Talos Vulnerability Report

TALOS-2021-1367

Accusoft ImageGear Palette box parser heap-based buffer overflow vulnerability

FEBRUARY 23, 2022

CVE NUMBER

CVE-2021-21938

Summary

A heap-based buffer overflow vulnerability exists in the Palette box parser functionality of Accusoft ImageGear 19.10. A specially-crafted file can lead to code execution. An attacker can provide a malicious file to trigger this vulnerability.

Tested Versions

Accusoft ImageGear 19.10

Product URLs

ImageGear - https://www.accusoft.com/products/imagegear-collection/

CVSSv3 Score

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

CWE

CWE-193 - Off-by-one Error

Details

The ImageGear library is a document-imaging developer toolkit that offers image conversion, creation, editing, annotation and more. It supports more than 100 formats such as DICOM, PDF, Microsoft Office and others.

A specially-crafted JPEG 2000 file can lead to a heap-based buffer overflow in the Palette box parser, due to a wrongly-sized heap buffer caused by an off-by-one error.

Trying to load a malformed JPEG 2000 file, we end up in the following situation:

```
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=0adacffc ebx=00000003 ecx=0000000f edx=00000000 esi=0adacfc0 edi=0ace9000
eip=6ebce13d esp=0019fac0 ebp=0019fae0 iopl=0         nv up ei pl nz na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00010206
MSVCR110!memcpy+0x21e:
6ebce13d f3a5            rep movs dword ptr es:[edi],dword ptr [esi]
```

Where the destination buffer has the following information:

```
0:000> !ext.heap -p -a edi
        address 0af11000 found in
        _DPH_HEAP_ROOT @ 2cb1000
        in busy allocation (  DPH_HEAP_BLOCK:         UserAddr         UserSize -         VirtAddr         VirtSize)
                                  d6b0d34:          af10fc0               40 -          af10000             2000
            ? Fuzzme!fuzzme+11f00
        6f08ab40 verifier!AVrfDebugPageHeapAllocate+0x00000240
        7793a65b ntdll!RtlDebugAllocateHeap+0x00000039
        778dff98 ntdll!RtlpAllocateHeap+0x00051808
        7788e5e0 ntdll!RtlpAllocateHeapInternal+0x00001280
        7788d34e ntdll!RtlAllocateHeap+0x0000003e
        6ebcdaff MSVCR110!malloc+0x00000049
        6ebcdba7 MSVCR110!operator new+0x0000001d
        6eda130b igCore19d!IG_mpi_page_set+0x000052db
        6ed656f1 igCore19d!IG_comm_is_comp_exist+0x000036a1
        6ed48aca igCore19d!IG_warning_set+0x000018da
        6e30e16e igJPEG2K19d!CPb_JPEG2K_init+0x000094ae
        6e30fe07 igJPEG2K19d!CPb_JPEG2K_init+0x0000b147
        6e31106c igJPEG2K19d!CPb_JPEG2K_init+0x0000c3ac
        6e3070a6 igJPEG2K19d!CPb_JPEG2K_init+0x000023e6
        6e30711e igJPEG2K19d!CPb_JPEG2K_init+0x0000245e
        6ed713d9 igCore19d!IG_image_savelist_get+0x00000b29
        6edb08d7 igCore19d!IG_mpi_page_set+0x000148a7
        6edb0239 igCore19d!IG_mpi_page_set+0x00014209
        6ed45757 igCore19d!IG_load_file+0x00000047
        00402219 Fuzzme!fuzzme+0x00000019
        00402524 Fuzzme!fuzzme+0x00000324
        0040668d Fuzzme!fuzzme+0x0000448d
        75330419 KERNEL32!BaseThreadInitThunk+0x00000019
        778b72ed ntdll!__RtlUserThreadStart+0x0000002f
        778b72bd ntdll!_RtlUserThreadStart+0x0000001b Instead, the source buffer:

0:000> !ext.heap -p -a esi
        address 0ae4efc0 found in
        _DPH_HEAP_ROOT @ 2cb1000
        in busy allocation (  DPH_HEAP_BLOCK:         UserAddr         UserSize -         VirtAddr         VirtSize)
                                  adb1e6c:          ae4ef80               7c -          ae4e000             2000
            ? Fuzzme!fuzzme+11f00
        6f08ab40 verifier!AVrfDebugPageHeapAllocate+0x00000240
        7793a65b ntdll!RtlDebugAllocateHeap+0x00000039
        778dff98 ntdll!RtlpAllocateHeap+0x00051808
        7788e5e0 ntdll!RtlpAllocateHeapInternal+0x00001280
        7788d34e ntdll!RtlAllocateHeap+0x0000003e
        6ebcdaff MSVCR110!malloc+0x00000049
        6ed964de igCore19d!AF_memm_alloc+0x0000001e
        6e30e7ed igJPEG2K19d!CPb_JPEG2K_init+0x00009b2d
        6e30e5eb igJPEG2K19d!CPb_JPEG2K_init+0x0000992b
        6e310e34 igJPEG2K19d!CPb_JPEG2K_init+0x0000c174
        6e3047be igJPEG2K19d+0x000747be
        6e37e2ad igJPEG2K19d!CPb_JPEG2K_init+0x000795ed
```

The information above clearly shows that the destination buffer is smaller than the source one. The access violation takes place during the parsing of the Palette box, in the following function:

```
int FUN_73ebe120(HIGDIBINFO *LPHIGDIBINFO,int enumIGColorSpaceIDs,int width,int heigth,
                int channelCount,AT_INT *channelDepths,IGDIBStd *IGDIBStd,int sizePalette,
                AT_RESOLUTION *lpResolution,undefined4 lphdib)

{
int iVar1;
size_t _Size;
void *_Dst;

if ((((enumIGColorSpaceIDs != 0) && (0 < width)) && (0 < heigth)) && (0 < channelCount)) {
    iVar1 = DIB_info_create(LPHIGDIBINFO,width,heigth,enumIGColorSpaceIDs,channelCount,channelDepths
                        );
    if (iVar1 == 0) {
    DIB_resolution_set(*LPHIGDIBINFO,lpResolution);
    if ((char)enumIGColorSpaceIDs == '\x03') {
        iVar1 = DIB_palette_alloc(*LPHIGDIBINFO);                                                   [1]
        if (iVar1 != 0) {
        return iVar1;
        }
        _Size = sizePalette << 2;
        _Dst = (void *)DIB_palette_pointer_get(*LPHIGDIBINFO);
        memcpy(_Dst,IGDIBStd,_Size);                                                                [3]
    }
    iVar1 = (*(code *)PTR_73f8874c)(*LPHIGDIBINFO,lphdib);
    }
    return iVar1;
}
return 0;
}
```

The memory violation takes place in `[3]`, and the allocation of the wrongly-sized buffer takes place in `[1]` in the function `DIB_palette_alloc`:

```
undefined4 call_IGDIB::DIB_palette_alloc(HIGDIBINFO hDIB)

{
[...]
size_buffer_palette = compute_size_palette(*hDIB->bits_depth_table_by_channel);                     [4]
(*hDIB->igdibstd_vftable->IGDIB::createIGPalette)((IGDIB *)hDIB,size_buffer_palette);
*in_FS_OFFSET = local_10;
return 0;
}
```

The size for the buffer is `size_buffer_palette`, which is calculated in `[4]`. The argument of the function called in `[4]` corresponds to the number of bits required for representing the Palette box's `NE` field. The `compute_size_palette` function is simply:

```
int __cdecl compute_size_palette(int bit_required)

{
if (bit_required < 9) {
    return 1 << ((byte)bit_required & 0x1f);
}
return 0;
}
```

The `compute_size_palette` function returns, if the argument does not exceed the value 8, the biggest representable value, using `bit_required` bits, plus one (e.g., for `bit_required = 1` the result would be 2; for `bit_required = 7` the result would be 128).

This number is later used in `[5]` to allocate the required space to parse the Palette box:

```
undefined ** __thiscall IGPalette::IGPalette(undefined **param_1_00,undefined *size_palette)

{
undefined *_Dst;

param_1_00[1] = (undefined *)0x0;
param_1_00[2] = size_palette;
if ((int)size_palette < 1) {
    param_1_00[1] = (undefined *)0x0;
}
else {
    _Dst = (undefined *)operator_new((int)size_palette << 2);                                    [5]
    param_1_00[1] = _Dst;
    if (_Dst != (undefined *)0x0) {
    memset(_Dst,0,(int)param_1_00[2] << 2);
    *param_1_00 = (undefined *)vftable;
    return param_1_00;
    }
}
param_1_00[2] = (undefined *)0x0;
*param_1_00 = (undefined *)vftable;
return param_1_00;
}
```

So, the Palette box parser uses the buffer allocated in `[5]` in the `memcpy` at `[3]`. The size of the allocated buffer can be simplified as `(1 << bit_required) << 2`. The problem resides in the calculation of the `bit_required` value. An off-by-one calculation exists during the computation of that value. The function that calculates the `bit_required` value is show here:

```
void __cdecl FUN_73ebfbc0(undefined4 *param_1,size_t *LPHIGDIBINFO)

{
[...]
    iVar3 = 0
    if (0 < iVar2) {
        do {
        if (*(int *)(param_1[0x18] + 4) == 0) {
            NE_field = param_1[0x13];
            bit_required = 0;
            while (NE_field = NE_field >> 1, NE_field != 0) {                                    [6]
            bit_required = bit_required + 1;
            }
            if ((*(char *)(param_1 + 0x16) != '\x03') || (iVar3 != 0)) {
            bit_required = *(int *)(param_1[5] + 4 + iVar3 * 0x20);
            }
        }
        else {
            bit_required = 8;
        }
        channelDepths[iVar3] = bit_required;
        iVar3 = iVar3 + 1;
        } while (iVar3 < iVar2);
    }
[...]
}
```

This is the while loop in assembly:

```
     MOV        ECX,dword ptr [EBX + NE_field]
     XOR        bit_required,bit_required
     SAR        ECX,1                                                                            [7]
     JZ         LAB_B
LAB_A
     INC        bit_required
     SAR        ECX,1
     JNZ        LAB_A
LAB_B
```

In `[6]`/`[7]` there is the actual calculation for computing the bits required for representing a value. The problem is that the computation starts with the right shift of one rather than the comparison with zero. This would count one bit less than the required one (e.g., with `NE_field = 1(0b...001)` the `bit_required` value would be 0, with `NE_field = 0x1f(0b...00011111)` the `bit_required` value would be 4). This leads to an off-by-one that can result in a heap-based buffer overflow in `[3]` because the calculated size, using the wrongly-calculated bits required, could be smaller than the real required size.

Crash Information

```
0:000> !analyze -v
*******************************************************************************
*                                                                             *
*                        Exception Analysis                                   *
*                                                                             *
*******************************************************************************


KEY_VALUES_STRING: 1

    Key  : AV.Fault
    Value: Write

    Key  : Analysis.CPU.mSec
    Value: 3218

    Key  : Analysis.DebugAnalysisManager
    Value: Create

    Key  : Analysis.Elapsed.mSec
    Value: 10243

    Key  : Analysis.Init.CPU.mSec
    Value: 468

    Key  : Analysis.Init.Elapsed.mSec
    Value: 11601

    Key  : Analysis.Memory.CommitPeak.Mb
    Value: 143

    Key  : Timeline.OS.Boot.DeltaSec
    Value: 166871

    Key  : Timeline.Process.Start.DeltaSec
    Value: 11

    Key  : WER.OS.Branch
    Value: rs5_release

    Key  : WER.OS.Timestamp
    Value: 2018-09-14T14:34:00Z

    Key  : WER.OS.Version
    Value: 10.0.17763.1

    Key  : WER.Process.Version
    Value: 1.0.1.1


NTGLOBALFLAG:  2000000

APPLICATION_VERIFIER_FLAGS:  0

APPLICATION_VERIFIER_LOADED: 1

EXCEPTION_RECORD:  (.exr -1)
ExceptionAddress: 6ebce13d (MSVCR110!memcpy+0x0000021e)
ExceptionCode: c0000005 (Access violation)
ExceptionFlags: 00000000
NumberParameters: 2
Parameter[0]: 00000001
Parameter[1]: 0af11000
Attempt to write to address 0af11000

FAULTING_THREAD:  00002848

PROCESS_NAME:  Fuzzme.exe

WRITE_ADDRESS:  0af11000

ERROR_CODE: (NTSTATUS) 0xc0000005 - The instruction at 0x%p referenced memory at 0x%p. The memory could not be %s.

EXCEPTION_CODE_STR:  c0000005

EXCEPTION_PARAMETER1:  00000001

EXCEPTION_PARAMETER2:  0af11000

STACK_TEXT:
0019fac4 6e30e18a     0af10fc0 0ae4ef80 0000007c MSVCR110!memcpy+0x21e
WARNING: Stack unwind information not available. Following frames may be wrong.
0019fae0 6e30fe07     0019fb6c 00000003 00000001 igJPEG2K19d!CPb_JPEG2K_init+0x94ca
0019fb54 6e31106c     0a996f50 0019fb6c 00000000 igJPEG2K19d!CPb_JPEG2K_init+0xb147
0019fb70 6e3070a6     0a996f50 0ac5ff50 a5d44a84 igJPEG2K19d!CPb_JPEG2K_init+0xc3ac
0019fb9c 6e30711e     0019fc3c 0ae2afa0 00000000 igJPEG2K19d!CPb_JPEG2K_init+0x23e6
0019fbb4 6ed713d9     0019fc3c 0ae2afa0 00000001 igJPEG2K19d!CPb_JPEG2K_init+0x245e
0019fbec 6edb08d7     00000000 0ae2afa0 0019fc3c igCore19d!IG_image_savelist_get+0xb29
0019fe68 6edb0239     00000000 0019ff10 00000001 igCore19d!IG_mpi_page_set+0x148a7
0019fe88 6ed45757     00000000 0019ff10 00000001 igCore19d!IG_mpi_page_set+0x14209
0019fea8 00402219     0019ff10 0019febc 00000001 igCore19d!IG_load_file+0x47
0019fec0 00402524     0019ff10 052c7fe0 0522df50 Fuzzme!fuzzme+0x19
0019ff28 0040668d     00000005 05226f68 0522df50 Fuzzme!fuzzme+0x324
0019ff70 75330419     0027b000 75330400 0019ffdc Fuzzme!fuzzme+0x448d
0019ff80 778b72ed     0027b000 7d8b826c 00000000 KERNEL32!BaseThreadInitThunk+0x19
0019ffdc 778b72bd     ffffffff 778d65c2 00000000 ntdll!__RtlUserThreadStart+0x2f
0019ffec 00000000     00406715 0027b000 00000000 ntdll!_RtlUserThreadStart+0x1b


STACK_COMMAND:  ~0s ; .cxr ; kb

SYMBOL_NAME:  MSVCR110!memcpy+21e

MODULE_NAME: MSVCR110

IMAGE_NAME:  MSVCR110.dll

FAILURE_BUCKET_ID:  INVALID_POINTER_WRITE_STRING_DEREFERENCE_AVRF_c0000005_MSVCR110.dll!memcpy

OS_VERSION:  10.0.17763.1

BUILDLAB_STR:  rs5_release
```

```
OSPLATFORM_TYPE:  x86

OSNAME:  Windows 10

IMAGE_VERSION:  11.0.50727.1

FAILURE_ID_HASH:  {77975e19-9d4d-daf1-6c0e-6a3a4c334a80}

Followup:     MachineOwner
---------
```

Timeline

2021-08-30 - Initial contact
2021-08-31 - Vendor acknowledged and created support ticket
2021-09-10 - Vendor closed support ticket and confirmed under review with engineering team
2021-11-30 - 60 day follow up
2021-12-01 - Vendor advised release planned for Q1 2022
2021-12-07 - 30 day disclosure extension granted
2022-01-06 - Final disclosure notification
2022-02-23 - Public disclosure

CREDIT

Discovered by Francesco Benvenuto of Cisco Talos.

VULNERABILITY REPORTS

PREVIOUS REPORT TALOS-2021-1362

NEXT REPORT TALOS-2021-1368