RING 0/-2 ROOKITS : COMPROMISING DEFENSES

DEFCON 2018 USA

ALEXANDRE BORGES

PROFILE AND TOC

- Malware and Security Researcher.
- Consultant, Instructor and Speaker on Malware Analysis, Memory Analysis, Digital Forensics, Rootkits and Software Exploitation.
- Member of Digital Law and Compliance Committee (CDDC/ SP)
- Reviewer member of the The Journal of Digital Forensics, Security and Law.
- Refereer on Digital Investigation:The International Journal of Digital Forensics & Incident Response
- Instructor at Oracle, (ISC)2 and Ex-instructor at Symantec.

TOC:

ACKNOWLEDGMENT

- ✓Alex Bazhaniuk
- ✓Alex Matrosov
- ✓Andrew Furtak
- ✓Bruce Dang
- ✓Corey Kallenberg
- ✓Dmytro Oleksiuk
- ✓Engene Rodionov
- ✓Joanna Rutkowska
- ✓John Loucaides
- ✓Oleksandr Bazhaniuk
- ✓Sergey Bratus
- ✓Vicent Zimmer
- ✓Yuriy Bulygin
- ✓Xeno Kovah

These professionals deserve my sincere "thank you" and deep respect for their researches on these topics. I have learned from their explanations and articles. By the way, I continue learning...

INTRODUCTION

RING 0/-2 ROOTKITS

RING 0:

- Kernel Callback methods
- WinDbg structures
- Kernel Drivers Structures
- Malicious Drivers
- Modern C2 communication
- Kernel Pools and APCs

ADVANCED MALWARES:

- MBR/VBR/UEFI rootkits
- Tecniques used by rootkits
- Kernel Code Signing Bypasses
- MBR + IPL infection
- BIOS, UEFI and boot architecture
- Boot Guard
- Secure Boot attacks
- WSMT (Windows SMM Security
- Mitigation Table)
- BIOS Guard
- BIOS/UEFI Protections

ROOTKITS: RING 0

ROOTKITS: RING 0

- Kernel Callback Functions, which are are a kind of "modern hooks" oftenly used by antivirus programs for monitoring and alerting the kernel modules about a specific event ocurrence. Therefore, they are used by malwares (kernel drivers) for evading defenses.
- Most known callback methods are:
  - PsSetLoadImageNotifyRoutine: it provides notification when a process, library or kernel memory is mapped into memory.
  - IoRegisterFsRegistrationChange: it provides notification when a filesystem becomes available.
  - IoRegisterShutdownNotification: the driver handler (IRP_MJ_SHUTDOWN) acts when the system is about going to down.
  - KeRegisterBugCheckCallback: it helps drivers to receive a notification (for cleaning tasks) before a system crash.

ALEXANDRE BORGES - MALWARE AND SECURITY RESEARCHER

ROOTKITS: RING 0

- PsSetCreateThreadNotifyRoutine: indicates a routine that is called every time when a thread starts or ends.
- PsSetCreateProcessNotifyRoutine: when a process starts or finishes, this callback is invoked (rootkits and AVs).
- DbgSetDebugPrintCallback: it is used for capturing debug messages.
- CmRegisterCallback( ) or CmRegisterCallbackEx( ) are called by drivers to register a RegistryCallback routine, which is called every time a thread performs an operation on the registry.

- Malwares have been using this type of callbacks for checking whether their persistence entries are kept and, just in case they were removed, so the malware adds them back.

ROOTKITS: RING 0

```
0: kd> dd nt!CmpCallBackCount L1
fffff801`aa733fcc  00000002

0: kd> dps nt!CallbackListHead L2
fffff801`aa769190  ffffc000`c8d62db0
fffff801`aa769198  ffffc000`c932c8b0

0: kd> dt nt!_LIST_ENTRY ffffc000`c8d62db0
 [ 0xffffc000`c932c8b0 - 0xfffff801`aa769190 ]
   +0x000 Flink            : 0xffffc000`c932c8b0 _LIST_ENTRY [ 0xfffff801`aa769190 - 0xffffc000`c8d62db0 ]
   +0x008 Blink            : 0xfffff801`aa769190 _LIST_ENTRY [ 0xffffc000`c8d62db0 - 0xffffc000`c932c8b0 ]
```

ROOTKITS: RING 0

```
0: kd> !list -t _LIST_ENTRY.Flink -x "dps" -a "L8" 0xffffc000`c932c8b0

ffffc000`c932c8b0  fffff801`aa769190 nt!CallbackListHead
.....
ffffc000`c932c8c8  01d3c3ba`27edfc12
ffffc000`c932c8d0  fffff801`6992a798 vsdatant+0x67798
ffffc000`c932c8d8  fffff801`69951a68 vsdatant+0x8ea68
ffffc000`c932c8e0  00000000`000a000a
.....
fffff801`aa7691c0  00000000`bee0bee0
fffff801`aa7691c8  fffff801`aa99b600 nt!HvpGetCellFlat
```

ALEXANDRE BORGES - MALWARE AND SECURITY RESEARCHER

ROOTKITS: RING 0

- At same way, PsSetCreateProcessNotifyRoutine( ) routine adds a driver-supplied callback routine to, or removes it from, a list of routines to be called whenever a process is created or deleted.

```
0: kd> dd nt!PspCreateProcessNotifyRoutineCount L1

fffff801`aab3f668  00000009

0: kd> .for (r $t0=0; $t0 < 9; r $t0=$t0+1) { r $t1=poi($t0 * 8 + nt!PspCreateProcessNotifyRoutine); .if ($t1 == 0) { .continue }; r $t1 = $t1 & 0xFFFFFFFFFFFFFFF0; dps $t1+8 L1;}
```

- Malwares composed by kernel drivers, which use the PsSetLegoNotifyRoutine( ) kernel callback to register a malicious routine that is called during the thread termination. The KTHREAD.LegoData field provides the direct address.

ROOTKITS: RING 0

```
0: kd> .for (r $t0=0; $t0 < 9; r $t0=$t0+1) { r $t1=poi($t0 * 8 + nt!PspCreateProcessNotifyRoutine); .if ($t1 == 0) { .continue }; r $t1 = $t1 & 0xFFFFFFFFFFFFFFF0; dps $t1+8 L1;}
ffffe001`134c8b08  fffff801`aa5839c4  nt!ViCreateProcessCallback
ffffe001`139e1138  fffff801`678175f0  cng!CngCreateProcessNotifyRoutine
ffffe001`13b43138  fffff801`67e6c610  kl1+0x414610
ffffe001`13bdb268  fffff801`685d1138  PGPfsfd+0x1c138
ffffe001`13b96858  fffff801`68a53000  ksecdd!KsecCreateProcessNotifyRoutine
ffffe001`14eeacc8  fffff801`68d40ec0  tcpip!CreateProcessNotifyRoutineEx
ffffe001`164ffce8  fffff801`67583c70  CI!I_PEProcessNotify
ffffe001`13b6e4b8  fffff801`68224a38  klflt!PstUnregisterProcess+0xfac
ffffe001`1653e4d8  fffff801`699512c0  vsdatant+0x8e2c0
```

ALEXANDRE BORGES - MALWARE AND SECURITY RESEARCHER

ROOTKITS: RING 0

kd> dt _KTHREAD

```
+0x288 SchedulerApcFill1 : [3] UChar
+0x28b QuantumReset      : UChar
+0x288 SchedulerApcFill2 : [4] UChar
+0x28c KernelTime        : Uint4B
+0x288 SchedulerApcFill3 : [64] UChar
+0x2c8 WaitPrcb          : Ptr64 _KPRCB
+0x288 SchedulerApcFill4 : [72] UChar
+0x2d0 LegoData          : Ptr64 Void
+0x288 SchedulerApcFill5 : [83] UChar
+0x2db CallbackNestingLevel : UChar
+0x2dc UserTime          : Uint4B
+0x2e0 SuspendEvent      : _KEVENT
+0x2f8 ThreadListEntry   : _LIST_ENTRY
+0x308 MutantListHead    : _LIST_ENTRY
+0x318 LockEntriesFreeList : _SINGLE_LIST_ENTRY
+0x320 LockEntries       : [6] _KLOCK_ENTRY
```

By now, we have seen malwares using KTHREAD.LegoData field for registering a malicious routine, which would be called during the thread termination.

ROOTKITS: RING 0

- ✓ Windows offers different types of drivers such as legacy drivers, filter drivers and minifilter drivers (malwares can be written using any one these types), which could be developed using WDM or WDF frameworks (of course, UMDF and KMDF take part)
- To analyze a malicious driver, remember this sequence of events:
  - The driver image is mapped into the kernel memory address space.
  - An associated driver object is created and registered with Object Manager, which calls the entry point and fills the DRIVER_OBJECT structure's fields.

DRIVER → DEVICE_OBJECT → DRIVER_OBJECT → HARDWARE RES.

(one or more objects)

I/O CreateDevice( )

ROOTKITS: RING 0

- Most ring 0 malwares install filter drivers for:
  - modifying aspects and behavior of existing drivers
  - filtering results of operations (reading file, for example)
  - adding new malicious features to a driver/devices (for example, keyloggers)

- Oftenly found in filter drivers (mainly the malicious one) for intercepting and altering data, a driver can easily “attach” (using IoAttachDevice( )) one device object to another device object (similar to a “pipeline) to receive I/O requests (see next slide).
- The AddDevice( ) routine is used to create an unamed Device Object and to attach it to a named Device Object (ex: aborges) from a layered driver (lower-level driver).

ROOTKITS: RING 0

- Each IRP will be processed by a dispatch routine, which is picked up from its MajorFunction Table.

- The correct dispatch routine will be called to handle the request, picking the IRP parameters from the own IO_STACK_LOCATION by calling the IoGetCurrentIrpStackLocation( ) routine.

- Additionally, these IRP parameters could be passed to the next IO_STACK_LOCATION by using the IoCopyCurrentIrpStackLocation( ) routine or even to the next driver by calling IoSkipCurrentStackLocation( ) routine.

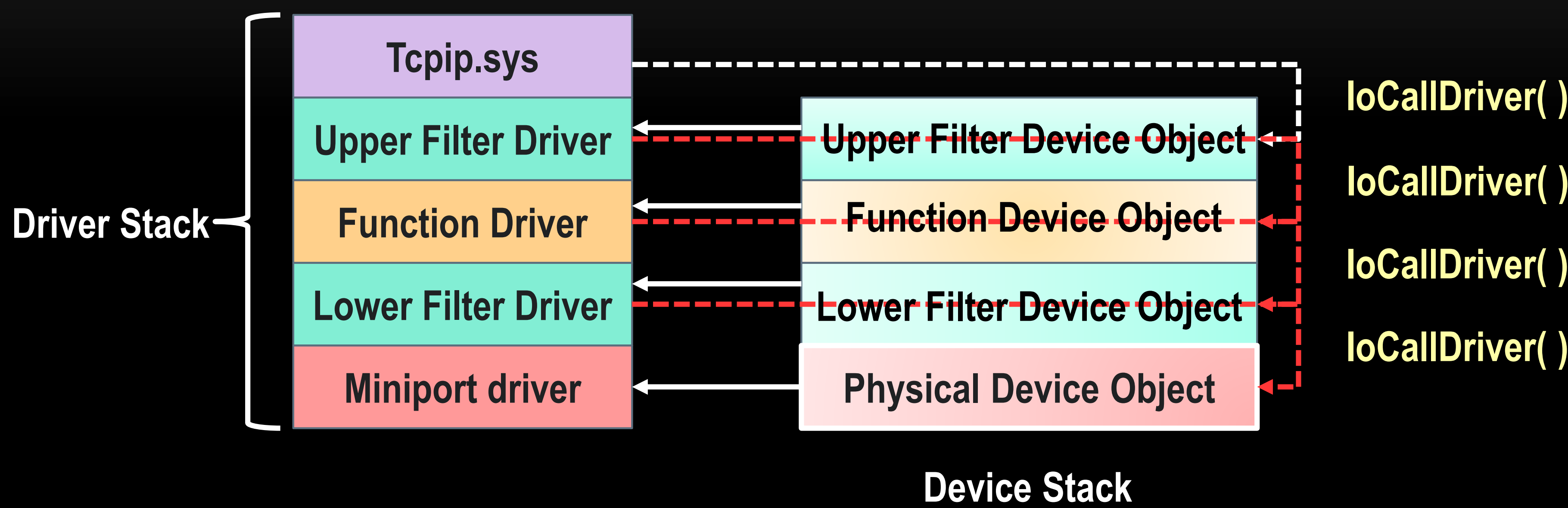
ROOTKITS: RING 0

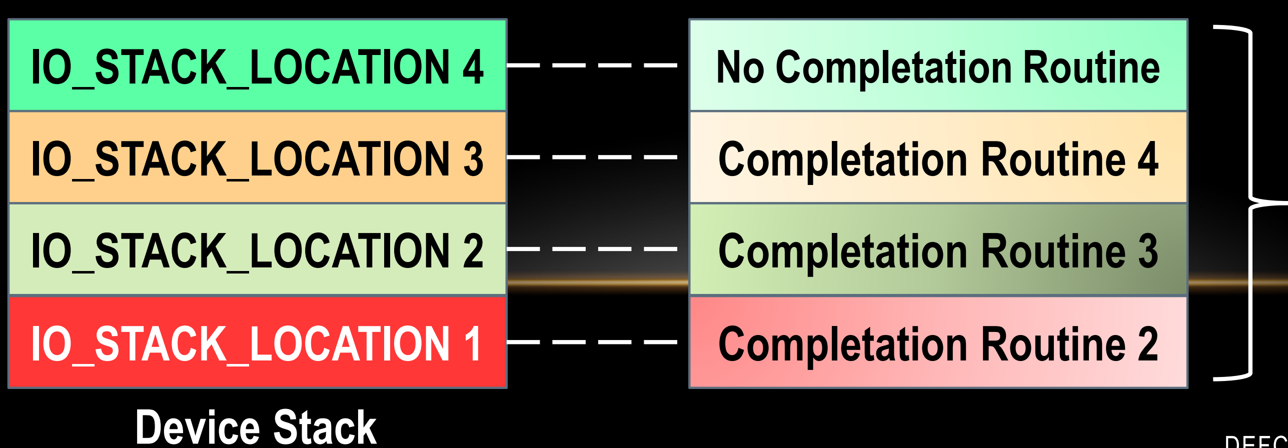
- Alternatively, this IRP could be passed down to the layered driver by using function such as IoCallDriver( ).

- Usually, rootkits use the same IoCallDriver( ) to send directly request to the filesystem driver, evading any kind of monitoring or hooking at middle of the path. ☺

ALEXANDRE BORGES - MALWARE AND SECURITY RESEARCHER

ROOTKITS: RING 0





The IoCompleteRequest( ) manages calling these routines in the correct order (bottom-up). ☺

ROOTKITS: RING 0

STATIC

```
0: kd> dt nt!_IRP
   +0x000 Type             : Int2B
   +0x002 Size             : Uint2B
   +0x004 AllocationProcessorNumber : Uint2B
   +0x006 Reserved         : Uint2B
   +0x008 MdlAddress       : Ptr64 _MDL
   +0x010 Flags            : Uint4B
   +0x018 AssociatedIrp    : <unnamed-tag>
   +0x020 ThreadListEntry  : _LIST_ENTRY
   +0x030 IoStatus         : _IO_STATUS_BLOCK
   +0x040 RequestorMode    : Char
   +0x041 PendingReturned  : UChar
   +0x042 StackCount       : Char
   +0x043 CurrentLocation  : Char
   +0x044 Cancel           : UChar
   +0x045 CancelIrql       : UChar
   +0x046 ApcEnvironment   : Char
   +0x047 AllocationFlags  : UChar
   +0x048 UserIosb         : Ptr64 _IO_STATUS_BLOCK
   +0x050 UserEvent        : Ptr64 _KEVENT
   +0x058 Overlay          : <unnamed-tag>
   +0x068 CancelRoutine    : Ptr64     void
   +0x070 UserBuffer       : Ptr64 Void
   +0x078 Tail             : <unnamed-tag>
```

DYNAMIC

IO_STACK_LOCATION

IO_STACK_LOCATION

IO_STACK_LOCATION

....

- A IRP is usually generated by the I/O Manager in response to requests.
- An IRP can be generated by drivers through the IoAllocateIrp( ) function.
- Analyzing malware, we are usually verify functions such as IoGetCurrentIrpStackLocation(), IoGetNextIrpStackLocation( ) and IoSkipCurrentIrpStackLocation( ).
- At end, each device holds the responsability to prepare the IO_STACK_LOCATION to the next level, as well a driver could call the IoSetCompletationRoutine( ) to set a completation routine up at CompletationRoutine field.

```
0: kd> dt nt!_IO_STACK_LOCATION
   +0x000 MajorFunction    : UChar
   +0x001 MinorFunction    : UChar
   +0x002 Flags            : UChar
   +0x003 Control          : UChar
   +0x008 Parameters       : <unnamed-tag>
   +0x028 DeviceObject     : Ptr64 _DEVICE_OBJECT
   +0x030 FileObject       : Ptr64 _FILE_OBJECT
   +0x038 CompletionRoutine : Ptr64     long
   +0x040 Context          : Ptr64 Void
```

ALEXANDRE BORGES - MALWARE AND SECURITY RESEARCHER

ROOTKITS: RING 0

```
0: kd> dt nt!_IO_STACK_LOCATION Parameters.
   +0x008 Parameters  :
      +0x000 Create       : <unnamed-tag>
      +0x000 CreatePipe   : <unnamed-tag>
      +0x000 CreateMailslot : <unnamed-tag>
      +0x000 Read         : <unnamed-tag>
      +0x000 Write        : <unnamed-tag>
      +0x000 QueryDirectory : <unnamed-tag>
      +0x000 NotifyDirectory : <unnamed-tag>
      +0x000 QueryFile    : <unnamed-tag>
      +0x000 SetFile      : <unnamed-tag>
      +0x000 QueryEa      : <unnamed-tag>
      +0x000 SetEa        : <unnamed-tag>
      +0x000 QueryVolume  : <unnamed-tag>
      +0x000 SetVolume    : <unnamed-tag>
      +0x000 FileSystemControl : <unnamed-tag>
      +0x000 LockControl  : <unnamed-tag>
      +0x000 DeviceIoControl : <unnamed-tag>
      +0x000 QuerySecurity : <unnamed-tag>
      +0x000 SetSecurity  : <unnamed-tag>
      +0x000 MountVolume  : <unnamed-tag>
      +0x000 VerifyVolume : <unnamed-tag>
```

Parameters field depends on the major and minor functions!

ROOTKITS: RING 0

Parameter field depends on **major and minor function number.** Thus, the IRPs being used are related to the action.

MEMBER NAME	IRPs that use this member
Create	IRP_MJ_CREATE
Read	IRP_MJ_READ
Write	IRP_MJ_WRITE
QueryFile	IRP_MJ_QUERY_INFORMATION
SetFile	IRP_MJ_SET_INFORMATION
QueryVolume	IRP_MJ_QUERY_VOLUME_INFORMATION
DeviceIoControl	IRP_MJ_DEVICE_CONTROL and IRP_MJ_INTERNAL_DEVICE_CONTROL
MountVolume	IRP_MN_MOUNT_VOLUME
VerifyVolume	IRP_MN_VERIFY_VOLUME
Scsi	IRP_MJ_INTERNAL_DEVICE_CONTROL (SCSI)
QueryDeviceRelations	IRP_MN_QUERY_DEVICE_RELATIONS
QueryInterface	IRP_MN_QUERY_INTERFACE
DeviceCapabilities	IRP_MN_QUERY_CAPABILITIES
FilterResourceRequirements	IRP_MN_FILTER_RESOURCE_REQUIREMENTS
ReadWriteConfig	IRP_MN_READ_CONFIG and IRP_MN_WRITE_CONFIG
SetLock	IRP_MN_SET_LOCK
QueryId	IRP_MN_QUERY_ID
QueryDeviceText	IRP_MN_QUERY_DEVICE_TEXT
UsageNotification	IRP_MN_DEVICE_USAGE_NOTIFICATION
WaitWake	IRP_MN_WAIT_WAKE
PowerSequence	IRP_MN_POWER_SEQUENCE
Power	IRP_MN_SET_POWER and IRP_MN_QUERY_POWER
StartDevice	IRP_MN_START_DEVICE
WMI	WMI minor IRPs

ROOTKITS: RING 0

```
0: kd> dt nt!_IO_STACK_LOCATION
   +0x000 MajorFunction     : UChar
   +0x001 MinorFunction     : UChar
   +0x002 Flags             : UChar
   +0x003 Control           : UChar
   +0x008 Parameters        : <unnamed-tag>
   +0x028 DeviceObject      : Ptr64 _DEVICE_OBJECT
   +0x030 FileObject        : Ptr64 _FILE_OBJECT
   +0x038 CompletionRoutine : Ptr64     long
   +0x040 Context           : Ptr64 Void
```

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

```
0: kd> dt nt!_DEVICE_OBJECT
   +0x000 Type              : Int2B
   +0x002 Size              : Uint2B
   +0x004 ReferenceCount    : Int4B
   +0x008 DriverObject      : Ptr64 _DRIVER_OBJECT
   +0x010 NextDevice        : Ptr64 _DEVICE_OBJECT
   +0x018 AttachedDevice    : Ptr64 _DEVICE_OBJECT
   +0x020 CurrentIrp        : Ptr64 _IRP
   +0x028 Timer             : Ptr64 _IO_TIMER
   +0x030 Flags             : Uint4B
   +0x034 Characteristics   : Uint4B
   +0x038 Vpb               : Ptr64 _VPB
   +0x040 DeviceExtension   : Ptr64 Void
   +0x048 DeviceType        : Uint4B
   +0x04c StackSize         : Char
   +0x050 Queue             : <unnamed-tag>
   +0x098 AlignmentRequirement : Uint4B
   +0x0a0 DeviceQueue       : _KDEVICE_QUEUE
   +0x0c8 Dpc               : _KDPC
```

ROOTKITS: RING 0

Malicious driver

```
kd> lm Dvm aborges
Browse full module list
start    end        module name
9a3c3000 9a3ca000   aborges        (no symbols)
    Loaded symbol image file: aborges.sys
    Image path: \SystemRoot\system32\drivers\aborges.sys
    Image name: aborges.sys
    Browse all global symbols  functions  data
    Timestamp:        Thu Feb 28 22:28:14 2013 (5130042E)
    CheckSum:         0000E646
    ImageSize:        00007000
    Translations:     0000.04b0 0000.04e4 0409.04b0 0409.04e4


kd> !object \driver\aborges
Object: 86862c60  Type: (851ea6e0) Driver
    ObjectHeader: 86862c48 (new version)
    HandleCount: 0  PointerCount: 15
    Directory Object: 8a252f50  Name:


kd> !drvobj \driver\aborges
Driver object (86862c60) is for:

Driver Extension List: (id , addr)

Device Object list:
85212888  85212a80  85212bb8  85214958
8640ac98  863c7860  86455bd0  8645b8e8
865d3d98  863faef8  86451900  868339f8
8683fd98
```

ROOTKITS: RING 0

```
kd> dt _DRIVER_OBJECT 86862c60
nt!_DRIVER_OBJECT
   +0x000 Type             : 0n4
   +0x002 Size             : 0n168
   +0x004 DeviceObject     : 0x85212888 _DEVICE_OBJECT
   +0x008 Flags            : 0x12
   +0x00c DriverStart      : 0x9a3c3000 Void
   +0x010 DriverSize       : 0x7000
   +0x014 DriverSection    : 0x86839ea8 Void
   +0x018 DriverExtension  : 0x86862d08 _DRIVER_EXTENSION
   +0x01c DriverName       : _UNICODE_STRING "\Driver\aborges"
   +0x024 HardwareDatabase : 0x82d8a270 _UNICODE_STRING
 "\REGISTRY\MACHINE\HARDWARE\DESCRIPTION\SYSTEM"
   +0x028 FastIoDispatch   : (null)
   +0x02c DriverInit       : 0x9a3c8f05     long  +0
   +0x030 DriverStartIo    : (null)
   +0x034 DriverUnload     : 0x9a3c3b36     void  +0
   +0x038 MajorFunction    : [28] 0x9a3c4f90     long  +0
```

ALEXANDRE BORGES - MALWARE AND SECURITY RESEARCHER

ROOTKITS: RING 0

```
kd> !drvobj 86862c60 3
Driver object (86862c60) is for:
 \Driver\aborges
Driver Extension List: (id , addr)

Device Object list:
85212888  85212a80  85212bb8  85214958
8640ac98  863c7860  86455bd0  8645b8e8
865d3d98  863faef8  86451900  868339f8
8683fd98

DriverEntry:   9a3c8f05    aborges
DriverStartIo: 00000000
DriverUnload:  9a3c3b36    aborges
AddDevice:     00000000

Dispatch routines:
[00] IRP_MJ_CREATE                     9a3c4f90    aborges+0x1f90
[01] IRP_MJ_CREATE_NAMED_PIPE          82aca0bf    nt!IopInvalidDeviceRequest
[02] IRP_MJ_CLOSE                      9a3c4e38    aborges+0x1e38
[03] IRP_MJ_READ                       9a3c5540    aborges+0x2540
[04] IRP_MJ_WRITE                      9a3c6290    aborges+0x3290
[05] IRP_MJ_QUERY_INFORMATION          82aca0bf    nt!IopInvalidDeviceRequest
[06] IRP_MJ_SET_INFORMATION            82aca0bf    nt!IopInvalidDeviceRequest
[07] IRP_MJ_QUERY_EA                   82aca0bf    nt!IopInvalidDeviceRequest
[08] IRP_MJ_SET_EA                     82aca0bf    nt!IopInvalidDeviceRequest
[09] IRP_MJ_FLUSH_BUFFERS              82aca0bf    nt!IopInvalidDeviceRequest
[0a] IRP_MJ_QUERY_VOLUME_INFORMATION   82aca0bf    nt!IopInvalidDeviceRequest
[0b] IRP_MJ_SET_VOLUME_INFORMATION     82aca0bf    nt!IopInvalidDeviceRequest
[0c] IRP_MJ_DIRECTORY_CONTROL          82aca0bf    nt!IopInvalidDeviceRequest
[0d] IRP_MJ_FILE_SYSTEM_CONTROL        82aca0bf    nt!IopInvalidDeviceRequest
[0e] IRP_MJ_DEVICE_CONTROL             9a3c3c82    aborges+0xc82
```

ROOTKITS: RING 0

```
kd> dt _DRIVER_OBJECT
nt!_DRIVER_OBJECT
   +0x000 Type             : Int2B
   +0x002 Size             : Int2B
   +0x004 DeviceObject     : Ptr32 _DEVICE_OBJECT
   +0x008 Flags            : Uint4B
   +0x00c DriverStart      : Ptr32 Void
   +0x010 DriverSize       : Uint4B
   +0x014 DriverSection    : Ptr32 Void
   +0x018 DriverExtension  : Ptr32 _DRIVER_EXTENSION
   +0x01c DriverName       : _UNICODE_STRING
   +0x024 HardwareDatabase : Ptr32 _UNICODE_STRING
   +0x028 FastIoDispatch   : Ptr32 _FAST_IO_DISPATCH
   +0x02c DriverInit       : Ptr32     long
   +0x030 DriverStartIo    : Ptr32     void
   +0x034 DriverUnload     : Ptr32     void
   +0x038 MajorFunction    : [28] Ptr32     long
```

```
mov   eax, [ebp+DriverObject]
mov   dword ptr [eax+38h], offset sub_12668
mov   ecx, [ebp+DriverObject]
mov   dword ptr [ecx+40h], offset sub_12760
mov   edx, [ebp+DriverObject]
mov   dword ptr [edx+44h], offset sub_12760
mov   eax, [ebp+DriverObject]
mov   dword ptr [eax+48h], offset sub_12760
mov   ecx, [ebp+DriverObject]
mov   dword ptr [ecx+70h], offset sub_12978
mov   eax, [ebp+DriverObject]
mov   dword ptr [eax+34h], offset sub_12606
```

```
Dispatch routines:
[00] IRP_MJ_CREATE                        9a3c4f90   aborges+0x1f90
[01] IRP_MJ_CREATE_NAMED_PIPE             82aca0bf   nt!IopInvalidDeviceRequest
[02] IRP_MJ_CLOSE                         9a3c4e38   aborges+0x1e38
[03] IRP_MJ_READ                          9a3c5540   aborges+0x2540
[04] IRP_MJ_WRITE                         9a3c6290   aborges+0x3290
[05] IRP_MJ_QUERY_INFORMATION             82aca0bf   nt!IopInvalidDeviceRequest
[06] IRP_MJ_SET_INFORMATION               82aca0bf   nt!IopInvalidDeviceRequest
[07] IRP_MJ_QUERY_EA                      82aca0bf   nt!IopInvalidDeviceRequest
[08] IRP_MJ_SET_EA                        82aca0bf   nt!IopInvalidDeviceRequest
[09] IRP_MJ_FLUSH_BUFFERS                 82aca0bf   nt!IopInvalidDeviceRequest
[0a] IRP_MJ_QUERY_VOLUME_INFORMATION      82aca0bf   nt!IopInvalidDeviceRequest
[0b] IRP_MJ_SET_VOLUME_INFORMATION        82aca0bf   nt!IopInvalidDeviceRequest
[0c] IRP_MJ_DIRECTORY_CONTROL             82aca0bf   nt!IopInvalidDeviceRequest
[0d] IRP_MJ_FILE_SYSTEM_CONTROL           82aca0bf   nt!IopInvalidDeviceRequest
[0e] IRP_MJ_DEVICE_CONTROL                9a3c3c82   aborges+0xc82
```

ROOTKITS: RING 0

```
kd> !fltkd.filters

Filter List: 85a88754 "Frame 0"
   FLT_FILTER: 86df0008 "abftldrv" "135000"
      FLT_INSTANCE: 86df4008 "abftldrv" "135000"
   FLT_FILTER: 85b56560 "FileInfo" "45000"
      FLT_INSTANCE: 85bb10b8 "FileInfo" "45000"
      FLT_INSTANCE: 85c74430 "FileInfo" "45000"
      FLT_INSTANCE: 85d71008 "FileInfo" "45000"
      FLT_INSTANCE: 85d92950 "FileInfo" "45000"


kd> dt _FLT_FILTER 86df0008
fltmgr!_FLT_FILTER
   +0x000 Base             : _FLT_OBJECT
   +0x014 Frame            : 0x85a886f8 _FLTP_FRAME
   +0x018 Name             : _UNICODE_STRING "abftldrv"
   +0x020 DefaultAltitude  : _UNICODE_STRING "135000"
   +0x028 Flags            : 6 (No matching name)
   +0x02c DriverObject     : 0x86ded938 _DRIVER_OBJECT
   +0x030 InstanceList     : _FLT_RESOURCE_LIST_HEAD
   +0x074 VerifierExtension : (null)
   +0x078 VerifiedFiltersLink : _LIST_ENTRY [ 0x0 - 0x0 ]
   +0x080 FilterUnload     :     (null)
   +0x084 InstanceSetup    :     0x8f5e663c     long  abftldrv!AbftldrvInstanceSetup+0
   +0x088 InstanceQueryTeardown : (null)
   +0x08c InstanceTeardownStart : (null)
   +0x090 InstanceTeardownComplete : (null)
   +0x094 SupportedContextsListHead : 0x86deed50 _ALLOCATE_CONTEXT_HEADER
   +0x098 SupportedContexts :     [6] (null)
   +0x0b0 PreVolumeMount   :     0x8f5da0cc     _FLT_PREOP_CALLBACK_STATUS  abftldrv!AbftldrvPreRedirect+0
   +0x0b4 PostVolumeMount  :     (null)
   +0x0b8 GenerateFileName :     0x8f5e28fa     long  abftldrv!AbftldrvGenerateFileName+0
   +0x0bc NormalizeNameComponent : (null)
   +0x0c0 NormalizeNameComponentEx : 0x8f5e29b2     long  abftldrv!AbftldrvNormalizeNameComponentEx+0
   +0x0c4 NormalizeContextCleanup : (null)
   +0x0c8 KtmNotification  : (null)
   +0x0cc Operations       : 0x86df0164 _FLT_OPERATION_REGISTRATION
   +0x0d0 OldDriverUnload  : (null)
```

ALEXANDRE BORGES - MALWARE AND SECURITY RESEARCHER

ROOTKITS: RING 0

```
kd> dx -id 0,0,85a88754 -r1 ((fltmgr!_FLT_OPERATION_REGISTRATION *) 0x86df0164)
((fltmgr!_FLT_OPERATION_REGISTRATION *) 0x86df0164)
: 0x86df0164 [Type: _FLT_OPERATION_REGISTRATION *]
    [+0x000] MajorFunction    : 0xec [Type: unsigned char]
    [+0x004] Flags            : 0x0 [Type: unsigned long]
    [+0x008] PreOperation     : 0x8f5da0cc [Type: _FLT_PREOP_CALLBACK_STATUS
(*) (_FLT_CALLBACK_DATA *,_FLT_RELATED_OBJECTS *,void * *)]
    [+0x00c] PostOperation    : 0x0 [Type: _FLT_POSTOP_CALLBACK_STATUS
(*) (_FLT_CALLBACK_DATA *,_FLT_RELATED_OBJECTS *,void *,unsigned long)]
    [+0x010] Reserved1        : 0x0 [Type: void *]

kd> u 0x8f5da0cc
abftldrv!AbftldrvPreRedirect:
8f5da0cc 8bff            mov     edi,edi
8f5da0ce 55              push    ebp
8f5da0cf 8bec            mov     ebp,esp
8f5da0d1 8b450c          mov     eax,dword ptr [ebp+0Ch]
8f5da0d4 8b4010          mov     eax,dword ptr [eax+10h]
8f5da0d7 85c0            test    eax,eax
8f5da0d9 7411            je      abftldrv!AbftldrvPreRedirect+0x20 (8f5da0ec)
8f5da0db 8b400c          mov     eax,dword ptr [eax+0Ch]
```

ROOTKITS: RING 0

- **Naturally, as closest at bottom of device stack occurs the infection (SCSI miniport drivers instead of targeting File System Drivers), so more efficient it is.**

- **Nowadays, most monitoring tools try to detect strange activities at upper layers.**

- **Malwares try to intercept requests (read / write operations) from hard disk by manipulating the MajorFunction array (IRP_MJ_DEVICE_CONTROL and IRP_INTERNAL_CONTROL) of the DRIVER_OBJECT structure. ☺**

ROOTKITS: RING 0

- Rootkits try to protect itself from being removed by modifying routines such as IRP_MJ DEVICE_CONTROL and hooking requests going to the disk (IOCTL_ATA_* and IOCTL_SCSI_*).

- Another easy approach is to hook the DriverUnload( ) routine for preventing the rootkit of being unloaded.

- However, any used tricks must avoid touching critical areas protected by KPP (Kernel Patch Guard) and one of tricky methods for find which are those areas is trying the following:

ROOTKITS: RING 0

Thanks, Alex Ionescu ☺

kd> !analyze –show 109

```
0    : A generic data region
1    : Modification of a function or .pdata
2    : A processor IDT
3    : A processor GDT
4    : Type 1 process list corruption
5    : Type 2 process list corruption
6    : Debug routine modification
7    : Critical MSR modification
8    : Object type
9    : A processor IVT
a    : Modification of a system service function
b    : A generic session data region
c    : Modification of a session function or .pdata
d    : Modification of an import table
e    : Modification of a session import table
f    : Ps Win32 callout modification
10   : Debug switch routine modification
11   : IRP allocator modification
12   : Driver call dispatcher modification
13   : IRP completion dispatcher modification
14   : IRP deallocator modification
```

ROOTKITS: RING 0

- **Most time, malwares have allocated a kind of hidden filesytem in free sectors to store configuration files and they are referred by random device object names generated during the boot.**

- **Few authors of ring 0 malwares are careless because they write malicious drivers that provide access to shared user-mode buffers using Neither method (METHOD_NEITHER), without any data validation, exposing it to memory corruption and, most time, leakage of information. Ridiculous. ☺**

ROOTKITS: RING 0

- Additionally, malwares composed by executable + drivers have been using APLC (Advanced Local Procedure Call) in the communication between user mode code and kernel drivers instead of using only IOCTL commands.

- Remember APLC interprocess-communication technique has been used since Windows Vista, as between lsass.exe and SRM(Security Reference Monitor). Most analysts are not used to seeing this approach.

- Malwares do not choose an specific driver during the boot for injection, but try to randomly pick up a driver by parsing structures such as _KLDR_DATA_TABLE_ENTRY.

ROOTKITS: RING 0

- Certainly, hooking the filesystem driver access is always a possible alternative:
  - IoCreateFile( ) → gets a handle to the filesystem.
  - ObReferenceObjectByHandle( ) → gets a pointer to FILE_OBJECT represented by the handle.
  - IoCreateDevice( ) → creates a device object (DEVICE_OBJECT) for use by a driver.
  - IoGetRelatedDeviceObject( ) → gets a pointer to DEVICE_OBJECT.
  - IoAttachDeviceToDeviceStack( ) → creates a new device object and attaches it to DEVICE_OBJECT pointer (previous function).

ROOTKITS: RING 0

- As it is done by AVs, malwares also hook functions such as ZwCreate( ) for intercepting all opened requests sent to devices. ☺
- After infecting a system by dropping kernel drivers, malwares usually force the system reboot calling ZwRaiseHardError( ) function and specifying OptionShutdownSystem as 5th parameter.
- Of course, it could be worse and the malware could use IoRegisterShutdownNotification( ) routine registers the driver to receive an IRP_MJ_SHUTDOWN IRP notification when the system is shutdown for restoring the malicious driver in the next boot just in case it is necessary.

ROOTKITS: RING 0

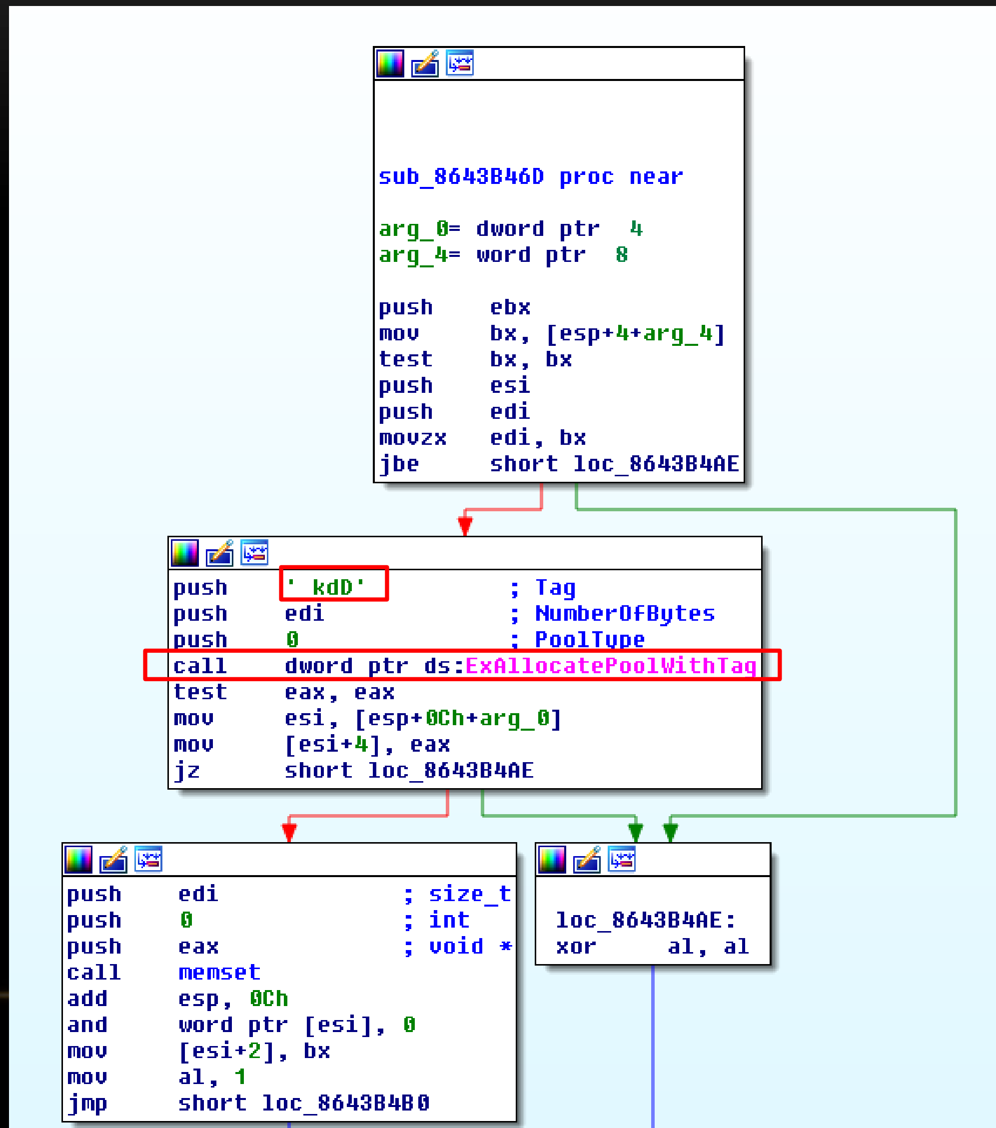
- Malwares continue allocating (usually RWX, although on Windows 8+ it could specify NonPagePoolNX) and marking their pages by using ExAllocatePoolWithTag( ) function (and other at same family ExAllocatePool*). Fortunately, it can be easily found by using memory analysis:

```
root@kali:~# more /root/volatility26/volatility/plugins/rootkitscanner.py
import volatility.poolscan as poolscan
import volatility.plugins.common as common
import volatility.utils as utils
import volatility.obj as obj

class RootkitPoolScanner(poolscan.SinglePoolScanner):
    """Configurable pool scanner"""

    checks = [
            # Replace XXXX with the 4-byte tag you're trying to find
            ('PoolTagCheck', dict(tag = "Ddk ")),
            # Replace > 0 with a size comparison test (i.e. >= 40, < 1000)
            ('CheckPoolSize', dict(condition = lambda x : x > 0)),
            # Assign a value of False or True depending on the desired allocations
            ('CheckPoolType', dict(paged = False, non_paged = True)),
            ]
```

ROOTKITS: RING 0



ROOTKITS: RING 0

```
kd> !poolfind Driv

Scanning large pool allocation table for tag 0x76697244 (Driv) (86711000 :
86911000)

85fce408 : tag Driv (Protected), size      0xf0, Nonpaged pool
85fd2158 : tag Driv, size     0x1b0, Nonpaged pool
85fd2470 : tag Driv (Protected), size      0xf0, Nonpaged pool
85fd0e50 : tag Driv, size     0x1b0, Nonpaged pool
85fa8698 : tag Driv (Protected), size      0xf0, Nonpaged pool
85fd5140 : tag Driv, size      0x10, Nonpaged pool
85fd5e50 : tag Driv, size     0x1b0, Nonpaged pool
8655e658 : tag Driv (Protected), size      0xf0, Nonpaged pool
85febb98 : tag Driv (Protected), size      0xf0, Nonpaged pool
85f911c8 : tag Driv (Protected), size      0xf0, Nonpaged pool
85f931e8 : tag Driv (Protected), size      0xf0, Nonpaged pool
85fbd248 : tag Driv, size     0x1b0, Nonpaged pool
85fbdb00 : tag Driv (Protected), size      0xf0, Nonpaged pool
85fc9800 : tag Driv (Protected), size      0xf0, Nonpaged pool
853e0540 : tag Driv (Protected), size      0xf0, Nonpaged pool
```

ROOTKITS: RING 0

0: kd> dt nt!_KTHREAD

```
+0x088 FirstArgument      : Ptr64 Void
+0x090 TrapFrame          : Ptr64 _KTRAP_FRAME
+0x098 ApcState           : _KAPC_STATE
+0x098 ApcStateFill       : [43] UChar
+0x0c3 Priority           : Char
+0x0c4 UserIdealProcessor : Uint4B
```

```
0: kd> dt _KAPC_STATE
ntdll!_KAPC_STATE
   +0x000 ApcListHead         : [2] _LIST_ENTRY
   +0x020 Process             : Ptr64 _KPROCESS
   +0x028 InProgressFlags     : UChar
   +0x028 KernelApcInProgress : Pos 0, 1 Bit
   +0x028 SpecialApcInProgress : Pos 1, 1 Bit
   +0x029 KernelApcPending    : UChar
   +0x02a UserApcPending      : UChar
```

- APC (user and kernel mode) are executed in the thread context, where normal APC executes at PASSIVE_LEVEL (thread is on alertable state) and special ones at APC_LEVEL (software interruption below DISPATCH LEVEL, where run Dispatch Procedure Calls).

- APC Injection → It allows a program to execute a code in a specific thread by attaching to an APC queue (without using the CreateRemoteThread( )) and preempting this thread in alertable state to run the malicious code. (QueueUserAPC( ), KeInitializeAPC( ) and KeInsertQueueAPC( )).

ADVANCED MALWARES AND ROOTKITS RING -2

ADVANCED MALWARES

- MBR rootkits: Petya and TLD4 (both in bootstrap code), Omasco (partition table) and Mebromi (MBR + BIOS, triggering SW System Management Interrupt (SMI) 0x29/0x2F for erasing the SPI flash)
- VBR rootkits: Rovnix (IPL) and Gapz (BPB – Bios Parameter Block, which it is specific for the filesystem)
- UEFI rootkits: replaces EFI boot loaders and, in some cases, they also install custom firmware executable (EFI DXE)
- Modern malwares alter the BPB (BIOS parameter block), which describes the filesystem volume, in the VBR.
- We should remember that a rough overview of a disk design is: MBR → VBR → IPL → NTFS

Locate the active partition and reads the first sector

It contains necessary boot code for loading the OS loader

Initial Program Loader. It has 15 sectors containing the bootstrap code for parsing the NTFS and locating the OS boot loader.

ADVANCED MALWARES

Boot Sector NTFS, Base Offset: 0

Offset	Title	Value
0	JMP instruction	EB 52 90
3	File system ID	NTFS
B	Bytes per sector	512
D	Sectors per cluster	8
E	Reserved sectors	0
10	(always zero)	00 00 00
13	(unused)	00 00
15	Media descriptor	F8
16	(unused)	00 00
18	Sectors per track	63
1A	Heads	255
1C	Hidden sectors	206,848

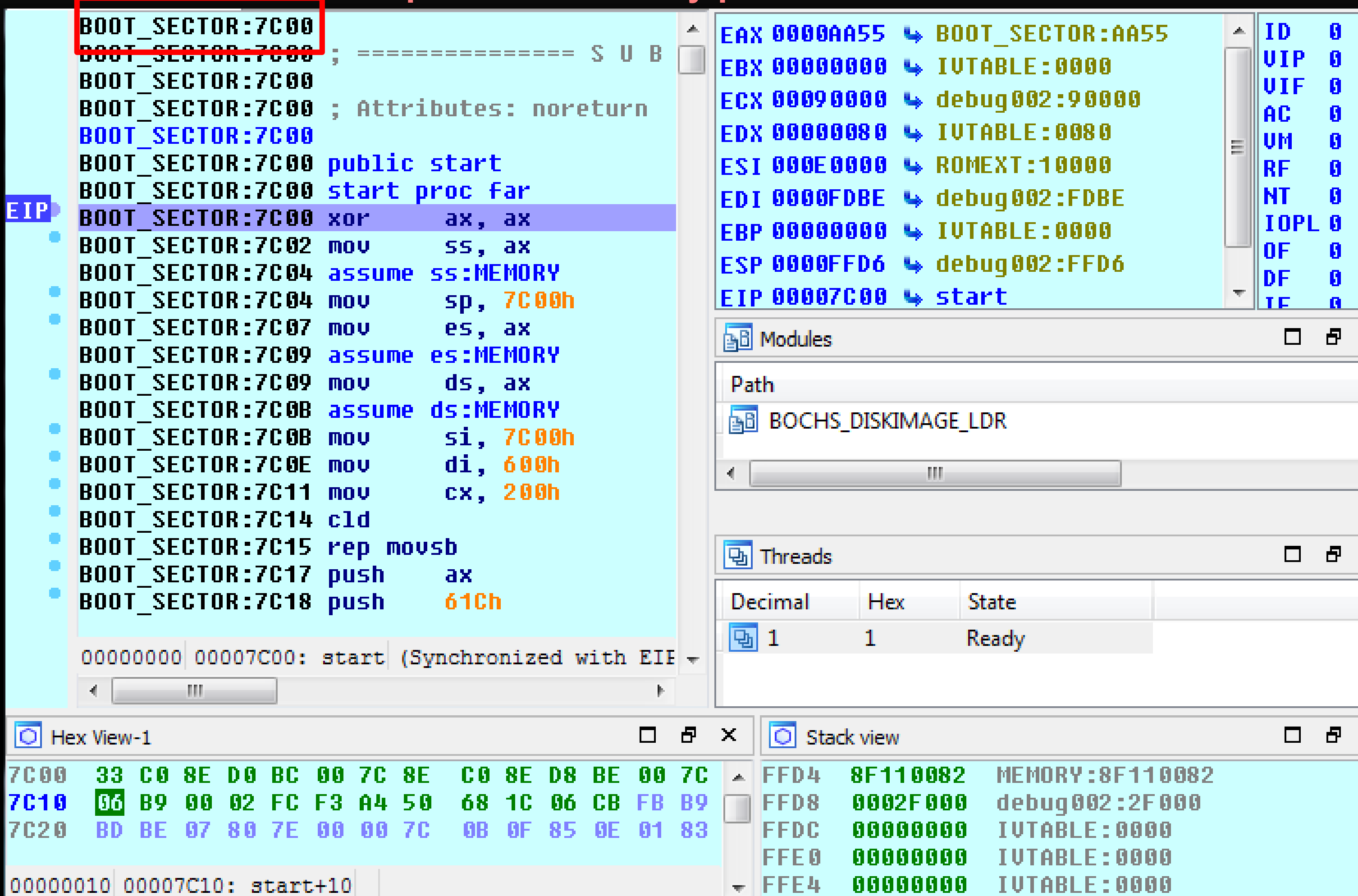
BIOS_PARAMETER__BLOCK_NTFS

Overwritten with an offset of the bootkit on the disk.

Thus, in this case, the malicious code will be executed instead of the IPL. ☺

ADVANCED MALWARES

expected MBR entry point and it must be included in the IDA Pro's load_file.



Eventually, analyzing and debugging the MBR/VBR (loaded as binary module) is unavoidable, but it's not so difficult as it seems. Furthermore, we never know when an advanced malware or a ransomwares (TDL4 and Petya) will attack us. ☺

ADVANCED MALWARES

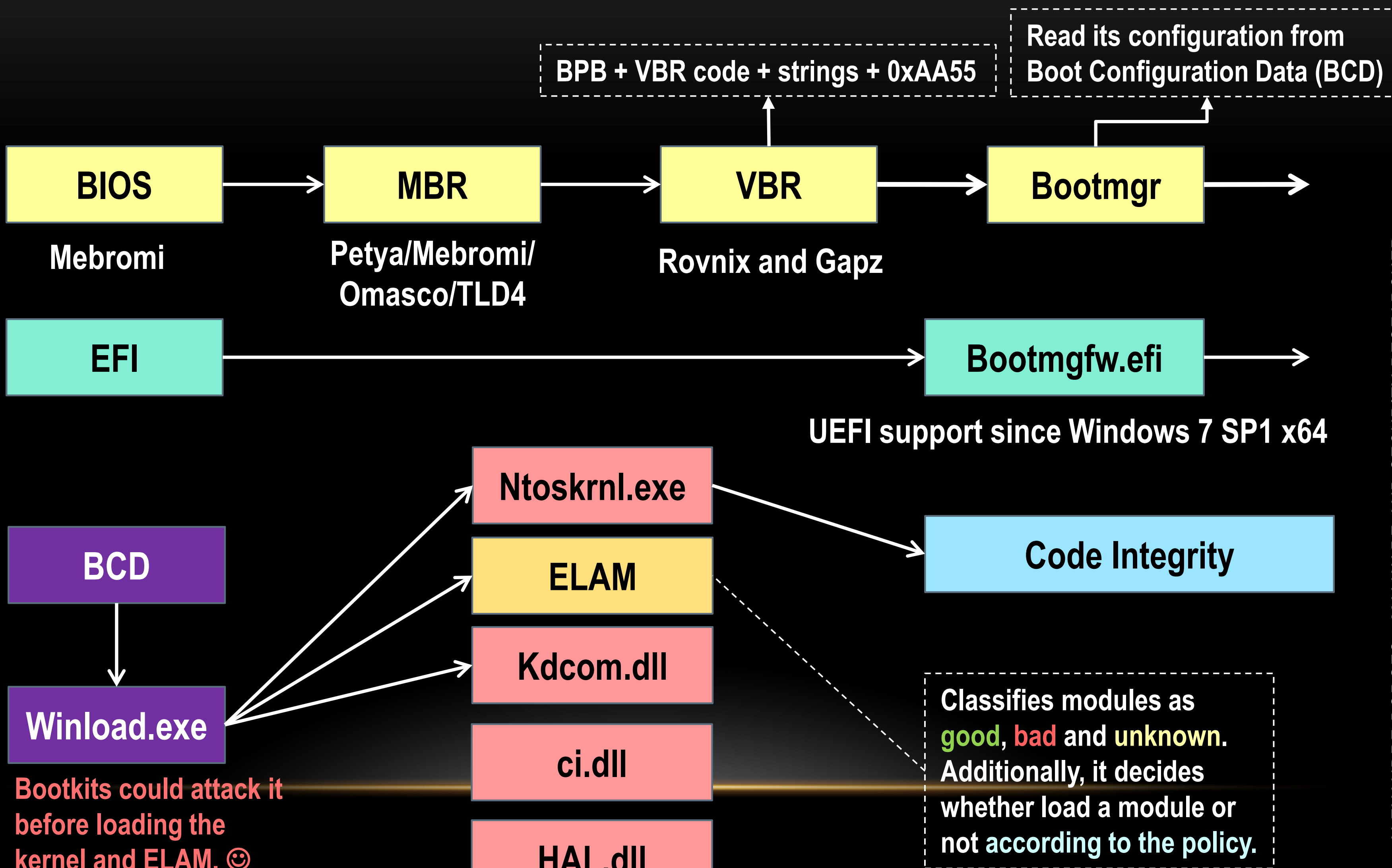
- MBR modifications (partition table or MBR code) and VBR+IPL modifications (BPB or IPL code) have been used as an effective way to bypass the KCS.
- As injecting code into the Windows kernel has turned out to be a bit more complicated, to modern malwares are used to bypassing the KCS (Kernel-Mode Code Signing Policy) by:
  - Disabling it → Booting the system on Testing Mode. Unfortunately, it is not so trivial because the Secure Boot must be disabled previously and, afterwards, it must be rebooted. ☺
  - Changing the kernel memory → MBR and/or VBR could be changed. However, as BIOS reads the MBR and handle over the execution to the code there, so changing memory could be lethal. ☺
  - Even trying to find a flaw in the firmware → it is not trivial and the Secure Boot must be disabled.

ADVANCED MALWARES

```
.text:00573DE3                 mov     [esp+8Ch+var_74], 'b'
.text:00573DE8                 mov     [esp+8Ch+var_73], 'c'
.text:00573DED                 mov     [esp+8Ch+var_72], 'd'
.text:00573DF2                 mov     [esp+8Ch+var_71], 'e'
.text:00573DF7                 mov     edi, ebx
.text:00573DF9                 mov     [esp+8Ch+var_70], 'd'
.text:00573DFE                 mov     [esp+8Ch+var_6F], 'i'
.text:00573E03                 rep stosd
.text:00573E05                 mov     [esp+8Ch+var_6E], 't'
.text:00573E0A                 mov     [esp+8Ch+var_6D], '.'
.text:00573E0F                 mov     [esp+8Ch+var_6C], 'e'
.text:00573E14                 mov     [esp+8Ch+var_6B], 'x'
.text:00573E19                 mov     [esp+8Ch+var_6A], 'e'
.text:00573E1E                 mov     [esp+8Ch+var_69], 0
.text:00573E23
.text:00573E23 loc_573E23:                             ; CODE XREF: sub_573DD0+61↓j
.text:00573E23                 mov     [esp+eax+8Ch+var_68], 0
.text:00573E2B                 add     eax, 4
.text:00573E2E                 cmp     eax, 20h
.text:00573E31                 jb      short loc_573E23
.text:00573E33                 lea     ecx, [esp+8Ch+var_68]
.text:00573E37                 test    edx, edx
.text:00573E39                 mov     byte ptr [esp+8Ch+var_68], '/'
.text:00573E3E                 mov     byte ptr [esp+8Ch+var_68+1], 's'
.text:00573E43                 mov     byte ptr [esp+8Ch+var_68+2], 'e'
.text:00573E48                 mov     byte ptr [esp+8Ch+var_68+3], 't'
.text:00573E4D                 mov     [esp+8Ch+var_64], ' '
.text:00573E52                 mov     edi, ecx
.text:00573E54                 mov     [esp+8Ch+var_63], 't'
.text:00573E59                 mov     [esp+8Ch+var_62], 'e'
.text:00573E5E                 mov     [esp+8Ch+var_61], 's'
.text:00573E63                 mov     [esp+8Ch+var_60], 't'
.text:00573E68                 mov     [esp+8Ch+var_5F], 's'
.text:00573E6D                 mov     [esp+8Ch+var_5E], 'i'
.text:00573E72                 mov     [esp+8Ch+var_5D], 'g'
.text:00573E77                 mov     [esp+8Ch+var_5C], 'n'
.text:00573E7C                 mov     [esp+8Ch+var_5B], 'i'
.text:00573E81                 mov     [esp+8Ch+var_5A], 'n'
.text:00573E86                 mov     [esp+8Ch+var_59], 'g'
.text:00573E8B                 mov     [esp+8Ch+var_58], ' '
.text:00573E90                 jz      loc_573F50
```

Setting TESTING mode is a very poor drive signature "bypassing". Actually, there are more elegant methods. ☺

ADVANCED MALWARES



ADVANCED MALWARES

- Malwares infect the bootmgr, which switches the processor execution from real mode to protected mode, and use the int 13h interrupt to access the disk drive, patch modules and load malicious drivers.
- The winload.exe roles are the following:
  - enables the protect mode.
  - checks the modules' integrity and loads the Windows kernel.
  - loads the several DLLs (among them, the ci.dll, which is responsible for Code Integrity) and ELAM (Early Launch Anti Malware, which was introduced on Windows 8 as callback methods and tries to prevent any strange code execution in the kernel).
  - loads drivers and few system registry data.

ADVANCED MALWARES

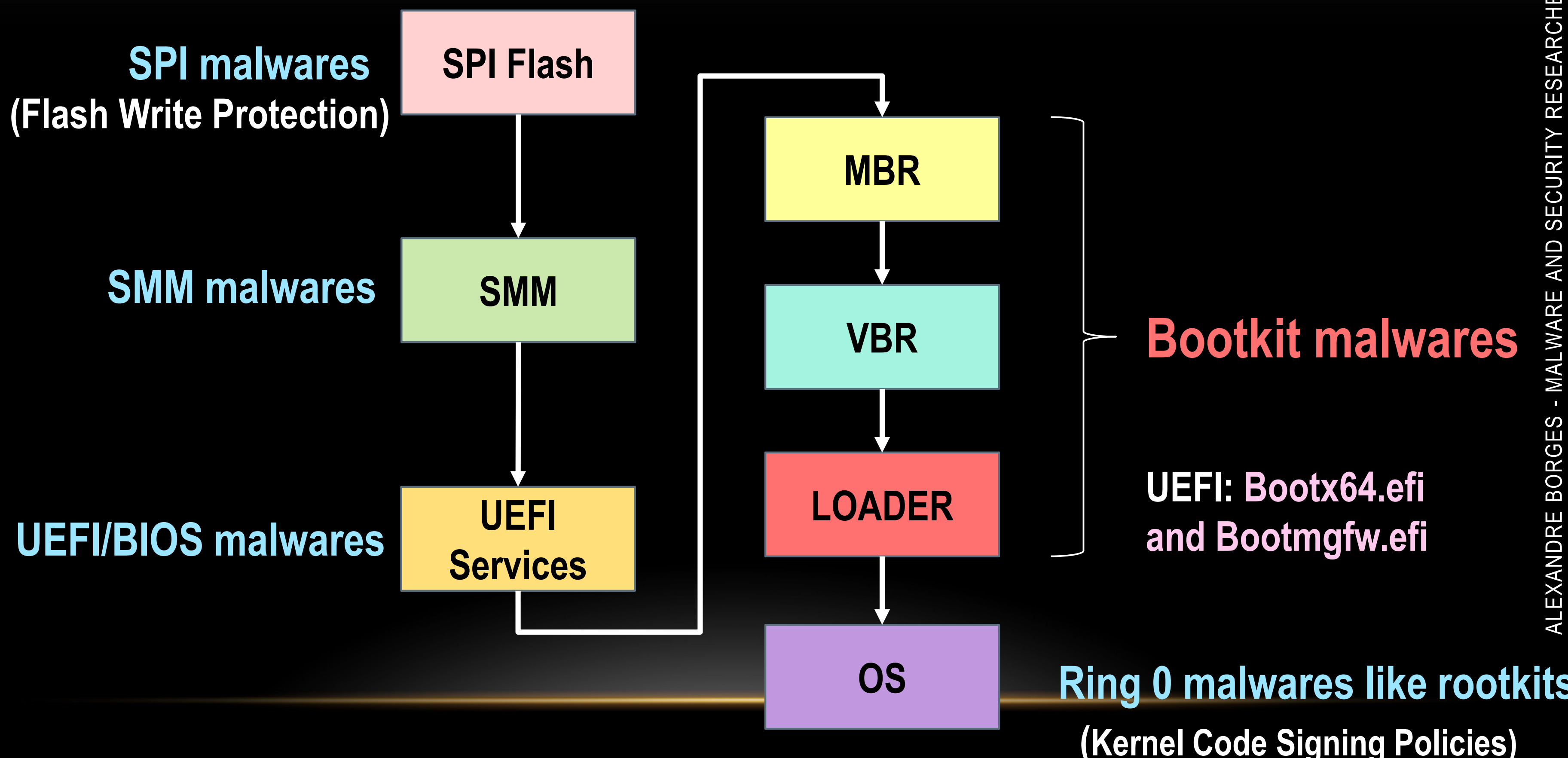
- Furthermore, if the integrity checking of the winload.exe is subverted, so a malicious code could be injected into the kernel because we wouldn't have an integrity control anymore.

- Most advanced rootkits continue storing/reading (opcode 0x42, 0x43 and 0x48) their configuration and payloads from encrypted hidden filesystems (usually, FAT32) and implementing modified symmetric algorithms (AES, RC4, and so on) in these filesystems.

ADVANCED MALWARES

- SMM basics:
  - Interesting place to hide malwares because is protected from OS and hypervisors.
  - The SMM executable code is copied into SMRAM and locked during the initialization.
  - To switch to SMM, it is necessary to triger a SMI (System Management Interrupt), save the current content into SMRAM and execute the SMI handler code.
  - A SMI could be generated from a driver (ring 0) by writing a value into APMC I/O / port B2h or using a I/O instruction restart CPU feature.
  - The return (and execution of the prior execution) is done by using RSM instruction.

ADVANCED MALWARES



ADVANCED MALWARES

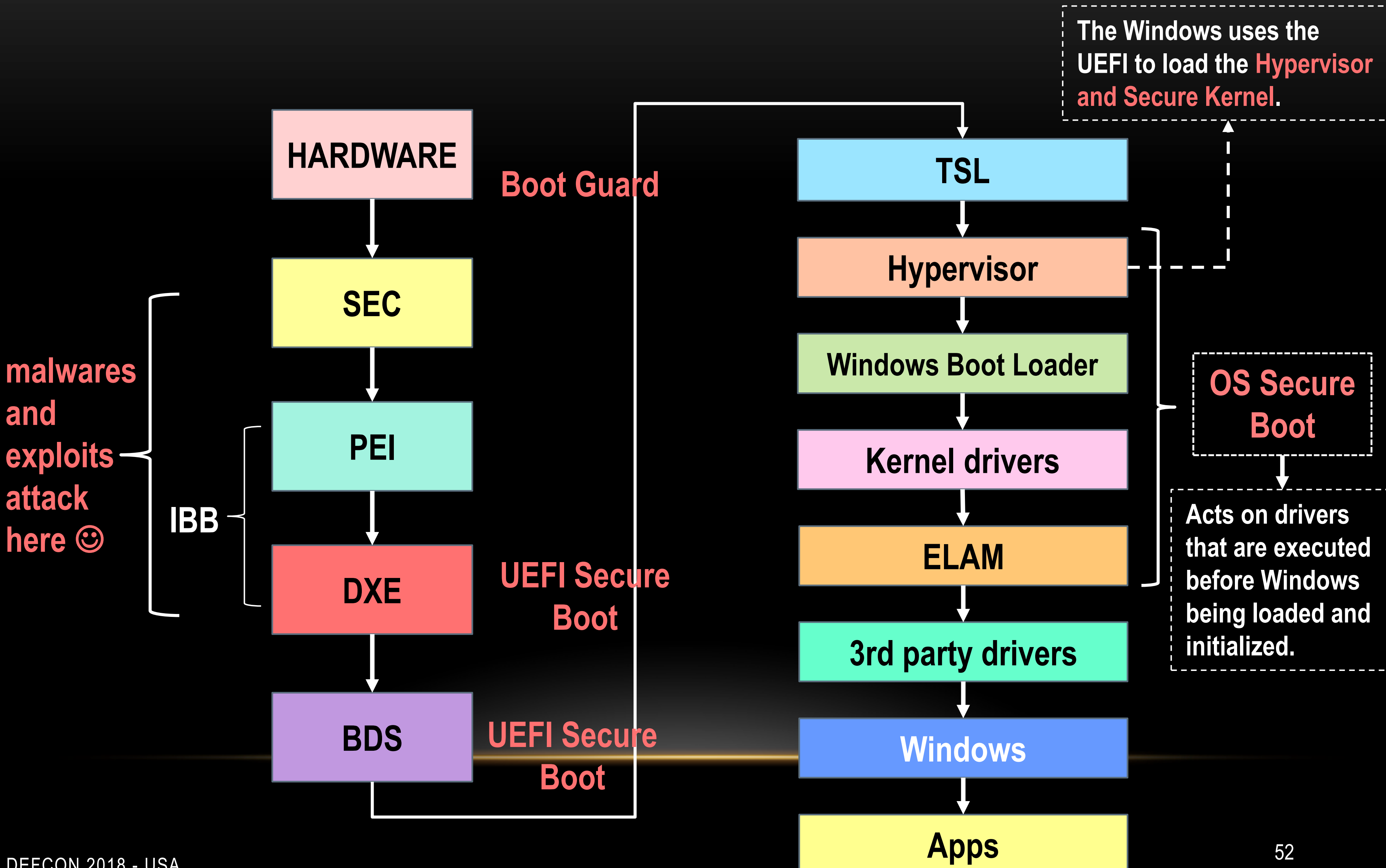
SEC → PEI → DXE → BDS → TSL → RT → AL

IBB – Initial Boot Block

After Life

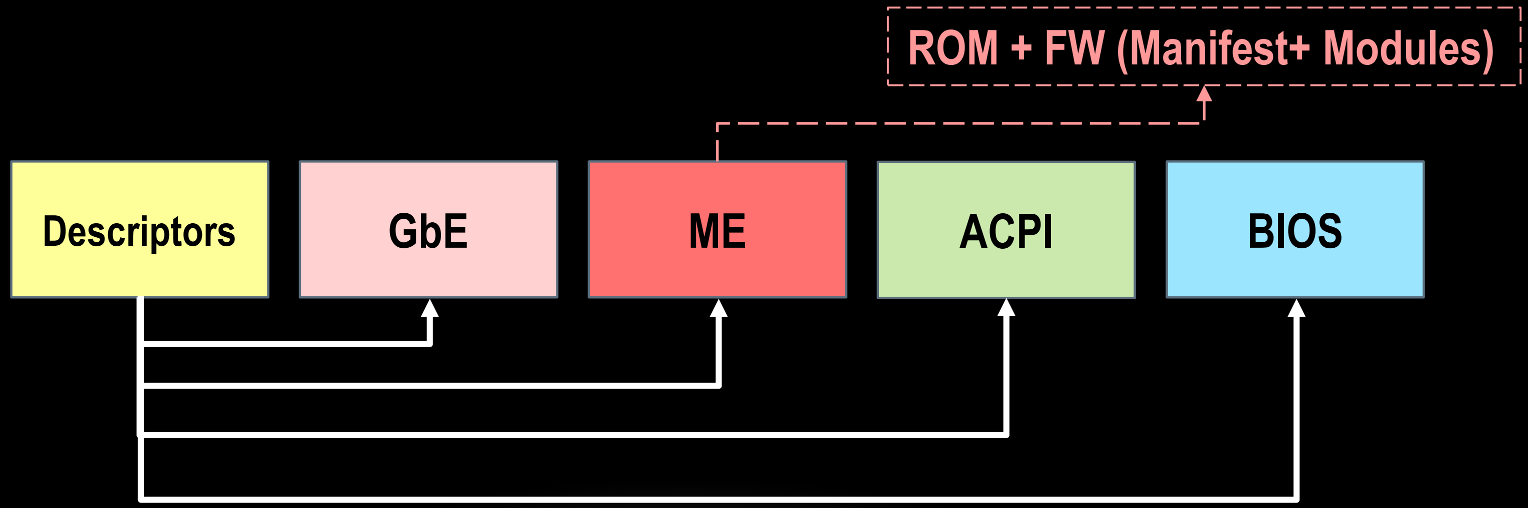
- SEC → Security (Caches, TPM and MTRR initialization)
- PEI → Pre EFI Initialization (SMM/Memory)
- DXE → Driver Execution Environment (platform + devices initialization , Dispatch Drivers, FV enumumeration)
- BDS → Boot Dev Select (EFI Shell + OS Boot Loader)
- TSL → Transient System Load
- RT → Run Time

ADVANCED MALWARES



ADVANCED MALWARES

- **Remember: the SPI Flash is composed by many regions such as Flash Descriptors, BIOS, ME (Management Engine), GbE and ACPI EC. Access Control table defines who can have READ/WRITE access to other regions.**



ME: has full access to the DRAM, invisible at same time, is always working (even then the system is shutdown) and has access to network interface. Conclusion: a nightmare. ☺

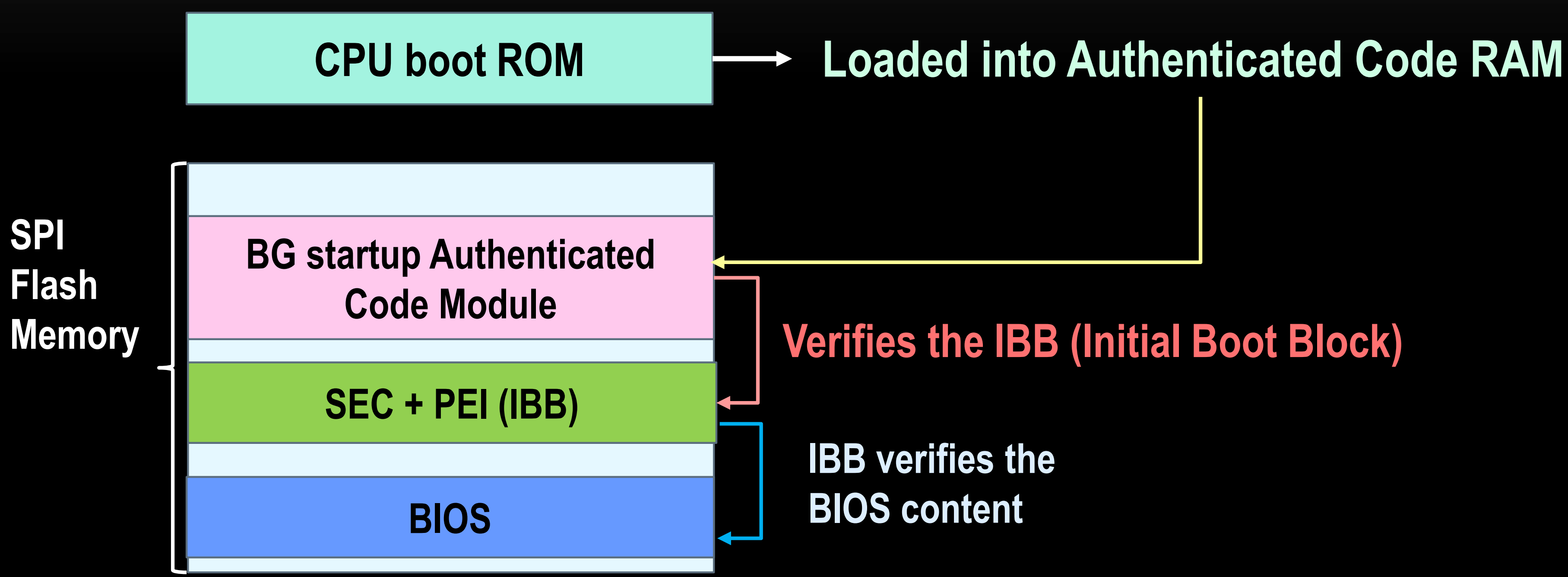
ADVANCED MALWARES

- Intel Boot Guard (controlled by ME), introduced by Intel, is used to validate the boot process through flashing a public key associated to BIOS signature into FPFs (Field Programmable Fuses) from Intel ME.

- Obviously, few vendors have been leaving closemnt fuse unset, so it could be lethal. ☺

- Of course, for a perfect Boot Guard working, the SPI region must be locked and the Boot Guard configuration must be set against a SMM driver rootkit. ☺

ADVANCED MALWARES



- Public key's hash, used for verifying the signature of the code with the ACM, is hard-coded within the CPU.
- It almost impossible to modify the BIOS without knowing the private key.
- At end, it works as a certificate chain. ☺

ADVANCED MALWARES

- ✓ Another protection feature named BIOS Guard is also running in the SMM, which protects the platform against not-authorized:
  - SPI Flash Access (through BIOS Guard Authenticated Code Module) → prevents an attacker to escalate privileges to SMM by writting a new image to SPI.
  - BIOS update → attacker (through a DXE driver) could update the BIOS to a flawed BIOS version.
  - Boot infection/corruption.
- ✓ BIOS Guard allows that only trusted modules (by ACM) be able to modify the SPI flash memory and protect us against rookit implants.

ADVANCED MALWARES

- Secure Boot:
  - Protects the entire path shown previously against bootkit infection.
  - Protects key components during kernel loading, key drivers and important system files, requesting a valid digital signature.
  - Prevents loading of any code that are not associated a valid digital signature.

ADVANCED MALWARES

- Two essential items on Secure Boot are:

  - Platform Key (PK – must be valid), which establishes a trust relationship between the platform owner and the platform firmware, verifies the Key Exchange Key (KEK).

  - KEK, which establishes a trust relationship between the OS and the platform firmware, verifies:

    - Authorized Database (db) → contains authorized signing certificates and digital signatures
    - Forbidden Database (dbx) → contains forbidden certificates and digital signatures.

ADVANCED MALWARES

- Obviously, if the Platform Key is corrupted, everything is not valid anymore because the SecureBoot turns out disabled when this fact happens. ☹

- Unfortunately, few vendors continue storing important Secure Boot settings in UEFI variables. However, if these UEFI variables are exploited through ring 0/-2 malware or bootkit, so the SecureBoot can be disabled.

ADVANCED MALWARES

- Without ensuring the UEFI image integrity, a rookit could load another UEFI image without being noticed. ☺

- UEFI BIOS supports TE (Terse Executable) format (signature 0x5A56 - VZ).

- As TE format doesn't support signatures, BIOS shouldn't load this kind of image because Signature checking would be skipped.

- Therefore, a rootkit could try to replace the typical PE/COFF loader by a TE EFI executable, so skipping the signature checking and disabling the Secure Boot.

ADVANCED MALWARES

- Fortunately, new releases of Windows 10 (version 1607 and later) has introduced an interesting SMM protection known as Windows SMM Security Mitigation Table (WSMT).

- In Windows 10, the firmware executing SMM must be "authorized and trusted" by VBS (Virtualized Based Security).

ADVANCED MALWARES

- These SMM Protections flags that can be used to enable or disable any WSMT feature.
  - FIXED_COMM_BUFFERS: it guarantees that any input/output buffers be filled by value within the expected memory regions.
  - SYSTEM_RESOURCE_PROTECTION: it works as an indication that the system won't allow out-of-band reconfiguration of system resources.
  - COMM_BUFFER_NESTED_PTR_PROTECTION: it is a validation method that try to ensure that any pointer whith the fixed communication buffer only refer to address ranges that are within a pre-defined memory region.

ADVANCED MALWARES

- chipsec_util.py spi dump spi.bin
- chipsec_uti.py decode spi.bin

Master Read/Write Access to Flash Regions

Region	CPU	ME
0 Flash Descriptor	R	R
1 BIOS	RW	
2 Intel ME		RW
3 GBe	RW	RW

Is the customer Safe? ☺

ALEXANDRE BORGES - MALWARE AND SECURITY RESEARCHER

ADVANCED MALWARES

```
[x][ =======================================================================
[x][ Module: BIOS Region Write Protection
[x][ =======================================================================
[*] BC = 0x01 << BIOS Control (b:d.f 00:31.0 + 0xDC)
    [00] BIOSWE           = 1 << BIOS Write Enable
    [01] BLE              = 0 << BIOS Lock Enable
    [02] SRC              = 0 << SPI Read Configuration
    [04] TSS              = 0 << Top Swap Status
    [05] SMM_BWP          = 0 << SMM BIOS Write Protection
[-] BIOS region write protection is disabled!

[*] BIOS Region: Base = 0x00600000, Limit = 0x009FFFFF
SPI Protected Ranges
------------------------------------------------------------
PRx (offset) | Value    | Base     | Limit    | WP? | RP?
------------------------------------------------------------
PR0 (74)     | 00000000 | 00000000 | 00000000 | 0   | 0
PR1 (78)     | 00000000 | 00000000 | 00000000 | 0   | 0
PR2 (7C)     | 00000000 | 00000000 | 00000000 | 0   | 0
PR3 (80)     | 00000000 | 00000000 | 00000000 | 0   | 0
PR4 (84)     | 00000000 | 00000000 | 00000000 | 0   | 0
```

BIOS Write Enable should be clear (BIOSWE=0) and BIOS Lock Enable should be set (BLE=1)! In this case, it is exactly the opposite!

SMM-based write-protection is disabled! Please, set SMM_BWP to 1 and lock the SMI configuration by setting GBL_SMI_LCK and TCO_LCK to 1!

None of Protect Range registers are protecting the flash against writes! The HSFS.FLOCKDN bit should also be set!

chipsec_main --module common.bios_wp

ADVANCED MALWARES

```
[*] running module: chipsec.modules.common.bios_kbrd_buffer

[*] running module: chipsec.modules.common.bios_smi
[x][ =======================================================================
[x][ Module: SMI Events Configuration
[x][ =======================================================================
[-] SMM BIOS region write protection has not been enabled (SMM_BWP is not used)

[*] Checking SMI enables..
    Global SMI enable: 1
    TCO SMI enable   : 1
[+] All required SMI events are enabled

[*] Checking SMI configuration locks..
[-] TCO SMI event configuration is not locked. TCO SMI events can be disabled
[+] SMI events global configuration is locked (SMI Lock)

[-] FAILED: Not all required SMI sources are enabled and locked
```

chipsec_main.py -m common.bios_smi

ADVANCED MALWARES

- The BIOS_CNTL register contains:

  - BIOS Write Enable(BWE) → if it is set to 1, an attacker could write to SPI flash.
  - BIOS Lock Enable (BLE) → if it is set to 1, it generates an SMI routine to run just in case the BWE goes from 0 to 1.

- Of course, there should be a SMM handler in order to prevent setting the BWE to 1.
- What could happen if SMI events were blocked? ☺
- The SMM BIOS write protection (SMM_BWP) , which protects the entire BIOS area, is not enabled. ☹

ADVANCED MALWARES

```
[*] running module: chipsec.modules.common.spi_lock
[x][ =======================================================================
[x][ Module: SPI Flash Controller Configuration Lock
[x][ =======================================================================
[*] HSFS = 0xE008 << Hardware Sequencing Flash Status Register (SPIBAR + 0x4)
    [00] FDONE              = 0 << Flash Cycle Done
    [01] FCERR              = 0 << Flash Cycle Error
    [02] AEL                = 0 << Access Error Log
    [03] BERASE             = 1 << Block/Sector Erase Size
    [05] SCIP               = 0 << SPI cycle in progress
    [13] FDOPSS             = 1 << Flash Descriptor Override Pin-Strap Status
    [14] FDV                = 1 << Flash Descriptor Valid
    [15] FLOCKDN            = 1 << Flash Configuration Lock-Down
[+] PASSED: SPI Flash Controller configuration is locked

[CHIPSEC] ***************************  SUMMARY  ***************************
[CHIPSEC] Time elapsed          0.014
[CHIPSEC] Modules total         1
[CHIPSEC] Modules failed to run 0:
[CHIPSEC] Modules passed        1:
[+] PASSED: chipsec.modules.common.spi_lock
[CHIPSEC] Modules failed        0:
[CHIPSEC] Modules with warnings 0:
[CHIPSEC] Modules skipped 0:
[CHIPSEC] *******************************************************************
```

chipsec_main.py -m common.spi_lock

ADVANCED MALWARES

- SPI Protect Range registers protect the flash chip against writes.
- They control Protected Range Base and Protected Range Limit fields, which set regions for Write Protect Enable bit and Read Protect Enable bit.
- If the Write Protect Enable bit is set, so regions from flash chip that are defined by Protected Range Base and Protected Range Limit fields are protected.
- However, SPI Protect Range registers DO NOT protect the entire BIOS and NVRAM.
- In a similar way to BLE, the HSFSS.FLOCKDN bit (from HSFSTS SPI MMIO Register) prevents any change to Write Protect Enable bit. Therefore, malware can't disable the SPI protected ranges for enabling access to the SPI flash memory.

ADVANCED MALWARES

python chipsec_main.py --module common.bios_ts

```
[+] loaded chipsec.modules.common.bios_ts
[*] running loaded modules ..

[*] running module: chipsec.modules.common.bios_ts
[x][ =======================================================================
[x][ Module: BIOS Interface Lock (including Top Swap Mode)
[x][ =======================================================================
[*] BiosInterfaceLockDown (BILD) control = 1
[*] BIOS Top Swap mode is disabled (TSS = 0)
[*] RTC TopSwap control (TS) = 0
[+] PASSED: BIOS Interface is locked (including Top Swap Mode)
```

ADVANCED MALWARES

- Top Swap Mode, which is enabled by BUC.TS in Root Complex range, is a feature that allows fault-tolerant update of the BIOS boot-block.
- Therefore, when Top Swap Configuration and swap boot-block range in SPI are not protected or even locked, any malware could force an execution redirect of the reset vector to backup bootblock because CPU will fetch the reset vector at 0xFFFEFFF0 instead of 0xFFFFFFF0 address.
- SMRR (System Management Range Registers) blocks the access to SMRAM (range of DRAM that is reserved by BIOS SMI handlers) while CPU is not in SMM mode, preventing it to execute any SMI exploit on cache.

ALEXANDRE BORGES - MALWARE AND SECURITY RESEARCHER

ADVANCED MALWARES

```
[*] running module: chipsec.modules.common.smrr
[x][ =======================================================================
[x][ Module: CPU SMM Cache Poisoning / System Management Range Registers
[x][ =======================================================================
[+] OK. SMRR range protection is supported

[*] Checking SMRR range base programming..
[*] IA32_SMRR_PHYSBASE = 0xCF800004 << SMRR Base Address MSR (MSR 0x1F2)
    [00] Type             = 4 << SMRR memory type
    [12] PhysBase         = CF800 << SMRR physical base address
[*] SMRR range base: 0x00000000CF800000
[*] SMRR range memory type is Write-through (WT)
[+] OK so far. SMRR range base is programmed

[*] Checking SMRR range mask programming..
[*] IA32_SMRR_PHYSMASK = 0xFF800800 << SMRR Range Mask MSR (MSR 0x1F3)
    [11] Valid            = 1 << SMRR valid
    [12] PhysMask         = FF800 << SMRR address range mask
[*] SMRR range mask: 0x00000000FF800000
[+] OK so far. SMRR range is enabled

[*] Verifying that SMRR range base & mask are the same on all logical CPUs..
[CPU0] SMRR_PHYSBASE = 00000000CF800004, SMRR_PHYSMASK = 00000000FF800800
[CPU1] SMRR_PHYSBASE = 00000000CF800004, SMRR_PHYSMASK = 00000000FF800800
[CPU2] SMRR_PHYSBASE = 00000000CF800004, SMRR_PHYSMASK = 00000000FF800800
[CPU3] SMRR_PHYSBASE = 00000000CF800004, SMRR_PHYSMASK = 00000000FF800800
[CPU4] SMRR_PHYSBASE = 00000000CF800004, SMRR_PHYSMASK = 00000000FF800800
[CPU5] SMRR_PHYSBASE = 00000000CF800004, SMRR_PHYSMASK = 00000000FF800800
[CPU6] SMRR_PHYSBASE = 00000000CF800004, SMRR_PHYSMASK = 00000000FF800800
[CPU7] SMRR_PHYSBASE = 00000000CF800004, SMRR_PHYSMASK = 00000000FF800800
[+] OK so far. SMRR range base/mask match on all logical CPUs
[*] Trying to read memory at SMRR base 0xCF800000..
[+] PASSED: SMRR reads are blocked in non-SMM mode

[+] PASSED: SMRR protection against cache attack is properly configured
```

chipsec_main.py -m common.smrr

ALEXANDRE BORGES - MALWARE AND SECURITY RESEARCHER

CONCLUSION

- Most security professionals have been facing problems to understand how to analyze malicious drivers because the theory is huge and not easy.

- Real customers are not aware about ring -2 threats and they don't know how to update systems' firmwares.

- All protections against implants are based on integrity (digital certificate and signature). However, what would it happen whether algorithms were broken (QC - quantum computation)?

THANK YOU FOR ATTENDING MY TALK!

- Malware and Security Researcher.
- Consultant, Instructor and Speaker on Malware Analysis, Memory Analysis, Digital Forensics, Rootkits and Software Exploitation.
- Member of Digital Law and Compliance Committee (CDDC/ SP)
- Reviewer member of the The Journal of Digital Forensics, Security and Law.
- Refereer on Digital Investigation:The International Journal of Digital Forensics & Incident Response
- Instructor at Oracle, (ISC)2 and Ex-instructor at Symantec.

LinkedIn:
http://www.linkedin.com/in/aleborges

Twitter: @ale_sp_brazil

Website:
http://blackstormsecurity.com

E-mail:
alexandreborges@blackstormsecurity.com