Talos Vulnerability Report

TALOS-2021-1374

Accusoft ImageGear TIFF parser heap-based buffer overflow vulnerabilities

FEBRUARY 23, 2022

CVE NUMBER

CVE-2021-21945,CVE-2021-21944

Summary

Two heap-based buffer overflow vulnerabilities exist in the TIFF parser functionality of Accusoft ImageGear 19.10. A specially-crafted file can lead to a heap buffer overflow. An attacker can provide a malicious file to trigger these vulnerabilities.

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-122 - Heap-based Buffer Overflow

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.

When a TIFF file, with specific tag requirements, is loaded, its data are parsed by the `FUN_10074d50` function.

The essential tags required to reach this “parser”:

- The value of the SamplesPerPixel tag, greater than 2
- The value of the BitsPerSample tag should be `0xc` (we will focus on this, but different values are parsed by the same function)
- The value of the PlanarConfiguration tag should be 2
- The presence of either TileOffsets or StripOffsets tag with `N` greater than 1

The funtion `FUN_10074d50`:

```
dword __cdecl
FUN_10074d50(uint ID_TIF_SAMPLES_PER_PIXEL,int sample_index,uint ID_TIF_BITS_PER_SAMPLE,
            int ID_TIF_IMAGE_WIDTH,byte *src_buff,void *dest_buff,size_t source_size)

{
 [...]

  if (true) {
    switch(ID_TIF_BITS_PER_SAMPLE) {
   [...]
    case 0xc:
      alternate_branch = false;
      loop_index = 0;
      width_index = ID_TIF_IMAGE_WIDTH;
      if (0 < ID_TIF_IMAGE_WIDTH) {
        do {
          if (0 < (int)ID_TIF_SAMPLES_PER_PIXEL) {
            ID_TIF_BITS_PER_SAMPLE = ID_TIF_SAMPLES_PER_PIXEL;
            do {
              if (alternate_branch) {
                if (alternate_branch) {
                  curren_src_byte_cur = src_buff + loop_index;
                  loop_index_+1 = loop_index + 1;
                  loop_index = loop_index + 2;
                  *(ushort *)((int)dest_buff + sample_index * 2) =
                       CONCAT11(*curren_src_byte_cur,src_buff[loop_index_+1]) & 0xfff;                  [1]
                  sample_index = sample_index + ID_TIF_SAMPLES_PER_PIXEL;
                  alternate_branch = false;
                }
              }
              else {
                *(ushort *)((int)dest_buff + sample_index * 2) =
                     (ushort)(src_buff[loop_index + 1] >> 4) |
                             (ushort)src_buff[loop_index] << 4;                                         [2]
                sample_index = sample_index + ID_TIF_SAMPLES_PER_PIXEL;
                loop_index = loop_index + 1;
                alternate_branch = true;
              }
[...]
}
```

In this function the `src_buff`, which represents a “row” of data, is copied into the `dest_buff`. This function implements the data parsing for the supported `BitsPerSample` values. When `BitsPerSample` is `0xc` the copy of the data is perfomed in a loop iterated `SamplesPerPixel` times. For every two iterations, there is a writing pattern where for each 3 bytes read from `src_buff`, there are 4 written into `dest_buff`. That loop is then iterated `ImageWidth` times.

The idea is that 16 bits (i.e., 2 bytes) of `dest_buff` are filled with 12 bits of `src_buff`. At `[2]` the first 12 bits of the source are manipulated, and the remaining 12, in order to complete 3 bytes read, are manipulated at `[1]`. It is important to note that the acess to `dest_buff` is not sequential, but instead, it is calculated using `sample_index` as base offset, incremented each iteration by `SamplesPerPixel`, a value taken from the homonymous TIFF tag.

The call to `FUN_10074d50` is originated by the `TIFF_parse` function, the “main” TIFF parser:

```
void TIFF_parse(mys_table_function *param_1,uint param_2,mys_tags_data *TIFF_tags,undefined4 param_4
               ,HIGDIBINFO param_5,subsapling_Y_Cb_Cr *YCbCr_subsamp)

{
  [...]

  dst_buff = (byte *)0x0;
  width_buff_size = 0;
  local_c = 0;
  multipler = (byte *)0x0;
  arr_of_dest_buff = (byte **)0x0;
  if (*(ushort *)&TIFF_tags->ID_TIF_SAMPLES_PER_PIXEL == 0) {
    AF_err_record_set("..\\..\\..\\..\\Common\\Formats\\tifread.c",0x1793,-0x80d,0,0,0,(LPCHAR)0x0);
    AF_error_check();
    return;
  }
  io_buff = (io_buffer *)
            AF_memm_alloc(param_2,(uint)*(ushort *)&TIFF_tags->ID_TIF_SAMPLES_PER_PIXEL * 0x34);
  if (io_buff == (io_buffer *)0x0) {
    AF_err_record_set("..\\..\\..\\..\\Common\\Formats\\tifread.c",0x1799,-1000,0,0,0,(LPCHAR)0x0);
    AF_error_check();
    return;
  }
  src_buff = (byte **)AF_memm_alloc(param_2,(uint)*(ushort *)&TIFF_tags->ID_TIF_SAMPLES_PER_PIXEL <<
                                            2);
  if (src_buff == (byte **)0x0) {
    sample_per_pixel_index = 0x17a1;
    lpExtraText_00 = src_buff;
  }
  else {
    OS_memset(src_buff,0,(uint)*(ushort *)&TIFF_tags->ID_TIF_SAMPLES_PER_PIXEL << 2);
    lpExtraText_00 =
         (byte **)AF_memm_alloc(param_2,(uint)*(ushort *)&TIFF_tags->ID_TIF_SAMPLES_PER_PIXEL << 2);
    if (lpExtraText_00 != (byte **)0x0) {
      if (TIFF_tags->ID_TIF_PHOTO_INTERP == IG_TIF_PHOTO_YCBCR) {
        [...]
      }
      else {
LAB_10177d86:
        if (TIFF_tags->ID_TIF_PLANAR_CONFIG == 1) {
          [...]
        }
        else {
          if (TIFF_tags->ID_TIF_PLANAR_CONFIG == 2) {
            if (TIFF_tags->ID_TIF_PHOTO_INTERP == IG_TIF_PHOTO_YCBCR) {
              [...]
            }
            else {
              sample_per_pixel_index = 0;
              if (*(short *)&TIFF_tags->ID_TIF_SAMPLES_PER_PIXEL != 0) {
                do {
                  loop_index = sample_per_pixel_index + 1;
                  iVar1 = (int)*(short *)((TIFF_tags->ID_TIF_BITS_PER_SAMPLE - 2) + loop_index * 2)
                          * TIFF_tags->ID_TIF_IMAGE_WIDTH + 7;
                  lpExtraText_00[sample_per_pixel_index] =
                       (byte *)((int)(iVar1 + (iVar1 >> 0x1f & 7U)) >> 3);                         [3]
                  sample_per_pixel_index = loop_index;
                } while (loop_index < (int)(uint)*(ushort *)&TIFF_tags->ID_TIF_SAMPLES_PER_PIXEL);
              }
              width_buff_size = IO_raster_size_get(param_5);
              dst_buff = (byte *)AF_memm_alloc(param_2,width_buff_size);                           [4]
              if (dst_buff == (byte *)0x0) {
                AF_err_record_set("..\\..\\..\\..\\Common\\Formats\\tifread.c",0x1843,-1000,0,
                                  param_2,width_buff_size,(LPCHAR)0x0);
                goto LAB_10178379;
              }
            }
            dVar3 = 0;
            sample_size_index = 0;
            piVar4 = io_buff;
            if (*(short *)&TIFF_tags->ID_TIF_SAMPLES_PER_PIXEL != 0) {
              do {
                dVar2 = IOb_init(param_1,param_2,piVar4,(int)lpExtraText_00[sample_size_index] * 5,1
                                );                                                                 [5]
                if (0 < (int)dVar2) {
                  dVar3 = 1;
                  local_c = 1;
                  break;
                }
                sample_size_index = sample_size_index + 1;
                piVar4 = (io_buffer *)&piVar4->size_buffer;
              } while (sample_size_index <
                       (int)(uint)*(ushort *)&TIFF_tags->ID_TIF_SAMPLES_PER_PIXEL);
            }
            sample_per_pixel_index = 0;
            param_5 = (HIGDIBINFO)0x0;
            for (local_18 = 0;
                (dVar3 == 0 &&
                (local_18 < (int)TIFF_tags->from_ID_TIF_STRIP_OFFSET_or_ID_TIF_TILE_OFFSETS));
                local_18 = local_18 + 1) {
              if ((TIFF_tags->ID_TIF_TILE_OFFSETS != 0) &&
                 (*(short *)&TIFF_tags->ID_TIF_SAMPLES_PER_PIXEL != 0)) {
                iVar1 = 0;
                piVar4 = io_buff;
                do {
                  perform_some_read_or_write_intofile
                            (piVar4,*(int *)(TIFF_tags->ID_TIF_TILE_OFFSETS +
                                       (TIFF_tags->
                                       result_strip_tile_offset_divided_sample_per_pixel *
                                       iVar1 + local_18) * 4) + TIFF_tags->IFD_Offset,0,0);        [6]
                  iVar1 = iVar1 + 1;
                  piVar4 = (io_buffer *)&piVar4->size_buffer;
                  dVar3 = local_c;
                  sample_per_pixel_index = (int)param_5;
                } while (iVar1 < (int)(uint)*(ushort *)&TIFF_tags->ID_TIF_SAMPLES_PER_PIXEL);
              }
              local_28 = 0;
              if (0 < (int)TIFF_tags->ID_TIF_ROWS_PER_STRIP) {
                do {
                  local_c = dVar3;
                  if ((int)TIFF_tags->ID_TIF_IMAGE_HEIGHT <= sample_per_pixel_index) break;
                  iVar1 = 0;
                  sample_per_pixel_index = (int)param_5;
                  if (TIFF_tags->ID_TIF_PHOTO_INTERP == IG_TIF_PHOTO_YCBCR) {
                    [...]
                  }
                  else {
```

```
                    piVar4 = io_buff;
                    if (*(short *)&TIFF_tags->ID_TIF_SAMPLES_PER_PIXEL != 0) {
                      do {
                        vert_buff = (byte *)get_data_from_file(piVar4,(uint)*lpExtraText_00);
                        src_buff[iVar1] = vert_buff;
                        if (vert_buff == (byte *)0x0) {
                          AF_err_record_set("..\\..\\..\\..\\Common\\Formats\\tifread.c",0x1887,
                                            -0x803,0,(AT_INT)*lpExtraText_00,0,(LPCHAR)0x0);
                          dVar3 = dVar3 + 1;
                          break;
                        }
                        iVar1 = iVar1 + 1;
                        piVar4 = (io_buffer *)&piVar4->size_buffer;
                      } while (iVar1 < (int)(uint)*(ushort *)&TIFF_tags->ID_TIF_SAMPLES_PER_PIXEL);
                    }
                    local_c = dVar3;
                    if (dVar3 != 0) break;
                    FUN_1017c970(TIFF_tags,param_4,src_buff,lpExtraText_00,dst_buff);                   [7]

                    [...]
}
```

This function is responsible for preparing the `src_buff` and `dest_buff` and calling the correct TIFF “sub-parser”. At `[5]`, `SamplesPerPixel` buffers are allocated, each with size calcualted at `[3]`. These buffers, in this specific scenario, are the same sizes, and each individually will be used as `src_buff`. The size of a `src_buff` is:

```
src_size = (((BitsPerSample & 0xffff) * width + 7) >> 3) * 5
```

Eventually, at `[6]` these buffers are filled.

At `[4]` the `dest_buff` is allocated, using as size the return value of the function `IO_raster_size_get`. The return value of `IO_raster_size_get`, in this specific case, can be simplified as:

```
dest_size = (((next_mult_of_8_of_BitsPerSample * SamplesPerPixel * ImageWidth) + 0x1f) >> 3) & 0xfffffffc
```

Where `ImageWidth` and `SamplesPerPixel` correspond to the homonymous TIFF tags. Instead, `next_mult_of_8_of_BitsPerSample` is the next multiple of 8 of `BitsPerSample`. That is, like the other two, a value directly taken from a TIFF tag.

The `TIFF_parse` then calls at `[7]` the `FUN_1017c970` function, that essentially calls `FUN_10074d50` with each `dest_buff` and `sample_index`, the variable used as base offset to access the `src_buff` that goes from 0 to `SamplesPerPixel`.

CVE-2021-21944 - TIFF parser - planar format. First 12 bits

A specially-crafted TIFF file can lead to a heap-based buffer overflow in the TIFF image parser, due to a missing boundary check.

Trying to load a malicious TIFF file, we end up in the following situation:

```
(dcc.182c): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000036 ebx=00000007 ecx=0bd58e08 edx=00000414 esi=000000fc edi=0bd5eef0
eip=70104f1f esp=0019f588 ebp=0019f5ac iopl=0         nv up ei pl nz na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00010206
igCore19d!IG_mpi_page_set+0x8eef:
70104f1f 66891471        mov     word ptr [ecx+esi*2],dx  ds:002b:0bd59000=????
```

This access violation take place at `[2]` in the `FUN_10074d50` function, when trying to copy the first 12 bits from `src_buff` to `dest_buff`.

From the allocation of `src_buff` and `dest_buff`, in the `TIFF_parse`, to `[2]`, the program does not check, taking into account the writing pattern used, a possible out-of-bounds access.

For example:

```
sample_index    = 0
ImageWidth      = 0x24
SamplesPerPixel = 0x7
BitsPerSample   = 0xc
```

We will obtain a `dest_size` of `0x1f8` and `src_size` of `0x10e`.

At the fifth iteration of the outer loop in `FUN_10074d50` (i.e., ImageWidth loop iteration 5) and the first of the inner one (i.e., SamplesPerPixel loop iteration 1) the element accessed at that iteration would be:

```
(sample_index + SamplesPerPixel*SamplesPerPixel) * width_iteration +
        sample_index + (sample_per_pixel_iteration * SamplesPerPixel) =
(0 + 7 * 7) * 5 + (0 + 1 * 7) = 0xfc
```

Because the `dest_size` is a 16 bits buffer, the element `0xfc` is located at offset `0x1f8` and `0x1f9`, and because the `dest_size` is only `0x1f8` bytes long we are accessing that heap buffer out-of-bound.

So based on the specific TIFF tags, the `dest_buff` could be bigger or smaller than a single `src_buff`. In either case a heap-base buffer oveflow could occur due to the specific writting pattern and the missing boundary check.

CVE-2021-21945 - TIFF parser - planar format. Second 12 bits

Trying to load a malicious TIFF file, we end up in the following situation:

```
(22a8.1bbc): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000006 ebx=00000007 ecx=0bd28fe0 edx=00000141 esi=00000015 edi=0bd48ff0
eip=6f3b4efd esp=0019f588 ebp=0019f5ac iopl=0         nv up ei pl nz na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00010206
igCore19d!IG_mpi_page_set+0x8ecd:
6f3b4efd 66891471        mov     word ptr [ecx+esi*2],dx  ds:002b:0bd2900a=????
```

This access violation takes place at `[1]` in the `FUN_10074d50` function, when trying to copy the second 12 bits from `src_buff` to `dest_buff`.

From the allocation of `src_buff` and `dest_buff`, in the `TIFF_parse`, to `[1]`, the program does not check, taking into account the writing pattern used, a possible out-of-bounds access.

For example with:

```
sample_index    = 0
ImageWidth      = 0x4
SamplesPerPixel = 0x7
BitsPerSample   = 0xc
```

We will obtain a `dest_size` of `0x38` and `src_size` of `0x1e`

At the first iteration of the outer loop in `FUN_10074d50` (i.e., ImageWidth loop iteration 0) and the fourth one of the inner one (i.e., SamplesPerPixel loop iteration 4) the element accessed at that iteration would be:

```
(sample_index + SamplesPerPixel*SamplesPerPixel) * width_iteration +
        sample_index + (sample_per_pixel_iteration * SamplesPerPixel) =
        7 * 4 = 0x1c
```

Because the `dest_size` is a 16 bits buffer, the element `0x1c` is located at offset `0x38` and `0x39` and because the `dest_size` is only `0x38` bytes long we are accessing that heap buffer out-of-bound.

Timeline

2021-09-10 - Initial contact
2021-09-14 - Vendor acknowledged and created support ticket
2021-09-21 - Vendor closed support ticket and confirmed under review with engineering team
2021-11-30 - 60 day follow up
2021-12-02 - 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-2021-1373

NEXT REPORT
TALOS-2021-1375