Talos Vulnerability Report

TALOS-2021-1276

Accusoft ImageGear PNG png_palette_process memory corruption vulnerability

JUNE 1, 2021

CVE NUMBER

CVE-2021-21808

Summary

A memory corruption vulnerability exists in the PNG png_palette_process functionality of Accusoft ImageGear 19.9. A specially crafted malformed file can lead to a heap buffer overflow. An attacker can provide malicious inputs to trigger this vulnerability.

Tested Versions

Accusoft ImageGear 19.9

Product URLs

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

CVSSv3 Score

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

CWE

CWE-120 - Buffer Copy without Checking Size of Input (‘Classic 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.

A specially crafted PNG file can lead to an out-of-bounds write in the `png_palette_process` function, due to a buffer overflow caused by a write without checking the destination buffer size.
Trying to load a malformed PNG file, we end up in the following situation:

```
===========================================================
VERIFIER STOP 0000000F: pid 0x35FD0: corrupted suffix pattern

    04F21000 : Heap handle
    04F55FF8 : Heap block
    00000001 : Block size
    04F55FF9 : corruption address
===========================================================
This verifier stop is not continuable. Process will be terminated
when you use the `go' debugger command.
===========================================================

(35fd0.33868): Break instruction exception - code 80000003 (first chance)
eax=002ff000 ebx=00000000 ecx=00000001 edx=0019d2b0 esi=7a58aa40 edi=00000000
eip=7a58dab2 esp=0019d250 ebp=0019d258 iopl=0         nv up ei pl nz na po nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000202
verifier!VerifierBreakin+0x42:
7a58dab2 cc              int     3
```

The memory corruption appearing in the crash details notifies us the block size is `1` byte. By inspecting the stack trace, we can identify where the corruption is happening during a free. Below we can see `MSVCR110!free+0x1a` is called with `04f55ff8` as first parameter:

```
STACK_TEXT:
0019d258 7a58dbb0     c0000421 00000000 00000000 verifier!VerifierBreakin+0x42
0019d580 7a58dead     0000000f 04f21000 04f55ff8 verifier!VerifierCaptureContextAndReportStop+0xf0
0019d5c4 7a58b945     0000000f 7a581e58 04f21000 verifier!VerifierStopMessage+0x2bd
0019d630 7a58bc2c     04f21000 00000000 04f55ff8 verifier!AVrfpDphReportCorruptedBlock+0x285
0019d6a0 7a58893a     04f21000 04f21af8 00000000 verifier!AVrfpDphCheckPageHeapBlock+0x1bc
0019d6cc 7a588ae0     04f21000 04f55ff8 0019d75c verifier!AVrfpDphFindBusyMemory+0xda
0019d6e8 7a58aad0     04f21000 04f55ff8 04f24750 verifier!AVrfpDphFindBusyMemoryAndRemoveFromBusyList+0x20
0019d704 77bcf796     04f20000 01000002 04f55ff8 verifier!AVrfDebugPageHeapFree+0x90
0019d76c 77b33be6     04f55ff8 1dcee6af 00000000 ntdll!RtlDebugFreeHeap+0x3e
0019d8c8 77b7778d     00000000 04f55ff8 04f55ff8 ntdll!RtlpFreeHeap+0xd6
0019d924 77b33ab6     00000000 00000000 00000000 ntdll!RtlpFreeHeapInternal+0x783
0019d940 7a5fdcc2     04f20000 00000000 04f55ff8 ntdll!RtlFreeHeap+0x46
0019d954 7a0569ad     04f55ff8 10000020 04f55ff8 MSVCR110!free+0x1a
WARNING: Stack unwind information not available. Following frames may be wrong.
0019d96c 7a140df1     1000001e 04f55ff8 7a261920 igCore19d!AF_memm_alloc+0x7ed
0019f5e4 7a141b84     0019fb30 1000001e 0db0afe8 igCore19d!IG_mpi_page_set+0xe50a1
0019f618 7a13f2b2     0019fb30 1000001e 0db0afe8 igCore19d!IG_mpi_page_set+0xe5e34
0019faa8 7a0310d9     0019fb30 0db0afe8 00000001 igCore19d!IG_mpi_page_set+0xe3562
0019fae0 7a070557     00000000 0db0afe8 0019fb30 igCore19d!IG_image_savelist_get+0xb29
0019fd5c 7a06feb9     00000000 05514f88 00000001 igCore19d!IG_mpi_page_set+0x14807
0019fd7c 7a005777     00000000 05514f88 00000001 igCore19d!IG_mpi_page_set+0x14169
0019fd9c 00498a3a     05514f88 0019fe0c 004801a4 igCore19d!IG_load_file+0x47
0019fe14 00498e36     05514f88 0019fe8c 004801a4 Fuzzme!fuzzme+0x4a
0019fee4 004daa53     00000005 05454f28 0545df20 Fuzzme!main+0x376
0019ff04 004da8a7     35cbeac8 004801a4 004801a4 Fuzzme!invoke_main+0x33
0019ff60 004da73d     0019ff70 004daad8 0019ff80 Fuzzme!__scrt_common_main_seh+0x157
0019ff68 004daad8     0019ff80 7628fa29 002ff000 Fuzzme!__scrt_common_main+0xd
0019ff70 7628fa29     002ff000 7628fa10 0019ffdc Fuzzme!mainCRTStartup+0x8
0019ff80 77b57c7e     002ff000 1dcec1bb 00000000 KERNEL32!BaseThreadInitThunk+0x19
0019ffdc 77b57c4e     ffffffff 77b788ce 00000000 ntdll!__RtlUserThreadStart+0x2f
0019ffec 00000000     004801a4 002ff000 00000000 ntdll!_RtlUserThreadStart+0x1b
```

Going down the stack, we can see the adresses `igCore19d!IG_mpi_page_set+0xe50a1`, which leads us to the function `FUN_101503a0` which is responsible to perform the `free` call:

```
LINE1    dword FUN_101503a0(mys_table_function *mys_table_function,uint kind_of_heap,int param_3,
LINE2                      PNG_object *PNG_Object,PNG_object *PNG_Object_2,HIGDIBINFO HIGDIBINFO,
LINE3                      undefined4 param_7,undefined4 param_8)
LINE4    {
LINE5
LINE6      local_8 = DAT_102bcea8 ^ (uint)&stack0xfffffffc;
LINE7      _kind_of_heap = kind_of_heap;
LINE8      _PNG_Object = PNG_Object;
LINE9      _HIGDIBINFO = HIGDIBINFO;
LINE10     local_20 = 0x200000f;
LINE11     local_1c = 0x1000100;
LINE12     local_18 = 0x4000f;
LINE13     local_14 = 0x10002;
LINE14     local_10 = 0x404040f;
LINE15     local_c = 0x1010202;
LINE16     local_28 = 0x408080f;
LINE17     local_24 = 0x1020204;
LINE18     local_1c00 = 0;
LINE19     local_1c44 = 0;
LINE20     local_1c30 = (png_to_be_defined *)AF_memm_alloc(kind_of_heap,0x60);
LINE21     uVar18 = (undefined)in_stack_ffffe3a8;
LINE22     if (local_1c30 == NULL) {
LINE23       AF_err_record_set("..\\..\\..\\..\\Common\\Formats\\pngread.c",0xb85,-1000,0,0x60,kind_of_heap,
LINE24                         NULL);
LINE25       _status_fastfail = kind_of_fastfail(local_8 ^ (uint)&stack0xfffffffc,extraout_DL,uVar18);
LINE26       return _status_fastfail;
LINE27     }
LINE28     _size_from_color = png_compute_size_from_color_and_width(PNG_Object);
LINE29     size_buff_to_alloc = (_size_from_color - (_size_from_color & 0x3f)) + 0x40;
LINE30     buff_64_bytes = (char *)AF_memm_alloc(kind_of_heap,size_buff_to_alloc);
LINE31     uVar18 = (undefined)in_stack_ffffe3a8;
LINE32     if (buff_64_bytes == NULL) {
LINE33       AF_memm_free_all(kind_of_heap);
LINE34       AF_err_record_set("..\\..\\..\\..\\Common\\Formats\\pngread.c",0xb93,-1000,0,_size_from_color,
LINE35                         kind_of_heap,NULL);
LINE36       _fastfail = kind_of_fastfail(local_8 ^ (uint)&stack0xfffffffc,extraout_DL_00,uVar18);
LINE37       return _fastfail;
LINE38     }
LINE39     _buff_2_64_bytes = (undefined4 *)AF_memm_alloc(kind_of_heap,size_buff_to_alloc);
LINE40     uVar18 = (undefined)in_stack_ffffe3a8;
LINE41     buff2_64_bytes = _buff_2_64_bytes;
LINE42     if (_buff_2_64_bytes == NULL) {
LINE43       AF_memm_free_all(kind_of_heap);
LINE44       AF_err_record_set("..\\..\\..\\..\\Common\\Formats\\pngread.c",0xb9d,-1000,0,_size_from_color,
LINE45                         kind_of_heap,NULL);
LINE46       _fastfail_2 = kind_of_fastfail(local_8 ^ (uint)&stack0xfffffffc,extraout_DL_01,uVar18);
LINE47       return _fastfail_2;
LINE48     }
LINE49     local_1c18 = AF_memm_alloc(kind_of_heap,_size_from_color);
LINE50     uVar18 = (undefined)in_stack_ffffe3a8;
LINE51     if (local_1c18 == NULL) {
LINE52       AF_memm_free_all(kind_of_heap);
LINE53       AF_err_record_set("..\\..\\..\\..\\Common\\Formats\\pngread.c",0xba6,-1000,0,_size_from_color,
LINE54                         kind_of_heap,NULL);
LINE55       _fastfail3 = kind_of_fastfail(local_8 ^ (uint)&stack0xfffffffc,extraout_DL_02,uVar18);
LINE56       return _fastfail3;
LINE57     }
LINE58     if (_size_from_color != 0) {
LINE59       uVar7 = _size_from_color >> 2;
LINE60       while (uVar7 != 0) {
LINE61         uVar7 = uVar7 - 1;
LINE62         *_buff_2_64_bytes = 0;
LINE63         _buff_2_64_bytes = _buff_2_64_bytes + 1;
LINE64       }
LINE65       uVar7 = _size_from_color & 3;
LINE66       while (kind_of_heap = _kind_of_heap, uVar7 != 0) {
LINE67         uVar7 = uVar7 - 1;
LINE68         *(undefined *)_buff_2_64_bytes = 0;
LINE69         _buff_2_64_bytes = (undefined4 *)((int)_buff_2_64_bytes + 1);
LINE70       }
LINE71     }
LINE72     raster_size = raster_size_from_HIGDIBINFO(HIGDIBINFO);
LINE73     __size_corrupted_buffer = raster_size;
LINE74     raster_size_buffer = (char *)AF_memm_alloc(kind_of_heap,raster_size);
LINE75     uVar7 = _kind_of_heap;
LINE76     uVar18 = (undefined)in_stack_ffffe3a8;
LINE77     _raster_size_buffer = raster_size_buffer;
LINE78     if (raster_size_buffer == NULL) {
LINE79       AF_memm_free_all(_kind_of_heap);
LINE80       AF_err_record_set("..\\..\\..\\..\\Common\\Formats\\pngread.c",0xbb4,-1000,0,raster_size,uVar7,
LINE81                         NULL);
LINE82       raster_size = kind_of_fastfail(local_8 ^ (uint)&stack0xfffffffc,extraout_DL_03,uVar18);
LINE83       return raster_size;
LINE84     }
LINE85     wrapper_memset(&local_1bf0,0,0x1bc8);
LINE86     uVar18 = (undefined)in_stack_ffffe3a8;
LINE87     local_1bec = 2;
LINE88     if (_PNG_Object->InterlaceType == 1) {
            [...]
LINE303    }
LINE304    else {
LINE305      iVar15 = 0;
LINE306      uVar7 = _kind_of_heap;
LINE307      if (0 < (int)_PNG_Object->Height) {
LINE308        while( true ) {
LINE309          iVar5 = FUN_10150fc0(mys_table_function,&local_1bf0,buff_64_bytes,(uint *)_size_from_color);
LINE310          uVar18 = (undefined)in_stack_ffffe3a8;
LINE311          uVar7 = _kind_of_heap;
LINE312          if (iVar5 != 0) break;
LINE313          uVar7 = __size_corrupted_buffer - 3;
LINE314          if (uVar7 < __size_corrupted_buffer) {
LINE315            uVar9 = __size_corrupted_buffer - uVar7 >> 2;
LINE316            _buff_2_64_bytes = (undefined4 *)(raster_size_buffer + uVar7);
LINE317            while (uVar9 != 0) {
LINE318              uVar9 = uVar9 - 1;
LINE319              *_buff_2_64_bytes = 0;
LINE320              _buff_2_64_bytes = _buff_2_64_bytes + 1;
LINE321            }
LINE322            uVar7 = __size_corrupted_buffer - uVar7 & 3;
LINE323            while (raster_size_buffer = _raster_size_buffer, uVar7 != 0) {
LINE324              uVar7 = uVar7 - 1;
LINE325              *(undefined *)_buff_2_64_bytes = 0;
LINE326              _buff_2_64_bytes = (undefined4 *)((int)_buff_2_64_bytes + 1);
LINE327            }
LINE328          }
LINE329          bVar12 = (byte)((_size_from_color - 1) / _PNG_Object->Width);
LINE330          if (bVar12 == 0) {
```

```
LINE331            bVar12 = 1;
LINE332          }
LINE333          local_1c3c = local_1c3c & 0xffffff00 | (uint)bVar12;
LINE334          png_process_colortype
LINE335                    (buff_64_bytes,(char *)buff2_64_bytes,raster_size_buffer,0,_size_from_color,bVar12
LINE336                     ,PNG_Object_2,param_8,*(int *)(param_3 + 0x10));
LINE337          wrapper_memcpy(local_1c18,buff_64_bytes,_size_from_color);
LINE338          wrapper_memcpy(buff_64_bytes,buff2_64_bytes,_size_from_color);
LINE339          wrapper_memcpy(buff2_64_bytes,local_1c18,_size_from_color);
LINE340          raster_size = FUN_1014f020(_HIGDIBINFO,mys_table_function,(int)raster_size_buffer,iVar15,
LINE341                                     __size_corrupted_buffer);
LINE342          uVar18 = (undefined)in_stack_ffffe3a8;
LINE343          uVar7 = _kind_of_heap;
LINE344          if ((raster_size != 0) || (iVar15 = iVar15 + 1, (int)_PNG_Object->Height <= iVar15)) break;
LINE345        }
LINE346      }
LINE347    }
LINE348    if (local_1620 != 0) {
LINE349      uVar9 = *(uint *)(local_1620 + 0x18);
LINE350      AF_memm_free(uVar9,*(void **)(local_1620 + 0x14));
LINE351      AF_memm_free_all(uVar9);
LINE352    }
LINE353    FUN_10102fd0(&local_1bf0);
LINE354    IO_byte_order_set(mys_table_function,1);
LINE355    AF_memm_free(uVar7,local_1c30);
LINE356    AF_memm_free(uVar7,buff_64_bytes);
LINE357    AF_memm_free(uVar7,buff2_64_bytes);
LINE358    AF_memm_free(uVar7,raster_size_buffer);
LINE359    AF_memm_free(uVar7,local_1c18);
LINE360    raster_size = AF_error_check();
LINE361    uVar11 = extraout_DL_08;
LINE362    if (raster_size != 0) {
LINE363      AF_err_error_get(0,raster_size - 1,NULL,0,NULL,&local_1c44,NULL,NULL,NULL,0);
LINE364      uVar11 = extraout_DL_09;
LINE365    }
LINE366    raster_size = kind_of_fastfail(local_8 ^ (uint)&stack0xfffffffc,uVar11,uVar18);
LINE367    return raster_size;
LINE368  }
```

The `free` invocation responsible for the exception is in LINE358 via the call to `AF_memm_free` function against the buffer `raster_size_buffer`. The `AF_memm_free` is somehow just some free wrapper on top of Imagegear memory allocator.

The `raster_size_buffer` is allocated in this same function `FUN_101503a0` in LINE74, using the size returned from the call to `raster_size_from_HIGDIBINFO` in LINE72. The pseudo-code is the following:

```
LINE369  dword raster_size_from_HIGDIBINFO(HIGDIBINFO HIGDIBINFO)
LINE370  {
LINE371    dword bit_depth;
LINE372    uint uVar1;
LINE373
LINE374    bit_depth = IGDIBStd::DIB_bit_depth_get(HIGDIBINFO);
LINE375    if (bit_depth == 1) {
LINE376      uVar1 = DIB1bit_packed_raster_size_get(HIGDIBINFO);
LINE377      return uVar1;
LINE378    }
LINE379    if (bit_depth == 4) {
LINE380      uVar1 = DIB_width_get(HIGDIBINFO);
LINE381      return uVar1;
LINE382    }
LINE383    uVar1 = DIBStd_raster_size_get(HIGDIBINFO);
LINE384    return uVar1;
LINE385  }
```

In this function `raster_size_from_HIGDIBINFO` we can see the value returned `uVar1` is depending on the `bit_depth` value extracted from the object `HIGDIBINFO`, which is in our case the value of `4`. The `HIGDIBINFO` is an object created during the init process through a call to the function `create_LPHIGDIBINFO_from_png` with the following pseudo code:

```
LINE386  dword create_LPHIGDIBINFO_from_png
LINE387                  (undefined4 kind_of_heap,int param_2,PNG_object *PNG_Object,
LINE388                  LPHIGDIBINFO LPHIGDIBINFO)
LINE389  {
LINE390    uVar2 = 0;
LINE391    bit_depth = (uint)PNG_Object->BitDepth;
LINE392    iVar3 = 0;
LINE393    if ((_DAT_102ae124 == 4) &&
LINE394       ((0x7fffffff < PNG_Object->Width || (0x7fffffff < PNG_Object->Height)))) {
LINE395      dVar1 = AF_err_record_set("..\\..\\..\\..\\Common\\Formats\\pngread.c",0x832,-0xd48,0,0,0,NULL);
LINE396      return dVar1;
LINE397    }
LINE398    switch(PNG_Object->ColorType) {
LINE399    default:
LINE400      uVar2 = 3;
LINE401      iVar3 = 1;
LINE402      break;
LINE403    case 4:
LINE404      uVar2 = 0x100;
LINE405      iVar3 = 1;
LINE406    case 0:
LINE407      uVar2 = uVar2 | 2;
LINE408      iVar3 = iVar3 + 1;
LINE409      if (bit_depth < 8) {
LINE410        uVar2 = 3;
LINE411      }
LINE412      break;
LINE413    case 6:
LINE414      if ((*(int *)(param_2 + 0x10) != 0) && ((bit_depth == 8 || (bit_depth == 0x10)))) {
LINE415        uVar2 = 0x100;
LINE416        iVar3 = 1;
LINE417      }
LINE418    case 2:
LINE419      uVar2 = uVar2 | 1;
LINE420      iVar3 = iVar3 + 3;
LINE421      goto LAB_1015196e;
LINE422    }
LINE423    if (bit_depth == 2) {
LINE424      bit_depth = 4;
LINE425    }
LINE426  LAB_1015196e:
LINE427    CreateLPHDIB(LPHIGDIBINFO,PNG_Object->Width,PNG_Object->Height,uVar2,iVar3,bit_depth);
LINE428    DIB_resolution_set(*LPHIGDIBINFO,&PNG_Object->png_phys_encoded);
LINE429    return 0;
LINE430  }
```

We can see that if the `bit_depth` LINE423 (is extracted directly from the file) is the value of `2`, it's converted into the value of `4` before creating the object `HIGDIBINFO` through the call to `CreateLPHDIB` LINE427.
Now we can inspect the function `DIB_width_get` which will explain us the computed raster size result with the following pseudo-code:

```
LINE431  AT_DIMENSION DIB_width_get(HIGDIBINFO higdibinfo)
LINE432  {
LINE433    return higdibinfo->size_X;
LINE434  }
```

As we can see then the buffer which is freed (`raster_size_buffer`) has a size that is the width taken directly from the file, as we can see in LINE433, and which has a value of `1`.

Now the corruption happens in the function `png_process_colortype` in LINE334, where we can see our third parameter is corresponding to our buffer `raster_size_buffer`.

```
LINE435  dword png_process_colortype
LINE436                  (char *param_1,char *param_2,char *raster_size_buffer,dword null_constant,
LINE437                  dword kind_of_size,byte param_6,PNG_object *PNG_Object,undefined4 param_8,
LINE438                  int param_9)
LINE439  {
LINE440   [...]
LINE454    switch(PNG_Object->ColorType) {
          [...]
LINE498    case 3:
LINE499      if (PNG_Object->BitDepth != 2) {
LINE500  LAB_101520cd:
LINE501        pPVar5 = (char *)wrapper_memcpy(raster_size_buffer,param_1 + 1,kind_of_size - 1);
LINE502        return (dword)pPVar5;
LINE503      }
LINE504  LAB_10151fa8:
LINE505      pPVar9 = (PNG_object *)png_palette_process(param_1,raster_size_buffer,kind_of_size);
LINE506      return (dword)pPVar9;
            [...]
LINE545    }
LINE546    return (dword)PNG_Object;
LINE547  }
```

We can see this buffer is used with the `png_palette_process` function LINE505 depending of the value `ColorType`, taken directly from the file too, which is in our case `3`:

```
LINE548  dword png_palette_process(char *param_1,char *buffer,uint kind_of_size)
LINE549  {
LINE550    [...]
LINE551    _max_size = 1;
LINE552    _tmp_max_size = 1;
LINE553    if (1 < kind_of_size) {
LINE554      do {
LINE555        bVar2 = 6;
LINE556        _computed_value._0_2_ = 0;
LINE557        uVar4 = 0xc0;
LINE558        iVar5 = 4;
LINE559        do {
LINE560          local_10 = (ushort)(byte)param_1[_tmp_max_size];
LINE561          uVar3 = (ushort)uVar4;
LINE562          uVar4 = uVar4 >> 2;
LINE563          bVar1 = bVar2 & 0x1f;
LINE564          bVar2 = bVar2 - 2;
LINE565          _computed_value._0_2_ = (ushort)_computed_value | (uVar3 & local_10) << bVar1;
LINE566          iVar5 = iVar5 + -1;
LINE567        } while (iVar5 != 0);
LINE568        _computed_value = (dword)(ushort)_computed_value;
LINE569        *buffer = (char)(_computed_value >> 8);
LINE570        buffer[1] = (char)_computed_value;
LINE571        _max_size = _tmp_max_size + 1;
LINE572        buffer = buffer + 2;
LINE573        _tmp_max_size = _max_size;
LINE574      } while (_max_size < kind_of_size);
LINE575    }
LINE576    return _max_size;
LINE577  }
```

And finally the vulnerability lies in the fact that there is no check against the buffer size `buffer` (in our case it has a size of 1) in LINE569 and LINE570 if `kind_of_size` is greater than `1` (in our case it's `2`).

Thus we're facing a one byte out of bounds write into this `buffer` causing a heap corruption in case the width is set to `1` and the bit_depth to `2`, which, with careful heap manipulation, could lead to code execution.

Crash Information

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

APPLICATION_VERIFIER_HEAPS_CORRUPTED_HEAP_BLOCK_SUFFIX (f)
Corrupted suffix pattern for heap block.
Most typically this happens for buffer overrun errors. Sometimes the application
verifier places non-accessible pages at the end of the allocation and buffer
overruns will cause an access violation and sometimes the heap block is
followed by a magic pattern. If this pattern is changed when the block gets
freed you will get this break. These breaks can be quite difficult to debug
because you do not have the actual moment when corruption happened.
You just have access to the free moment (stop happened here) and the
allocation stack trace (!heap -p -a HEAP_BLOCK_ADDRESS)
Arguments:
Arg1: 04f21000, Heap handle used in the call.
Arg2: 04f55ff8, Heap block involved in the operation.
Arg3: 00000001, Size of the heap block.
Arg4: 04f55ff9, Corruption address.

KEY_VALUES_STRING: 1

    Key  : AVRF.Code
    Value: f

    Key  : AVRF.Exception
    Value: 1

    Key  : Analysis.CPU.mSec
    Value: 2749

    Key  : Analysis.DebugAnalysisManager
    Value: Create

    Key  : Analysis.Elapsed.mSec
    Value: 31375

    Key  : Analysis.Init.CPU.mSec
    Value: 3218

    Key  : Analysis.Init.Elapsed.mSec
    Value: 63862224

    Key  : Analysis.Memory.CommitPeak.Mb
    Value: 172

    Key  : Timeline.OS.Boot.DeltaSec
    Value: 501140

    Key  : Timeline.Process.Start.DeltaSec
    Value: 63861

    Key  : WER.OS.Branch
    Value: vb_release

    Key  : WER.OS.Timestamp
    Value: 2019-12-06T14:06:00Z

    Key  : WER.OS.Version
    Value: 10.0.19041.1

    Key  : WER.Process.Version
    Value: 1.0.1.1


NTGLOBALFLAG:  2100000

APPLICATION_VERIFIER_FLAGS:  0

APPLICATION_VERIFIER_LOADED: 1

EXCEPTION_RECORD:  (.exr -1)
ExceptionAddress: 7a58dab2 (verifier!VerifierBreakin+0x00000042)
   ExceptionCode: 80000003 (Break instruction exception)
  ExceptionFlags: 00000000
NumberParameters: 1
   Parameter[0]: 00000000

FAULTING_THREAD:  00033868

PROCESS_NAME:  Fuzzme.exe

ERROR_CODE: (NTSTATUS) 0x80000003 - {EXCEPTION}  Breakpoint  A breakpoint has been reached.

EXCEPTION_CODE_STR:  80000003

EXCEPTION_PARAMETER1:  00000000

STACK_TEXT:  
0019d258 7a58dbb0     c0000421 00000000 00000000 verifier!VerifierBreakin+0x42
0019d580 7a58dead     0000000f 04f21000 04f55ff8 verifier!VerifierCaptureContextAndReportStop+0xf0
0019d5c4 7a58b945     0000000f 7a581e58 04f21000 verifier!VerifierStopMessage+0x2bd
0019d630 7a58bc2c     04f21000 00000000 04f55ff8 verifier!AVrfpDphReportCorruptedBlock+0x285
0019d6a0 7a58893a     04f21000 04f21af8 00000000 verifier!AVrfpDphCheckPageHeapBlock+0x1bc
0019d6cc 7a588ae0     04f21000 04f55ff8 0019d75c verifier!AVrfpDphFindBusyMemory+0xda
0019d6e8 7a58aad0     04f21000 04f55ff8 04f24750 verifier!AVrfpDphFindBusyMemoryAndRemoveFromBusyList+0x20
0019d704 77bcf796     04f20000 01000002 04f55ff8 verifier!AVrfDebugPageHeapFree+0x90
0019d76c 77b33be6     04f55ff8 1dcee6af 00000000 ntdll!RtlDebugFreeHeap+0x3e
0019d8c8 77b7778d     00000000 04f55ff8 04f55ff8 ntdll!RtlpFreeHeap+0xd6
0019d924 77b33ab6     00000000 00000000 00000000 ntdll!RtlpFreeHeapInternal+0x783
0019d940 7a5fdcc2     04f20000 00000000 04f55ff8 ntdll!RtlFreeHeap+0x46
0019d954 7a0569ad     04f55ff8 10000020 04f55ff8 MSVCR110!free+0x1a
WARNING: Stack unwind information not available. Following frames may be wrong.
0019d96c 7a140df1     1000001e 04f55ff8 7a261920 igCore19d!AF_memm_alloc+0x7ed
0019f5e4 7a141b84     0019fb30 1000001e 0db0afe8 igCore19d!IG_mpi_page_set+0xe50a1
0019f618 7a13f2b2     0019fb30 1000001e 0db0afe8 igCore19d!IG_mpi_page_set+0xe5e34
0019faa8 7a0310d9     0019fb30 0db0afe8 00000001 igCore19d!IG_mpi_page_set+0xe3562
0019fae0 7a070557     00000000 0db0afe8 0019fb30 igCore19d!IG_image_savelist_get+0xb29
0019fd5c 7a06feb9     00000000 05514f88 00000001 igCore19d!IG_mpi_page_set+0x14807
0019fd7c 7a005777     00000000 05514f88 00000001 igCore19d!IG_mpi_page_set+0x14169
```

```
0019fd9c 00498a3a     05514f88 0019fe0c 004801a4 igCore19d!IG_load_file+0x47
0019fe14 00498e36     05514f88 0019fe8c 004801a4 Fuzzme!fuzzme+0x4a
0019fee4 004daa53     00000005 05454f28 0545df20 Fuzzme!main+0x376
0019ff04 004da8a7     35cbeac8 004801a4 004801a4 Fuzzme!invoke_main+0x33
0019ff60 004da73d     0019ff70 004daad8 0019ff80 Fuzzme!__scrt_common_main_seh+0x157
0019ff68 004daad8     0019ff80 7628fa29 002ff000 Fuzzme!__scrt_common_main+0xd
0019ff70 7628fa29     002ff000 7628fa10 0019ffdc Fuzzme!mainCRTStartup+0x8
0019ff80 77b57c7e     002ff000 1dcec1bb 00000000 KERNEL32!BaseThreadInitThunk+0x19
0019ffdc 77b57c4e     ffffffff 77b788ce 00000000 ntdll!__RtlUserThreadStart+0x2f
0019ffec 00000000     004801a4 002ff000 00000000 ntdll!_RtlUserThreadStart+0x1b


STACK_COMMAND:  ~0s ; .cxr ; kb

SYMBOL_NAME:  verifier!VerifierBreakin+42

MODULE_NAME: verifier

IMAGE_NAME:  verifier.dll

FAILURE_BUCKET_ID:  BREAKPOINT_AVRF_80000003_verifier.dll!VerifierBreakin

OS_VERSION:  10.0.19041.1

BUILDLAB_STR:  vb_release

OSPLATFORM_TYPE:  x86

OSNAME:  Windows 10

IMAGE_VERSION:  10.0.19041.1

FAILURE_ID_HASH:  {59a738c4-b581-efeb-feb5-548af1fa6817}

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

Timeline

2021-03-24 - Vendor Disclosure

2021-05-31 - Public Release

CREDIT

Discovered by Emmanuel Tacheau of Cisco Talos.

VULNERABILITY REPORTS

PREVIOUS REPORT

TALOS-2021-1275

NEXT REPORT

TALOS-2021-1296