PROFILE AND TOC

TOC:

- Introduction
- Rootkits: Ring 0
- Advanced malware and Rootkits: Ring -2

- 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.
- Referee on Digital Investigation: The International Journal of Digital Forensics & Incident Response
- Instructor at Oracle, (ISC)2 and Ex-instructor at Symantec.
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 MALWARE:

- MBR/VBR/UEFI rootkits
- Techniques 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 a kind of “modern hook” oftenly used by antivirus programs for monitoring and alerting the kernel modules about a specific event occurrence. Therefore, they are used by malware samples (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.
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.

- Malware has been using `CmRegisterCallback()` 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
ffff801`aa733fcc 00000002

0: kd> dps nt!CallbackListHead L2
ffff801`aa769190 ffffc000`c8d62db0
ffff801`aa769198 ffffc000`c932c8b0

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

0: kd> !list -t _LIST_ENTRY.Flink -x "dps" -a "L8" 0xfffffc000`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

.....

ffff801`aa7691c0  00000000`bee0bee0
ffff801`aa7691c8  fffff801`aa99b600 nt!HvpGetCellFlat
ROOTKITS: RING 0

- At the same way, `PsSetCreateProcessNotifyRoutine()` routine adds a driver-supplied callback routine to a list of routines to be called when 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 & 0xFFFFFFFFFFFFFFFF; dps $t1+8 L1;}
  ```

- Malware 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 malicious address and the routine redirects the execution flow to the malicious code.
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 & 0xFFFFFFFFFFFFFFFF0; dps $t1+8 L1;}

fferfe001`13c8b808 fffff801`aa5839c4 nt!ViCreateProcessCallback
fferfe001`139e1138 fffff801`678175f0 cng!CngCreateProcessNotifyRoutine
fferfe001`13b43138 fffff801`67e6c610 kl1+0x414610
fferfe001`13bdb268 fffff801`685d1138 PGPfsfd+0x1c138
fferfe001`13b96858 fffff801`68a53000 ksecdd!KsecCreateProcessNotifyRoutine
fferfe001`1642eacc8 fffff801`68d40ec0 tcpip!CreateProcessNotifyRoutineEx
fferfe001`164f6fca8 fffff801`67583c70 CII!PEProcessNotify
fferfe001`13b6e4b8 fffff801`68224a38 klflt!PstUnregisterProcess+0xfac
fferfe001`1653e4d8 fffff801`699512c0 vsdatant+0x8e2c0
By now, we have seen malware samples using KTHREAD.LegoData field for registering a malicious routine, which would be called during the thread termination.
Windows offers different types of drivers such as legacy drivers, filter drivers and minifilter drivers (malware can be written using any one these types), which could be developed using either 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.
ROOTKITS: RING 0

• Most ring 0 malware 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).

• The AddDevice( ) function is used to create an unnamed Device Object and to attach it to a named Device Object (ex: aborges) from a layered driver (lower-level driver).

• 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).
ROOTKITS: RING 0

• Each IRP packet 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 to the next IO_STACK_LOCATION by using the IoCopyCurrentIrpStackLocation() routine or even to the next driver by calling IoSkipCurrentStackLocation() routine.
• Alternatively, the IRP packet 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. 😊
ROOTKITS: RING 0

Driver Stack

- Tcpi.sys
- Upper Filter Driver
- Function Driver
- Lower Filter Driver
- Miniport driver

Device Stack

- Upper Filter Device Object
- Function Device Object
- Lower Filter Device Object
- Physical Device Object

The IoCompleteRequest() manages calling these routines in the correct order (bottom-up). 😊
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 responsibility to prepare the IO_STACK_LOCATION to the next level, as well a driver could call the IoSetCompletionRoutine() to set a completion routine up at CompletionRoutine field.
ROOTKITS: RING 0

Parameters field depends on the major and minor functions!
Parameter field depends on major and minor function number. Thus, the IRPs being used are related to the action.
ROOTKITS: RING 0

Malicious driver

```
kd> lmDrvm aborges
Browse full module list
start end module name
9a3c0000 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 : 0x4
+0x002 Size : 0x168
+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
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!IoInvalidDeviceRequest
[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!IoInvalidDeviceRequest
[06] IRP_MJ_SET_INFORMATION  82aca0bf  nt!IoInvalidDeviceRequest
[07] IRP_MJ_QUERY_EA  82aca0bf  nt!IoInvalidDeviceRequest
[08] IRP_MJ_SET_EA  82aca0bf  nt!IoInvalidDeviceRequest
[09] IRP_MJ_FLUSH_BUFFERS  82aca0bf  nt!IoInvalidDeviceRequest
[0a] IRP_MJ_QUERY_VOLUME_INFORMATION  82aca0bf  nt!IoInvalidDeviceRequest
[0b] IRP_MJ_SET_VOLUME_INFORMATION  82aca0bf  nt!IoInvalidDeviceRequest
[0c] IRP_MJ_DIRECTORY_CONTROL  82aca0bf  nt!IoInvalidDeviceRequest
[0d] IRP_MJ_FILE_SYSTEM_CONTROL  82aca0bf  nt!IoInvalidDeviceRequest
[0e] IRP_MJ_DEVICE_CONTROL  9a3c3c82  aborges+0xc82
ROOTKITS: RING 0

kd> !fltkd.filters

Filter List
 FLT FILTER 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
fltctx: _FLT_FILTER
+0x000 Base : _FLT_OBJECT
+0x014 Frame : 0x85a886f8 _FLT.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 : 0x86ded50 _ALLOCATE_CONTEXT_HEADER
+0x098 SupportedContexts : [6] (null)
+0x0b0 PreVolumeMount : 0x8f5a0cc _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)
+0x0cc RenameNotification : (null)
+0x0cc Operations : 0x86df0164 _FLT_OPERATION_REGISTRATION
+0x0d0 OldDriverUnload : (null)
ROOTKITS: RING 0
ROOTKITS: RING 0

- Naturally, as closest at bottom of device stack the infection occurs (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.

- Malware 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

kd> !analyze –show 109

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

Thanks, Alex Ionescu 😊
Most time, malware has allocated a kind of hidden filesystem 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 malware 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. 😊
Additionally, malware 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.

Malware also do not choose an specific driver for injection, but try to randomly pick up a driver by parsing structures such as _KLDR_DATA_TABLE_ENTRY.
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, malware also hook functions such as ZwCreate() for intercepting all opened requests sent to devices.

• After infecting a system by dropping kernel drivers, malware 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

- Malware 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:

```python
root@kali:~# more /root/volatility26/volatility/plugins/rootkitsscanner.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)

<table>
<thead>
<tr>
<th>Address</th>
<th>Tag</th>
<th>Size</th>
<th>Protection</th>
<th>Pool Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>85fce408</td>
<td>Driv</td>
<td>0x0f0</td>
<td>Protected</td>
<td>Nonpaged pool</td>
</tr>
<tr>
<td>85fd2158</td>
<td>Driv</td>
<td>0x1b0</td>
<td></td>
<td>Nonpaged pool</td>
</tr>
<tr>
<td>85fd2470</td>
<td>Driv</td>
<td>0x0f0</td>
<td>Protected</td>
<td>Nonpaged pool</td>
</tr>
<tr>
<td>85fd0e50</td>
<td>Driv</td>
<td>0x1b0</td>
<td></td>
<td>Nonpaged pool</td>
</tr>
<tr>
<td>85fa8698</td>
<td>Driv</td>
<td>0x0f0</td>
<td>Protected</td>
<td>Nonpaged pool</td>
</tr>
<tr>
<td>85fd5140</td>
<td>Driv</td>
<td>0x10</td>
<td></td>
<td>Nonpaged pool</td>
</tr>
<tr>
<td>85fd5e50</td>
<td>Driv</td>
<td>0x1b0</td>
<td></td>
<td>Nonpaged pool</td>
</tr>
<tr>
<td>8655e658</td>
<td>Driv</td>
<td>0x0f0</td>
<td>Protected</td>
<td>Nonpaged pool</td>
</tr>
<tr>
<td>85febb98</td>
<td>Driv</td>
<td>0x0f0</td>
<td>Protected</td>
<td>Nonpaged pool</td>
</tr>
<tr>
<td>85f911c8</td>
<td>Driv</td>
<td>0x0f0</td>
<td>Protected</td>
<td>Nonpaged pool</td>
</tr>
<tr>
<td>85f931e8</td>
<td>Driv</td>
<td>0x0f0</td>
<td>Protected</td>
<td>Nonpaged pool</td>
</tr>
<tr>
<td>85fbd248</td>
<td>Driv</td>
<td>0x1b0</td>
<td></td>
<td>Nonpaged pool</td>
</tr>
<tr>
<td>85fbd00</td>
<td>Driv</td>
<td>0x0f0</td>
<td>Protected</td>
<td>Nonpaged pool</td>
</tr>
<tr>
<td>85fc9800</td>
<td>Driv</td>
<td>0x0f0</td>
<td>Protected</td>
<td>Nonpaged pool</td>
</tr>
<tr>
<td>853e0540</td>
<td>Driv</td>
<td>0x0f0</td>
<td>Protected</td>
<td>Nonpaged pool</td>
</tr>
</tbody>
</table>
```
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 MALWARE AND ROOTKITS RING -2
ADVANCED MALWARE

- **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 malware 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.**
Overwritten with an offset of the bootkit on the disk.

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

\[ \text{BIOS\_PARAMETER} \_
\text{BLOCK\_NTFS} \]

### Table: Boot Sector NTFS, Base Offset: 0

<table>
<thead>
<tr>
<th>Offset</th>
<th>Title</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>JMP instruction</td>
<td>EB 52 90</td>
</tr>
<tr>
<td>3</td>
<td>File system ID</td>
<td>NTFS</td>
</tr>
<tr>
<td>B</td>
<td>Bytes per sector</td>
<td>512</td>
</tr>
<tr>
<td>D</td>
<td>Sectors per cluster</td>
<td>8</td>
</tr>
<tr>
<td>E</td>
<td>Reserved sectors</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>(always zero)</td>
<td>00 00 00</td>
</tr>
<tr>
<td>13</td>
<td>(unused)</td>
<td>00 00</td>
</tr>
<tr>
<td>15</td>
<td>Media descriptor</td>
<td>F8</td>
</tr>
<tr>
<td>16</td>
<td>(unused)</td>
<td>00 00</td>
</tr>
<tr>
<td>18</td>
<td>Sectors per track</td>
<td>63</td>
</tr>
<tr>
<td>1A</td>
<td>Heads</td>
<td>255</td>
</tr>
<tr>
<td>1C</td>
<td>Hidden sectors</td>
<td>206,848</td>
</tr>
</tbody>
</table>
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. 😊
• MBR modifications (partition table or MBR code) and VBR+IPL modifications (BPB or IPL code) are effective ways to bypass the KCS, which validates the driver signature.

• As injecting code into the Windows kernel has turned out to be a bit more complicated, modern malware samples 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.
Setting TESTING mode is a very poor drive signature "bypassing". Actually, there are more elegant methods. 😊
ADVANCED MALWARE

BIOS -> MBR -> VBR -> Bootmgr

- BIOS: Mebromi
- MBR: Petya/Mebromi/Omasco/TLD4
- VBR: Rovnix and Gapz
- Bootmgr: Bootmgfw.efi

UEFI support since Windows 7 SP1 x64

BPB + VBR code + strings + 0xAA55

Read its configuration from Boot Configuration Data (BCD)

BCD -> Winload.exe

- Winload.exe: Code Integrity
- Ntoskrnl.exe
- ELAM
- Kdcom.dll
- ci.dll
- HAL.dll

Classifies modules as good, bad and unknown. Additionally, it decides whether load a module or not according to the policy.

Bootkits could attack it before loading the kernel and ELAM. 😊
ADVANCED MALWARE

- Malware infect the `bootmgr`, which is responsible to switch 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`'s tasks 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 MALWARE

• Thus, 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 MALWARE

• SMM basics:
  • Interesting place to hide malware 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 trigger 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 MALWARE

- SPI malware (Flash Write Protection)
- SMM malware
- UEFI/BIOS malware

SPI Flash → SMM → UEFI Services → MBR → VBR → LOADER → OS

UEFI: Bootx64.efi and Bootmgfw.efi

Ring 0 malware like rootkits (Kernel Code Signing Policies)

UEFI/BIOS malware

SMM malware

SPI malware
ADVANCED MALWARE

SEC ➔ PEI ➔ DXE ➔ BDS ➔ TSL ➔ RT ➔ AL

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

IBB – Initial Boot Block

After Life

ALEXANDRE BORGES - MALWARE AND SECURITY RESEARCHER
ADVANCED MALWARE

The Windows uses the UEFI to load the Hypervisor and Secure Kernel.

OS Secure Boot
Acts on drivers that are executed before Windows being loaded and initialized.

The Windows uses the UEFI to load the Hypervisor and Secure Kernel.

IBB
malware and exploits attack here 😎

Hardware

Boot Guard

SEC

PEI

DXE

BDS

TSL

Hypervisor

Windows Boot Loader

Kernel drivers

ELAM

3rd party drivers

Windows

Apps

ALEXANDRE BORGES - MALWARE AND SECURITY RESEARCHER
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 which component (master) can have READ/WRITE access to other regions.

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

• 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) within Intel ME (Management Engine).

• Obviously, few vendors have been leaving closemmt fuse unset, so it could be lethal.

• Of course, for a perfect working of the Boot Guard, the SPI region must be locked and the Boot Guard configuration must be set and protected against a SMM driver rootkit.
CPU boot ROM

Loaded into Authenticated Code RAM

BG startup Authenticated Code Module

Verifies the IBB (Initial Boot Block)

SEC + PEI (IBB)

IBB verifies the BIOS content

BIOS

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

✓ Another protection feature named BIOS Guard that runs within 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 writing 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 only allows trusted modules (by ACM) to modify the SPI flash memory and protect us against rootkit implants.
• **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.

  ✓ **Secure Boot** prevents loading any code that is not associated to a valid digital signature.
ADVANCED MALWARE

• Two essential items in 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 platform firmware and **OS,** verifies:

    • **Authorized Database (db)** ➔ contains authorized signing certificates and digital signatures
    • **Forbidden Database (dbx)** ➔ contains forbidden certificates and digital signatures.
ADVANCED MALWARE

• 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 MALWARE

• Without ensuring the UEFI image integrity, a rootkit 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.
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 MALWARE

• 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 MALWARE

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

<table>
<thead>
<tr>
<th>Region</th>
<th>CPU</th>
<th>ME</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 Flash Descriptor</td>
<td>R</td>
<td>R</td>
</tr>
<tr>
<td>1 BIOS</td>
<td>RW</td>
<td></td>
</tr>
<tr>
<td>2 Intel ME</td>
<td>RW</td>
<td>RW</td>
</tr>
<tr>
<td>3 GBe</td>
<td>RW</td>
<td>RW</td>
</tr>
</tbody>
</table>

Is the customer Safe? 😊
ADVANCED MALWARE

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!

### BIOS Region Write Protection

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
<th>Base</th>
<th>Limit</th>
<th>WP?</th>
<th>RP?</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>0x00060000</td>
<td>0x000000</td>
<td>0x000000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0x01</td>
<td>0x00000000</td>
<td>0x000000</td>
<td>0x000000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0x02</td>
<td>0x00000000</td>
<td>0x000000</td>
<td>0x000000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0x03</td>
<td>0x00000000</td>
<td>0x000000</td>
<td>0x000000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0x04</td>
<td>0x00000000</td>
<td>0x000000</td>
<td>0x000000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0x05</td>
<td>0x00000000</td>
<td>0x000000</td>
<td>0x000000</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

chipsec_main --module common.bios_wp
ADVANCED MALWARE

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

[*] running module: chipsec.modules.common.bios_smi

[X] Module: SMI Events Configuration

[-] 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 MALWARE

• 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 MALWARE

chipsec_main.py -m common.spi_lock
ADVANCED MALWARE

• **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 MALWARE

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][ 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)
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 0xFFFEBFFF0 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.
ADVANCED MALWARE

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

[X] Module: CPU SMM Cache Poisoning / System Management Range Registers

[+] 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)
[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
```
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’ firmware.

- 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!

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

Twitter:  @ale_sp_brazil

Website:  
http://blackstormsecurity.com

E-mail:  
alexandreborges@blackstormsecurity.com

- 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)
- Referee on Digital Investigation: The International Journal of Digital Forensics & Incident Response
- Instructor at Oracle, (ISC)2 and Ex-instructor at Symantec.