Talos Vulnerability Report

TALOS-2021-1414

Apple macOS ImageIO DDS image out-of-bounds read vulnerability

JANUARY 25, 2022

CVE NUMBER

CVE-2021-30939

Summary

An out-of-bounds read vulnerability exists in the DDS image parsing functionality of ImageIO library on Apple macOS Big Sur 11.6.1 and iOS 15.1. A specially-crafted DDS file can disclose sensitive memory content which can aid in exploitation of other vulnerabilities. An attacker can deliver a malicious file to trigger this vulnerability.

Tested Versions

Apple iOS 15.1
Apple macOS Big Sur 11.6.1

Product URLs

http://apple.com

CVSSv3 Score

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

CWE

CWE-125 - Out-of-bounds Read

Details

macOS and iOS are a series of operating systems developed by Apple that exclusively run on Apple hardware. In many aspects, the desktop and mobile OS have a shared codebase. ImageIO is a core component used to render and display different image file formats on both operating systems. Image parsers, and corresponding attack surface, implemented in ImageIO can be reached through default applications and preview handlers without user interaction.

There exists a vulnerability in the apple texture encoder library (part of ImageIO) when handling specially malformed DirectDraw Surface texture image files.

A DDS file is composed of: - Magic value (0x04 bytes) - `DDS ` - DDS header (0x7C bytes) - starts with `7C 00 00 00` - Data

Among other data, DDS header contains image width and height. If DDS image passes the prior checks, `BCReadPlugin` object is created inside ImageIO, and its class method `initialize(...)` is called. `BCReadPlugin` is the main class that handles parsing of the DDS file format. ImageIO reads rest of the DDS header (`0x7C` bytes) to local stack buffer which `varDDSHeader` at [1] in the following disassembly. There is another class member inside the `BCReadPlugin` object that stores the current parsing offset into the file which is initially set to 4 (actual starting point of the header).

```
__text:00007FFF28C1F2C7                mov     [r12+BC_READ_PLUGIN.curr_image_offset], 4
__text:00007FFF28C1F2D3                lea     rdi, [rbp+varIIOScanner] ; this
__text:00007FFF28C1F2DA                lea     rsi, [rbp+varDDSHeader] ; void *                                [1]
__text:00007FFF28C1F2E1                mov     edx, 4          ; unsigned __int64
__text:00007FFF28C1F2E6                mov     ecx, 7Ch ; '|'  ; unsigned __int64
__text:00007FFF28C1F2EB                call    IIOScanner::getBytesAtOffset
__text:00007FFF28C1F2F0                cmp     rax, 7Ch                                                        [2]
__text:00007FFF28C1F2F4                jnz     exit_func
```

Additionally, at [2] above , we can see that `IIOScanner::getBytesAtOffset(...)` must successfully read 0x7C bytes starting from offset 0x4 of the image file, which means the file size must be at least 0x80 bytes.

```
__text:00007FFF28C1F2FA                mov     eax, dword ptr [rbp+varDDSHeader+4]
__text:00007FFF28C1F300                test    al, 1           ; flags
__text:00007FFF28C1F302                jz      DDSD_CAPS_ERROR
__text:00007FFF28C1F308                test    al, 2
__text:00007FFF28C1F30A                jz      DDSD_HEIGHT_ERROR
__text:00007FFF28C1F310                test    al, 4
__text:00007FFF28C1F312                jz      DDSD_WIDTH_ERROR
__text:00007FFF28C1F318                bt      eax, 0Ch
__text:00007FFF28C1F31C                jnb     DDSD_PIXELFORMAT_ERROR
```

Then ImageIO start to check the `flags` field in the header, which is a 4 byte field at the offset `0x8` of the file (or `+0x4` from start of the header), to see if some bits are set. Following checks on the bits must pass in order to continue parsing of the image. Having minimum required bits set with hex bytes `D7 D5 00 00` or `FF FF FF FF` at the `flags` location will succeed.

```
__text:00007FFF28C1F322                add     [r12+BC_READ_PLUGIN.curr_image_offset], 7Ch ; '|'
__text:00007FFF28C1F32B                mov     byte ptr [r12+13Fh], 0
__text:00007FFF28C1F334                mov     r8d, dword ptr [rbp+varDDSHeader+8]
__text:00007FFF28C1F33B                mov     r15d, dword ptr [rbp+varDDSHeader+0Ch]
__text:00007FFF28C1F342                cmp     r13, r15        ; compare image size, image width
__text:00007FFF28C1F345                jbe     BAD_DIMENSTION_ERROR_WIDTH
__text:00007FFF28C1F34B                cmp     r13, r8         ; compare image size, image height
__text:00007FFF28C1F34E                jbe     BAD_DIMENSTION_ERROR_HEIGHT
__text:00007FFF28C1F354                mov     ecx, dword ptr [rbp+varDDSHeader+10h]
__text:00007FFF28C1F35A                cmp     r13, rcx        ; compare image size, pitchOrLinearSize
__text:00007FFF28C1F35D                jbe     PITCHORLINEARSIZE_ERROR
__text:00007FFF28C1F363                lea     ebx, [r15+3]    ; image width + 3
__text:00007FFF28C1F367                shr     ebx, 2
__text:00007FFF28C1F36A                mov     [rbp+varImageHeight], r8
__text:00007FFF28C1F371                lea     ecx, [r8+3]
__text:00007FFF28C1F375                shr     ecx, 2
__text:00007FFF28C1F378                bt      eax, 11h        ; flag
__text:00007FFF28C1F37C                mov     [r12+1A0h], ebx ; width rounded up to the multiple of 4 >> 2              [3]
__text:00007FFF28C1F384                mov     [r12+1A4h], ecx ; height rounded up to the multiple of 4 >> 2             [4]
__text:00007FFF28C1F38C                jnb     short loc_7FFF28C1F3E1
```

Next, it continues to compare the image file size with fields `width`, `height`, `pitchOrLinearSize`, which are at file offset `0x0C`, `0x10`, `0x14` respectively. They each must be smaller than the file size. At [3] and [4] above, `width` and `height` is rounded up to the nearest multiple of 4 and quotient when divided by 4 is saved inside instance of `BCReadPlugin`. `width` and `height` each should not be 0 in order to trigger the bug, but `pitchOrLinearSize` can be 0.

Also if bit at the offset 0x11 of the `flags`, which is `DDSD_MIPMAPCOUNT`, is set then check is done on `mipMapCount` but it only logs the information and continues if check doesn't pass. `mipMapCount` field inside the header is at file offset `0x1C`, and must not be 0 as well. If it is larger than 1, there is additional check that happens later in the function but for demonstration purposes setting the field to `1` will skip this check.

Then, four-character codes at file offset `0x54` for specifying compressed or custom format is checked to invoke the appropriate decoder. In the attached PoC image, `DXT1` is set. With `DXT3` and `DXT5`, it is also possible to trigger the bug with no or minimum changes to the original POC image.

At this point, current file offset is increased by `0x7C`, resulting `0x80`. File offset is now pointing to data section.

```
__text:00007FFF28C1F42C                lea     eax, ds:0[rbx*8] ; quotient * 8
__text:00007FFF28C1F433                test    ebx, ebx
__text:00007FFF28C1F435                mov     ecx, 8
__text:00007FFF28C1F43A                cmovnz  ecx, eax
__text:00007FFF28C1F43D                imul    rcx, rdx        ; rdx height            [5]
__text:00007FFF28C1F441                mov     rax, rcx
__text:00007FFF28C1F444                shr     rax, 2
__text:00007FFF28C1F448                mov     edi, 83F1h
__text:00007FFF28C1F44D                cmp     rax, r13        ; r13 image file size    [6]
__text:00007FFF28C1F450                jb      loc_7FFF28C1F7B1
```

This specific check on `width` and `height` fields in the header is done if the invoked decoder is `DXT1`. Previously saved quotient is multiplied by 8 and multiplied by image height again at [5]. Another constraint is that this result divided by 4 must be less than the actual file size [6]. This is roughly equal to `width * height / 2`.

```
__text:00007FFF28C1F7C1 loc_7FFF28C1F7C1:                      ; CODE XREF: BCReadPlugin::initialize(IIODictionary *)+993↓j
__text:00007FFF28C1F7C1                mov     eax, [r12+1A0h] ; (image width + 3 ) / 4
__text:00007FFF28C1F7C9                mov     ebx, [r12+1A4h] ; (image height + 3 ) / 4
__text:00007FFF28C1F7D1                imul    rbx, rax
__text:00007FFF28C1F7D5                call    __ZN12BCReadPlugin13bytesPerBlockEj ; BCReadPlugin::bytesPerBlock(uint)
__text:00007FFF28C1F7DA                mov     ecx, eax
__text:00007FFF28C1F7DC                mov     rax, rbx
__text:00007FFF28C1F7DF                mul     rcx
__text:00007FFF28C1F7E2                jo      exit_func
__text:00007FFF28C1F7E8                mov     rbx, rax
__text:00007FFF28C1F7EB                mov     rdi, qword ptr [r12+BC_READ_PLUGIN.session] ; this
__text:00007FFF28C1F7F0                call    __ZN19IIOImageReadSession7getSizeEv ; IIOImageReadSession::getSize(void)
__text:00007FFF28C1F7F5                cmp     rbx, rax        ; compare computed value with the image file size
```

Yet another, final, check against `width` and `height` is performed above. `((width + 3) >> 2) * ((height + 3) >> 2) * blocksize` should be less than the file size. Block size is `0x8` bytes in this case.

```
__text:00007FFF28C1F84D loc_7FFF28C1F84D:                      ; CODE XREF: BCReadPlugin::initialize(IIODictionary *)+657↓j
__text:00007FFF28C1F84D                mov     esi, r15d
__text:00007FFF28C1F850                shr     esi, cl
__text:00007FFF28C1F852                add     esi, 3
__text:00007FFF28C1F855                mov     edi, esi
__text:00007FFF28C1F857                shr     edi, 2
__text:00007FFF28C1F85A                cmp     esi, 7
__text:00007FFF28C1F85D                cmovbe  edi, r8d
__text:00007FFF28C1F861                imul    edi, edi
__text:00007FFF28C1F864                imul    edi, eax
__text:00007FFF28C1F867                mov     [rdx+rcx*8+30h], rbx    ; rbx: current file offset
__text:00007FFF28C1F86C                mov     [rdx+rcx*8+130h], rdi
__text:00007FFF28C1F874                add     rbx, rdi
__text:00007FFF28C1F877                inc     rcx
__text:00007FFF28C1F87A                cmp     r9, rcx
__text:00007FFF28C1F87D                jnz     short loc_7FFF28C1F84D
```

Then this loop initializes two array class members of `BCTextureImpl` object. Here `rdx` register stores pointer to `BCTextureImpl` class instance. Offset `0x30` and `0x130` each is start address of the arrays. `rcx` being the index, at first iteration current file offset (`rbx`: `0x80`) is stored at index 0 of the array (offset `+0x30`). This array is referenced later again when calculating memory address for the decoding routine to use.

After `BCReadPlugin::initialize(...)` returns, later `BCReadPlugin::copyImageBlockSet(...)` is called which in turn calls `BCReadPlugin::decodeDXTCtoRGBX(...)`

```
__text:00007FFF28C20149 loc_7FFF28C20149:                       ; CODE XREF: BCReadPlugin::decodeDXTCtoRGBX(IIOImageReadSession
*,vImage_Buffer *,CGImageAlphaInfo,bool)+A0↑j
__text:00007FFF28C20149                                         ; BCReadPlugin::decodeDXTCtoRGBX(IIOImageReadSession *,vImage_Buffer
*,CGImageAlphaInfo,bool)+AC↑j ...
__text:00007FFF28C20149                 mov     [rbp+texelType], rdx ; jumptable 00007FFF28C2007E case 36283
__text:00007FFF28C2014D                 mov     [rbp+blockType], rbx
__text:00007FFF28C20151                 mov     r15, [r13+1B8h]
__text:00007FFF28C20158                 lea     rsi, [rbp+var_48] ; void **
__text:00007FFF28C2015C                 mov     qword ptr [rsi], 0
__text:00007FFF28C20163                 mov     rdi, r14        ; this
__text:00007FFF28C20166                 xor     edx, edx        ; bool
__text:00007FFF28C20168                 call    IIOImageReadSession::retainBytePointer(void **,bool)             [7]
__text:00007FFF28C2016D                 mov     r12, rax        ; r12 points to actual image data
__text:00007FFF28C20170                 mov     eax, [r13+0BCh]
__text:00007FFF28C20177                 cmp     rax, 1Fh
__text:00007FFF28C2017B                 mov     [rbp+var_30], r14
__text:00007FFF28C2017F                 ja      short loc_7FFF28C20198
```

At [7], return value of `IIOImageReadSession::retainBytePointer(...)` is a pointer to the actual file contents. Then from the array that was initialized in `BCReadPlugin::initialize(...)`, element is copied to register `rcx` in the following disassembly. When parsing the POC image, array index is 0 (`rax`). Remember that at index 0 of the array (offset `0x30`) was value `0x80`.

```
__text:00007FFF28C20186                 mov     rdx, r14
__text:00007FFF28C20189                 mov     rcx, [r15+rax*8+30h]   ; Array that was initialized in BCReadPlugin::initialize(...)
__text:00007FFF28C2018E                 mov     r14, [r15+rax*8+130h]
__text:00007FFF28C20196                 jmp     short loc_7FFF28C201A5
...
__text:00007FFF28C201A5 loc_7FFF28C201A5:                       ; CODE XREF: BCReadPlugin::decodeDXTCtoRGBX(IIOImageReadSession
*,vImage_Buffer *,CGImageAlphaInfo,bool)+1B8↑j
__text:00007FFF28C201A5                 add     r12, rcx        ; start of the image + 0x80                      [8]
__text:00007FFF28C201A8                 test    r12b, 0Fh
__text:00007FFF28C201AC                 jz      loc_7FFF28C20258
```

At [8] above, `r12` register points to file `start of the file contents + 0x80`. If the file size is `0x80`, which is a valid size without data section, this pointer will now point to out of bounds memory beyoud the end of the data read from the file.

```
__text:00007FFF28C202E4                 lea     rax, [rbp+src]
__text:00007FFF28C202EB                 mov     [rax+at_block_buffer_t.blocks], r12
```

In the above code , local variable `src` is a structure of type `at_block_buffer_t`, and pointer to `start of the file contents + 0x80` is saved in the member `blocks`. The complete structure is as follows:

```
struct at_block_buffer_t
{
  void *blocks;
  size_t rowBytes;
  size_t sliceBytes;
};
```

Actual data decoding is invoked by the following code:

```
__text:00007FFF28C203B5                 lea     rsi, [rbp+src]  ; src
__text:00007FFF28C203BC                 lea     rdx, [rbp+imageSize] ; dest
__text:00007FFF28C203C0                 mov     r14, qword ptr [rbp+texelAlphaType]
__text:00007FFF28C203C4                 mov     rdi, r14        ; encoder
__text:00007FFF28C203C7                 xor     ecx, ecx        ; flags
__text:00007FFF28C203C9                 call    _at_encoder_decompress_texels
__text:00007FFF28C203CE                 test    rax, rax
__text:00007FFF28C203D1                 jz      short loc_7FFF28C203FB
```

Pointer to `src` is passed as an argument to `_at_encoder_decompress_texels(...)` which is a stub to `DXTCEncoder::DecompressTexels` in this case. From this point on, any operation done on the `blocks` pointer will cause out-of-bound access. With GuardMalloc turned on in MacOS Big Sur, parsing the PoC image will cause access violation with the stack trace like following.

```
* thread #1, queue = 'com.apple.main-thread', stop reason = EXC_BAD_ACCESS (code=1, address=0x100b22000)
  * frame #0: 0x00007fff2ae9061b libate.dylib`decode_bc1 + 27
    frame #1: 0x00007fff2aecd9ea libate.dylib`DecodeRow(void*, unsigned long) + 191
    frame #2: 0x00007fff2aecd8b8 libate.dylib`DXTCEncoder::DecompressTexels(at_block_buffer_t const&, at_texel_region_t const&, at_flags_t)
const + 608
    frame #3: 0x00007fff2ae978da libate.dylib`at_encoder_decompress_texels + 855
    frame #4: 0x00007fff28abda19 ImageIO`BCReadPlugin::decodeDXTCtoRGBX(IIOImageReadSession*, vImage_Buffer*, CGImageAlphaInfo, bool) + 1015
    frame #5: 0x00007fff28abdc8d ImageIO`BCReadPlugin::copyImageBlockSet(InfoRec*, CGImageProvider*, CGRect, CGSize, __CFDictionary const*)
+ 453
    frame #6: 0x00007fff28a78a68 ImageIO`IIO_Reader::CopyImageBlockSetProc(void*, CGImageProvider*, CGRect, CGSize, __CFDictionary const*) +
100
    frame #7: 0x00007fff28a965f7 ImageIO`IIOImageProviderInfo::copyImageBlockSetWithOptions(CGImageProvider*, CGRect, CGSize, __CFDictionary
const*) + 663
    frame #8: 0x00007fff28a789a0 ImageIO`IIOImageProviderInfo::CopyImageBlockSetWithOptions(void*, CGImageProvider*, CGRect, CGSize,
__CFDictionary const*) + 680
    frame #9: 0x00007fff250532d0 CoreGraphics`imageProvider_retain_data + 77
    frame #10: 0x00007fff25053246 CoreGraphics`CGDataProviderRetainData + 75
    frame #11: 0x00007fff250530f6 CoreGraphics`CGAccessSessionCreate + 98
    frame #12: 0x00007fff25022923 CoreGraphics`CGDataProviderCopyData + 187
    frame #13: 0x00007fff2504fa7c CoreGraphics`CGImageGetDataProviderInternal + 89
    frame #14: 0x00007fff2504ef13 CoreGraphics`CGDataProviderCreateForDestinationWithImage + 92
    frame #15: 0x00007fff2504dbac CoreGraphics`img_image + 1173
```

Without GuardMalloc, out of bounds memory will be accessed in blocks via one or multiple calls to `DecodeRow` which in turn invokes the appropriate decoder selected previously. Unless the out of bounds access hits a invalid memory, arbitrary data will be decoded and presented as pixels. Presented analysis was performed on macOS Big Sur, but similar behaviour was observed on iOS to confirm presence of the same vulnerability.

The root cause of this bug is that decoding starts at file offset `+0x80` so any `width`, `height` field check should be against `file size - 0x80` which is the size of the actual image data, but ImageIO checks against the entire image file size. By varying the expected image size by modifying `width` and `height` fields, function `DXTCEncoder::DecompressTexels` can be made to read multiple blocks which ends up reading adjacent heap data. Contents of the heap is then directly rendered as pixels thereby potentially leaking heap addresses and other information that could aid futher exploitation if leaked data can be accessed in the context of a vulnerable application.

Timeline

2021-11-15 - Vendor Disclosure
2021-12-13 - Vendor Patched
2022-01-25 - Public Release

CREDIT

Discovered by Jaewon Min of Cisco Talos.
