Talos Vulnerability Report

TALOS-2020-1101

Pixar OpenUSD Binary File Format Compressed Value Reps Code Execution Vulnerabilities

NOVEMBER 12, 2020

CVE NUMBER

CVE-2020-6155

Summary

A heap overflow vulnerability exists in the Pixar OpenUSD 20.05 while parsing compressed value rep arrays in binary USD files. A specially crafted malformed file can trigger a heap overflow, which can result in remote code execution. To trigger this vulnerability, the victim needs to access an attacker-provided malformed file.

Tested Versions

Pixar OpenUSD 20.05
Apple macOS Catalina 10.15.3

Product URLs

https://openusd.org

CVSSv3 Score

8.8 - CVSS:3.0/AV:N/AC:L/PR:N/UI:R/S:U/C:H/I:H/A:H

CWE

CWE-122 - Heap-based Buffer Overflow

Details

OpenUSD stands for open Universal Scene Descriptor and is a software suite by Pixar that facilitates, among other things, interchange of arbitrary 3-D scenes that may be composed of many elemental assets.

Most notably, USD and its backing file format `usd` are used on Apple iOS and macOS as part of ModelIO framework in support of SceneKit and ARKit for sharing and displaying 3D scenes in, for example, augmented reality applications. On macOS, these files are automatically rendered to generate thumbnails, while on iOS they can be shared via iMessage and opened with user interaction.

The USD binary file format consists of a header pointing to a table of contents that, in turn, points to individual sections that comprise the whole file. Section `FIELDS` contains two arrays, the second one labeled as `reps`. Compressed in binary, this array holds an array of 64-bit integers which encode references to extra data. Encoding consists of three top bits specifying whether the value is inline, compressed or array. Inlined values are directly represented in the first 48 bits, otherwise those 48 bits represent an offset into the file where the actual value can be found. Besides three most significant bits, five are unused and next eight encode one of 54 different value types (ints, floats, matrices, vectors , special types…). Two distinct paths of triggering this heap overflow are present in the way referenced values are accessed and decoded.

When a value from `reps` array is referenced, if that particular value happens to be an array and is compressed, the following code is invoked for integer types:

```
template <class Reader, class T>
static inline
typename std::enable_if<
    std::is_same<T, int>::value ||
    std::is_same<T, unsigned int>::value ||
    std::is_same<T, int64_t>::value ||
    std::is_same<T, uint64_t>::value>::type
_ReadPossiblyCompressedArray(
    Reader reader, ValueRep rep, VtArray<T> *out, CrateFile::Version ver, int)
{
    // Version 0.5.0 introduced compressed int arrays.
    if (ver < CrateFile::Version(0,5,0) || !rep.IsCompressed()) {
        _ReadUncompressedArray(reader, rep, out, ver);
    }
    else {
        // Read total elements.
            out->resize(ver < CrateFile::Version(0,7,0) ?                    [0]
                    reader.template Read<uint32_t>() :
                    reader.template Read<uint64_t>());
        if (out->size() < MinCompressedArraySize) {
            reader.ReadContiguous(out->data(), out->size());
        } else {
            _ReadCompressedInts(reader, out->data(), out->size());           [1]
        }
    }
}
```

At [0] above, depending on the file version type (greater than 5, but either less or more than 7) either a 32 bit or 64 bit integer size is read from the file. If the size is larger than a minimum compressed size this indicates that the data is indeed compressed so `_readCompressedInts` is invoked at [1]. Following the code:

```
template <class Reader, class Int>
static inline void
_ReadCompressedInts(Reader reader, Int *out, size_t size)
{
    using Compressor = typename std::conditional<
        sizeof(Int) == 4,
        Usd_IntegerCompression,
        Usd_IntegerCompression64>::type;
    std::unique_ptr<char[]> compBuffer(
        new char[Compressor::GetCompressedBufferSize(size)]);
            auto compSize = reader.template Read<uint64_t>();                       [3]
    reader.ReadContiguous(compBuffer.get(), compSize);                              [4]
        Compressor::DecompressFromBuffer(compBuffer.get(), compSize, out, size);
}
```

In the above code, at [3] another 64 bit size value is read from the file and is then directly used as a size of a read operation at [4]. Notice that the destination size buffer is allocated based on previously read argument from [0]. If the latter size value is bigger than the one calculated from the first, this will lead to a heap buffer overflow.

To trigger this vulnerability, first a specific `reps` entry needs to be placed in the `reps` array inside `FIELDS` section. Here is an example of such entry, decoded:

```
580000000000 - offset into file where the value is encoded
03 - type (int)
a0 - flags (1010 0000 binary , is array, not inline, is compressed)
```

The specific referenced value to be placed at offset 0x58 into the file can be:

```
44 00 00 00 00 00 00 00 - read as size value at [0]
41 41 00 00 00 00 00 00 - read as size value at [3]
00 00 00 00 00 00 00 00 - start of buffer
00 00 00 00 00 00 00 00 - ...
```

Another path to this vulnerability lies in the way other types, like floats, are parsed. If the type specified in the encoded `rep` entry is 0x8 for `float`, instead of 0x3 for `int`, the following code is invoked:

```
template <class Reader, class T>
static inline
typename std::enable_if<
    std::is_same<T, GfHalf>::value ||
    std::is_same<T, float>::value ||
    std::is_same<T, double>::value>::type
_ReadPossiblyCompressedArray(
    Reader reader, ValueRep rep, VtArray<T> *out, CrateFile::Version ver, int)
{
    // Version 0.6.0 introduced compressed floating point arrays.
    if (ver < CrateFile::Version(0,6,0) || !rep.IsCompressed()) {
        _ReadUncompressedArray(reader, rep, out, ver);
        return;
    }

    out->resize(ver < CrateFile::Version(0,7,0) ?                                   [4]
                reader.template Read<uint32_t>() :
                reader.template Read<uint64_t>());
    auto odata = out->data();
    auto osize = out->size();

    if (osize < MinCompressedArraySize) {
        // Not stored compressed.
        reader.ReadContiguous(odata, osize);
        return;
    }

    // Read the code
    char code = reader.template Read<int8_t>();
    if (code == 'i') {                                                              [5]
        // Compressed integers.
        vector<int32_t> ints(osize);
        _ReadCompressedInts(reader, ints.data(), ints.size());                      [6]
        std::copy(ints.begin(), ints.end(), odata);
    }
```

Similarly as before, a version check is performed at [4] and size is read. Additionally, a type of value is encoded by a char literal (either `i` or `t`) which is checked at [5]. If this literal is `i`, the same `_ReadCompressedInts` function is invoked which can lead to heap buffer overflow.

In this case, the potential attacker controls both the size of allocated memory buffer, the size of the overflow as well as all the data of the overflow which comes directly from the file being read. With proper memory manipulation and control, this can lead to arbitrary code execution.

Timeline

2020-07-01 - Vendor Disclosure

2020-07-14 - Talos tested and confirmed fix with latest beta of macOS Catalina 10.15.6
2020-11-12 - Public Release

CREDIT

Discovered by Aleksandar Nikolic of Cisco Talos.