REboot: Bootkits Revisited

Samuel Chevet

29 May 2014

Agenda

- Describe what a bootkit is
- How the Windows boot process works
- State of the art in the real world
- REboot project
- Conclusion

Plan

1. Bootkit

Rootkit

- Type of "malicious" software
- Kernel-Land
- Full control
- Hide malicious stuff
- Adding / Replacing portions of OS
- Proprietary software protections used it sometimes

Bootkit

Problem with x64 version

- Driver signing is mandatory
- Buy or steal certificate ?
- Kernel Protection

New attack

- Compromise the boot process
- Subvert 64-bit kernel mode driver signing
- Load malicious driver
- Botnets: Spam, steal credentials, DDOS, . . .

Bootkit

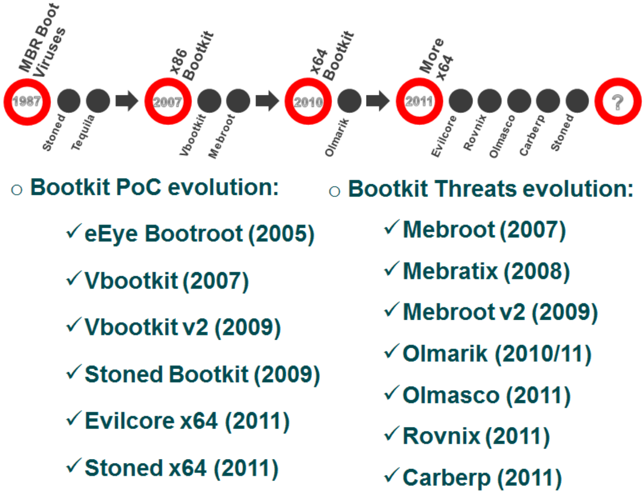
Problem with x64 version

- Driver signing is mandatory
- Buy or steal certificate ?
- Kernel Protection

New attack

- Compromise the boot process
- Subvert 64-bit kernel mode driver signing
- Load malicious driver
- Botnets: Spam, steal credentials, DDOS, . . .

Bootkit



○ **Bootkit PoC evolution:**

- ✓ eEye Bootroot (2005)
- ✓ Vbootkit (2007)
- ✓ Vbootkit v2 (2009)
- ✓ Stoned Bootkit (2009)
- ✓ Evilcore x64 (2011)
- ✓ Stoned x64 (2011)

○ **Bootkit Threats evolution:**

- ✓ Mebroot (2007)
- ✓ Mebratix (2008)
- ✓ Mebroot v2 (2009)
- ✓ Olmarik (2010/11)
- ✓ Olmasco (2011)
- ✓ Rovnix (2011)
- ✓ Carberp (2011)

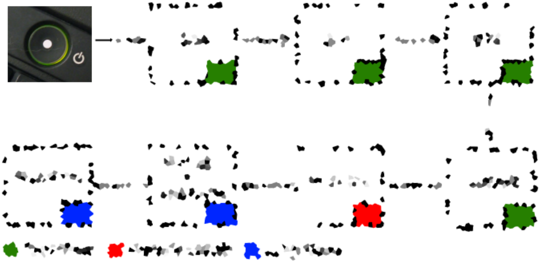
Bootkits' evolution (`http://www.welivesecurity.com/` ©)

Plan

2 Basics

- Boot process
  - BIOS
  - MBR
  - VBR
  - BootMGR
  - Winload
- Chain of trust

Boot process



Plan

2 Basics

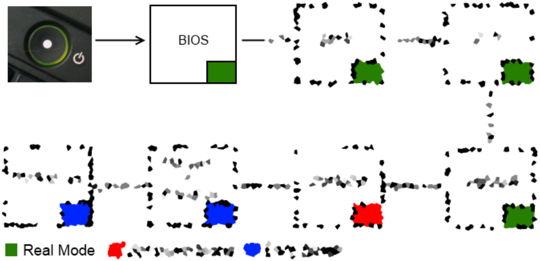
- Boot process
  - BIOS
  - MBR
  - VBR
  - BootMGR
  - Winload
- Chain of trust

BIOS

- Initialize and test the system hardware components
- Executed in Real mode
- Transfer execution to some other medium :
    - Disk drive
    - CD-ROM
    - Network boot
- **Load first sector of hardware drive at 0000:7C00**
- First sector is called Master Boot Record(MBR)

Some bogus BIOSes jump to 07C0:0000 instead of 0000:7C00

Boot process



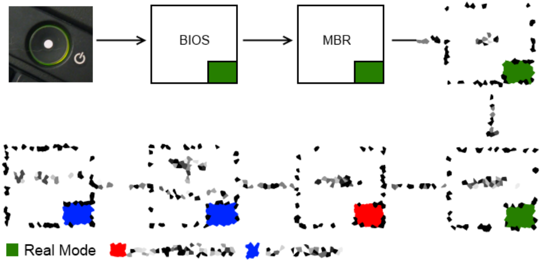
Plan

2 Basics
- Boot process
  - BIOS
  - MBR
  - VBR
  - BootMGR
  - Winload
- Chain of trust

Master Boot Record

- Executed in Real mode
- Copies itself to 0000:0600
- Searches bootable partition inside partition table
- **Copies first sector of bootable partition at 0000:7C00**
- Jump to 0000:7C00

Boot process



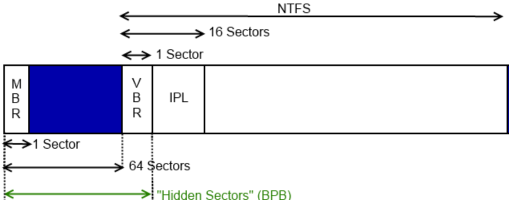
Plan

2 Basics

- Boot process
    - BIOS
    - MBR
    - VBR
    - BootMGR
    - Winload
- Chain of trust

Volume Boot Record

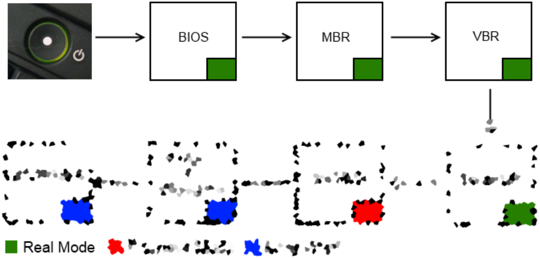
- 1 sector containing Bios Parameter Block (BPB)
- BPB structure is completely different from FAT to NTFS
- **BPB uses HiddenSectors field to load Initial Program Loader (IPL)**
- Jump to it



Initial Program Loader

- Ability to read FAT32 and NTFS
- **Load BootMGR at 2000h:0000h (0x20000)**
- Jump to it
- Or NTLDR for older version (branch is still here ;))

Boot process



🟩 Real Mode

Plan

2 Basics

- Boot process
  - BIOS
  - MBR
  - VBR
  - BootMGR
  - Winload
- Chain of trust

BootMGR

- **Map a 32 bit embedded executable to 0x400000**
- Activate protected mode
- Load GDT, IDT
- Checksum of the embedded file

BootMGR 32

- Ability to use symbols (.pdb) from Microsoft
- BmMain(x), BmFwVerifySelfIntegrity(x), ImgpLoadPEImage()
- Check for hibernation state

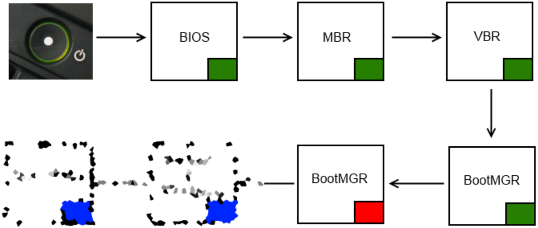
Hibernation state TRUE

- Load Winresume.exe

Hibernation state FALSE

- Mount BCD database, and enumerate boot entries, settings, . . .
- Change CPU mode to 64 bits
- **Load Winload.exe (BmpLaunchBootEntry(x, x, x))**

Boot process



🟩 Real Mode 🟥 Protected Mode

Plan

2 Basics
- Boot process
  - BIOS
  - MBR
  - VBR
  - BootMGR
  - Winload
- Chain of trust

Winload

- Setup minimal 64 bits kernel
- Enable paging
- Get Boot Options (DISABLE_INTEGRITY_CHECKS, TESTSIGNING, . . .)
- Load BCD entries
- **Fill LOADER_PARAMETER_BLOCK**
- Load SYSTEM Hives (system32\config\system)
- **Load Ntoskrnl.exe, hal.dll, SERVICE_BOOT_START drivers**
- Create PsLoadedModuleList

Winload

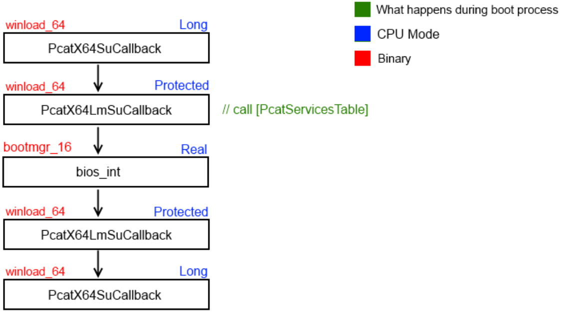
GDT Entry

- Code entry for long mode
- Code entry for protected mode
- Data entry for protected mode
- Tss for long mode
- Code entry for real mode
- Data entry for real mode
- Data entry for framebuffer (0x000B8000)

Winload

BIOS interruption while in Long mode

- Winload needs to read / write files
- Print UI, get keyboard input, . . .
- **Winload is able to execute BIOS interruption**



🟩 What happens during boot process

🟦 CPU Mode

🟥 Binary

winload_64 — Long

PcatX64SuCallback

↓

winload_64 — Protected

PcatX64LmSuCallback — // call [PcatServicesTable]

↓

bootmgr_16 — Real

bios_int

↓

winload_64 — Protected

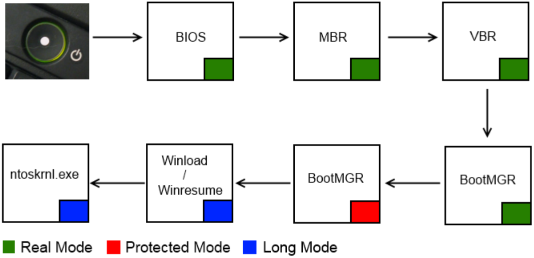
PcatX64LmSuCallback

↓

winload_64 — Long

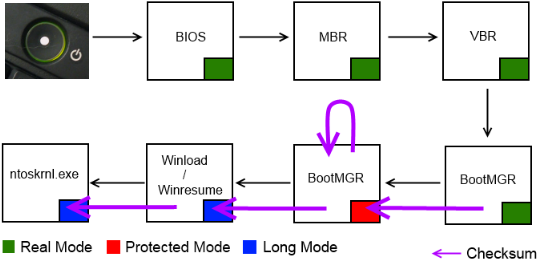
PcatX64SuCallback

Boot process



🟩 Real Mode 🟥 Protected Mode 🟦 Long Mode

Chain of trust



BIOS → MBR → VBR → BootMGR → BootMGR → Winload / Winresume → ntoskrnl.exe

🟩 Real Mode 🟥 Protected Mode 🟦 Long Mode

← Checksum

Plan

3 State of the art

- Type of infection
- Payload
- Problems

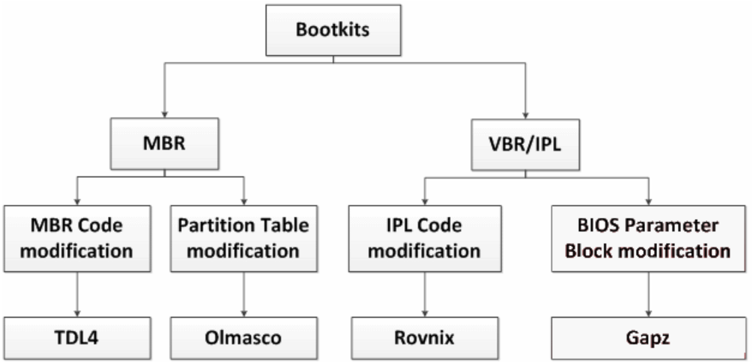
State of the art

- In 2010, bad guys started to attack 64 bits system
- TDL, aka Alureon family of malware

Some Bootkits

- TDL4
- Turla
- gapz
- xpaj
- Cidox
- yurn
- prioxer
- rovnix
- ...

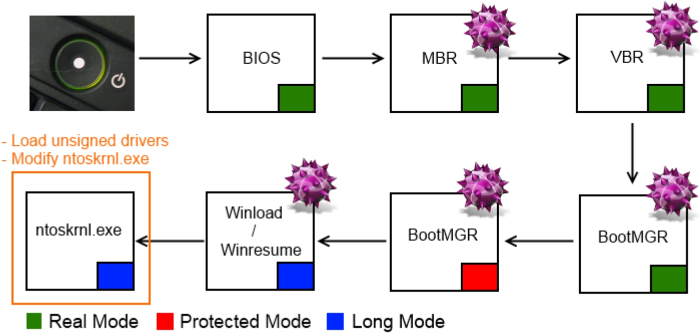
Type of infection



Bootkit techniques (`http://www.welivesecurity.com/` ©)

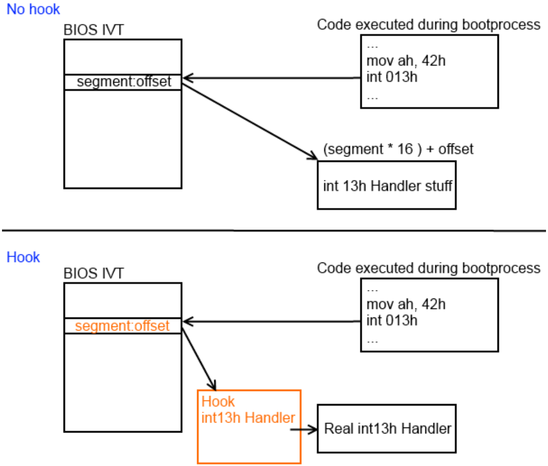
Payload

- Keep control during all bootprocess stages until Ntoskrnl.exe loading
- Final malicious payload is injected during Ntoskrnl.exe stage



Payload

- BIOS provides interruptions
- int 013h (Function : 042h) : Extended Read Sectors
- Hook this interruption
- Same technique used in all infection methods



Hook interest

- Scan all disk read operations inside hook
- Patch file in memory
- Setup new trampoline in next stage
- (Ex : from MBR -> VBR, VBR -> BootMGR, . . .)
- Final goal is to reach Ntoskrnl.exe loading
- Load unsigned drivers
- Disable Kernel Protection

Open Source Project

- StonedBootkit
- VBootkit
- DreamBoot
- . . .

Problems

- Focused only on executable (VBR, BootMGR_16, BootMGR_32, Windload)
- Most bootkits rely on code modifications and hooks:
  - Those are setuped based on patterns matching and hardcoded offsets
  - Require to patch the chain of trust
- Those techniques are not reliable:
  - Not generic across all Windows versions
  - TrueCrypt & BitLocker are not supported (one project setup two hook layers)
  - Can easily be detected

Plan

4 REboot

- Research
- Real mode to Protected mode
- Protected mode to Long mode
- Winload to Ntoskrnl
- Payload

Research

- Create a proof of concept able to control all bootprocess stages until Windows kernel startup
- Not based on currently well known techniques

Goal

- Find a new way to implement bootkits on Windows using generic methods
- **Bypass Windows bootprocess chain of trust**
- **Load unsigned drivers at boot**

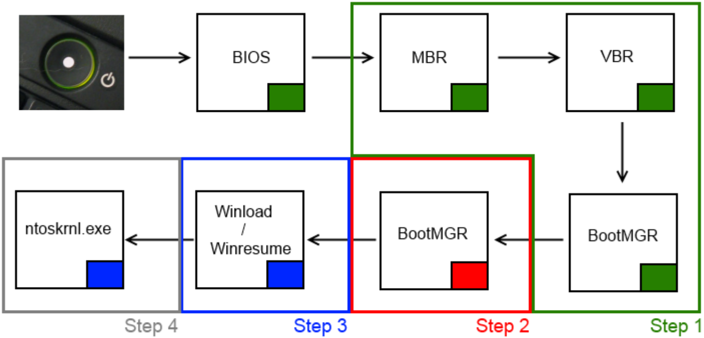
Ideas

- Main problems are CPU mode switches while booting:
  - Real mode (16 bits)
  - Protected mode (32 bits)
  - Long mode (64 bits)
- We want to be able to execute arbitrary code at each stage
- **Without using hooks or scanning patterns in memory**
- So we only use provided processor features!

Four main steps

1. From Real mode (16 bits) to Protected mode (32 bits)
2. From Protected mode to Long mode (64 bits, Winload)
3. From Winload to Ntoskrnl
4. Payload execution

4 Steps



🟩 Real Mode 🟥 Protected Mode 🟦 Long Mode

Plan

4. REboot
   - Research
   - Real mode to Protected mode
   - Protected mode to Long mode
   - Winload to Ntoskrnl
   - Payload

Virtual 8086 mode

- Virtual 8086 mode is a sub-mode of Protected mode
- V86 allows to execute 8086 code under protected mode
- NTVDM
- Virtual machine (VM) bit in the EFLAGS (bit #17) register is set
- We need only one task
- popf does not work, use iret or 386 TSS
- **Trap on privileged instruction, like lgdt**

Virtual 8086 mode

Problem encountered

- At first we used an I/O privilege level (IOPL) equal to 3
- Only exceptions during privilegied instructions
- TPM BIOS interruption (0x1A) setup a protected mode
- False positive detection of BootMGR

Virtual 8086 mode

Solution

- Use IOPL equal to 1
- When an interruption is trying to be executed
    1. We setup back real mode CPU
    2. Execute it
    3. We go back to v8086 mode

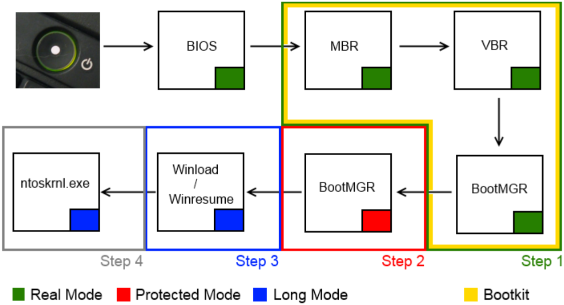
Virtual 8086 mode

Step by Step

- Setup Protected mode
- Load original MBR
- Setup and enable VM 86 mode
- Jump to original MBR
- Manage all exceptions
- **GP Handler executed during lgdt instruction**

Real mode to Protected mode

First step has been solved using V8086 mode



Plan

4 REboot
- Research
- Real mode to Protected mode
- Protected mode to Long mode
- Winload to Ntoskrnl
- Payload

Protected mode to Long mode

- With V8086 mode, we control until BootMGR_32
- BootMGR_32 must :
  - Prepare Long mode in case of 64 bits kernel
  - Setup new GDT and IDT
  - Enable paging
- **This new IDT must be placed on an allocated page**
- All these operations are carried out by ImgArchPcatStartBootApplication() function

Protected mode to Long mode

ImgArchPcatStartBootApplication()

- Setup a page for new GDT and IDT
- Use sidt instruction to get current IDT entries (created by BootMGR_16) and copy them to the new one
- Test IMAGE_FILE_HEADER->Machine for starting 32 bits application or 64 bits

ImgPcatStart64BitApplication()

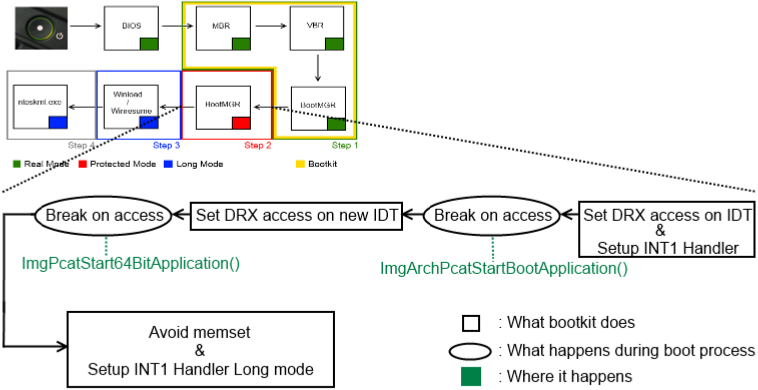
- Case for 64 bits application
- **Reset all new IDT entries because it is invalid for Long mode**

Protected mode to Long mode

When in protected mode we can :

- Use Debug registers (dr0 . . . dr3)
- Setup Debug Interrupt (0x1)
- We control until Winload execution

Protected mode to Long mode



Break on access ← Set DRX access on new IDT ← Break on access ← Set DRX access on IDT & Setup INT1 Handler

ImgPcatStart64BitApplication()

ImgArchPcatStartBootApplication()

Avoid memset
&
Setup INT1 Handler Long mode

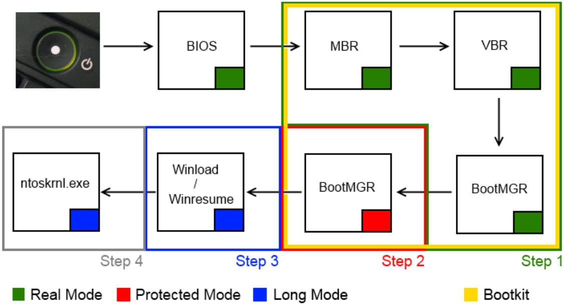
☐ : What bootkit does

○ : What happens during boot process

🟩 : Where it happens

Protected mode to Long mode

Second step has been solved using debug registers



Plan

4. REboot
   - Research
   - Real mode to Protected mode
   - Protected mode to Long mode
   - Winload to Ntoskrnl
   - Payload

Winload to Ntoskrnl

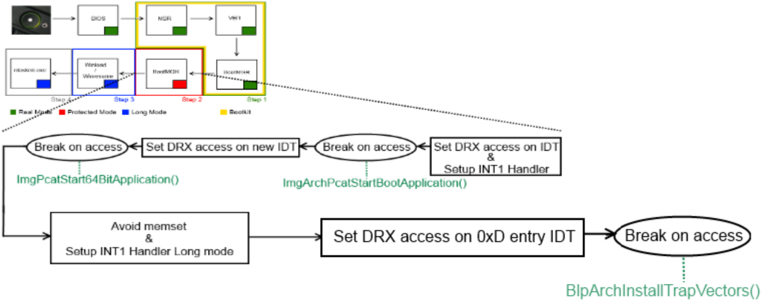
- With debug registers, we control until Winload
- Winload starts with an empty IDT_64

BlpArchInstallTrapVectors()

- Retrieve IDTR with ArchGetIdtRegister() and setup new Long mode entries

- We can setup a DRX on access on these entries before switching from Protected mode to Long mode

Winload to Ntoskrnl



Set DRX access on IDT & Setup INT1 Handler

Break on access

ImgArchPcatStartBootApplication()

Set DRX access on new IDT

Break on access

ImgPcatStart64BitApplication()

Avoid memset & Setup INT1 Handler Long mode

Set DRX access on 0xD entry IDT

Break on access

BlpArchInstallTrapVectors()

☐ : What bootkit does

⬭ : What happens during boot process

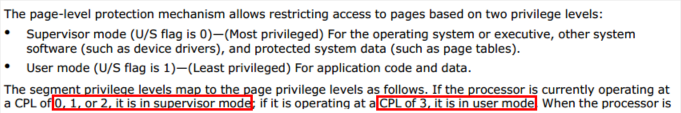
🟩 : Where it happens

Winload to Ntoskrnl

- Now we can control execution "inside" Winload
- We want to monitor the transition between Winload and Ntosknrl
- **Winload will setup a new GDT and IDT before jumping to kernel**
- We can follow these operations by tracing privileged instructions
- So we run Winload's code at ring 1 privilege (DPL=1)

Why ring 1?

- Winload sections are in paged area



The page-level protection mechanism allows restricting access to pages based on two privilege levels:

- Supervisor mode (U/S flag is 0)—(Most privileged) For the operating system or executive, other system software (such as device drivers), and protected system data (such as page tables).
- User mode (U/S flag is 1)—(Least privileged) For application code and data.

The segment privilege levels map to the page privilege levels as follows. If the processor is currently operating at a CPL of 0, 1, or 2, it is in supervisor mode; if it is operating at a CPL of 3, it is in user mode. When the processor is

Intel 64 and IA-32 Architectures Developer's Manual: Vol. 3A 4-38

Ring 1

- Setup new Code / Data segment with DPL = 1
- Setup General Protection fault handler
- Fill rsp0 field inside TSS_64

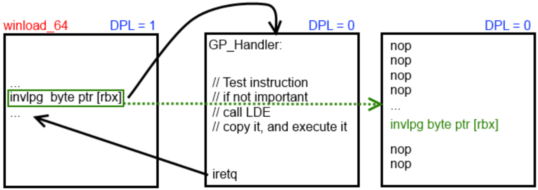
GP Handler

- Check where the fault occured
- Check what privileged instruction occured
- Copy it and execute it somewhere else
- Or "emulate" it

Ring 1

Example

- mov ds, ax
- mov rax, cr3
- jmp far . . .
- . . .



winload_64 DPL = 1

```
...
invlpg  byte ptr [rbx]
...
```

DPL = 0

```
GP_Handler:

 // Test instruction
 // if not important
 // call LDE
 // copy it, and execute it


 iretq
```

DPL = 0

```
nop
nop
nop
nop
...
invlpg byte ptr [rbx]

nop
nop
```

Ring 1 : Special cases

mov ds, ax

- In PcatX64SuCallback
- Winload wants to update data segment to perform a BIOS interrupt (swich from long mode to real mode)
- At this point, restore ring0 to avoid any problem
- Wait come back from real mode (jmp far 10h:343D31h)

jmp far XX:YYYY

- Fault occurs because DPL != RPL
- Update cs, ss and ip before iretq

mov ss, ax

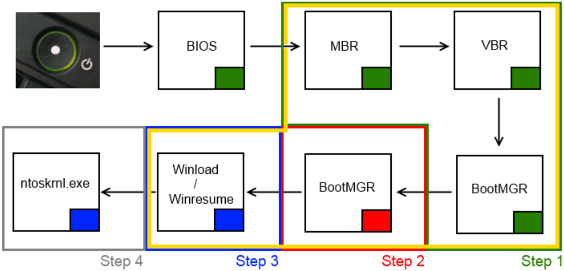
- Happen just after jmp:far
- Avoid instruction

Winload to Ntoskrnl

- All other cases can be copied and executed from somewhere else
- Last case is lgdt fword ptr [rax]
- In function : OslArchTransferToKernel
- Just before jumping into Ntoskrnl.exe
- First parameter of KiSystemStartup() is LOADER_PARAMETER_BLOCK
- +0x10 : _LDR_DATA_TABLE_ENTRY (boot driver)

Winload to Ntoskrnl

Third step has been solved using ring protection


BIOS
MBR
VBR
BootMGR
BootMGR
Winload / Winresume
ntoskrnl.exe
Step 4
Step 3
Step 2
Step 1

🟩 Real Mode 🟥 Protected Mode 🟦 Long Mode 🟨 Bootkit

Plan

4. REboot
   - Research
   - Real mode to Protected mode
   - Protected mode to Long mode
   - Winload to Ntoskrnl
   - Payload

Just before Ntoskrnl.exe

- Inject our own driver in the PsLoadModuleList
- We have access to ntoksrnl's APIs
- But we cannot use it because kernel is not initialised
- So replace EntryPoint of known drivers
- But most of driver's entry point are called from hal.dll, kernel is still not fully initialised
- So replace export function of kdcom.dll (KdDebuggerInitialize1)

Payload

- We do not want to inject specific payload
- Goal is loading unsigned drivers
- Use undocumented method to avoid signature checking

Undocumented method

- IoCreateDriver(PUNICODE_STRING DriverName, PDRIVER_INITIALIZE InitializationFunction)
- Function exported by Ntoskrnl.exe in order to create a driver object
- DriverName can be null

Payload

- We do not want to inject specific payload
- Goal is loading unsigned drivers
- Use undocumented method to avoid signature checking

Undocumented method

- IoCreateDriver(PUNICODE_STRING DriverName, PDRIVER_INITIALIZE InitializationFunction)
- Function exported by Ntoskrnl.exe in order to create a driver object
- DriverName can be null

Payload

InitializationFunction

- Open and Read (PE) driver file
- Map sections in memory
- Resolve imports
- Fix image relocations
- Fill information of DRIVER_OBJECT
- Call entry point

Driver example

- Patch msv1_0!MsvpPasswordValidate from LSASS process
- Escalate privileges of any cmd.exe command
- Change behavior of CTRL+ALT+DEL
- . . .

Plan

5 Conclusion

Demo

Demo time !

TODO

Still work to be done

- Implementing UEFI (without SecureBoot)
- More work to do with BitLocker or TrueCrypt: Extract passphrase at boot

Conclusion

- Real interest to use bootkit techniques, for loading unsigned drivers
- REBoot uses no memory modifications!
- Chain of trust defeated
- Works on all 64 bits Windows versions
- Virtual environments or emulated environments
- Physical machines with BIOS or UEFI legacy
- Does not work if UEFI Secureboot is present

Questions ?

Thank you for your attention