Unreal Engine, the Autodesk suite, the Apple VisionPro, NVIDIA Omniverse, etc.

The aim of this *Preliminary Security Review and Threat Model Assessment* was to gain a general understanding of the security posture of the project, to provide project maintainers with valuable recommendations and starting points to incrementally improve it.

Specifically, the main goals were to:

- Perform a high-level Threat Modeling Assessment to understand the common and typical use-cases, high-risk functionalities, and their associated threats.
- Perform an overall high-level manual review of the source code, and the secure-coding practices employed.
- Perform an automated source code analysis with SAST tools like Semgrep and CodeQL.
- Perform a review of the dependencies in use to detect outdated and vulnerable dependencies.
- Perform a review the use of CI and GitHub Workflows.
- Perform a review of the current state of fuzzing coverage, addressing issues with the fuzzers, and improving the coverage of critical codepaths.

The scope of this audit is the **OpenEXR** commit `8bc3faebc66b92805f7309fa7a2f46a66e5cc5c9` released on **January 1, 2025**.

It is important to note that Security Assessments are time-boxed activities performed at a specific point in time; thus, they are unable to guarantee that a software is or will be free of bugs.

3.3. Methodology

The source code audit was carried out following a standard Shielder methodology developed during years of experience. Different testing techniques and approaches were employed.

Moreover, manual and tool-driven techniques were used to analyze the source code. The audit was assisted by SAST tools like CodeQL and Semgrep with publicly available C/C++ queries and rules.

Parallel to the source code audit, a preliminary fuzzing setup and campaign was conducted. The pre-existing fuzzer on OSS-Fuzz were fixed to correctly build and collect coverage, then a grammar-based fuzzer was developed thanks to libprotobuf-mutator.

Finally, Shielder performed a review of the whole release process, including the managing of dependencies, for misconfigurations leading to supply chain attacks, and a review of the documentation for insecure recommendations and/or insecure defaults. This included, for instance, reviewing the configured GitHub actions and workflows for typical attack scenarios, and the approach that OpenEXR employs when using third-party packages in its project.

3.4. Threat Model

Threat Modeling a specification can be slightly more complex when compared to "standalone" applications. When doing so, many things must be taken in consideration since independent implementations could make different choices based on what the specification allows and on its design.

The OpenEXR file format is fairly complex, it supports many parsing mechanism (Scanlines, Tiled, Deep Data), multi-part images, arbitrary number of channels, a number of compression algorithms, etc. The complexity and feature richness of the specification increase the attack surface.

For the reference implementation instead, the main threat is posed by the parsing of untrusted EXR files.

Additionally, the team has reviewed and audited the documented APIs to find "situational" code - e.g. non-core - that may introduce vulnerabilities in the software using the library.

3.5. Audit Summary

The **OpenEXR** project is implemented following robust C++ secure coding principles. However, due to the flexibility of the project, it is affected by some security issues in the design of its parsing logic.

The main problems arise from excessive reliance on the values extracted from the many different kinds of headers (file, part, chunk) while parsing the file.

The Shielder team was able to identify **1** (one) critical, **1** (one) medium and **2** (two) low findings.

3.6. Recommendations

The following list outlines further recommendations for the project's maintainers to harden the security posture of the project.

Perform Null-pointer Checks

The OpenEXR library comes in two different flavors, the C++ APIs and the C core library. Many of the C++ APIs rely on the core library under-the-hood. In both C and C++, dereferencing a null-pointer results in an undefined behavior that could vary between compilers, operating systems, or even execution environments. Most of the cases it will cause a segmentation fault or a memory corruption.

It is recommended to perform null-pointer checks after dereferencing pointers and before accessing them, aborting execution if the target pointer is `NULL`/`nullptr`.

The following list outlines recommendations for third-party developers using the OpenEXR library or specification.

Set a Limit on the Maximum Image Size

The OpenEXR file format defines many information about the final image inside of the file header, such as the size of data/display window. Those fields contain user-supplied values that the library trusts to allocate the memory needed to render the final image.

It is recommended to use the `setMaxImageSize` function from OpenEXR to limit the size of the images that will be loaded based on the target use of the library. This will avoid arbitrarily large memory allocations and crash the application when opening small EXR images that however define huge `data-window` dimensions.

3.7. Long Term Improvements

Due to fast-evolving nature of the Security field and the time-constrained nature of Security Audits, there still is room for long-term improvements to the overall security of the project's ecosystem.

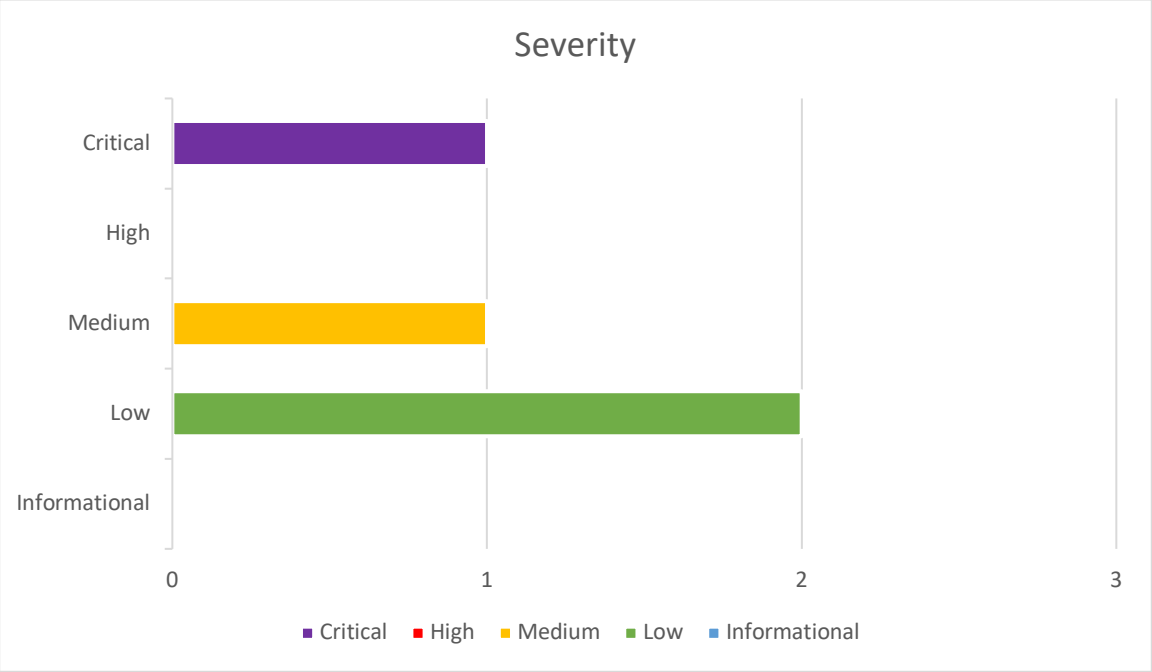
Perform an In-Depth Code Audit

When complexity and flexibility meet in an unsafe memory language like C or C++, it's not uncommon to introduce bugs with a serious security impact.

Considering the major presence of the library in the motion industry, it is recommended to perform a more in-depth audit of the codebase, focusing on finding other bugs that might be exploited by attackers in the wild.

3.8. Results Summary

The following chart shows the number of findings found per severity:



ID	Finding	Severity	Status
1	Heap-Based Buffer Overflow in Deep Scanline Parsing via Forged Unpacked Size	CRITICAL	Closed
2	Out-Of-Memory via Unbounded File Header Values	LOW	Closed
3	Out of Bounds Heap Read due to Bad Pointer Arithmetic in LossyDctDecoder_execute	MEDIUM	Closed
4	ScanLineProcess::run_fill NULL Pointer Write in “reduceMemory” Mode	LOW	Closed

3.9. Findings Severity Classification

Severity	Description
CRITICAL	Vulnerability that allows to compromise the whole application, host and/or infrastructure. In some cases, it allows access, in read and/or write, to highly sensitive data, totally impacting the resources in terms of confidentiality, integrity and availability. Usually, such vulnerabilities can be exploited without the need of valid credentials, without considerable difficulty and with the possibility of automated, highly reliable, and remotely triggerable attacks. Vulnerabilities marked with this severity must be resolved quickly, especially in production environment.
HIGH	Vulnerability that significantly affects the confidentiality, integrity, and availability of confidential and sensitive data. However, the prerequisites for the attack affect its likelihood of success, such as the presence of controls or mitigations and the need of a certain set of privileges.
MEDIUM	Vulnerability that allows to obtain only a limited or less sensitive set of data, partially compromising confidentiality. In some cases, it may affect the integrity and availability of the information, but with a lower level of severity. In addition, the chances of success of such vulnerability may depend on external factors and/or conditions outside the attacker's control.
LOW	Vulnerability resulting in a limited loss of confidentiality, integrity, and availability of data. In some cases, it depends on conditions not aligned to a real scenario or requires that the attacker has access to credentials with a high level of privileges. In addition, a low severity vulnerability may provide useful information to successfully exploit a higher impact vulnerability.
INFORMATIONAL	Problems that do not directly impact confidentiality, integrity, and availability. Usually, these problems indicate the absence of security mechanisms or the improper configuration of them. Mitigation of this type of problem increases the general level of security of the system and allows in some cases to prevent potential new vulnerabilities and/or limit the impact of existing ones.

3.10. Remediation Status Classification

Status	Description
Open	Vulnerability not mitigated or insufficient mitigation.
Not reproducible	Vulnerability not reproducible due to environment changes or to mitigation of other vulnerabilities required in the reproduction steps.
Closed	Vulnerability mitigated. The security patch applied is reasonably robust.

4. Fuzzing Strategy

During the assessment, a preliminary fuzzing campaign was conducted to improve the current fuzzing coverage.

The OpenEXR project was initially on-boarded into OSS-Fuzz in 2020 after a Google's Project Zero campaign. Since then, new fuzzers were developed, and the project matured from a fuzzing and security point-of-view in later versions.

After a brief analysis of the current fuzzers the team discovered that the coverage build was failing due to issues with the compilation process.

Due to the nature of the OSS-Fuzz infrastructure, fuzzers are forced to use a limited amount of memory, and test-cases are limited in size. This is implemented in OpenEXR with two flags (`reduceMemory` and `reduceTime`) that whenever set to `True` will avoid to parse files that are too big.

Moreover, OSS-Fuzz corpus is not public but thanks to the wide distribution of the OpenEXR format the team was able to gather a large sample of initial corpus made of real-world images that were particularly big in size.

Finally, OpenEXR has a total of four implementations of the Huffman decoding algorithm: a fast version and a slow version written in both C and C++. The fast implementations are used most of the time, however in some systems the slow ones will be used (GNU/Linux on ARM). Currently only the fast versions are fuzzed.

For these reasons the fuzzing campaign was focused on test cases that OSS-Fuzz could have missed or ignored. This was done by unsetting the `reduce*` flags, gathering a big and new corpus, and fuzzing the different implementations.

This was partially successful since it uncovered some weaknesses, though they were limited and confined instances, reflecting the project's fuzzing maturity.

In view of this, a new grammar-based fuzzer employing *libprotobuf-mutator* was developed, modeling the EXR format as a Protobuf object. Due to the limited time-frame for this, only the "Simple Scanline" image type was implemented.

5. Findings Details

Analysis results are discussed in this section.

5.1. Heap-Based Buffer Overflow in Deep Scanline Parsing via Forged Unpacked Size

Severity	CRITICAL
Affected Resources	src/lib/OpenEXRCore/chunk.c src/lib/OpenEXRCore/internal_zip.c
Status	Closed

Update

This was fixed in https://github.com/AcademySoftwareFoundation/openexr/pull/1974, which was merged and released in version `v3.3.3`.

Description

The OpenEXRCore code is vulnerable to a heap-based buffer overflow during a write operation when decompressing `ZIPS`-packed deep scan-line EXR files with a maliciously forged chunk header.

When parsing `STORAGE_DEEP_SCANLINE` chunks from an EXR file, the following code (from `src/lib/OpenEXRCore/chunk.c`) is used to extract the chunk information:

```
if (part->storage_mode == EXR_STORAGE_DEEP_SCANLINE)
// ...SNIP...
        cinfo->sample_count_data_offset = dataoff;
        cinfo->sample_count_table_size  = (uint64_t) ddata[0];
        cinfo->data_offset              = dataoff + (uint64_t) ddata[0];
        cinfo->packed_size              = (uint64_t) ddata[1];
        cinfo->unpacked_size            = (uint64_t) ddata[2];
// ...SNIP...
```

By storing this information, the code that will later decompress and reconstruct the chunk bytes knows how much space the uncompressed data will occupy.

This size is carried along in the chain of decoding/decompression until the `undo_zip_impl` function in `src/lib/OpenEXRCore/internal_zip.c`:

```
static exr_result_t
undo_zip_impl (
    exr_decode_pipeline_t* decode,
    const void*            compressed_data,
    uint64_t               comp_buf_size,
    void*                  uncompressed_data,
    uint64_t               uncompressed_size,
    void*                  scratch_data,
```

```
    uint64_t               scratch_size)
{
    size_t       actual_out_bytes;
    exr_result_t res;

    if (scratch_size < uncompressed_size) return EXR_ERR_INVALID_ARGUMENT;

    res = exr_uncompress_buffer (
        decode->context,
        compressed_data,
        comp_buf_size,
        scratch_data,
        scratch_size,
        &actual_out_bytes);

    if (res == EXR_ERR_SUCCESS)
    {
        decode->bytes_decompressed = actual_out_bytes;
        if (comp_buf_size > actual_out_bytes)
            res = EXR_ERR_CORRUPT_CHUNK;
        else
            internal_zip_reconstruct_bytes (
                uncompressed_data, scratch_data, actual_out_bytes);
    }

    return res;
}
```

The `uncompressed_size` comes from the `unpacked_size` extracted earlier, and the `uncompressed_data` is a buffer allocated by making space for the size "advertised" in the chunk information.

However, `scratch_data` and `actual_out_bytes` will contain, after decompression, the uncompressed data and its size, respectively.

The vulnerability lies in the fact that the `undo_zip_impl` function lacks code to check whether `actual_out_bytes` is greater than `uncompressed_size`.

The effect is that, by setting the `unpacked_size` in the chunk header smaller than the actual chunk decompressed data, it is possible - in the `internal_zip_reconstruct_bytes` function - to overflow past the boundaries of a heap chunk.

Impact

An attacker might exploit this vulnerability by feeding a maliciously crafted file to a program that uses the OpenEXR libraries, thus gaining the capability to write an arbitrary amount of bytes in the heap. This could potentially result in code execution in the process.

Proof of Concept

1. Compile the `exrcheck` binary in a macOS or GNU/Linux machine with ASAN.
2. Download the `heap_overflow.exr` file from the https://github.com/ShielderSec/poc repository.
3. Open the file with the following command:

   ```
   exrcheck heap_overflow.exr
   ```

4. Notice that `exrcheck` crashes with an ASAN stack-trace.

Suggested Remediations

Values coming from the file header should not be trusted. Implement code to verify whether the amount of data that was decompressed would fit in the advertised `unpacked_size`.

References

- https://cwe.mitre.org/data/definitions/122.html

5.2. Out-Of-Memory via Unbounded File Header Values

Severity	LOW
Affected Resources	src/lib/OpenEXRUtil/ImfCheckFile.cpp:145-258
Status	Closed

Update

This was partially fixed (handling too large data allocations) in https://github.com/AcademySoftwareFoundation/openexr/pull/1979, which was merged and released in version `v3.3.3`.

A residual risk has been accepted, due to the impossibility of imposing default upper/lower bounds to the dimensions advertised in the OpenEXR header.

Description

The OpenEXR file format defines many information about the final image inside of the file header, such as the size of data/display window.

The application trusts the value of `dataWindow` size provided in the header of the input file and performs computations based on this value.

This may result in unintended behaviors, such as excessively large number of iterations and/or huge memory allocations.

A concrete example of this issue is present in the function `readScanline()` in `ImfCheckFile.cpp` at line 235, that performs a for-loop using the `dataWindow min.y` and `max.y` coordinates that can be arbitrarily large.

```
in.setFrameBuffer (i);

int step = 1;

//
// try reading scanlines. Continue reading scanlines
// even if an exception is encountered
//
for (int y = dw.min.y; y <= dw.max.y; y += step) // <-- THIS LOOP IS EXCESSIVE
BECAUSE OF DW.MAX
{
    try
    {
        in.readPixels (y);
    }
    catch (...)
    {
        threw = true;

        //
```

```
            // in reduceTime mode, fail immediately - the file is corrupt
            //
            if (reduceTime) { return threw; }
        }
    }
```

Another example occurs in the `EnvmapImage::resize` function that in turn calls `Array2D<T>::resizeEraseUnsafe` passing the `dataWindow` X and Y coordinates and perform a huge allocation.

On some system, the allocator will simply return `std::bad_alloc` and crash. On other systems such as macOS, the allocator will happily continue with a "small" pre-allocation and allocate further memory whenever it is accessed. This is the case with the `EnvmapImage::clear` function that is called right after and fills the image RGB values with zeros, allocating tens of Gigabytes.

Impact

An attacker might exploit this vulnerability by feeding a maliciously crafted file to a program that uses the OpenEXR libraries, thus causing a denial of service by stalling the application or exhaust the memory by stalling the application in a loop which contains a memory leakage.

Proof of Concept

1. Compile the `exrcheck` binary in a macOS or GNU/Linux machine with ASAN.
2. Download the `oom_crash.exr` file from the https://github.com/ShielderSec/poc repository.
3. Open the file with the following command:

   ```
   exrcheck oom_crash.exr
   ```

4. Notice that `exrenvmap/exrcheck` crashes with an ASAN stack-trace.

Suggested Remediations

Add a reasonable and safe upper/lower bound limit to the data window size. Third-party developers would be able to disable such limit during compilation or runtime time if the handling of huge images is intended.

It should be noted that `exrcheck` has already some kind of checks to reduce the memory and time usage but they are not effective in this scenario and only cover the `ImfCheckFile` entrypoint.

References

- https://cwe.mitre.org/data/definitions/502.html

5.3. Out of Bounds Heap Read due to Bad Pointer Arithmetic in LossyDctDecoder_execute

Severity	MEDIUM
Affected Resources	src/lib/OpenEXRCore/internal_dwa_decoder.h:650
Status	Closed

Update

This was fixed in https://github.com/AcademySoftwareFoundation/openexr/pull/1977, which was merged and released in version `v3.3.3`.

Description

The OpenEXRCore code is vulnerable to a heap-based buffer overflow during a read operation due to bad pointer math when decompressing DWAA-packed scan-line EXR files with a maliciously forged chunk.

In the `LossyDctDecoder_execute` function (from `src/lib/OpenEXRCore/internal_dwa_decoder.h`, when SSE2 is enabled), the following code is used to copy data from the chunks:

```
// no-op conversion to linear
for (int y = 8 * blocky; y < 8 * blocky + maxY; ++y)
{
    __m128i* restrict dst = (__m128i *) chanData[comp]->_rows[y];
    __m128i const * restrict src = (__m128i const *)&rowBlock[comp][(y & 0x7)
* 8];

    for (int blockx = 0; blockx < numFullBlocksX; ++blockx)
    {
        _mm_storeu_si128 (dst, _mm_loadu_si128 (src)); //

        src += 8 * 8; // <--- si128 pointer incremented as a uint16_t
        dst += 8;
    }
}
```

The issue arises because the `src` pointer, which is a `si128` pointer, is incremented by `8*8`, as if it were a `uint16_t` pointer (64 * uint16_t == 128 bytes). In non-block aligned chunks (width/height not a multiple of 8), this can cause `src` to point past the boundaries of the chunk.

Impact

An attacker might exploit this vulnerability by feeding a maliciously crafted file to a program that uses the OpenEXR libraries, thus crashing the application and in some scenarios also leaking data, for example addresses that could allow them to break the ASLR mitigation.

Proof of Concept

1. Compile the `exrcheck` binary in a macOS or GNU/Linux machine with ASAN which supports SSE2.
2. Download the `dwadecoder_crash.exr` file from the https://github.com/ShielderSec/poc repository.
3. Open the file with the following command:

   ```
   exrcheck dwadecoder_crash.exr
   ```

4. Notice that `exrcheck` crashes with an ASAN stack-trace.

Suggested Remediations

Implement stricter checks to validate the information about the compressed and uncompressed data sizes defined in the chunk.

References

- https://cwe.mitre.org/data/definitions/125.html

5.4. ScanLineProcess::run_fill NULL Pointer Write in "reduceMemory" Mode

Severity	LOW
Affected Resources	src/lib/OpenEXR/ImfDeepScanLineInputFile.cpp:963
Status	Closed

Update

This was fixed in https://github.com/AcademySoftwareFoundation/openexr/pull/1980, which was merged and released in version `v3.3.3`.

Description

When reading a deep scanline image with a large sample count in `reduceMemory` mode, it is possible to crash a target application with a NULL pointer dereference in a write operation. In the `ScanLineProcess::run_fill` function, implemented in `src/lib/OpenEXR/ImfDeepScanLineInputFile.cpp`, the following code is used to write the `fillValue` in the sample buffer:

```
switch (fills.type)
  {
    case OPENEXR_IMF_INTERNAL_NAMESPACE::UINT:
    {
    unsigned int fillVal = (unsigned int) (fills.fillValue);
    unsigned int* fillptr = static_cast<unsigned int*> (dest);

    for ( int32_t s = 0; s < samps; ++s )
      fillptr[s] = fillVal; // <--- POTENTIAL CRASH HERE
    break;
    }
```

However, when `reduceMemory` mode is enabled in the `readDeepScanLine` function in `src/lib/OpenEXRUtil/ImfCheckFile.cpp`, with large sample counts, the sample data will not be read, as shown below:

```
// limit total number of samples read in reduceMemory mode
//
if (!reduceMemory || fileBufferSize + bufferSize < gMaxBytesPerDeepScanline)
// <--- CHECK ON LARGE SAMPLE COUNTS AND reduceMemory
{
// SNIP...
  try
  {
    in.readPixels (y);
  }
```

Therefore, in those cases, the sample buffer would not be allocated, resulting in a potential write operation on a NULL pointer.

Impact

An attacker might exploit this vulnerability by feeding a maliciously crafted file to a program that uses the OpenEXR libraries, thus crashing the application.

Proof of Concept

1. Compile the `exrcheck` binary in a macOS or GNU/Linux machine with ASAN.
2. Download the `runfill_crash.exr` file from the https://github.com/ShielderSec/poc repository.
3. Open the `runfill_crash.exr` file with the following command:

   ```
   exrcheck -m runfill_crash.exr
   ```

4. Notice that `exrcheck` crashes with an ASAN stack-trace.

Suggested Remediations

Check that every image's `fill` is valid and has a valid base address before writing data to it.

References

- https://cwe.mitre.org/data/definitions/476.html