Talos Vulnerability Report

TALOS-2022-1649

Callback technologies CBFS Filter handle_ioctl_0x830a0_systembuffer null pointer dereference vulnerability

NOVEMBER 22, 2022

CVE NUMBER

CVE-2022-43590

SUMMARY

A null pointer dereference vulnerability exists in the handle_ioctl_0x830a0_systembuffer functionality of Callback technologies CBFS Filter 20.0.8317. A specially-crafted I/O request packet (IRP) can lead to denial of service. An attacker can issue an ioctl to trigger this vulnerability.

CONFIRMED VULNERABLE VERSIONS

The versions below were either tested or verified to be vulnerable by Talos or confirmed to be vulnerable by the vendor.

Callback technologies CBFS Filter 20.0.8317

PRODUCT URLS

CBFS Filter - https://www.callback.com/cbfsfilter/

CVSSV3 SCORE

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

CWE

CWE-476 - NULL Pointer Dereference

DETAILS

A windows device driver is almost like a kernel DLL that, once loaded, provides additional features. In order to communicate with these device drivers, Windows has a major component named Windows I/O Manager. The Windows IO Manager is responsible for the interface between user applications and device drivers. It implements I/O Request Packets (IRP) to enable the communication with the devices drivers, answering to all I/O requests. For more information see the Microsoft website.

The driver is responsible for creating a device interface with different functions to answer to specific codes, named major code function. If the designer wants to implement customized functions into a driver, there is one major function code named `IRP_MJ_DEVICE_CONTROL`. By handling such major code function, device drivers will support specific I/O Control Code (IOCTL) through a dispatch routine.

The Windows I/O Manager provides three different methods to enable the shared memory: Buffered I/O, Direct I/O, Neither I/O.
Without getting into the details of the IO Manager mechanisms, the method Buffered I/O is often the easiest one for handling memory user buffers from a device perspective.
The I/O Manager is providing all features to enable device drivers sharing buffers between userspace and kernelspace. It will be responsible for copying data back and forth.

Let’s see some examples of routines (which you should not copy as is) that explain how things work.
When creating a driver, you’ll have several functions to implement, and you’ll find some dispatcher routines to handle different IRP as follows:

```
extern "C"
NTSTATUS DriverEntry(_In_ PDRIVER_OBJECT pDriverObject, _In_ PUNICODE_STRING RegistryPath)
{
 [...]
        pDriverObject->DriverUnload = DriverUnload;
        pDriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] = DriverIOctl;
        pDriverObject->MajorFunction[IRP_MJ_CREATE] = DriverCreate;
        pDriverObject->MajorFunction[IRP_MJ_CLOSE] = DriverClose;
[...]
}
```

The DriverEntry is the function `main` for a driver. This is the place where initializations start.

We can see for example the `pDriverObject` which is a `PDRIVER_OBJECT` object given by the system to associate different routines, to be called against specific codes, into the `Majorfunction` table `IRP_MJ_DEVICE_CONTROL` for `DriverIOctl` etc.

Then later inside the driver you’ll see the implementation of the `DriverIOctl` routine responsible for handling the IOCTL code. It can be something like below:

```
NTSTATUS DriverIOctl(PDEVICE_OBJECT pDevObject, PIRP pIrp)
{
   [...]
    auto pIrpSp = IoGetCurrentIrpStackLocation(pIrp);
    switch (pIrpSp->Parameters.DeviceIoControl.IoControlCode)
    {

    case IO_CREATE_EXAMPLE:
            ioctl_inbuffer_data = (ioctl_inbuffer*)pIrp->AssociatedIrp.SystemBuffer;
            auto InputBufferLength = pIrpSp->Parameters.DeviceIoControl.InputBufferLength;
            auto OutputBufferLength = pIrpSp->Parameters.DeviceIoControl.OutputBufferLength;
     [...] some code

        pIrp->IoStatus.Information = 0;
        pIrp->IoStatus.Status = status;

        break;
     }

     pIrp->IoStatus.Information = some value;
     pIrp->IoStatus.Status = status;
     return status;
}
```

First we can see the `pIrp` pointer to an IRP structure (the description would be out of the scope of this document). Keep in mind this pointer will be useful for accessing data.
So here for example we can observe some switch-case implementation depending on the `IoControlCode` IOCTL. When the device driver gets an IRP with code value `IO_CREATE_EXAMPLE`, it performs the operations below the case. To get into the buffer data exchanged between userspace and kernelspace and vice-versa, we'll look into `SystemBuffer` passed as an argument through the `pIrp` pointer.

On the device side, the pointer inside an IRP represents the user buffer, usually a field named `Irp->AssociatedIrp.SystemBuffer`, when the buffered I/O mechanism is chosen. The specification of the method is indicated by the code itself.
On the userspace side, an application would need to gain access to the device driver symbolic link if it exists, then send some ioctl requests as follows:

```
success  = ::DeviceIoControl(
    ghDevice,
    IO_CREATE_EXAMPLE,                  // control code
    &gpIoctl,                           // input buffer
    sizeof(struct ioctl_inbuffer),      // input buffer length
    &gpIoctl,                           // output buffer
    sizeof(struct ioctl_inbuffer),      // output buffer length
    &returned,
    nullptr
);
```

Such a call will result in an IRP with a major code `IRP_MJ_DEVICE_CONTROL` and a control code to `IO_CREATE_EXAMPLE`. The buffer passed from userspace here as input `gpIoctl`, and output will be accessible from the device driver in the kernelspace via `pIrp->AssociatedIrp.SystemBuffer`. The lengths specified on the `DeviceIoControl` parameters will be used to build the IRP, and the device would be able to get them into the `InputBufferLength` and the `OutputBufferLength` respectively.

Now below we'll see one examples of incorrect checking of a null value, which can lead to different behaviors and more frequently to a local denial of service and blue screen of death (BSOD) through the usage of the device driver `cbfilter20`.

While investigating into the dump analysis we can see the following :

```
CONTEXT:  ffffd780568f6540 -- (.cxr 0xffffd780568f6540)
rax=0000000000000000 rbx=0000000000000000 rcx=0000000000000016
rdx=0000000000000000 rsi=0000000000000000 rdi=a2e64eada2e64ead
rip=fffff8042b49a79c rsp=ffffd780568f6f40 rbp=ffffd780568f70a0
 r8=0000000000000000  r9=0000000000000000 r10=0000000000000000
r11=0000000000000000 r12=0000000000000000 r13=0000000000000001
r14=ffffb7011ffe4e30 r15=ffffb7011c6ec1c0
iopl=0         nv up ei ng nz na po nc
cs=0010  ss=0018  ds=002b  es=002b  fs=0053  gs=002b             efl=00050286
nt!ExFreeHeapPool+0x76c:
fffff804`2b49a79c 488b18          mov     rbx,qword ptr [rax] ds:002b:00000000`00000000=????????????????
```

which is the consequence of passing a null pointer to `nt!ExFreePool` we can observe if we put some breakpoint just before the call.

The handler for the ioctl code 0x830a0 is the following function named `handle_ioctl_0x830A0` with the following pseudo-code

```
LINE1   MACRO_STATUS __fastcall handle_ioctl_0x830A0(struct _DEVICE_OBJECT *a1, PIRP a2, unsigned int a3)
LINE2   {
LINE3     unsigned __int64 l_InputBufferLength; // r9
LINE4     systembuffer_830a0 *SystemBuffer; // rdx
LINE5     MACRO_STATUS result; // rax
LINE6
LINE7     l_InputBufferLength = a2->Tail.Overlay.CurrentStackLocation->Parameters.DeviceIoControl.InputBufferLength;
LINE8     if ( (unsigned int)l_InputBufferLength < 0x38 )
LINE9       return STATUS_INVALID_PARAMETER;
LINE10    SystemBuffer = (systembuffer_830a0 *)a2->AssociatedIrp.SystemBuffer;
LINE11    if ( l_InputBufferLength < (unsigned __int64)(unsigned __int16)SystemBuffer->input_buffer_size + 0x38 )
LINE12      return STATUS_INVALID_PARAMETER;
LINE13    result = handle_ioctl_0x830a0_systembuffer(a3, SystemBuffer);
LINE14    a2->IoStatus.Information = 0i64;
LINE15    return result;
LINE16  }
```

Some checks are performed at LINE8 and LINE11 respectively checking minimum length and double-checking value added from the `SystemBuffer->input_buffer_size`.

Then a call is made to `handle_ioctl_0x830a0_systembuffer` LINE13 passing the `SystemBuffer` as a second argument. Investigating the following pseudo-code for `handle_ioctl_0x830a0_systembuffer`:

```
LINE17   MACRO_STATUS __fastcall handle_ioctl_0x830a0_systembuffer(unsigned int a1, systembuffer_830a0 *systemBuffer)
LINE18   {
        [...]
LINE41
LINE42     p_ListEntry = &ListEntry;
LINE43     ListEntry = (PSLIST_ENTRY)&ListEntry;
LINE44     SourceString.Buffer = (wchar_t *)((char *)systemBuffer + (unsigned __int16)systemBuffer->offset);
LINE45     SourceString.MaximumLength = systemBuffer->input_buffer_size;
LINE46     SourceString.Length = SourceString.MaximumLength;
LINE47     lunicode_buffer = 0i64;
LINE48     l_result = copy_string_into_dest_with_tag(&lunicode_buffer, &SourceString, gTag, 0);
LINE49     if ( !(_DWORD)l_result )
LINE50     {
LINE51       v5 = 0i64;
LINE52       if ( !kind_validate_4bytes(&lunicode_buffer) )
LINE53       {
LINE54         l_index = 0i64;
LINE55         if ( (lunicode_buffer.Length & 0xFFFE) != 0 )
LINE56         {
LINE57           l_buffer = lunicode_buffer.Buffer;
LINE58           do
LINE59           {
LINE60             l_buffer[v5] = l_buffer[l_index];
LINE61             l_buffer = lunicode_buffer.Buffer;
LINE62             if ( !(_DWORD)l_index
LINE63               || lunicode_buffer.Buffer[(unsigned int)(l_index - 1)] != '\\'
LINE64               || lunicode_buffer.Buffer[l_index] != '\\' )
LINE65             {
LINE66               v5 = (unsigned int)(v5 + 1);
LINE67             }
LINE68             l_index = (unsigned int)(l_index + 1);
LINE69           }
LINE70           while ( (unsigned int)l_index < lunicode_buffer.Length >> 1 );
LINE71         }
LINE72         lunicode_buffer.Length = 2 * v5;
LINE73         sort_of_split_string(&lunicode_buffer);
LINE74       }
LINE75       v8 = sub_0_FFFFF80053AEC6A4(systemBuffer->guint_num_of_unit);
LINE76       if ( v8 )
LINE77       {
            [...]
LINE155      }
LINE156      else
LINE157      {
LINE158        ExFreePoolWithTag(lunicode_buffer.Buffer, gTag);
LINE159        return STATUS_INVALID_PARAMETER;
LINE160      }
LINE161    }
LINE162    return l_result;
LINE163  }
```

we can see the responsible call to `ExFreePool` leading to the BSOD is done at `LINE158` as `ExFreePool` is an alias of `ExFreePoolWithTag` as you can see below from windows kernel code build 19043.

```
POOLCODE:FFFFF800505BB140 ; Exported entry 225. ExFreePool
POOLCODE:FFFFF800505BB140 ; Exported entry 226. ExFreePoolWithTag
POOLCODE:FFFFF800505BB140
POOLCODE:FFFFF800505BB140 ; =============== S U B R O U T I N E =======================================
POOLCODE:FFFFF800505BB140
POOLCODE:FFFFF800505BB140
POOLCODE:FFFFF800505BB140 ; void __stdcall ExFreePoolWithTag(PVOID P, ULONG Tag)
POOLCODE:FFFFF800505BB140                 public ExFreePoolWithTag
POOLCODE:FFFFF800505BB140 ExFreePoolWithTag proc near            ; CODE XREF: VrpOriginalKeyNameParameterCleanup+24↑p
POOLCODE:FFFFF800505BB140                                        ; CmQueryLayeredKey+186↑p ...
POOLCODE:FFFFF800505BB140                 sub     rsp, 28h        ; ExFreePool
POOLCODE:FFFFF800505BB144                 call    ExFreeHeapPool
POOLCODE:FFFFF800505BB149                 add     rsp, 28h
POOLCODE:FFFFF800505BB14D                 retn
POOLCODE:FFFFF800505BB14D ; ---------------------------------------------------------------------------
POOLCODE:FFFFF800505BB14E                 db 0CCh
POOLCODE:FFFFF800505BB14E ExFreePoolWithTag endp
```

In order to understand why we do have a null pointer passed as parameter at LINE158, we have to look into the function `copy_string_into_dest_with_tag` which is responsible for building the `lunicode_buffer` variable pointer.

```
LINE164  MACRO_STATUS __fastcall copy_string_into_dest_with_tag(
LINE165          PUNICODE_STRING DestinationString,
LINE166          PCUNICODE_STRING SourceString,
LINE167          ULONG Tag,
LINE168          char a4)
LINE169  {
            [...]
LINE173
LINE174    Length = SourceString->Length + 2;
LINE175    if ( !a4 )
LINE176      Length = SourceString->Length;
LINE177    if ( Length )
LINE178    {
LINE179      l_buffer = (wchar_t *)ExAllocatePoolWithTag(NonPagedPoolNx, Length, Tag);
LINE180      if ( !l_buffer )
LINE181      {
LINE182        l_buffer = (wchar_t *)ExAllocatePoolWithTag(NonPagedPool, Length, Tag);
LINE183        if ( !l_buffer )
LINE184        {
LINE185          DestinationString->Buffer = 0i64;
LINE186          result = 0xC000009Ai64;
LINE187          DestinationString->Length = 0;
LINE188          DestinationString->MaximumLength = Length;
LINE189          return result;
LINE190        }
LINE191      }
LINE192      DestinationString->Buffer = l_buffer;
LINE193      DestinationString->Length = 0;
LINE194      DestinationString->MaximumLength = Length;
LINE195      RtlCopyUnicodeString(DestinationString, SourceString);
LINE196      if ( a4 )
LINE197        DestinationString->Buffer[(unsigned __int64)DestinationString->Length >> 1] = 0;
LINE198    }
LINE199    else
LINE200    {
LINE201      DestinationString->Buffer = 0i64;
LINE202      DestinationString->Length = 0;
LINE203      DestinationString->MaximumLength = 0;
LINE204    }
LINE205    return 0i64;
LINE206  }
```

At LINE177, a test is made with `Length` and if null it will create an null content `DestinationString` at LINE201-LINE203. While `a4` corresponds to the fourth parameter passed as null value, the `Length` is derived directly for the `SourceString->Length` at LINE176, which is in fact computed earlier LINE46 and LINE45, directly derived from our input buffer values `systemBuffer->input_buffer_size` LINE45.

Thus creating some crafted packet can lead to passing a null pointer to `ExFreePoolWithTag` then causing the denial of service through a BSOD.

Crash Information

```
1: kd> !analyze -v
*******************************************************************************
*                                                                             *
*                        Bugcheck Analysis                                    *
*                                                                             *
*******************************************************************************

SYSTEM_SERVICE_EXCEPTION (3b)
An exception happened while executing a system service routine.
Arguments:
Arg1: 00000000c0000005, Exception code that caused the bugcheck
Arg2: fffff8042b49a79c, Address of the instruction which caused the bugcheck
Arg3: ffffd780568f6540, Address of the context record for the exception that caused the bugcheck
Arg4: 0000000000000000, zero.

Debugging Details:
------------------


KEY_VALUES_STRING: 1

    Key  : Analysis.CPU.mSec
    Value: 2468

    Key  : Analysis.DebugAnalysisManager
    Value: Create

    Key  : Analysis.Elapsed.mSec
    Value: 3398

    Key  : Analysis.Init.CPU.mSec
    Value: 88249

    Key  : Analysis.Init.Elapsed.mSec
    Value: 14709296

    Key  : Analysis.Memory.CommitPeak.Mb
    Value: 131

    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


BUGCHECK_CODE:  3b

BUGCHECK_P1: c0000005

BUGCHECK_P2: fffff8042b49a79c

BUGCHECK_P3: ffffd780568f6540

BUGCHECK_P4: 0

CONTEXT:  ffffd780568f6540 -- (.cxr 0xffffd780568f6540)
rax=0000000000000000 rbx=0000000000000000 rcx=0000000000000016
rdx=0000000000000000 rsi=0000000000000000 rdi=a2e64eada2e64ead
rip=fffff8042b49a79c rsp=ffffd780568f6f40 rbp=ffffd780568f70a0
 r8=0000000000000000  r9=0000000000000000 r10=0000000000000000
r11=0000000000000000 r12=0000000000000000 r13=0000000000000001
r14=ffffb7011ffe4e30 r15=ffffb7011c6ec1c0
iopl=0         nv up ei ng nz na po nc
cs=0010  ss=0018  ds=002b  es=002b  fs=0053  gs=002b             efl=00050286
nt!ExFreeHeapPool+0x76c:
fffff804`2b49a79c 488b18          mov     rbx,qword ptr [rax] ds:002b:00000000`00000000=????????????????
Resetting default scope

PROCESS_NAME:  830a0.exe

STACK_TEXT:
ffffd780`568f6f40 fffff804`2bbc2149     : ffffb701`1c6eb030 fffff804`00000000 00000000`00000000 00000000`00040244 : nt!ExFreeHeapPool+0x76c
ffffd780`568f7020 fffff804`2eff61b9     : ffffb701`1c693cd0 ffffd780`568f70a0 ffffb701`1aecfb40 00000000`00000001 : nt!ExFreePool+0x9
ffffd780`568f7050 fffff804`2eff6983     : ffffb701`1c693cd0 ffffb701`1aecfb40 00000000`00000001 00000000`00000000 :
cbfilter20!handle_ioctl_0x830a0_systembuffer+0xf5
ffffd780`568f70d0 fffff804`2eff656c     : 00000000`00000000 00000000`00000000 00000000`00000000 ffffb701`1ffe4e00 :
cbfilter20!handle_ioctl_0x830A0+0x33
ffffd780`568f7100 fffff804`2efbc7a0     : 00000000`000830a0 00000000`00000001 00000000`0000004e 00000000`00000000 :
cbfilter20!ExtraDeviceDispatchRoutine+0xd4
ffffd780`568f7130 fffff804`2efa57f3     : b7011ffe`00000000 ffffb701`1c693cd0 00000000`00000001 ffffb701`1aecfb40 :
cbfilter20!DispatchDeviceControl+0x1fc
ffffd780`568f7160 fffff804`2b4a07d5     : 00000000`0000000e 00000000`00000000 ffffb701`1c693cd0 00000000`00000001 :
cbfilter20!fn_IRP_MJ_DEVICE_CONTROL+0x73
ffffd780`568f71c0 fffff804`2b886a08     : ffffd780`568f7540 ffffb701`1c693cd0 00000000`00000001 ffffb701`2062b080 : nt!IofCallDriver+0x55
ffffd780`568f7200 fffff804`2b8862d5     : 00000000`000830a0 ffffd780`568f7540 00000000`00000005 ffffd780`568f7540 :
nt!IopSynchronousServiceTail+0x1a8
ffffd780`568f72a0 fffff804`2b885cd6     : 00000000`00000000 00000000`00000000 00000000`00000000 00000000`00000000 :
nt!IopXxxControlFile+0x5e5
ffffd780`568f73e0 fffff804`2b619ab5     : 00000000`00000000 00000000`00000000 00000000`00000000 00000000`00000000 :
nt!NtDeviceIoControlFile+0x56
ffffd780`568f7450 00007ff9`95ccce54     : 00007ff9`937db04b 00000000`00000000 00000002`0000000c 00000000`00000101 :
nt!KiSystemServiceCopyEnd+0x25
000000b3`6931f5d8 00007ff9`937db04b     : 00000000`00000000 00000002`0000000c 00000000`00000101 00002602`06faad51 :
ntdll!NtDeviceIoControlFile+0x14
000000b3`6931f5e0 00007ff9`946b5611     : 00000000`000830a0 00000000`00000000 000000b3`6931f670 00007ff9`00000000 :
KERNELBASE!DeviceIoControl+0x6b
000000b3`6931f650 00007ff7`00154cbb     : 00000000`00000000 00007ff7`00160760 000000b3`6931f6e0 00000000`00000000 :
KERNEL32!DeviceIoControlImplementation+0x81
000000b3`6931f6a0 00000000`00000000     : 00007ff7`00160760 000000b3`6931f6e0 00000000`00000000 000000b3`6931f940 : 830a0!SendData+0xeb [..]


SYMBOL_NAME:  nt!ExFreePool+9

IMAGE_NAME:  Pool_Corruption

MODULE_NAME: Pool_Corruption

STACK_COMMAND:  .cxr 0xffffd780568f6540 ; kb

BUCKET_ID_FUNC_OFFSET:  9

FAILURE_BUCKET_ID:  0x3B_c0000005_nt!ExFreePool
```

```
OS_VERSION:  10.0.19041.1

BUILDLAB_STR:  vb_release

OSPLATFORM_TYPE:  x64

OSNAME:  Windows 10

FAILURE_ID_HASH:  {c9913766-80de-cdf5-a1a8-15c856d3f064}

Followup:     Pool_corruption
---------
```

TIMELINE

2022-11-04 - Vendor Disclosure
2022-11-04 - Initial Vendor Contact
2022-11-22 - Public Release

CREDIT

Discovered by Emmanuel Tacheau of Cisco Talos.