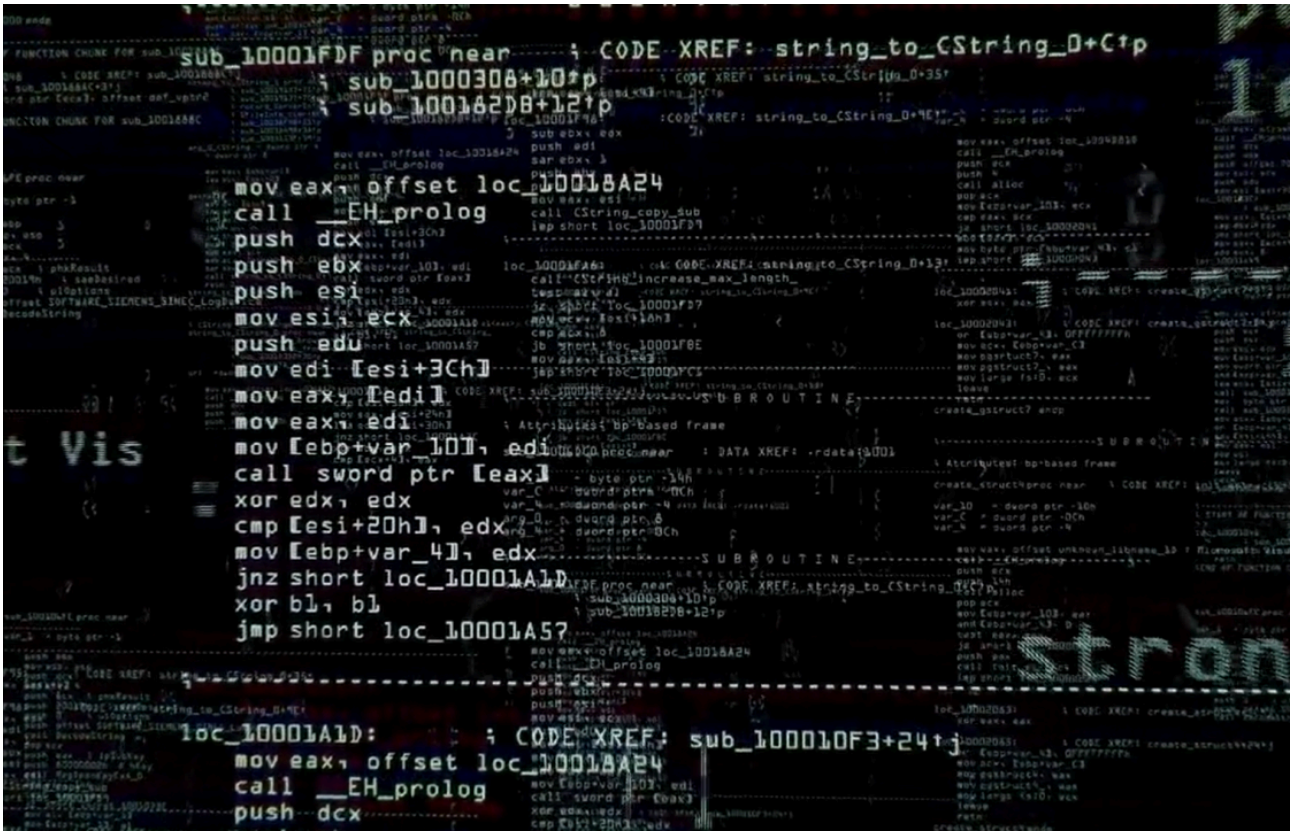
Stuxnet drivers: detailed analysis

By Artem Baranov

Archived: 2026-04-05 13:57:01 UTC



📌 Chapters:

- Introduction
- Some basic terms
- Howto
- Exploring Win11 disk subsystem
- Set up a secure environment
- Overview of the driver
- Patching kernel data
- Securing disk I/O operations
- Securing file I/O operations
- Tracing kernel mode code
- About PPL'ed processes

📝 Introduction

GMER is a well-known powerful anti-rootkit tool, which has been used for years by Windows IT pros to detect the presence of rootkits in the system. A rootkit is a kind of malicious software intended to hide the components and artifacts of malware. Historically, rootkits can be divided into two types: user mode (Ring 3) and kernel mode (Ring 0). Nowadays, there are also malicious implants designed to work at the hypervisor (Ring -1) and SMM (Ring -2) privilege levels. We're gonna focus on the most common type, the kernel mode rootkit, and will simply refer to it as a rootkit.

Rootkits were popular in the Windows x86 era, when there were no restrictions on intercepting anything in privileged kernel mode. Typically, rootkits use three types of techniques: hooking, patching, DKOM. We won't delve into them in detail, but it's worth noting that the first one is used to replace a pointer to the necessary function in the function table, the second involves inline code patching and the third is used to modify members of kernel objects such as KTHREAD or EPROCESS.

The authors of other well-known anti-rootkits have dropped their support due to the growing dominance of the x64 platform and the emergence of new Windows versions. The restrictions imposed on kernel mode code on this platform have affected not only rootkits but also anti-rootkits. Rootkits have lost the ability to gain control over the system making anti-rootkit checks useless. Nevertheless, some rootkits were able to bypass the new security perimeter by rebooting the system with the test signing bootloader option. One of them was Necurs, along with several other bootkits.

Unlike other anti-rootkits, GMER has an x64 version of the driver, although doesn't support modern versions of Windows. It has an impressive arsenal of clever tricks for detecting the presence of rootkits and unhooking them, which can be useful nowadays. Since malware aims to detect the launch of GMER, it drops the driver to disk with a random name and deletes it once it's loaded. This way, the malware can't block its loading. Within the tool, the driver, which is located in the resource section, is packed twice: the tool itself is packed with UPX and its unpacked version stores the compressed driver. The tool's executable is also landed on the disk with a random name.

📖 Some basic terms

Anti-rootkit - a standalone tool/utility or component within a security product that is designed to deeply inspect Windows environment at both user and kernel mode levels to detect system anomalies.

Direct Kernel Object Manipulation (DKOM) - a rootkit technique that means modification of Windows kernel objects through direct access to them without any API.

Disk driver - a Windows driver called disk.sys that is responsible of processing disk I/O operations usually coming from the partition manager (PartMgr.sys), volume manager (Volmgrx.sys) or any other clients via \PhysicalDriveX objects. In fact, the classpnp.sys driver dispatches all disk.sys driver requests.

Disk port driver - while disk.sys implements a high-level logic of communication with various disk types connected to different interfaces, the disk port driver is designed to communicate with a specific disk device. There are several common disk port drivers such as atapi.sys, scsiport.sys, ataport.sys, storport.sys.

🔬 Howto

To extract the driver, the dropper first should be unpacked. The driver inside the resource section of the dropper is compressed with zlib. In order to decompress it, I personally simply used the built-in VirusTotal decompressor. In the Relations tab, you can find a link to the report with the decompressed driver.

C:\test>upx -d -o C:\test\gmer\gmer_unpacked.exe C:\test\gmer\gmer.exe

To make the analysis process faster, I ran GMER on my Win7 SP1 x64 VM and took a kernel memory dump. Since the dump contains the driver with already initialized variables and decrypted text strings, we can determine the purpose of each pointer in the disassembled version of the driver. Next, we need to rebase the driver loaded into IDA so that the offsets of both drivers are identical.

```
kd> lmDvmafaoyaoc
Browse full module list
start             end                 module name
fffff880`03220000 fffff880`03230000   afaoyaoc   (no symbols)
    Loaded symbol image file: afaoyaoc.sys
    Image path: \??\C:\Users\Artem\AppData\Local\Temp\afaoyaoc.sys
    Image name: afaoyaoc.sys
    Browse all global symbols  functions  data
    Timestamp:        Wed Mar  9 10:28:57 2016 (56DFD0B9)
    CheckSum:         0000E358
    ImageSize:        00010000
    Translations:     0000.04b0 0000.04e4 0409.04b0 0409.04e4
    Information from resource tables:
```

Edit->Segments->Rebase program...

```
.text:FFFFF88003221008          ; __int64 __fastcall sub_FFFFF88003221008(PRKPROCESS Process)
.text:FFFFF88003221008          sub_FFFFF88003221008 proc near          ; CODE XREF: sub_FFFFF8800322132C+650↓p
.text:FFFFF88003221008                                                  ; sub_FFFFF8800322132C+6DA↓p ...
.text:FFFFF88003221008 40 53                    push    rbx
.text:FFFFF8800322100A 48 83 EC+                sub     rsp, 20h
.text:FFFFF8800322100A 20
.text:FFFFF8800322100E 48 8B D9                 mov     rbx, rcx
.text:FFFFF88003221011
.text:FFFFF88003221011          loc_FFFFF88003221011:                   ; DATA XREF: .rdata:FFFFF8800322B770↓o
.text:FFFFF88003221011 FF 15 91+                call    cs:MmIsAddressValid
```

🔐 Set up a secure environment

Any anti-rootkit shouldn't trust the environment in which it works. Members of kernel mode objects, pointers in dispatch tables, the integrity of the WinNT kernel executable, and drivers - all this stuff may already be compromised before an anti-rootkit is launched.

A secure environment means a set of artifacts such as kernel object pointers, their values, kernel function pointers, that have been retrieved or restored in a secure way and are suitable for further use.

GMER takes the following actions to initialize its own secure environment in *DriverEntry*.

- To construct names of driver objects of interest on the stack, such as \Filesystem\Ntfs, \Driver\Disk, \Driver\atapi.
- To select kernel object offsets depending on the OS version, including, EPROCESS.Peb, EPROCESS.UniqueProcessId, KPROCESS.ThreadListHead, ETHREAD.StartAddress, etc.
- In case of an unknown Windows version, it obtains those offsets through manual analysis of the appropriate ntoskrnl functions, such as *PsGetProcessPeb*, *PsGetProcessSectionBaseAddress*, etc.

- Dynamically resolves some important imports such as *IofCompleteRequest* and *IofCallDriver* using *MmGetSystemRoutineAddress*.
- To map the NT layer DLL ntdll.dll, which is used further to get additional information.
- To obtain the start address of loaded Ntfs.sys and Fastfat.sys, locate the entry point and scan it for a specific signature to retrieve the real address of their IRP_MJ_CREATE handlers.
- To get information about loaded ataport.sys and scsiport.sys. These drivers are responsible for low-level disk communication.
- To use its own PE export parser and get the addresses of the sensitive functions listed below.

```
Process and thread functions:   ZwReadVirtualMemory, ZwSuspendThread,
                                ZwQueryInformationProcess, ZwQueryInformationThread,
                                PsGetThreadTeb, PsGetProcessPeb, PsGetProcessId,
                                PsGetThreadProcess, PsGetThreadId, PsGetThreadProcessId

Registry:                       ZwOpenKey, ZwEnumerateKey, ZwQueryValueKey,
                                ZwDeleteKey, ZwDeleteValueKey

Other:                          ZwQuerySystemInformation, ZwClose, NtShutdownSystem
```

For most of the kernel objects offsets, GMER obtains them by analyzing the following functions.

```
  PsGetProcessPeb                           -> EPROCESS.Peb
  PsGetProcessId                            -> EPROCESS.UniqueProcessId
  PsGetProcessInheritedFromUniqueProcessId  -> EPROCESS.InheritedFromUniqueProcessId
  PsGetProcessSectionBaseAddress            -> EPROCESS.SectionBaseAddress
  PsGetProcessPriorityClass                 -> EPROCESS.Win32Process
  IoThreadToProcess                         -> ETHREAD.ThreadsProcess
  PsGetCurrentThreadId                      -> ETHREAD.ClientID
  PsGetThreadWin32Thread                    -> KTHREAD.Win32Thread
  KeDelayExecutionThread                    -> KiWaitListHead (WinXP, 2k3)
  KeWaitForSingleObject                     -> KiWaitInListHead, KiWaitOutListHead (Win2k)
  KeSetAffinityThread                       -> KiDispatcherReadyListHead (Win2k)
```

The following driver function looks up for most offsets due to their simple structure at the beginning.

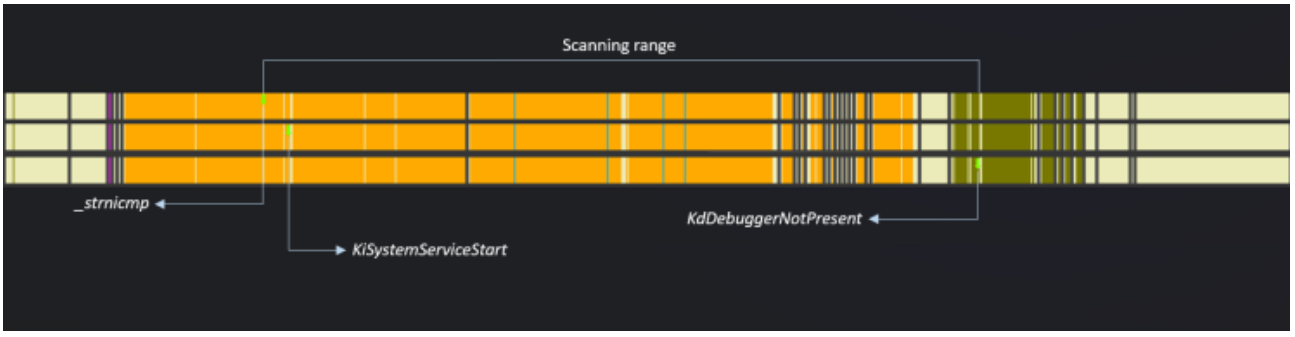
```
;============================================================
mov r64, qword ptr [r64+offs] ; offs - kernel object offset
;============================================================

fnGetKernelObjects_Offs proc near
    xor eax, eax
    cmp word ptr [rcx], 0x8B48 ; mov reg64
    jnz NextCheck
    movzx eax, word ptr [rcx+3] ; [+offs]
    retn

    NextCheck:
        cmp word ptr [rcx], 0x818A ; add reg64
        jnz NextCheck1
        movzx eax, word ptr [rcx+2] ; [+offs]
        retn

    NextCheck1:
        mov edx, eax

    NextCheck2:
        cmp byte ptr [rcx], 0x48 ; dec r16/r32
        jnz Inc1
        cmp byte ptr [rcx+1], 0x8b ; mov eax
        jnz Inc1
        cmp byte ptr [rcx+2], 0x80 ; mov eax, [eax+offs]
        mov eax, [rcx+3] ; [+offs]
        cmp eax, 0x400 ; if it's the valid offset, it should be less 0x400
        jb Return

    Inc1:
        add edx, 1
        add rcx, 1
        cmp edx, 0xA
        jb NextCheck1

    Return:
    retn
```

The situation varies when it comes to finding the corresponding EPROCESS and ETHREAD fields used to collect information about threads, as different Windows versions use a different number of lists.

GMER obtains the addresses of nt!Zw exports (services) in a tricky way. First, it obtains the *KeServiceDescriptorTable* address by finding a 8-byte signature "8B F8 C1 EF 07 83 E7 20" and two other values in ntoskrnl, which represent the following instructions inside the *KiSystemServiceStart* function. As a starting point, it takes the address of the *nt!strnicmp* function and scans it to the *KdDebuggerNotPresent* variable.

```
KiSystemServiceStart:
    48 89 A3 90 00 00 00    mov     [rbx+90h], rsp
    8B F8                   mov     edi, eax
    C1 EF 07                shr     edi, 7
    83 E7 20                and     edi, 20h
    25 FF 0F 00 00          and     eax, 0FFFh

KiSystemServiceRepeat:
    4C 8D 15 55 63 9D 00    lea     r10, KeServiceDescriptorTable
    4C 8D 1D 8E 3B 8F 00    lea     r11, KeServiceDescriptorTableShadow
    F7 43 78 80 00 00 00    test    dword ptr [rbx+78h], 80h
    74 13                   jz      short loc_14042B58E
```

The disposition of the target ntoskrnl functions.



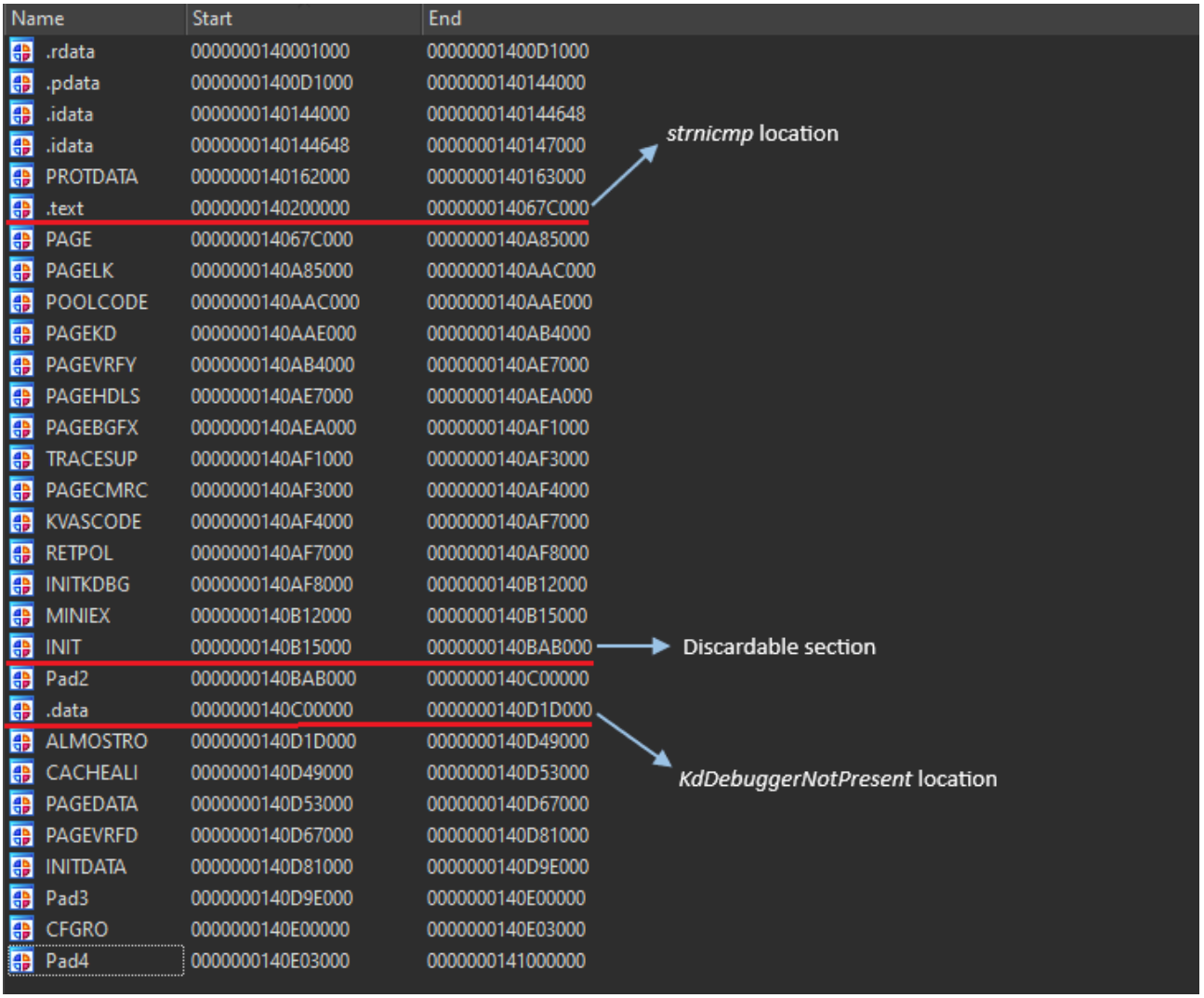
To obtain the address of a specific *ntoskrnl!Zw* function, it maps Ntdll and retrieves from it the address of the required export function. Next, it grabs the System Service Number (SSI) from the second instruction of the export, which is identical for all of them: mov eax, SSN (B8 3F 00 00 00).

With these pieces together, the process of obtaining ntoskrnl exports for Zw services looks as follows:

1. To get an export function address from the mapped Ntdll.
2. To take the SSN from the second instruction of the function.
3. To use this SSN as an index in the *KeServiceDescriptorTable.KiServiceTable* array and calculate the appropriate *Zw* function address. Note that the pointers in this array are protected by PatchGuard.

```
; rax->ntdll export, rdi->KeServiceDescriptorTable,

mov eax, [rax+4]    ; after rax->SSN (mov eax, SSN)
cmp rax, [rdi+SSDT.NumberOfServices]
jnb short jRet

; FunctionAddress = KiServiceTable + (FunctionOffset >> 4)

mov rcx, [rdi+SSDT.KiServiceTable]
movsxd rax, dword ptr [rcx+rax*4]
sar rax, 4
add rax, rcx ; rax->ntoskrnl Zw function
cmp rax, rbx
jbe short jRet
```

An interesting fact is that when scanning ntoskrnl data between strnicmp and *KdDebuggerNotPresent* to find the address of *KeServiceDescriptorTable*, the driver doesn't validate the current pointer with *MmIsAddressValid*. Since the space between these symbols belongs to multiple ntoskrnl sections, one of them may be INIT, which may be already discarded from memory.



📖 Overview of the driver

The driver provides its user mode counterpart with various valuable interfaces aimed at obtaining trustworthy data about the Windows environment. These features are available via the appropriate IOCTL codes listed below. After obtaining the necessary data, GMER compares it with the data obtained through regular Windows API and report to the user about the detected anomalies.

Basically, to supply the requested data, the driver leverages Windows kernel API, low-level disk and file system access skipping the intermediate filters, DKOM and custom implementation of some Windows kernel functions.

IOCTL number	Description	Implementation details
0x7201C004	Query ataport.sys and scsiport.sys image bases	ZwQuery* API
0x7201C008	Secure disk RW	See below
0x7201C00C	Query device objects pointers for the provided device interface class	Ntoskrnl API
0x7201C010	Query an object manager directory object	Ntoskrnl API
0x7201C014	Query an object manager symbolic link information	Ntoskrnl API
0x7201C018	Query system module information	ZwQuery* API
0x7201C01C	Duplicate the driver object by device name	-
0x7201C020	Replace or restore the disk port driver's IAT entry for IofCompleteRequest	See below
0x7201C024	Obtain the disk port driver object path	-
0x7201C028	Read kernel mode data	Custom implementation
0x7201C02C	Prepare for disk write operation	See below
0x7201C030	Pulse a keyboard port and cause BSOD	MANUALLY_INITIATED_CRASH
0x7201C034	Write kernel mode data	Custom implementation
0x9876C010	Cause BSOD	MANUALLY_INITIATED_CRASH
0x9876C050	Shutdown system	Ntoskrnl API
0x9876C054	Write file secure	See below
0x9876C058	Check the classpnp driver on hooks	See below
0x9876C080	Enumerate threads and processes	DKOM
0x9876C0A0	Copy KiServiceTable	
0x9876C0A4	System registry API with Zw*Key	Zw*Key
0x9876C0C0	Read file secure	See below
0x9876C0C4	R/W executable file secure	See below
0x9876C0C8	Delete file secure	Ntoskrnl API
0x9876C0CC	Perform a specific IO request on the specified device object	See below

During initialization, the driver creates a separate thread to execute the following operations in the System process context: shutdown system, read process memory, suspend thread, query information about thread, process, system, and system registry. When the GMER's DeviceIoControl handler, which is executed in the current process context, recognizes one of those IOCTLs, it builds the context structure with a pointer to a specific handler and sets the appropriate event that triggers another thread to execute it.

📚 Exploring Win11 disk subsystem

Before we start discussing the topic, let's take a look at some basic aspects of Windows disk subsystem.

The Disk.sys driver is responsible for dispatching storage devices I/O. The client must specify the offset from the beginning of the disk and data length. The driver in turn redirects this request to one of the corresponding disk port drivers.

We can start by exploring disk device stack and go deeper.

```
2: kd> !devobj \Device\Harddisk0\DR0
Device object (ffffe5035398f080) is for:
 DR0 \Driver\disk DriverObject ffffe503537aada0

2: kd> !devstack ffffe5035398f080
  !DevObj           !DrvObj            !DevExt           ObjectName
  ffffe5035381f8d0  \Driver\partmgr    ffffe5035381fa20
> ffffe5035398f080  \Driver\disk       ffffe5035398f1d0  DR0
  ffffe50351bf1e30  \Driver\ACPI       ffffe503531ef460
  ffffe503535d8050  \Driver\iaStorVD   ffffe503535d81a0  00000055
```

As we can see there are four devices on that device stack.

- The first one belongs to the partition manager and presents the device of the raw disk partition. Partmgr forwards the disk I/O request further to the disk driver, providing it an LBA instead of a partition offset.
- The second device belongs to the disk driver itself, described above.
- The next device was created by the common driver acpi.sys which, essentially simply forwards disk I/O requests to the port driver. The responsibilities of Acpi. sys include support for power management and Plug and Play (PnP) device enumeration.
- The latter belongs to the iaStorVD disk port driver (Intel Rapid Storage Technology) and is used in many computers with Intel chipsets. Being a disk port driver, it's responsible for low-level communication with a specific storage device type, including, initializing the storage controller and identifying and attaching storage devices.

But the whole picture of processing disk requests is a bit more complicated, since there are more drivers that are indirectly involved in it. Let's inspect those driver objects on the disk device stack.

```
2: kd> !drvobj \Driver\disk 7
Driver object (ffffe503537aada0) is for:
 \Driver\disk

...

DriverEntry:   fffff8075f858010 disk!GsDriverEntry
DriverUnload:  fffff8075f8cb660 CLASSPNP!ClassUnload
AddDevice:     fffff8075f8c42b0 CLASSPNP!ClassAddDevice

Dispatch routines:
[00] IRP_MJ_CREATE                      fffff8075f878980    CLASSPNP!ClassGlobalDispatch
[01] IRP_MJ_CREATE_NAMED_PIPE           fffff8075c7517e0    nt!IopInvalidDeviceRequest
[02] IRP_MJ_CLOSE                       fffff8075f878980    CLASSPNP!ClassGlobalDispatch
[03] IRP_MJ_READ                        fffff8075f878980    CLASSPNP!ClassGlobalDispatch
[04] IRP_MJ_WRITE                       fffff8075f878980    CLASSPNP!ClassGlobalDispatch
...
[0e] IRP_MJ_DEVICE_CONTROL              fffff8075f878980    CLASSPNP!ClassGlobalDispatch
[0f] IRP_MJ_INTERNAL_DEVICE_CONTROL     fffff8075f878980    CLASSPNP!ClassGlobalDispatch
...

2: kd> !drvobj \Driver\iaStorVD 7
Driver object (ffffe503511e8d50) is for:
 \Driver\iaStorVD

...

DriverEntry:   fffff8075e9ba000 iaStorVD
DriverUnload:  fffff8075e7fb770 iaStorVD
AddDevice:     fffff8075e9ff2c0 storport!RaDriverAddDevice

Dispatch routines:
[00] IRP_MJ_CREATE                      fffff8075ea60050    storport!RaDriverCreateIrp
[01] IRP_MJ_CREATE_NAMED_PIPE           fffff8075c7517e0    nt!IopInvalidDeviceRequest
[02] IRP_MJ_CLOSE                       fffff8075ea5ffb0    storport!RaDriverCloseIrp
...
[0e] IRP_MJ_DEVICE_CONTROL              fffff8075e9c5ab0    storport!RaDriverDeviceControlIrp
[0f] IRP_MJ_INTERNAL_DEVICE_CONTROL     fffff8075e9c4180    storport!RaDriverScsiIrp
...
[16] IRP_MJ_POWER                       fffff8075e9d02d0    storport!RaDriverPowerIrp
...
```

As we can see both drivers redirect their driver dispatch routines to another drivers. In the case of disk.sys its table of driver dispatch routines points to Classpnp functions. The driver name stands for "Class Plug and Play Driver" and is responsible for managing Plug and Play (PnP) devices. Classpnp handles the device requests addressed to disk.sys. *classpnp!ClassGlobalDispatch*, in turn, simply redirects the execution flow to the appropriate classpnp dispatch function. Therefor, these items of the disk driver dispatch table are perfect targets for any rootkit (if it can defeat PatchGuard first).

```
; IN rcx -> device object, rdx -> irp
CLASSPNP!ClassGlobalDispatch:

mov rax, qword ptr [rdx+0B8h] ; rax -> current stack location
mov r8, qword ptr [rcx+40h] ; r8 -> device extension
movzx r9d, byte ptr [rax] ; r9d -> major function
mov rax, qword ptr [r8+1C0h] ; rax -> handlers table
jmp qword ptr [rax+r9*8]
```

If we know the pointer to the disk device object, we can get information about the real dispatch table of the classpnp driver.

```
0xfffffa80`025b5790 - disk device object
+0x40 - DeviceExtension
+0x1c0 - dispatch table
```

```
kd> dps poi(poi(fffffa80`025b5790+40)+1c0)
fffffa80`025b1618  fffff880`01759b30 CLASSPNP!ClassCreateClose
fffffa80`025b1620  fffff880`0174b3c0 CLASSPNP!ClassDispatchUnimplemented
fffffa80`025b1628  fffff880`01759b30 CLASSPNP!ClassCreateClose
fffffa80`025b1630  fffff880`0173c8a0 CLASSPNP!ClassReadWrite
fffffa80`025b1638  fffff880`0173c8a0 CLASSPNP!ClassReadWrite
fffffa80`025b1640  fffff880`0174b3c0 CLASSPNP!ClassDispatchUnimplemented
fffffa80`025b1648  fffff880`0174b3c0 CLASSPNP!ClassDispatchUnimplemented
fffffa80`025b1650  fffff880`0174b3c0 CLASSPNP!ClassDispatchUnimplemented
fffffa80`025b1658  fffff880`0174b3c0 CLASSPNP!ClassDispatchUnimplemented
fffffa80`025b1660  fffff880`0173d670 CLASSPNP!ClassShutdownFlush
fffffa80`025b1668  fffff880`0174b3c0 CLASSPNP!ClassDispatchUnimplemented
fffffa80`025b1670  fffff880`0174b3c0 CLASSPNP!ClassDispatchUnimplemented
fffffa80`025b1678  fffff880`0174b3c0 CLASSPNP!ClassDispatchUnimplemented
fffffa80`025b1680  fffff880`0174b3c0 CLASSPNP!ClassDispatchUnimplemented
fffffa80`025b1688  fffff880`0173de50 CLASSPNP!ClassDeviceControlDispatch
fffffa80`025b1690  fffff880`0174ce30 CLASSPNP!ClassInternalIoControl
```

The second driver iaStorVD.sys relies on its counterpart storport.sys. The latter is a general-purpose storage driver responsible for managing communication between storage devices and the operating system itself. While iaStorVD.sys provides functionality for managing RAID arrays and handling I/O requests for devices connected to the Intel RAID controller, storport.sys is directly responsible for communication with storage devices. Thus, Storport.sys operates at a lower level than iaStorVD.sys, providing basic communication and data transfer functionality with the storage hardware.

iaStorVD.sys also creates one more device incorporated in another device stack that ends up in Pci.sys. It's used to handle requests for the PCI bus driver Pci.sys.

```
2: kd> !devstack ffffe50351be3050
  !DevObj           !DrvObj            !DevExt           ObjectName
> ffffe50351be3050  \Driver\iaStorVD   ffffe50351be31a0  RaidPort0
  ffffe503521e4d50  \Driver\ACPI       ffffe503531e6460
  ffffe503516e8360  \Driver\pci        ffffe503516e84b0  NTPNP_PCI0009
!DevNode ffffe503512f1a20 :
  DeviceInst is "PCI\VEN_8086&DEV_9A0B&SUBSYS_00008086&REV_00\3&11583659&0&70"
  ServiceName is "iaStorVD"
```

🛠️ Patching kernel data

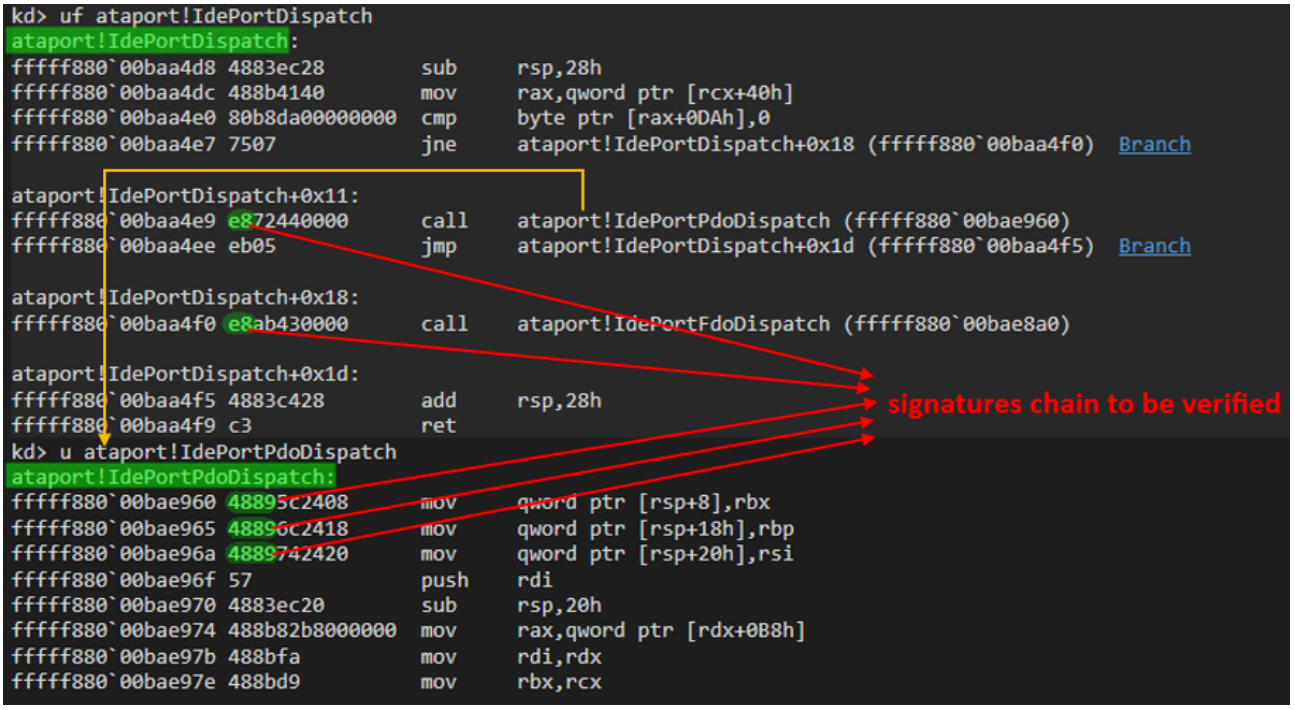
GMER aims to patch kernel mode code and disk port driver data in two cases mentioned below. In both of them, it is interested in intercepting control of the disk I/O operation before the disk port driver returns control to the client. The driver parses the SCSI_REQUEST_BLOCK structure and, in particular, its Cdb structure to obtain information about the LBA and the size of the requested data.

- The anti-rootkit driver overwrites the IAT entry of the disk port driver that matches the *IofCompleteRequest* function with a GMER's one.
- It also implements run-time code patching. First 0xF bytes of *ataport!IdePortDispatch->ataport!IdePortPdoDispatch* are subject to this modification in the case of a sector write operation.

GMER puts the following instruction at the beginning of *ataport!IdePortPdoDispatch*. A pointer to its implementation of *IofCompleteRequest* follows the instruction.

```
IdePortPdoDispatch pseudocode after modification

ataport!IdePortPdoDispatch:
ff2500000000 jmp qword ptr [ataport!IdePortPdoDispatch+0x6]
ff2500000006 ptr_to_fnGMER_IofCompleteRequest
```

While the address of *ataport!IdePortDispatch* can simply be obtained from the driver's dispatch function table, to find the address of *ataport!IdePortPdoDispatch* GMER needs to check the body of the first function for a specific signature chain. This signature chain is presented below.



Below you can see part of the code that implements the interception of the *ataport!IdePortPdoDispatch* function. The first call is used to modify that function and copy the old 0xF bytes. The second one saves the copied old bytes to the global driver data.

```
098 48 8D 05 27 F3 FF FF  lea     rax, fnGMER_IofCompleteRequest2_Internal
098 48 89 44 24 56        mov     [rsp+98h+pfnGMER_IofCompleteRequest2_Internal], rax
098 48 8D 47 0F           lea     rax, [rdi_ptr_ataport_IdePortPdoDispatch+0Fh]
098 48 89 05 F7 49 00 00  mov     cs:ptr_ataport_IdePortPdoDispatch_plus_f_bytes, rax
098 4C 8D 4C 24 60        lea     r9, [rsp+98h+ataport_IdePortPdoDispatch_f_bytes_old_local]
098 45 8B C5              mov     r8d, r13d_value_f
098 48 8D 54 24 50        lea     rdx, [rsp+98h+ataport_IdePortPdoDispatch_hook]
098 48 8B CF              mov     rcx, rdi_ptr_ataport_IdePortPdoDispatch
098 E8 12 F0 FF FF        call    fnPatchKernelData ; IN rcx -> pKernelData, rdx -> data to copy

098 45 33 C9              xor     r9d, r9d
098 45 8B C5              mov     r8d, r13d_value_f
098 48 8D 54 24 60        lea     rdx, [rsp+98h+ataport_IdePortPdoDispatch_f_bytes_old_local]
098 48 8D 0D 05 26 00 00  lea     rcx, fn_ataport_IdePortPdoDispatch_f_bytes_old
098 E8 FB EF FF FF        call    fnPatchKernelData ; IN rcx -> pKernelData, rdx -> data to copy
```

It's worth noting that the GMER function for patching kernel mode code isn't safe for use in multiprocessor systems (SMP). Instead of raising IRQL on all CPUs, the driver does this only on the current one. GMER implements a typical method for patching kernel mode data as follows.

```
GMERPatchKernelData(PVOID pTargetData, PVOID pSource, ULONG Length,
                    OUT PVOID OldData)
{
    PMDL Mdl;
    BOOL f;
    BOOL bPagesLocked;
    PVOID pMappedTargetData;
    KSPIN_LOCK SpinLock;
    KIRQL Irql;

    Mdl = IoAllocateMdl(pTargetData, Length, 0, 0, NULL);

    f = Mdl->MdlFlags & (MDL_MAPPED_TO_SYSTEM_VA | MDL_PAGES_LOCKED |
         MDL_SOURCE_IS_NONPAGED_POOL);

    if(!f) {
        MmProbeAndLockPages(Mdl, KernelMode, IoModifyAccess);
        bPagesLocked = TRUE;
    }

    pMappedTargetData = MmMapLockedPagesSpecifyCache(Mdl, KernelMode,
        MmCached, NULL, 0, NormalPagePriority);

    //Raising IRQL only on the current CPU isn't enough in SMP systems
    Irql = KeAcquireSpinLockRaiseToDpc(&SpinLock);

    if(OldData) memmove(OldData, pMappedTargetData, Length);

    memmove(pMappedTargetData, pSource, Length);

    KeReleaseSpinLock(&SpinLock, Irql);

    MmUnmapLockedPages(pMappedTargetData, Mdl);

    if(bPagesLocked) MmUnlockPages(Mdl);

    IoFreeMdl(Mdl);
}
```

🔑 Securing disk I/O operations

The driver provides GMER with the ability to work with disk at a low level by addressing directly to the Atapi/Ataport or Scsiport disk drivers. It builds a request packet and sends it to one of them through the IRP_MJ_INTERNAL_DEVICE_CONTROL request depending on which one is active in the system.

The driver receives the name of the disk device object from its user-mode counterpart. Before sending a request to the disk device, the driver obtains the pointer to the lowest device on the stack using *IoGetBaseFileSystemDeviceObject*, thus bypassing all potentially non trusted devices.

```
struct _CDB10 {

UCHAR OperationCode;

...

UCHAR LogicalUnitNumber : 3;

UCHAR LogicalBlockByte0;

UCHAR LogicalBlockByte1;

UCHAR LogicalBlockByte2;

UCHAR LogicalBlockByte3;

...

UCHAR Control;

} CDB10, *PCDB10;
```

Before calling the disk driver, GMER patches its IAT entry matching the *IofCompleteRequest* function to GMER's one.

```
.text:FFFFF8800338F09A     jGetPE_IATEntry_for_IofCompleteRequest: ; CODE XREF: fnReplaceDiskDriverIATEntryWithGMERIofCompleteRequest+29↑j
.text:FFFFF8800338F09A 048                 mov     r11, cs:pIofCompleteRequest_0
.text:FFFFF8800338F0A1 048                 mov     rdx, r11
.text:FFFFF8800338F0A4 048                 mov     rcx, rbx
.text:FFFFF8800338F0A7 048                 call    fnGetPE_IATEntry_ByName ; IN rcx->disk port driver base address, rdx->pIofCompleteRequest; OU
.text:FFFFF8800338F0A7
.text:FFFFF8800338F0AC 048                 mov     cs:pAtaport_Atapi_Scsiport_IATEntry_For_IofCompleteRequest, rax
.text:FFFFF8800338F0B3 048                 test    rax, rax
.text:FFFFF8800338F0B6 048                 jnz     short jPatchDiskDriverIATEntry
.text:FFFFF8800338F0B6
.text:FFFFF8800338F0B8 048                 mov     rcx, cs:ScsiportSys_ImageBase
.text:FFFFF8800338F0BF 048                 test    rcx, rcx
.text:FFFFF8800338F0C2 048                 jz      short jPatchDiskDriverIATEntry
.text:FFFFF8800338F0C2
.text:FFFFF8800338F0C4 048                 mov     rdx, r11
.text:FFFFF8800338F0C7 048                 call    fnGetPE_IATEntry_ByName ; IN rcx->disk port driver base address, rdx->pIofCompleteRequest; OU
.text:FFFFF8800338F0C7
.text:FFFFF8800338F0CC 048                 mov     cs:pAtaport_Atapi_Scsiport_IATEntry_For_IofCompleteRequest, rax
.text:FFFFF8800338F0CC
.text:FFFFF8800338F0D3
.text:FFFFF8800338F0D3     jPatchDiskDriverIATEntry:               ; CODE XREF: fnReplaceDiskDriverIATEntryWithGMERIofCompleteRequest+17↑j
.text:FFFFF8800338F0D3                                             ; fnReplaceDiskDriverIATEntryWithGMERIofCompleteRequest+7A↑j ...
.text:FFFFF8800338F0D3 048                 test    rax, rax
.text:FFFFF8800338F0D6 048                 jz      short jRet_2
.text:FFFFF8800338F0D6
.text:FFFFF8800338F0D8 048                 lea     rcx, fnGMER_IofCompleteRequest
.text:FFFFF8800338F0DF 048                 cmp     [rax], rcx
.text:FFFFF8800338F0E2 048                 jz      short jRet_2
.text:FFFFF8800338F0E2
.text:FFFFF8800338F0E4 048                 mov     [rsp+48h+pGMER_IofCompleteRequest], rcx
.text:FFFFF8800338F0E9 048                 lea     r9, Ataport_Atapi_Scsiport_IATEntry_For_IofCompleteRequest_OrigValue
.text:FFFFF8800338F0F0 048                 mov     r8d, 8
.text:FFFFF8800338F0F6 048                 lea     rdx, [rsp+48h+pGMER_IofCompleteRequest]
.text:FFFFF8800338F0FB 048                 mov     rcx, rax
.text:FFFFF8800338F0FE 048                 call    fnPatchKernelData ; IN rcx -> pKernelData, rdx -> data to copy, r8d -> number of bytes OUT r9
```

GMER supports IOCTL code 0x7201C008 to perform secure disk I/O operation for its user-mode application. Below you can see its pseudo code, which omits minor operations.

```
//the user mode app supplies the driver with the name of the target disk //device, type of operation (RW), data buffer, and its size

//get the disk device object pointer by its name
IoGetDeviceObjectPointer(&unDeviceName, &pDeviceObject);

//the previous call returned the top device object in the device object //stack, get the lowest since we need to bypass all potentially attached //filters
pDeviceObject = IoGetBaseFileSystemDeviceObject(pDeviceObject);

pDiskPortDriverObject = pDeviceObject->DriverObject;

pDataBuf = ExAllocatePool(DataBufSize);
pEvent = ExAllocatePool(sizeof(KEVENT));

//patch the disk port driver's IAT entry of IofCompleteRequest to GMER's //one
fnReplaceDiskPortDriverIATEntryWithGMERIofCompleteRequest(pDiskPortDriverObject);

ZeroMemory(pDataBuf);
KeInitializeEvent(pEvent);
KeClearEvent(pEvent);

Srb = ExAllocatePool(sizeof(SCSI_REQUEST_BLOCK));
ZeroMemory(Srb, sizeof(SCSI_REQUEST_BLOCK));

//initialize the SCSI_REQUEST_BLOCK structure and CDB10 -> OperationCode = //SCSIOP_READ or SCSIOP_WRITE
InitializeSrbAndCdbBlock(Srb);

pIrp = IoAllocateIrp();

pMdl = IoAllocateMdl(pDataBuf);

MmProbeAndLockPages(Mdl);

InitializeIrpAndIoStackLocation(Irp);

//check IdePortDispatch for potential hooks and restore if possible
fnCheck_ataport_IdePortDispatch(pDiskPortDriverObject);

//check StartIo for hooks and restore if possible
fnCheck_ataport_StartIo_Pointer(pDiskPortDriverObject);

IofCallDriver(pDeviceObject, pIrp);
KeWaitForSingleObject(pEvent);

fnRestoreDiskPortDriverIATEntry(pDiskPortDriverObject,    DiskPort_IATEntry_IofCompleteRequest_OrigValue);
```

GMER has several I/O completion routines that are designed to be used in multiple anti-rootkit scenarios when processing disk I/O operations.

- The first is used to secure (intercept) the disk read operation (IRP_MJ_INTERNAL_DEVICE_CONTROL, SCSIOP_READ) globally (IOCTL 0x7201C020). Its pointer replaces *IofCompleteRequest* in the disk port driver IAT entry and is used to copy read data from the system buffer to the GMER's one.
- Another one is involved in securing the disk write op (IRP_MJ_INTERNAL_DEVICE_CONTROL, SCSIOP_WRITE) globally (IOCTL 0x7201C02C). The driver uses a pointer to this function in the patching code for *ataport!IdePortPdoDispatch*. This routine is used to prohibit write access to disk sectors (LBA) supplied from user mode .
- The latter is used to dispatch the initiated disk I/O request coming from the GMER app (IOCTL 0x7201C008).

In addition to those IOCTLs, GMER has a feature to scan the classpnp handlers for potential run-time hooks (0x9876C058). The driver obtains the handler offset inside the classpnp driver file, its SYSTEM_MODULE_ENTRY.ImageBase, and checks its prologue for two signature sequences: 0x55 0x8B 0xEC, 0x8B 0xFF 0x55.

```
0x55 push ebp
0x8B 0xEC mov ebp, esp ; usual prologue

0x8B 0xFF mov edi, edi
0x55      push ebp     ; first instruction is for hot patching support
```

But it's unclear why the 64-bit GMER driver checks the classpnp dispatch functions for x86 instructions...

🗃️ Securing file I/O operations

GMER secures file operations as follows.

- To open a file, the driver calls the *IoCreateFileSpecifyDeviceObjectHint* API, sending a request directly to the FSD, skipping possible intermediate filters. To obtain a pointer to the FSD device object that is the lowest on the stack, it uses *IoGetBaseFileSystemDeviceObject* (*IoGetDeviceAttachmentBaseRef*).
- If the function fails, GMER tries to check *IofCallDriver* for hooks, but only its the old version, which has a jump instruction to the *IopfCallDriver* pointer at the beginning.
- In addition to checking *IofCallDriver* for hooks, the driver also checks and restores the original IRP_MJ_CREATE handlers for Ntfs and Fastfat driver objects. As was mentioned above, GMER obtains these handlers at the driver initialization phase.
- After obtaining a handle to the requested file, the driver uses the ordinary APIs *ZwReadFile*, *ZwWriteFile*, *ZwDeleteFile*, *ZwQueryInformationFile*, *ZwSetInformationFile*, *ZwClose*.

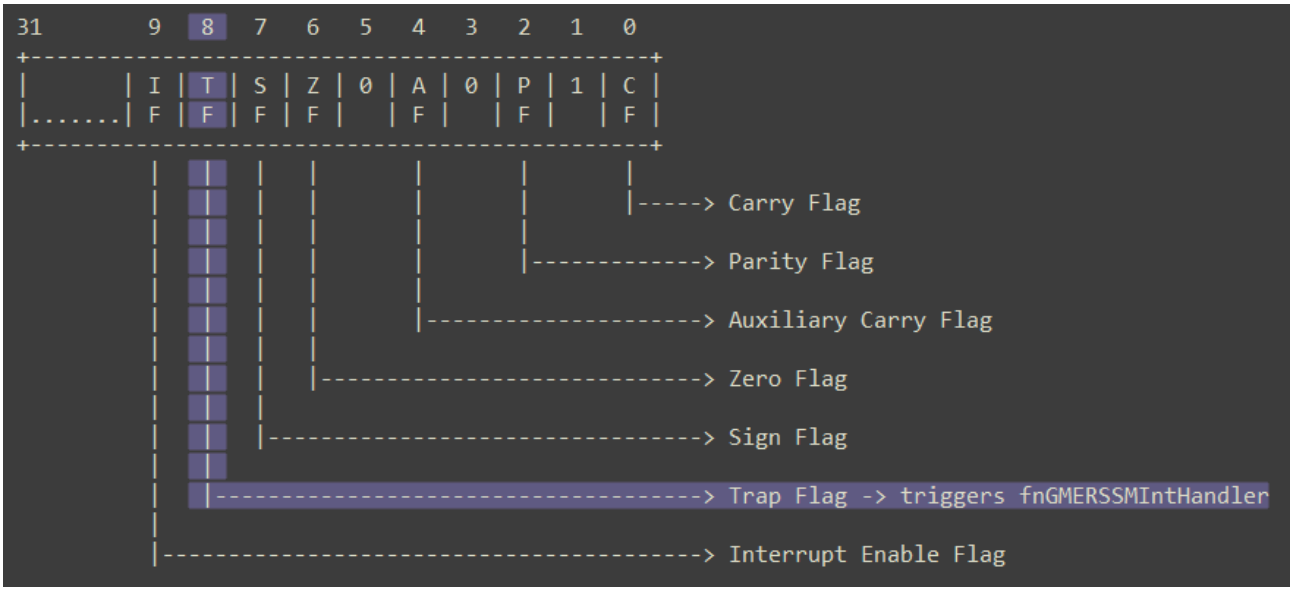
These operations are performed in the System process context.

The following pseudocode demonstrates how GMER secures file I/O operations.

```
KeAttachProcess(SystemProcess);

Status = IoCreateFileSpecifyDeviceObjectHint(pLowestFSDeviceObject);

if( Status!=STATUS_SUCCESS )
{
    fnInterceptOrRestoreIopfCallDriverIfExists();
    fnInterceptOrRestoreNtfsOrFastFatIrpMjCreate();

    Status = IoCreateFileSpecifyDeviceObjectHint(pLowestFSDeviceObject);
}

if( Status!=STATUS_SUCCESS ) ZwOpenFile();

...

fnInterceptOrRestoreIopfCallDriverIfExists();
fnInterceptOrRestoreNtfsOrFastFatIrpMjCreate();

KeDetachProcess();
```

⛓Tracing kernel mode code

Along with information about system anomalies, GMER is also capable of providing details about possible code execution flow violations involved in processing file system operations. This feature is based on code tracing or single-step CPU mode, when the driver code intercepts control after each executed CPU instruction and saves information about each system module to which this instruction belongs. This mode is activated by setting the trap flag in the RFLAGS register {pushfq; pop rax; or eax 100h; push rax; popfq}.



Below you can see the driver code responsible for intercepting the int 1 handler and the corresponding x64 structures.

```
// the format of the x64 IDT entry
union _KIDTENTRY64 {
    USHORT OffsetLow;
    USHORT Selector;
    USHORT IstIndex : 3;
    USHORT Reserved0 : 5;
    USHORT Type : 5;
    USHORT Dpl : 2;
    USHORT Present : 1;
    USHORT OffsetMiddle;
    ULONG OffsetHigh;
    ULONG Reserved1;
};

// the format of the interrupt descriptor table register (IDTR)
struct _KDESCRIPTOR {
    USHORT Pad[3];
    USHORT Limit;
    PVOID Base;
} kDescriptor;

rdx -> GMER's int 1 handler
============================

lea rcx, [rsp+kDescriptor]
sidt fword ptr [rcx] ; read IDTR to [rcx] variable

cli ; mask (disable) interrupts

mov rbx, [rsp+kDescriptor.Base]
mov r11, rdi
shr r11, 20h
mov rax, rdi
mov [rbx+10h], di   ; pIdt[1].OffsetLow
shr rax, 10h
mov [rbx+18h], r11d ; pIdt[1].OffsetHigh
mov [rbx+16h], ax   ; pIdt[1].OffsetMiddle

sti ; enable interrupts
============================
```

The driver interrupt handler is simply a wrapper, it prepares the necessary data on the stack and calls the real handler.

```
.text:FFFFF880033E9282     fnGMERSSMIntHandler proc near          ; DATA XREF: fnActivateSingleStepModeInterrupt+115↑o
.text:FFFFF880033E9282 000                cli
.text:FFFFF880033E9283 000                pushfq
.text:FFFFF880033E9284 008                push    r15
.text:FFFFF880033E9286 010                push    r14
.text:FFFFF880033E9288 018                push    r13
.text:FFFFF880033E928A 020                push    r12
.text:FFFFF880033E928C 028                push    r11
.text:FFFFF880033E928E 030                push    r10
.text:FFFFF880033E9290 038                push    r9
.text:FFFFF880033E9292 040                push    r8
.text:FFFFF880033E9294 048                push    rdi
.text:FFFFF880033E9295 050                push    rsi
.text:FFFFF880033E9296 058                push    rdx             ; -> IofCallDriver(DeviceObject, _Irp_); [rbp + 20h]
.text:FFFFF880033E9297 060                push    rcx             ; -> IofCallDriver(_DeviceObject_, Irp); [rbp + 18h]
.text:FFFFF880033E9298 068                push    rbx
.text:FFFFF880033E9299 070                push    rax
.text:FFFFF880033E929A 078                push    rbp
.text:FFFFF880033E929B 080                mov     rbp, rsp
.text:FFFFF880033E929E 080                mov     r9, cs:pSSM_Handler_Context
.text:FFFFF880033E92A5 080                mov     r8, rbp
.text:FFFFF880033E92A8 080                lea     rdx, [rbp+90h]  ; RSP: +0 RIP, +8 CS, +A RFLAGS
.text:FFFFF880033E92AF 080                mov     rcx, [rbp+80h]  ; rcx -> RIP saved by CPU before executing the handler
.text:FFFFF880033E92B6 080                cmp     cs:pfnSSMIntHandlerInternal, 0
.text:FFFFF880033E92BE 080                jz      short loc_FFFFF880033E92C6
.text:FFFFF880033E92BE
.text:FFFFF880033E92C0 080                call    cs:pfnSSMIntHandlerInternal
.text:FFFFF880033E92C0
```

When tracing code, the driver maintains a context structure with several arrays that store information about system modules and their call stack frames. Then this data will be copied to the provided user buffer and analyzed by the application for the presence of unknown system modules.

```
struct SSM_INT_CONTEXT {
    ULONG dwSize;
    ...
    PKTHREAD pCurrentThread;
    PVOID pIofCallDriver;
    PVOID pGMERCallerFunction;
    PVOID pSystemModuleInformation;
    ...
    ULONG numUnknownAddresses;
    ULONG numCallStackImageBaseArray;
    PVOID CallStackImageBaseArray;
    ULONG numIntFrameRIPArray;
    ULONG dwUnused1;
    PVOID IntFrameRIPArray;
    DWORD numCallStackDrvObjDevObjArray;
    DWORD dwUnused;
    PVOID CallStackDrvObjDevObjArray;
};
```

The driver single step mode dispatch function looks as follows.

```
rcx -> RIP from the trap (int) frame
r9 -> SSM_INT_CONTEXT from the GMER's int 1 handler
============================
mov rax, gs:188h ; rax -> ptr to current KTHREAD
mov r10, rcx

; make sure that we handle instruction from our thread
cmp [r9+SSM_INT_CONTEXT.pCurrentThread], rax
jnz jRet

; align RIP to the section start
mov rax, rcx
and rax, 0FFFFFFFFFFFF0000h
cmp [r9+SSM_INT_CONTEXT.GMERDriverTextSectionStart], rax
jz jHandleCallFromGMERCodeOrOther ; instruction originated from GMER code

save_RIP_to_IntFrameRIPArray(rcx)

; check if the instruction originated from IofCallDriver
cmp rcx, cs:pIofCallDriver
jz jHandleCallFromIofCallDriver

jmp jHandleCallFromGMERCodeOrOther

jHandleCallFromIofCallDriver:

mov rdx, [r8+18h] ; IofCallDriver rcx -> device object
mov rax, [rdx+DEVICE_OBJECT.DriverObject] ; rax -> driver object

save_ptr_to_devobj_and_drvobj_and_drvobjStartIO_in_array3

jHandleCallFromGMERCodeOrOther:

; iterate items in the system module information buffer to determine which ; driver the executed instruction belongs
mov rax, [r9+SSM_INT_CONTEXT.pBuffer_system_module_information]
mov r11d, dword ptr [rax+SYSTEM_MODULE_INFORMATION.Count]
lea rdx, [rax+SYSTEM_MODULE_INFORMATION.SystemModuleEntry.ImageBase]

jNextModuleEntry:

cmp r10, [rdx] ; r10 -> saved RIP, rdx -> ImageBase from current entry
lea rcx, [rdx-10h] ; get its rcx -> system module entry
jb jIncCounterAndNext

mov eax, [rdx+8] ; rdx + 8 = .SystemModuleEntry.ImageSize
add rax, [rdx] ; rdx -> .SystemModuleEntry.ImageBase
cmp r10, rax ; r10 -> saved RIP, rax -> module image base
jb jModuleFound

jIncCounterAndNext:
add r8d, 1
add rdx, size SYSTEM_MODULE_ENTRY
cmp r8d, r11d ; r11d -> .Count
jb jNextModuleEntry

........

jModuleFound:
add_found_module_entry_to_CallStackImageBaseArray_if_not_exist([rcx+SYSTEM_MODULE_ENTRY_1.ImageBase])
```

The above function is involved in code tracing in two scenarios - calling the FSD driver directly via *IofCallDriver* and *ZwQueryDirectoryFile*.

```
//build an IRP_MJ_READ request
IoBuildSynchronousFsdRequest();

//start recording information about executing instructions
EnableTracing();

//call the FSD
IofCallDriver();

//stop recording information
DisableTracing();
```

The second scenario.

📦About PPL'ed processes

PPL is a widely known built-in Windows security feature designed to provide high-end protection for trusted Windows services such as anti-malware processes. It's recognized as a robust security measure aimed at protecting running processes from any form of modification or other destructive impact. Access checks for PPL-protected processes are implemented at the opening process handle level, without any exceptions for kernel mode code. As a result, these processes cannot be terminated using *ZwOpenProcess* and *ZwTerminateProcess* calls made by the kernel mode driver.

The required access checks can only be successfully passed by code that works on another PPL protection level with an equal or higher one. GMER isn't an exception; like other renowned security tools, including, Process Explorer, Process Hacker, Process Informer, WinArk, it can't terminate PPL'ed processes. This limitation arises not only because it employs a simple trick involving a pair of aforementioned functions after attaching to the System process, but also due to its constraints when working on modern Windows versions. To terminate a PPL-protected process, kernel mode code requires a pointer to the kernel object of the process and a pointer to an internal ntoskrnl function, *PspTerminateProcess*, capable of process termination by a pointer rather than its handle.

GMER terminates the process as shown below (IOCTL 0x9876C094).

```
//Attach to the System process to increase the likelihood of successfully //obtaining the handle to the target process
KeStackAttachProcess(pSystemProcess);

//Obtain a handle; in case of a PPL'ed process, we'd fail here
ZwOpenProcess(&hTargetProcess, PROCESS_TERMINATE,..., Pid);

//GMER needs the object pointer to reset
//EPROCESS->Flags.BreakOnTermination before termination, otherwise
//the system may fail and result in a BSOD
ObReferenceObjectByHandle(hTargetProcess,..., &pTargetProcess,...);

ResetEPROCESSBreakOnTerminationFlagDependingOSVersion(pTargetProcess);

//Free the EPROCESS pointer
ObDereferenceObject(pTargetProcess);

//Terminate process by handle
ZwTerminateProcess(hTargetProcess);

//Close handle
ZwClose(hTargetProcess);

//Leave the System process context
KeUnstackDetachProcess(pSystemProcess);
```

To make the process of finding *PspTerminateProcess* more reliable across Windows versions, a signature chain candidate should consist of unique byte sequences. GMER can easily find many Windows kernel undocs, so locating *PspTerminateProcess* shouldn't be difficult for it.

```
PAGE:0000000140721CFF                   jTerminateProcess:                      ; CODE XREF: NtTerminateProcess+1D0↓j
PAGE:0000000140721CFF 48 8B BC 24 90 00 00 00           mov     rdi_pEPROCESS, [rsp+78h+pEPROCESS]
PAGE:0000000140721D07 48 83 CB FF                       or      rbx, -1
PAGE:0000000140721D0B 45 8B CF                          mov     r9d, r15d
PAGE:0000000140721D0E 45 8B C5                          mov     r8d, r13d
PAGE:0000000140721D11 48 8B D6                          mov     rdx, rsi_pCurrentThread ; pThread
PAGE:0000000140721D14 48 8B CF                          mov     rcx, rdi_pEPROCESS ; Process
PAGE:0000000140721D17 8B 87 40 04 00 00                 mov     eax, dword ptr [rdi_pEPROCESS+_EPROCESS.UniqueProcessId]
PAGE:0000000140721D1D 66 01 9E E4 01 00 00              add     [rsi_pCurrentThread+1E4h], bx ; KTHREAD.KernelApcDisable
PAGE:0000000140721D24 90                                nop
PAGE:0000000140721D25 89 84 24 80 00 00 00              mov     [rsp+78h+Pid], eax
PAGE:0000000140721D2C E8 43 01 00 00                    call    PspTerminateProcess
PAGE:0000000140721D2C
PAGE:0000000140721D31 BA 50 73 54 65                    mov     edx, 'eTsP'     ; Tag
PAGE:0000000140721D36 48 8B CF                          mov     rcx, rdi_pEPROCESS ; Object
PAGE:0000000140721D39 8B D8                             mov     ebx, eax
PAGE:0000000140721D3B E8 40 A9 BC FF                    call    ObfDereferenceObjectWithTag
```

If we delve deeper, we find that the key function in the process of checking PPL protection is an open procedure for the process object type (*PsProcessType*) called *PspProcessOpen*. This is the only purpose of this function, which is responsible for comparing PSPROTECTION values of both processes. Before implementing PPL, the process kernel object didn't have the open procedure.

Below, you can see the process of obtaining a handle to the PPL'ed process, starting with the call of *NtOpenProcess* and ending with the actual validation of the protection.

```
NtOpenProcess(IN Pid, OUT hProcess) - Windows kernel API (ntoskrnl)
    |
    |
    |->PsOpenProcess - Process Manager's internal function
           |
           |
           |->ObOpenObjectByPointer - Object Manager API creates handle by pEPROCESS
              |
              |
              |->ObpCreateHandle - Object Manager private function creates handle
                     |
                     |
                     |->ObpIncrementHandleCountEx - _guard_dispatch_icall(PsProcessType->TypeInfo.OpenProcedure)
                            |
                            |
                            |->PspProcessOpen(EPROCESS pSourceProcess, EPROCESS pTargetProcess)
                                   |
                                   |
                                   |->PsTestProtectedProcessIncompatibility(EPROCESS pSourceProcess, EPROCESS pTargetProcess)
                                          |
                                          |
                                          |->PspCheckForInvalidAccessByProtection(PS_PROTECTION Source, PS_PROTECTION Target)
                                                 |
                                                 |
                                                 |->RtlTestProtectedAccess(PS_PROTECTION Source, PS_PROTECTION Target)
```

The process of removing PPL protection could be simplified with just zeroing EPROCESS_Protection value that is used by the Windows kernel to set the corresponding level of protection. It's related to DKOM and there are several projects on GitHub demonstrating this method. It can also be used by attackers or defenders for the opposite purpose to enable the protection for specific processes, making them inaccessible for any kind of modification. Below you can see the corresponding structures describing PPL.

```
//Win11 23H2
typedef enum _PS_PROTECTED_SIGNER
{
    PsProtectedSignerNone = 0,
    PsProtectedSignerAuthenticode = 1,
    PsProtectedSignerCodeGen = 2,
    PsProtectedSignerAntimalware = 3,
    PsProtectedSignerLsa = 4,
    PsProtectedSignerWindows = 5,
    PsProtectedSignerWinTcb = 6,
    PsProtectedSignerWinSystem = 7,
    PsProtectedSignerApp = 8,
    PsProtectedSignerMax = 9
} PS_PROTECTED_SIGNER;

typedef enum _PS_PROTECTED_TYPE
{
    PsProtectedTypeNone = 0,
    PsProtectedTypeProtectedLight = 1,
    PsProtectedTypeProtected = 2,
    PsProtectedTypeMax = 3
} PS_PROTECTED_TYPE;

typedef struct _PS_PROTECTION
{
    union
    {
        UCHAR Level;
        struct
        {
            UCHAR Type : 3; //PS_PROTECTED_TYPE
            UCHAR Audit : 1;
            UCHAR Signer : 4; //PS_PROTECTED_SIGNER
        };
    };
} PS_PROTECTION, *PPS_PROTECTION;

//Win11 23H2 offsets
typedef struct _EPROCESS
```

```
{
/*0x000*/ struct _KPROCESS Pcb;
/*0x438*/ struct _EX_PUSH_LOCK ProcessLock;
/*0x440*/ VOID*  UniqueProcessId;
/*0x448*/ struct _LIST_ENTRY ActiveProcessLinks;
/*0x458*/ struct _EX_RUNDOWN_REF RundownProtect;
...
/*0x878*/ UINT8  SignatureLevel;
/*0x879*/ UINT8  SectionSignatureLevel;
/*0x87A*/ struct _PS_PROTECTION Protection;
```

To disable PPL, the protection byte or all three fields should be set to zero (see Kernel Driver Utility, KDU).

```
PsProtection->Signer = PsProtectedSignerNone;
PsProtection->Type   = PsProtectedTypeNone;
PsProtection->Audit  = 0;
```

To enable protection for the process (EDRSandblast).

Source: http://artemonsecurity.blogspot.de/2017/04/stuxnet-drivers-detailed-analysis.html