Talos Vulnerability Report

TALOS-2020-1094

Pixar OpenUSD binary file format compressed sections code execution vulnerabilities

NOVEMBER 12, 2020

CVE NUMBER

CVE-2020-6147, CVE-2020-6148, CVE-2020-6149, CVE-2020-6150, CVE-2020-6156, CVE-2020-13493

Summary

A heap overflow vulnerability exists in Pixar OpenUSD 20.05 when the software parses compressed sections 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 open 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, the 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 the ModelIO framework in support of SceneKit and ARKit for sharing and displaying 3-D 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.

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. Format specifies that if the format version is 4 or higher, sections content is compressed. Four instances of a heap overflow vulnerability exist in a way USD processes compressed data of specific sections. These are `FIELDS`,`FIELDSETS`, `PATHS` and `SPECS`.

CVE-2020-6147 - USDC file format FIELDS section decompression heap overflow

Following code is responsible for decoding the `FIELDS` section data (from `pxr/usd/usd/crateFile.cpp`):

```
if (auto fieldsSection = _toc.GetSection(_FieldsSectionName)) {
    reader.Seek(fieldsSection->start);                              [1]
    if (Version(_boot) < Version(0,4,0)) {                          [2]
        _fields = reader.template Read<decltype(_fields)>();
    } else {
        // Compressed fields in 0.4.0.
        auto numFields = reader.template Read<uint64_t>();          [3]
        _fields.resize(numFields);

        // Create temporary space for decompressing.
        std::unique_ptr<char[]> compBuffer(
            new char[Usd_IntegerCompression::
                     GetCompressedBufferSize(numFields)]);          [4]
        vector<uint32_t> tmp(numFields);
        auto fieldsSize = reader.template Read<uint64_t>();         [5]
        reader.ReadContiguous(compBuffer.get(), fieldsSize);        [6]
```

In the above code, parser navigates to the beginning of the `FIELDS` section at [1]. If version check at [2] resolves that compressed fields are used, a 64 bit value of `numFields` is read at [3] and is subsequently used to calculate the size of the buffer at [4]. Another 64 bit value , `fieldSize` is read at [5] which is subsequently used to initiate a buffer read at [6]. Data from the file is read into buffer that is sized based on `numFields`, but is bounded by `fieldsSize`. Both values are under direct control and no check is performed to insure fieldsSize bytes can fit into the allocated buffer. This can result in buffer overflow on the heap.

CVE-2020-6148 - USDC file format FIELDSETS section decompression heap overflow

Following code is responsible for decoding the `FIELDSETS` section data (from `pxr/usd/usd/crateFile.cpp`):

```
// Compressed fieldSets in 0.4.0.
auto numFieldSets = reader.template Read<uint64_t>();          [7]
_fieldSets.resize(numFieldSets);

// Create temporary space for decompressing.
std::unique_ptr<char[]> compBuffer(
    new char[Usd_IntegerCompression::
             GetCompressedBufferSize(numFieldSets)]);          [8]
vector<uint32_t> tmp(numFieldSets);
std::unique_ptr<char[]> workingSpace(
    new char[Usd_IntegerCompression::
             GetDecompressionWorkingSpaceSize(numFieldSets)]);

auto fsetsSize = reader.template Read<uint64_t>();             [9]
reader.ReadContiguous(compBuffer.get(), fsetsSize);            [10]
```

Like in the previous vulnerability, parsing the `FIELDSETS` section starts with reading `numFiledSets` at [7] which is used to allocate a properly sized buffer at [8]. Then, `fsetsSize` 64 bit value is read at [9] and used as a read size argument to a file buffer read at [10]. Destination buffer size is calculated based on `numFieldSets`, but file read at [10] reads `fsetsSize` bytes. This constitutes an buffer overflow on the heap.

CVE-2020-6149 - USDC file format PATHS section decompression heap overflow

Following code is responsible for decoding the `PATHS` section data (from `pxr/usd/usd/crateFile.cpp`):

```
_paths.resize(reader.template Read<uint64_t>());                   [11]
std::fill(_paths.begin(), _paths.end(), SdfPath());

WorkArenaDispatcher dispatcher;
// VERSIONING: PathItemHeader changes size from 0.0.1 to 0.1.0.
Version fileVer(_boot);
if (fileVer == Version(0,0,1)) {
    _ReadPathsImpl<_PathItemHeader_0_0_1>(reader, dispatcher);
} else if (fileVer < Version(0,4,0)) {
    _ReadPathsImpl<_PathItemHeader>(reader, dispatcher);
} else {                                                            [12]
    // 0.4.0 has compressed paths.
    _ReadCompressedPaths(reader, dispatcher);
}
```

First, a 64 bit value of number of paths is read at [11] and a familiar file version check is performed. If the file version is `4` or higher, code at [12] proceeds to call `_ReadCompressedPaths` :

```
size_t numPaths = reader.template Read<uint64_t>();                [13]

pathIndexes.resize(numPaths);
elementTokenIndexes.resize(numPaths);
jumps.resize(numPaths);

// Create temporary space for decompressing.
std::unique_ptr<char[]> compBuffer(
    new char[Usd_IntegerCompression::GetCompressedBufferSize(numPaths)]);      [14]
std::unique_ptr<char[]> workingSpace(
    new char[Usd_IntegerCompression::
             GetDecompressionWorkingSpaceSize(numPaths)]);

// pathIndexes.
auto pathIndexesSize = reader.template Read<uint64_t>();               [15]
reader.ReadContiguous(compBuffer.get(), pathIndexesSize);              [16]
```

Again, a 64 bit value of `numPaths` is read into a `size_t` typed variable at [13] and is then used to calculate the size of buffer allocation at [14]. Another 64 bit value of `pathIndexesSize` is read at [15] and is subsequently used as a buffer read size value at [16]. Since the destination buffer size is based on `numPaths` value, but file read size on `pathIndexesSize`, buffer can be made too small to fit the whole read which will result in a buffer overflow on the heap.

CVE-2020-6150 - USDC file format SPECS section decompression heap overflow

Following code is responsible for decoding the `SPECS` section data (from `pxr/usd/usd/crateFile.cpp`):

```
// Version 0.4.0 specs are compressed
auto numSpecs = reader.template Read<uint64_t>();              [17]
_specs.resize(numSpecs);

// Create temporary space for decompressing.
std::unique_ptr<char[]> compBuffer(
    new char[Usd_IntegerCompression::
             GetCompressedBufferSize(numSpecs)]);              [18]
vector<uint32_t> tmp(_specs.size());
std::unique_ptr<char[]> workingSpace(
    new char[Usd_IntegerCompression::
             GetDecompressionWorkingSpaceSize(numSpecs)]);

// pathIndexes.
auto pathIndexesSize = reader.template Read<uint64_t>();       [19]
reader.ReadContiguous(compBuffer.get(), pathIndexesSize);      [20]
```

As with previous vulnerabilities, a 64 bit value is read into `numSpecs` at [17] which is used to calculate the size of the buffer at [18]. Actual size of the section data is read at [19] into a 64 bit value `pathIndexesSize`. Notice the reuse of variable name, indicating code copy/pasting. This size value is used as a file read size at [20] and can result in a heap based buffer overflow if buffer allocated at [18] is too small.

CVE-2020-6156 - USDC file format path element token index decompression heap overflow

Section `PATHS` in the `USDC` binary file contains three arrays, each encoding parts of a path value. Similarly to already reported CVE-2020-6149, a heap buffer overflow vulnerability exists in a way compressed path elements are processed. Following code is invoked :

```
// Read number of encoded paths.
size_t numPaths = reader.template Read<uint64_t>();                              [1]

pathIndexes.resize(numPaths);
elementTokenIndexes.resize(numPaths);
jumps.resize(numPaths);

// Create temporary space for decompressing.
std::unique_ptr<char[]> compBuffer(
    new char[Usd_IntegerCompression::GetCompressedBufferSize(numPaths)]);        [2]
std::unique_ptr<char[]> workingSpace(
    new char[Usd_IntegerCompression::
             GetDecompressionWorkingSpaceSize(numPaths)]);

// pathIndexes.
auto pathIndexesSize = reader.template Read<uint64_t>();
reader.ReadContiguous(compBuffer.get(), pathIndexesSize);
Usd_IntegerCompression::DecompressFromBuffer(
    compBuffer.get(), pathIndexesSize, pathIndexes.data(), numPaths,
    workingSpace.get());

// elementTokenIndexes.
auto elementTokenIndexesSize = reader.template Read<uint64_t>();                  [3]
reader.ReadContiguous(compBuffer.get(), elementTokenIndexesSize);                 [4]
Usd_IntegerCompression::DecompressFromBuffer(
    compBuffer.get(), elementTokenIndexesSize,
    elementTokenIndexes.data(), numPaths, workingSpace.get());
```

First, at [1] , a total number of paths to be reconstructed is read. This value is used to calculate the size of compressed buffer at [2]. Then, at [3] , a compressed size of `element token indices` is read at [3] and a large buffer read at [4]. Notice that while the `compBuffer` is sized based on value read from [1], the value in `elementTokenIndexSize` is used as a read size argument. Both of these value come directly frome the file and, if mismatched, can result in a heap buffer overflow where both allocation and overflow size , as well as overflow contents are under direct control.

CVE-2020-13493 - USDC file format path jumps decompression heap overflow

Similarly to above, a heap buffer overflow vulnerability exists in a way path jumps are processed. Following code is invoked right after the previously discussed:

```
// jumps.
auto jumpsSize = reader.template Read<uint64_t>();                                   [5]
reader.ReadContiguous(compBuffer.get(), jumpsSize);                                [6]
Usd_IntegerCompression::DecompressFromBuffer(
    compBuffer.get(), jumpsSize, jumps.data(), numPaths,
    workingSpace.get());
```

As in the previous case, destination buffer `compBuffer` size is calculated based on calculation at [2], but a second value read at [5] is used as a size argument to a `ReadContiguous` call at [6]. If the allocated buffer is smaller than the size read at [6], this will lead to a heap based 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.

Crash Information

Including crash context for only one instance of the vulnerability. Tested on latest version of macOS Catalina 10.15.3.

```
    thread #2, queue = 'com.apple.quicklook.thumbnailservicecontext.thumbnailgeneration', stop reason = EXC_BAD_ACCESS (code=EXC_I386_GPFLT)
          frame #0: 0x00007fff6f4f0930 libsystem_platform.dylib`_platform_memmove$VARIANT$Haswell + 496
            frame #1: 0x00007fff3ded2644 ModelIO`___lldb_unnamed_symbol96151$$ModelIO + 158
            frame #2: 0x00007fff3ded1d67 ModelIO`___lldb_unnamed_symbol96148$$ModelIO + 341
            frame #3: 0x00007fff3debf7ef ModelIO`___lldb_unnamed_symbol95659$$ModelIO + 867
            frame #4: 0x00007fff3debf273 ModelIO`___lldb_unnamed_symbol95658$$ModelIO + 517
            frame #5: 0x00007fff3debee89 ModelIO`___lldb_unnamed_symbol95657$$ModelIO + 417
            frame #6: 0x00007fff3debe40d ModelIO`___lldb_unnamed_symbol95646$$ModelIO + 599
            frame #7: 0x00007fff3dfc558d ModelIO`___lldb_unnamed_symbol101370$$ModelIO + 261
            frame #8: 0x00007fff3df7177e ModelIO`___lldb_unnamed_symbol100320$$ModelIO + 258
            frame #9: 0x00007fff3d95c195 ModelIO`___lldb_unnamed_symbol34985$$ModelIO + 1221
            frame #10: 0x00007fff3d95b321 ModelIO`___lldb_unnamed_symbol34982$$ModelIO + 2545
            frame #11: 0x00007fff3d95a593 ModelIO`___lldb_unnamed_symbol34979$$ModelIO + 867
            frame #12: 0x00007fff3df85979 ModelIO`___lldb_unnamed_symbol100531$$ModelIO + 209
            frame #13: 0x00007fff3df85ae0 ModelIO`___lldb_unnamed_symbol100532$$ModelIO + 200
            frame #14: 0x00007fff3d61231a ModelIO`___lldb_unnamed_symbol2606$$ModelIO + 386
            frame #15: 0x00007fff3d592ed6 ModelIO`___lldb_unnamed_symbol670$$ModelIO + 794
            frame #16: 0x00007fff3d592b17 ModelIO`___lldb_unnamed_symbol669$$ModelIO + 97
            frame #17: 0x00007fff43f488d4 SceneKit`loadMDLAssetWithURL + 113
            frame #18: 0x00007fff43dbe878 SceneKit`-[SCNSceneSource _createSceneRefWithOptions:statusHandler:] + 788
            frame #19: 0x00007fff43dbe4a5 SceneKit`-[SCNSceneSource _sceneWithClass:options:statusHandler:] + 1525
            frame #20: 0x00007fff43dbdcd6 SceneKit`-[SCNSceneSource sceneWithClass:options:statusHandler:] + 117
            frame #21: 0x00007fff43dbdc45 SceneKit`-[SCNSceneSource sceneWithClass:options:error:] + 88
            frame #22: 0x0000000102ab25c9 SceneKitQLThumbnailExtension`___lldb_unnamed_symbol1$$SceneKitQLThumbnailExtension + 414
            frame #23: 0x00007fff438e8fb2 QuickLookThumbnailing`__145-[QLThumbnailServiceContext
generateThumbnailOfSize:minimumSize:scale:badgeType:withFileURLHandler:additionalResourcesWrapper:completionHandler:]_block_invoke + 432
            frame #24: 0x00007fff6f2a0583 libdispatch.dylib`_dispatch_call_block_and_release + 12

        Nearby code:
        libsystem_platform.dylib`_platform_memmove$VARIANT$Haswell:
        ->  0x7fff6f4f0930 <+496>: c5 fc 10 46 e0     vmovups ymm0, ymmword ptr [rsi - 0x20]
            0x7fff6f4f0935 <+501>: 49 89 fb           mov    r11, rdi
            0x7fff6f4f0938 <+504>: 48 83 ef 01        sub    rdi, 0x1
            0x7fff6f4f093c <+508>: 48 83 e7 e0        and    rdi, -0x20
            0x7fff6f4f0940 <+512>: 4c 89 d9           mov    rcx, r11
            0x7fff6f4f0943 <+515>: 48 29 f9           sub    rcx, rdi
            0x7fff6f4f0946 <+518>: 48 29 ce           sub    rsi, rcx
            0x7fff6f4f0949 <+521>: 48 29 ca           sub    rdx, rcx
            0x7fff6f4f094c <+524>: c5 fc 10 4e e0     vmovups ymm1, ymmword ptr [rsi - 0x20]
            0x7fff6f4f0951 <+529>: c4 c1 7c 11 43 e0  vmovups ymmword ptr [r11 - 0x20], ymm0

            rax = 0x00007ffa0b55eb10
            rbx = 0x0000000000000000
            rcx = 0x4141414141414141
            rdx = 0x4141414141414141
            rsi = 0x414141424400d36f
            rdi = 0x4141c13b4c972c51
             r8 = 0x4141414141414141
             r9 = 0x0000000000000000
            r10 = 0x00000000fffbffff
            r11 = 0x00007ff9089658e2
            r12 = 0x0000700008c499a8
            r13 = 0x0000700008c499a8
            r14 = 0x4141414141414141
            r15 = 0x00007ffa0b55eb10
            rsp = 0x0000700008c498e0
            rbp = 0x0000700008c498e0
            rip = 0x00007fff6f4f0930 libsystem_platform.dylib`_platform_memmove$VARIANT$Haswell + 496
            fla = o d I t s z a p c
```

Timeline

2020-07-01 - Vendor Disclosure to Pixar & Apple

2020-11-12 - Public Release

CREDIT

Discovered by Aleksandar Nikolic of Cisco Talos.
