Talos Vulnerability Report

TALOS-2021-1433

Webroot Secure Anywhere IOCTL GetProcessCommand and B_03 out-of-bounds read vulnerability

MARCH 15, 2022

CVE NUMBER

CVE-2021-40425,CVE-2021-40424

Summary

An out-of-bounds read vulnerability exists in the IOCTL GetProcessCommand and B_03 of Webroot Secure Anywhere 21.4. A specially-crafted executable can lead to denial of service. An attacker can issue an ioctl to trigger this vulnerability.

Tested Versions

Webroot Secure Anywhere 21.4

Product URLs

Secure Anywhere - https://www.webroot.com/us/en/home/products/av

CVSSv3 Score

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

CWE

CWE-125 - Out-of-bounds Read

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 https://docs.microsoft.com/en-us/windows-hardware/drivers/kernel/example-i-o-request—an-overview.

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`. 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 the 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 two examples of out-of-bounds read, 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 `WRCore_x64`.

CVE-2021-40424 - Out-of-bounds GetProcessCommandLine

The `GetProcessCommandLine` IOCTL request could cause an out-of-bounds read in the device driver `WRCore_x64`, as shown below:

```
WRCore_x64+0x190b:
fffff80f`ad60190b 8b4024          mov     eax,dword ptr [rax+24h] ds:ffff8488`e3a0b004=????????
```

When inspecting the memory, we see the out-of-bounds read in non-mapped memory

```
3: kd> dq ffff8488e3a0afe0
ffff8488`e3a0afe0  00000000`00000000 00000000`00000000
ffff8488`e3a0aff0  00000000`00000000 55555555`55555555
ffff8488`e3a0b000  ????????`???????? ????????`????????
ffff8488`e3a0b010  ????????`???????? ????????`????????
ffff8488`e3a0b020  ????????`???????? ????????`????????
ffff8488`e3a0b030  ????????`???????? ????????`????????
ffff8488`e3a0b040  ????????`???????? ????????`????????
ffff8488`e3a0b050  ????????`???????? ????????`????????
```

The call stack is the following, clearly indicating the out-of-bounds happening in `WRCore_x64+0x190b`

```
3: kd> k
 # Child-SP          RetAddr               Call Site
00 ffffe781`793f6c18 fffff802`1946046f     nt!KeBugCheckEx
01 ffffe781`793f6c20 fffff802`192b5500     nt!MiSystemFault+0x18d0bf
02 ffffe781`793f6d20 fffff802`1941b35e     nt!MmAccessFault+0x400
03 ffffe781`793f6ec0 fffff80f`ad60190b     nt!KiPageFault+0x35e
04 ffffe781`793f7050 fffff80f`ad61decf     WRCore_x64+0x190b
05 ffffe781`793f7080 fffff80f`ad61c6d3     WRCore_x64+0x1decf
06 ffffe781`793f7110 fffff802`193830b7     WRCore_x64+0x1c6d3
07 ffffe781`793f7140 fffff802`199d7f1a     nt!IopfCallDriver+0x53
08 ffffe781`793f7180 fffff802`19454db1     nt!IovCallDriver+0x266
09 ffffe781`793f71c0 fffff802`1968b308     nt!IofCallDriver+0x1af5a1
0a ffffe781`793f7200 fffff802`1968abd5     nt!IopSynchronousServiceTail+0x1a8
0b ffffe781`793f72a0 fffff802`1968a5d6     nt!IopXxxControlFile+0x5e5
0c ffffe781`793f73e0 fffff802`1941ebb5     nt!NtDeviceIoControlFile+0x56
0d ffffe781`793f7450 00007ffa`a450ce54     nt!KiSystemServiceCopyEnd+0x25
0e 00000026`34d4fbf8 00000000`00000000     0x00007ffa`a450ce54
```

This corresponds to the following pseudo-code named `get_hprocess_from_webroot_irp`. The function `get_hprocess_from_webroot_irp` has an argument named here `webroot_irp`; we'll see later where it's coming from.

```
LINE1  __int64 __fastcall get_hprocess_from_webroot_irp(webroot_irp *webroot_irp)
LINE2  {
LINE3    if ( webroot_irp->possible_size_of_self <= 32ui64 )
LINE4      j_rtl_failfast_wrappoer((__int64)webroot_irp);
LINE5    return (unsigned int)webroot_irp->getProcessInfo->hprocess;
LINE6  }
```

The out-of-bounds read happens at LINE5 while attempting to read what should be the `hprocess`.

The function `get_hprocess_from_webroot_irp` is called at LINE28 from a function named `GetProcessCommandLine`. The function `GetProcessCommandLine` is an IOCTL handler which is responsible for returning information against some specific process ID. Here below, the pseudo-code corresponds to `GetProcessCommandLine`:

```
LINE8  MACRO_STATUS __fastcall GetProcessCommandLine(IRP *pIrp, _IO_STACK_LOCATION *io_stack_loc)
LINE9  {
                [...]
LINE20
LINE21   if ( io_stack_loc->Parameters.DeviceIoControl.InputBufferLength < 0x18
LINE22     || io_stack_loc->Parameters.DeviceIoControl.OutputBufferLength < 0x18 )
LINE23   {
LINE24     return STATUS_SINGLE_STEP|STATUS_OBJECT_NAME_EXISTS;
LINE25   }
LINE26   build_webroot_irp(&webroot_irp, (getProcessInfo *)pIrp->AssociatedIrp.SystemBuffer);
LINE27   process_list_related_0 = (process_related *)get_process_list_related_0();
LINE28   p_systemBuffer = get_hprocess_from_webroot_irp(&webroot_irp);
LINE29   sub_14000F8EC(process_list_related_0, &_result, p_systemBuffer);
LINE30   nt_status = 0;
LINE31   if ( !_result.some_ptr )
LINE32   {
LINE33     nt_status = STATUS_UNSUCCESSFUL;
LINE34 LABEL_11:
LINE35     sub_14000159C(&_result);
LINE36     sub_14001C190(&webroot_irp);
LINE37     return (unsigned __int64)nt_status;
LINE38   }
LINE39   field_28_of_irp = get_field_28_of_irp(&webroot_irp);
LINE40   size_to_check = (unsigned __int64)*(unsigned __int16 *)sub_14000D5E8(_result.some_ptr) >> 1;
LINE41   v8 = *(_WORD **)sub_140001A8C(&webroot_irp);
LINE42   if ( !v8 || field_28_of_irp < size_to_check )
LINE43   {
LINE44     // set some length
LINE45     sub_14000199C(&webroot_irp, size_to_check);
LINE46     pIrp->IoStatus.Information = 24i64;
LINE47     goto LABEL_11;
LINE48   }
LINE49   v9 = sub_14000D5E8(_result.some_ptr);
LINE50   v10 = &word_140028480;
LINE51   if ( *(_QWORD *)(v9 + 8) )
LINE52     v10 = *(const wchar_t **)(v9 + 8);
LINE53   _mm_lfence();
LINE54   ProbeForWrite(v8, size_to_check, 1u);
LINE55   perform_write_into_dest(v8, field_28_of_irp, (__int64)v10, size_to_check);
LINE56   _mm_lfence();
LINE57   sub_140001810(&webroot_irp, size_to_check);
LINE58   pIrp->IoStatus.Information = 24i64;
LINE59   sub_14000159C(&_result);
LINE60   sub_14001C190(&webroot_irp);
LINE61   return 0i64;
LINE62 }
```

The function `build_webroot_irp`, called at LINE26, is responsible for building the `webroot_irp` object, which will be used as an argument in the culprit function.

```
LINE63 webroot_irp *__fastcall build_webroot_irp(webroot_irp *webroot_irp, getProcessInfo *getProcessInfoData)
LINE64 {
LINE65   webroot_irp->table_function = &off_14002AAA0;
LINE66   webroot_irp->byte8 = 0;
LINE67   webroot_irp->qword10 = 0i64;
LINE68   webroot_irp->PERESOURCE = 0i64;
LINE69   webroot_irp->qword20 = 0i64;
LINE70   webroot_irp->possible_size_of_self = 56i64;
LINE71   webroot_irp->getProcessInfo = getProcessInfoData;
LINE72   if ( !getProcessInfoData )
LINE73     j_rtl_failfast_wrappoer((__int64)webroot_irp);
LINE74   webroot_irp->table_function = &off_14002AAA0;
LINE75   return webroot_irp;
LINE76 }
```

When looking at LINE26, we can see our buffer `AssociatedIrp.SystemBuffer` is corresponding to the parameter named here `getProcessInfoData`. The function `build_webroot_irp` is associating the buffer `getProcessInfoData` to the variable `webroot_irp->getProcessInfo` as we can see at LINE71.

When reversing a bit more the functions around this spot, we can deduce the expected structure `getProcessInfo` format should correspond to something like:

```
00000000 getProcessInfo  struc ; (sizeof=0x38, mappedto_561)
00000000 qword_1         dq ?
00000008 qword_2         dq ?
00000010 qword_3         dq ?
00000018 qword_4         dq ?
00000020 dword_1         dd ?
00000024 hprocess        dd ?
00000028 length          dq ?
00000030 output_buffer   dq ?
00000038 getProcessInfo  ends
```

The `getProcessInfoData` corresponds to the user buffer we can control from userspace as input buffer.
We can see here the `hprocess` at offset 0x24. And this is the `hprocess` field from the structure `getProcessInfo` which corresponds to the read value in the culprit function `get_hprocess_from_webroot_irp` without precaution, causing the out-of-bounds read. The issue is the checks done in LINE21 and LINE22 are not preventing the out-of-bounds read happening, as they check only for valid input up to input size of 0x18 bytes.

Crash Information

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

PAGE_FAULT_IN_NONPAGED_AREA (50)
Invalid system memory was referenced.  This cannot be protected by try-except.
Typically the address is just plain bad or it is pointing at freed memory.
Arguments:
Arg1: ffff8488e3a0b004, memory referenced.
Arg2: 0000000000000000, value 0 = read operation, 1 = write operation.
Arg3: fffff80fad60190b, If non-zero, the instruction address which referenced the bad memory
    address.
Arg4: 0000000000000002, (reserved)

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

Unable to load image \??\C:\Program Files\Webroot\Core\WRCore.x64.sys, Win32 error 0n2

KEY_VALUES_STRING: 1

    Key  : Analysis.CPU.mSec
    Value: 3078

    Key  : Analysis.DebugAnalysisManager
    Value: Create

    Key  : Analysis.Elapsed.mSec
    Value: 4492

    Key  : Analysis.Init.CPU.mSec
    Value: 7921

    Key  : Analysis.Init.Elapsed.mSec
    Value: 22374

    Key  : Analysis.Memory.CommitPeak.Mb
    Value: 83

    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


VIRTUAL_MACHINE:  HyperV

BUGCHECK_CODE:  50

BUGCHECK_P1: ffff8488e3a0b004

BUGCHECK_P2: 0

BUGCHECK_P3: fffff80fad60190b

BUGCHECK_P4: 2

READ_ADDRESS:  ffff8488e3a0b004 Special pool

MM_INTERNAL_CODE:  2

IMAGE_NAME:  WRCore.x64.sys

MODULE_NAME: WRCore.x64

FAULTING_MODULE: 0000000000000000

BLACKBOXBSD: 1 (!blackboxbsd)


BLACKBOXNTFS: 1 (!blackboxntfs)


BLACKBOXWINLOGON: 1

PROCESS_NAME:  webroot_ioctl_a01.exe

TRAP_FRAME:  ffffe781793f6ec0 -- (.trap 0xffffe781793f6ec0)
NOTE: The trap frame does not contain all registers.
Some register values may be zeroed or incorrect.
rax=ffff8488e3a0afe0 rbx=0000000000000000 rcx=ffffe781793f70b8
rdx=ffff8488e3a0afe0 rsi=0000000000000000 rdi=0000000000000000
rip=fffff80fad60190b rsp=ffffe781793f7050 rbp=ffff8488df571d80
 r8=ffff8488df571d80  r9=000000000000000e r10=fffff80fad61c554
r11=0000000000000000 r12=0000000000000000 r13=0000000000000000
r14=0000000000000000 r15=0000000000000000
iopl=0         nv up ei pl nz na po nc
WRCore_x64+0x190b:
fffff80f`ad60190b 8b4024          mov     eax,dword ptr [rax+24h] ds:ffff8488`e3a0b004=????????
Resetting default scope

STACK_TEXT:
ffffe781`793f6c18 fffff802`1946046f     : 00000000`00000050 ffff8488`e3a0b004 00000000`00000000 ffffe781`793f6ec0 : nt!KeBugCheckEx
ffffe781`793f6c20 fffff802`192b5500     : 00000000`00001000 00000000`00000000 ffffe781`793f6f40 00000000`00000000 :
nt!MiSystemFault+0x18d0bf
ffffe781`793f6d20 fffff802`1941b35e     : 00000000`00000001 00000000`00001000 00000000`00000000 00000000`00000000 : nt!MmAccessFault+0x400
ffffe781`793f6ec0 fffff80f`ad60190b     : fffff802`1968b308 fffff802`199eaf31 ffff8488`e377d0d8 fffff802`199f3a48 : nt!KiPageFault+0x35e
ffffe781`793f7050 fffff80f`ad61decf     : ffff8488`e0205650 ffff8488`e2afe9a0 ffff8488`e377d0d8 fffff802`199e311a : WRCore_x64+0x190b
ffffe781`793f7080 fffff80f`ad61c6d3     : 00000000`00000000 ffff8488`df5196a0 ffff8488`df5195d0 ffff8488`e0205650 : WRCore_x64+0x1decf
ffffe781`793f7110 fffff802`193830b7     : ffff8488`df5195d0 00000000`00000000 ffff8488`00000001 ffff8488`df571d80 : WRCore_x64+0x1c6d3
ffffe781`793f7140 fffff802`199d7f1a     : ffff8488`df5195d0 ffff8488`df571d80 00000000`20206f49 00000000`00000000 : nt!IopfCallDriver+0x53
ffffe781`793f7180 fffff802`19454db1     : ffff8488`df5195d0 00000000`00000002 00000000`00000028 ffff8488`e0205650 : nt!IovCallDriver+0x266
ffffe781`793f71c0 fffff802`1968b308     : ffffe781`793f7540 ffff8488`df5195d0 00000000`00000001 ffffe781`793f7540 :
nt!IofCallDriver+0x1af5a1
ffffe781`793f7200 fffff802`1968abd5     : 00000000`9c412804 ffffe781`793f7540 00000000`00000000 ffffe781`793f7540 :
nt!IopSynchronousServiceTail+0x1a8
ffffe781`793f72a0 fffff802`1968a5d6     : 00000000`00000000 00000000`00000000 00000000`00000000 00000000`00000000 :
nt!IopXxxControlFile+0x5e5
ffffe781`793f73e0 fffff802`1941ebb5     : 00000000`000000e8 00000000`00000000 00000000`77566d4d ffffe781`793f74a8 :
nt!NtDeviceIoControlFile+0x56
```

```
ffffe781`793f7450 00007ffa`a450ce54     : 00000000`00000000 00000000`00000000 00000000`00000000 00000000`00000000 :
nt!KiSystemServiceCopyEnd+0x25
00000026`34d4fbf8 00000000`00000000     : 00000000`00000000 00000000`00000000 00000000`00000000 00000000`00000000 : 0x00007ffa`a450ce54


STACK_COMMAND:  .thread ; .cxr ; kb

FAILURE_BUCKET_ID:  AV_VRF_R_INVALID_IMAGE_WRCore.x64.sys

OS_VERSION:  10.0.19041.1

BUILDLAB_STR:  vb_release

OSPLATFORM_TYPE:  x64

OSNAME:  Windows 10

FAILURE_ID_HASH:  {4487bf4b-fcaa-ab42-3f1b-e48f350aad47}

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

CVE-2021-40425 - Out-of-bounds IOCTL_B03

Another IOCTL request with specific invalid data causes a similar issue in the device driver `WRCore_x64`, as shown below:

```
WRCore_x64+0x192b:
fffff803`8a87192b 8b4020          mov     eax,dword ptr [rax+20h] ds:ffffe48f`d8063000=????????
```

Investigating the call stack should lead us to the culprit routine :

```
3: kd> k
 # Child-SP          RetAddr               Call Site
00 ffff810a`4aae4478 fffff801`03d24b12     nt!DbgBreakPointWithStatus
01 ffff810a`4aae4480 fffff801`03d240f6     nt!KiBugCheckDebugBreak+0x12
02 ffff810a`4aae44e0 fffff801`03c092b7     nt!KeBugCheck2+0x946
03 ffff810a`4aae4bf0 fffff801`03c5c46f     nt!KeBugCheckEx+0x107
04 ffff810a`4aae4c30 fffff801`03ab1500     nt!MiSystemFault+0x18d0bf
05 ffff810a`4aae4d30 fffff801`03c1735e     nt!MmAccessFault+0x400
06 ffff810a`4aae4ed0 fffff803`8a87192b     nt!KiPageFault+0x35e
07 ffff810a`4aae5060 fffff803`8a88e079     WRCore_x64+0x192b
08 ffff810a`4aae5090 fffff803`8a88c760     WRCore_x64+0x1e079
09 ffff810a`4aae5110 fffff801`03b7f0b7     WRCore_x64+0x1c760
0a ffff810a`4aae5140 fffff801`041d3f1a     nt!IopfCallDriver+0x53
0b ffff810a`4aae5180 fffff801`03c50db1     nt!IovCallDriver+0x266
0c ffff810a`4aae51c0 fffff801`03e87308     nt!IofCallDriver+0x1af5a1
0d ffff810a`4aae5200 fffff801`03e86bd5     nt!IopSynchronousServiceTail+0x1a8
0e ffff810a`4aae52a0 fffff801`03e865d6     nt!IopXxxControlFile+0x5e5
0f ffff810a`4aae53e0 fffff801`03c1abb5     nt!NtDeviceIoControlFile+0x56
10 ffff810a`4aae5450 00007ff9`b24ece54     nt!KiSystemServiceCopyEnd+0x25
11 00000045`4e31fb28 00007ff9`aff5b07b     ntdll!NtDeviceIoControlFile+0x14
12 00000045`4e31fb30 00007ff9`b09a5611     KERNELBASE!DeviceIoControl+0x6b
13 00000045`4e31fba0 00007ff7`e9271142     KERNEL32!DeviceIoControlImplementation+0x81
14 00000045`4e31fbf0 00000000`00000000     poc+0x1142
```

So it turns out the `WRCore_x64+0x192b` looks to be where the out-of-bounds is occurring. This corresponds to the following pseudo-code:

```
LINE77  __int64 __fastcall sub_14000191C(webroot_irp *iob)
LINE78  {
LINE79    if ( iob->self_size <= 0x20ui64 )
LINE80      j_rtl_failfast_wrappoer((__int64)iob);
LINE81    return (unsigned int)iob->SystemBuffer->field_20;
LINE82  }
```

We can see at LINE81 a variable here named `field_20` from `iob->SystemBuffer`.
The subroutine `sub_14000191C` is called directly from the ioctl dispatcher routine `ioctl_2` associated with following pseudo code:

```
LINE84  __int64 __fastcall ioctl_2(PIRP pIrp, _IO_STACK_LOCATION *psStackLocation)
LINE85  {
            [...]
LINE93
LINE94    if ( psStackLocation->Parameters.DeviceIoControl.InputBufferLength < 32
LINE95      || psStackLocation->Parameters.DeviceIoControl.OutputBufferLength < 32 )
LINE96    {
LINE97      return 0xC0000004i64;
LINE98    }
LINE99    SystemBuffer = pIrp->AssociatedIrp.SystemBuffer;
LINE100   memset(&iob.qword10, 0, 24);
LINE101   iob.byte8 = 0;
LINE102   iob.self_size = 64i64;
LINE103   iob.SystemBuffer = SystemBuffer;
LINE104   if ( !SystemBuffer )
LINE105     goto LABEL_11;
LINE106   iob.table_function = &off_14002B920;
LINE107   process_list_related_0 = get_process_list_related_0();
LINE108   v5 = sub_14000191C(&iob);
LINE109   sub_14000F8EC(process_list_related_0, &a2, v5);
LINE110   some_ptr = a2.some_ptr;
LINE111   if ( a2.some_ptr )
LINE112   {
LINE113     pIrp->IoStatus.Information = 64i64;
LINE114     v7 = sub_14000D850(some_ptr, &iob);
LINE115   }
LINE116   else
LINE117   {
LINE118     v7 = STATUS_UNSUCCESSFUL;
LINE119   }
LINE120   sub_14000159C(&a2);
LINE121   iob.table_function = &off_14002B920;
LINE122   if ( iob.PERESOURCE )
LINE123   {
LINE124     if ( iob.self_size <= 0x20ui64 )
LINE125 LABEL_11:
LINE126       j_rtl_failfast_wrappoer((__int64)pIrp);
LINE127   }
LINE128   sub_1400076C8(&iob.byte8);
LINE129   return v7;
LINE130 }
```

At LINE108 we can see the call directly to the culprit subroutine using the variable `iob`.
The `iob.SystemBuffer` is directly derived from `SystemBuffer` which is in fact the IOCTL input `SystemBuffer` as indicated at LINE99.

When doing some reverse around function the attended `SystemBuffer` is structured like below:

```
00000000 ioctl_03        struc ; (sizeof=0x40, mappedto_625)
00000000 vftable         dq ?
00000008 field_8         dq ?
00000010 field_10        dq ?
00000018 field_18        dq ?
00000020 hprocess        dd ?                    ; XREF: sub_14000191C+F/r
00000024 field_24        dd ?
00000028 self_size       dq ?
00000030 SystemBuffer    dq ?
00000038 field_38        dq ?
00000040 ioctl_03        ends
```

We can see here the `hprocess` field at offset 0x20, which again checks the size length for input buffer and output buffer done at LINE94 and at LINE95. They are not big enough, as they allow an out-of-bounds read to happen when the function `sub_14000191C` is called.

Crash Information

```
3: kd> !analyze -v
Connected to Windows 10 19041 x64 target at (Thu Dec  2 18:11:33.195 2021 (UTC + 1:00)), ptr64 TRUE
Loading Kernel Symbols
...............................................................
................................................................
...........................................
Loading User Symbols
.............
Loading unloaded module list
.........
*******************************************************************************
*                                                                             *
*                        Bugcheck Analysis                                    *
*                                                                             *
*******************************************************************************

PAGE_FAULT_IN_NONPAGED_AREA (50)
Invalid system memory was referenced.  This cannot be protected by try-except.
Typically the address is just plain bad or it is pointing at freed memory.
Arguments:
Arg1: ffffe48fd8063000, memory referenced.
Arg2: 0000000000000000, value 0 = read operation, 1 = write operation.
Arg3: fffff8038a87192b, If non-zero, the instruction address which referenced the bad memory
    address.
Arg4: 0000000000000002, (reserved)

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

*** WARNING: Unable to verify checksum for poc.exe

KEY_VALUES_STRING: 1

    Key  : Analysis.CPU.mSec
    Value: 7952

    Key  : Analysis.DebugAnalysisManager
    Value: Create

    Key  : Analysis.Elapsed.mSec
    Value: 11787

    Key  : Analysis.Init.CPU.mSec
    Value: 172905

    Key  : Analysis.Init.Elapsed.mSec
    Value: 24533381

    Key  : Analysis.Memory.CommitPeak.Mb
    Value: 92

    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:  50

BUGCHECK_P1: ffffe48fd8063000

BUGCHECK_P2: 0

BUGCHECK_P3: fffff8038a87192b

BUGCHECK_P4: 2

READ_ADDRESS:  ffffe48fd8063000 Special pool

MM_INTERNAL_CODE:  2

IMAGE_NAME:  WRCore.x64.sys

MODULE_NAME: WRCore.x64

FAULTING_MODULE: 0000000000000000

PROCESS_NAME:  poc.exe

TRAP_FRAME:  ffff810a4aae4ed0 -- (.trap 0xffff810a4aae4ed0)
NOTE: The trap frame does not contain all registers.
Some register values may be zeroed or incorrect.
rax=ffffe48fd8062fe0 rbx=0000000000000000 rcx=ffff810a4aae50c8
rdx=ffffe48fd6ab8f70 rsi=0000000000000000 rdi=0000000000000000
rip=fffff8038a87192b rsp=ffff810a4aae5060 rbp=ffff810a4aae5100
 r8=ffff8383152450b2  r9=0000000000000002 r10=fffff80103f14940
r11=0000000000000000 r12=0000000000000000 r13=0000000000000000
r14=0000000000000000 r15=0000000000000000
iopl=0         nv up ei pl nz na pe nc
WRCore_x64+0x192b:
fffff803`8a87192b 8b4020          mov     eax,dword ptr [rax+20h] ds:ffffe48f`d8063000=????????
Resetting default scope

STACK_TEXT:
ffff810a`4aae4478 fffff801`03d24b12     : ffff810a`4aae45e0 fffff801`03b8f200 fffff803`8a870000 00000000`00000000 :
nt!DbgBreakPointWithStatus
ffff810a`4aae4480 fffff801`03d240f6     : fffff803`00000003 ffff810a`4aae45e0 fffff801`03c1e110 ffff810a`4aae4b30 :
nt!KiBugCheckDebugBreak+0x12
ffff810a`4aae44e0 fffff801`03c092b7     : 00000000`00000000 00000000`00000000 ffffe48f`d8063000 ffffe48f`d8063000 : nt!KeBugCheck2+0x946
ffff810a`4aae4bf0 fffff801`03c5c46f     : 00000000`00000050 ffffe48f`d8063000 00000000`00000000 ffff810a`4aae4ed0 : nt!KeBugCheckEx+0x107
ffff810a`4aae4c30 fffff801`03ab1500     : 00000000`00000000 00000000`00000000 ffff810a`4aae4f50 00000000`00000000 :
nt!MiSystemFault+0x18d0bf
ffff810a`4aae4d30 fffff801`03c1735e     : ffff8383`0d000100 fffff801`03a9b422 ffff8383`0d000340 00000000`000000ff : nt!MmAccessFault+0x400
ffff810a`4aae4ed0 fffff803`8a87192b     : ffff8383`15774060 fffff803`8a88f986 00000000`00000000 ffffe48f`00000000 : nt!KiPageFault+0x35e
ffff810a`4aae5060 fffff803`8a88e079     : ffffe48f`d240b0d8 fffff801`041df11a ffff8383`15245e30 fffff801`03e87308 : WRCore_x64+0x192b
ffff810a`4aae5090 fffff803`8a88c760     : 00000000`00000000 ffffe48f`d6ab8ea0 00000000`00000000 ffffe48f`d5c48440 : WRCore_x64+0x1e079
ffff810a`4aae5110 fffff801`03b7f0b7     : ffffe48f`d6ab8ea0 00000000`00000000 ffffe48f`00000001 ffffe48f`d0f16290 : WRCore_x64+0x1c760
ffff810a`4aae5140 fffff801`041d3f1a     : ffffe48f`d6ab8ea0 ffffe48f`d0f16290 00000000`20206f49 00000000`00000000 : nt!IopfCallDriver+0x53
ffff810a`4aae5180 fffff801`03c50db1     : ffffe48f`d6ab8ea0 00000000`00000002 00000000`00000028 ffffe48f`d5c48440 : nt!IovCallDriver+0x266
ffff810a`4aae51c0 fffff801`03e87308     : ffff810a`4aae5540 ffffe48f`d6ab8ea0 00000000`00000001 ffff810a`4aae5540 :
nt!IofCallDriver+0x1af5a1
ffff810a`4aae5200 fffff801`03e86bd5     : 00000000`9c412c0c ffff810a`4aae5540 00000000`00000000 ffff810a`4aae5540 :
nt!IopSynchronousServiceTail+0x1a8
```

```
ffff810a`4aae52a0 fffff801`03e865d6     : 00007ff9`b046d246 00000000`00000000 00000000`00000000 00000000`00000000 :
nt!IopXxxControlFile+0x5e5
ffff810a`4aae53e0 fffff801`03c1abb5     : 00000000`00000000 00000000`00000000 00000000`00000000 00000000`00000000 :
nt!NtDeviceIoControlFile+0x56
ffff810a`4aae5450 00007ff9`b24ece54     : 00007ff9`aff5b07b 00007ff9`b049f4e8 00000002`0000000c 000001c9`c66c0101 :
nt!KiSystemServiceCopyEnd+0x25
00000045`4e31fb28 00007ff9`aff5b07b     : 00007ff9`b049f4e8 00000002`0000000c 000001c9`c66c0101 00000045`4e31fb50 :
ntdll!NtDeviceIoControlFile+0x14
00000045`4e31fb30 00007ff9`b09a5611     : 00000000`9c412c0c 00000000`00000000 00000000`00000000 00007ff9`b03bdd3e :
KERNELBASE!DeviceIoControl+0x6b
00000045`4e31fba0 00007ff7`e9271142     : 00000000`00000000 00000000`00000000 00000000`00000000 00000000`00000000 :
KERNEL32!DeviceIoControlImplementation+0x81
00000045`4e31fbf0 00000000`00000000     : 00000000`00000000 00000000`00000000 00000000`00000000 000001c9`c66d15c0 : poc+0x1142


STACK_COMMAND:  .thread ; .cxr ; kb

FAILURE_BUCKET_ID:  AV_VRF_R_INVALID_IMAGE_WRCore.x64.sys

OS_VERSION:  10.0.19041.1

BUILDLAB_STR:  vb_release

OSPLATFORM_TYPE:  x64

OSNAME:  Windows 10

FAILURE_ID_HASH:  {4487bf4b-fcaa-ab42-3f1b-e48f350aad47}

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

Timeline

2021-12-16 - Vendor disclosure

2022-03-15 - Public Release

CREDIT

Discovered by Emmanuel Tacheau of Cisco Talos.

VULNERABILITY REPORTS

PREVIOUS REPORT
TALOS-2022-1464

NEXT REPORT
TALOS-2022-1512