Talos Vulnerability Report

TALOS-2022-1647

Callback technologies CBFS Filter handle_ioctl_83150 null pointer dereference vulnerability

NOVEMBER 22, 2022

CVE NUMBER

CVE-2022-43588

SUMMARY

A null pointer dereference vulnerability exists in the handle_ioctl_83150 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 example of sending a correct output buffer length directly without providing the driver some previous context which can lead to different behaviors and more frequently a local denial of service and blue screen of death through the usage of the device driver `cbfilter20`.

After the system has normally booted and the driver is running, sending an IOCTL 0x83150 with a valid buffer input size and some specific data length leads to the following situation

```
  CONTEXT:  ffff9c0ee2b66560 -- (.cxr 0xffff9c0ee2b66560)
    rax=0000000000001000 rbx=0000000000000000 rcx=0000000000000000
    rdx=ffffc408362e0a10 rsi=ffffc4083492a570 rdi=0000000000000000
    rip=fffff80258bc4eb4 rsp=ffff9c0ee2b66f60 rbp=ffff9c0ee2b67150
     r8=ffffc40839fe1a30  r9=0000000000083150 r10=fffff80258ba5780
    r11=0000000000000000 r12=ffffc4083a7740c0 r13=ffffc408362e0940
    r14=ffffc40839fe1a01 r15=0000000000000000
    iopl=0         nv up ei pl nz na pe nc
    cs=0010  ss=0018  ds=002b  es=002b  fs=0053  gs=002b             efl=00050302
    cbfilter20!handle_ioctl_83150+0xd0:
    fffff802`58bc4eb4 488b4308        mov     rax,qword ptr [rbx+8] ds:002b:00000000`00000008=????????????????
    Resetting default scope

    PROCESS_NAME:  python.exe
```

When looking at pseudo code corresponding to the culprit function we can see the following:

```
LINE1   __int64 __fastcall handle_ioctl_83150(_DEVICE_OBJECT *a1, PIRP a2)
LINE2   {
        [...]
LINE71    v3 = 0i64;
LINE72    Object = 0i64;
LINE73    v4 = 0;
LINE74    SectionHandle = 0i64;
LINE75    Handle = 0i64;
LINE76    v48 = 0i64;
LINE77    buffer_0x10000 = 0i64;
LINE78    v50 = 0i64;
LINE79    CurrentStackLocation = a2->Tail.Overlay.CurrentStackLocation;
LINE80    SystemBuffer = (struct_SystemBuffer *)a2->AssociatedIrp.SystemBuffer;
LINE81    if ( CurrentStackLocation->Parameters.DeviceIoControl.InputBufferLength != SystemBuffer->size_len + 0x20i64 )
LINE82    {
LINE83      l_FileObject = 0i64;
LINE84  invalid_param:
LINE85      v14 = STATUS_INVALID_PARAMETER;
LINE86      goto LABEL_68;
LINE87    }
LINE88    FileObject = CurrentStackLocation->FileObject;
LINE89    must_be_1 = SystemBuffer->must_be_1;
LINE90    if ( (must_be_1 & 1) != 0 )
LINE91    {
LINE92      v4 = 1;
LINE93      v11 = 0i64;
LINE94      v12 = 4096i64;
LINE95    }
LINE96    else
LINE97    {
LINE98      if ( (must_be_1 & 2) == 0 )
LINE99      {
LINE100       l_FileObject = (__int64)CurrentStackLocation->FileObject;
LINE101       goto invalid_param;
LINE102     }
LINE103     v11 = 4096i64;
LINE104     v12 = 0i64;
LINE105   }
LINE106   v53 = v12;
LINE107   MaxCount.QuadPart = v11;
LINE108   v59 = v12;
LINE109   v52 = v11;
LINE110   FsContext2 = (__int64)FileObject->FsContext2;
LINE111   v58 = FsContext2;
LINE112   v39 = *(_QWORD *)(FsContext2 + 8);
LINE113   v54 = v39;
LINE114   v14 = ObReferenceObjectByHandle(SystemBuffer->handle, 0, 0i64, 0, &Object, 0i64);
LINE115   v38 = v14;
LINE116   if ( v14 < 0 )
LINE117   {
LINE118     Object = 0i64;
LINE119     v3 = 0i64;
LINE120 LABEL_9:
LINE121     l_FileObject = v39;
LINE122     goto free_buffer;
LINE123   }
LINE124   v15 = *((_QWORD *)Object + 4);
LINE125   v40 = v15;
LINE126   v55 = v15;
LINE127   a2->IoStatus.Information = 0i64;
        [...]
LINE380   a2->IoStatus.Status = v14;
LINE381   return (unsigned int)v14;
LINE382 }
```

The crash is happening at LINE110, while dereferencing `FileObject->FsContext2`. The issue is happening as there is no null check done against `FileObject` and is assumed to be always valid, which is not the case if the IRP packet is sent directly. The `FileObject` is derived directly from `CurrentStackLocation` at LINE88, which is derived itself from the IRP packet at LINE79. A specially-crafted I/O request packet bypassing the checks done at LINE80 & LINE90 will lead to denial of service immediately.

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: fffff80258bc4eb4, Address of the instruction which caused the bugcheck
Arg3: ffff9c0ee2b66560, 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: 2608

    Key  : Analysis.DebugAnalysisManager
    Value: Create

    Key  : Analysis.Elapsed.mSec
    Value: 3502

    Key  : Analysis.Init.CPU.mSec
    Value: 32186

    Key  : Analysis.Init.Elapsed.mSec
    Value: 670833

    Key  : Analysis.Memory.CommitPeak.Mb
    Value: 96

    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: fffff80258bc4eb4

BUGCHECK_P3: ffff9c0ee2b66560

BUGCHECK_P4: 0

CONTEXT:  ffff9c0ee2b66560 -- (.cxr 0xffff9c0ee2b66560)
rax=0000000000001000 rbx=0000000000000000 rcx=0000000000000000
rdx=ffffc408362e0a10 rsi=ffffc4083492a570 rdi=0000000000000000
rip=fffff80258bc4eb4 rsp=ffff9c0ee2b66f60 rbp=ffff9c0ee2b67150
 r8=ffffc40839fe1a30  r9=0000000000083150 r10=fffff80258ba5780
r11=0000000000000000 r12=ffffc4083a7740c0 r13=ffffc408362e0940
r14=ffffc40839fe1a01 r15=0000000000000000
iopl=0         nv up ei pl nz na pe nc
cs=0010  ss=0018  ds=002b  es=002b  fs=0053  gs=002b             efl=00050302
cbfilter20!handle_ioctl_83150+0xd0:
fffff802`58bc4eb4 488b4308        mov     rax,qword ptr [rbx+8] ds:002b:00000000`00000008=????????????????
Resetting default scope

PROCESS_NAME:  python.exe

STACK_TEXT:
ffff9c0e`e2b66f60 fffff802`58bbc878     : ffffc408`3492a570 ffffc408`362e0940 ffffeb80`01414701 00000000`00000000 :
cbfilter20!handle_ioctl_83150+0xd0
ffff9c0e`e2b67130 fffff802`58ba57f3     : c40839fe`00000000 ffffc408`362e0940 00000000`00000001 ffffc408`3492a570 :
cbfilter20!DispatchDeviceControl+0x2d4
ffff9c0e`e2b67160 fffff802`5509b7d5     : 00000000`0000000e 00000000`00000000 ffffc408`362e0940 00000000`00000001 :
cbfilter20!fn_IRP_MJ_DEVICE_CONTROL+0x73
ffff9c0e`e2b671c0 fffff802`55481a08     : ffff9c0e`e2b67540 ffffc408`362e0940 00000000`00000001 ffffc408`3ad7e0c0 : nt!IofCallDriver+0x55
ffff9c0e`e2b67200 fffff802`554812d5     : 00000000`00083150 ffff9c0e`e2b67540 00000000`00000005 ffff9c0e`e2b67540 :
nt!IopSynchronousServiceTail+0x1a8
ffff9c0e`e2b672a0 fffff802`55480cd6     : 00007ff8`86a68e80 00000000`00000000 00000000`00000000 00000000`00000000 :
nt!IopXxxControlFile+0x5e5
ffff9c0e`e2b673e0 fffff802`55214ab5     : 00000000`00000000 00000000`00000000 00000000`00000000 000000e2`7d7ecb18 :
nt!NtDeviceIoControlFile+0x56
ffff9c0e`e2b67450 00007ff8`c726ce54     : 00007ff8`c4aab04b ffffffff`ff9c9830 00000000`00000000 000000e2`7d7eeb48 :
nt!KiSystemServiceCopyEnd+0x25
000000e2`7d7eea78 00007ff8`c4aab04b     : ffffffff`ff9c9830 00000000`00000000 000000e2`7d7eeb48 000000e2`7d7eec40 :
ntdll!NtDeviceIoControlFile+0x14
000000e2`7d7eea80 00007ff8`c6a05611     : 00000000`00083150 000000e2`7d7eec40 00000236`fb968d00 00007ff8`86b0694d :
KERNELBASE!DeviceIoControl+0x6b
000000e2`7d7eeaf0 00007ff8`86a4648c     : 00000000`00000022 000000e2`7d7eec40 000000e2`7d7eec40 00000000`00000001 :
KERNEL32!DeviceIoControlImplementation+0x81
000000e2`7d7eeb40 00000000`00000022     : 000000e2`7d7eec40 000000e2`7d7eec40 00000000`00000001 00000000`00000000 :
win32file!PyInit_win32file+0xf98c
000000e2`7d7eeb48 000000e2`7d7eec40     : 000000e2`7d7eec40 00000000`00000001 00000000`00000000 000000e2`00000000 : 0x22
000000e2`7d7eeb50 000000e2`7d7eec40     : 00000000`00000001 00000000`00000000 000000e2`00000000 000000e2`7d7eeba0 : 0x000000e2`7d7eec40
000000e2`7d7eeb58 00000000`00000001     : 00000000`00000000 000000e2`00000000 000000e2`7d7eeba0 00000000`00000000 : 0x000000e2`7d7eec40
000000e2`7d7eeb60 00000000`00000000     : 000000e2`00000000 000000e2`7d7eeba0 00000000`00000000 000000e2`7d7eeba8 : 0x1


SYMBOL_NAME:  cbfilter20!handle_ioctl_83150+d0

MODULE_NAME: cbfilter20

IMAGE_NAME:  cbfilter20.sys

STACK_COMMAND:  .cxr 0xffff9c0ee2b66560 ; kb

BUCKET_ID_FUNC_OFFSET:  d0

FAILURE_BUCKET_ID:  0x3B_c0000005_cbfilter20!handle_ioctl_83150

OS_VERSION:  10.0.19041.1
```

```
BUILDLAB_STR:  vb_release

OSPLATFORM_TYPE:  x64

OSNAME:  Windows 10

FAILURE_ID_HASH:  {2a1744df-b799-38ae-0bd4-b13f23c537e7}

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

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.

VULNERABILITY REPORTS

PREVIOUS REPORT

TALOS-2022-1648