eEye BootRoot

This presentation will cover the eEye BootRoot project, an exploration of technology that boot sector code can use to subvert the Windows NT-family kernel and retain the potential for execution, even after Windows startup—a topic made apropos by the recent emergence of Windows rootkits into mainstream awareness. We will provide some brief but technical background on the Windows startup process, then discuss BootRoot and related technology, including a little-known stealth technique for low-level disk access. Finally, we will demonstrate the proof-of-concept BootRootKit, loaded from a variety of bootable media.

Derek Soeder is a Software Engineer and after-hours researcher at eEye Digital Security. In addition to participating in the ongoing development of eEye’s Retina Network Security Scanner product, Derek has also produced a number of internal technologies and is responsible for the discovery of multiple serious security vulnerabilities. His main areas of interest include operating system internals and machine code-level manipulation.

Ryan Permeh is a Senior Software Engineer at eEye Digital Security. He focuses mainly on the Retina and SecureIIS product lines. He has worked in the porting of nmap and libnet to Windows, as well as helping with disassembly and reverse engineering, and exploitation efforts within the eEye research team.
Introduction

- Explores the capabilities of custom boot sector code on NT-family Windows
  - What can it do? Anything – it's privileged code on the CPU
  - The trick is keeping control while allowing the OS to function

- Overview
  - BIOS boot process and Windows startup
  - eEye BootRoot: how it works, capabilities and shortcomings
  - Demo: eEye BootRootKit backdoor

- Required Knowledge
  - x86 real and protected modes, some Windows kernel

digital self defense
Booting Up

BIOS Handoff to Bootstrap Code

Booting Up – Summary

- BIOS transfers execution to code from some other medium
  - Disk drive (fixed or removable)
  - CD-ROM
  - Network boot

- Windows startup from a hard drive installation
  - Hard drive Master Boot Record
  - Windows bootstrap loader
  - NTLDR
  - OSLOADER.EXE
  - NTDETECT.COM
  - NTOSKRNL.EXE, HAL.DLL, boot drivers
Booting Up – Disk Drive

- BIOS loads first sector of drive (200h bytes) at 0000h:7C00h
  - Executes in real mode
  - SS:SP < 0000h:0400h, DS = 0040h (BIOS data area)

- For hard drives, the first sector is the Master Boot Record
  - Copies itself to 0000h:0600h
  - Locates a bootable partition in the partition table
  - Executes the first sector of the boot partition at 0000h:7C00h

- Partition boot sector is always part of the operating system
  - Loads and executes the next boot stage of the OS

---

Booting Up – MBR Partition Table

Master Boot Record Layout

<table>
<thead>
<tr>
<th>Offset</th>
<th>Type</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>Boot Indicator</td>
<td>1 Byte</td>
<td>bit 7: partition bootable</td>
</tr>
<tr>
<td>0001</td>
<td>Starting Head</td>
<td>1 Byte</td>
<td>bits 7..0: cylinder</td>
</tr>
<tr>
<td>0002</td>
<td>Starting Sector</td>
<td>2 Bytes</td>
<td>bits 7..0: cylinder</td>
</tr>
<tr>
<td>0003</td>
<td>Ending Cylinder</td>
<td>1 Byte</td>
<td>bits 7..0: cylinder</td>
</tr>
<tr>
<td>0004</td>
<td>System ID</td>
<td>1 Byte</td>
<td>volume type</td>
</tr>
<tr>
<td>0005</td>
<td>Ending Head</td>
<td>1 Byte</td>
<td>bits 7..0: cylinder</td>
</tr>
<tr>
<td>0006</td>
<td>Ending Sector</td>
<td>2 Bytes</td>
<td>bits 7..0: cylinder</td>
</tr>
<tr>
<td>0007</td>
<td>Linear Sectors</td>
<td>4 Bytes</td>
<td>bits 7..0: cylinder</td>
</tr>
<tr>
<td>0008</td>
<td>Size in Sectors</td>
<td>4 Bytes</td>
<td>bits 7..0: cylinder</td>
</tr>
</tbody>
</table>

---

Source: NTFS.com Hard Drive Partition - Partition Table
http://www.ntfs.com/partition-table.htm
Booting Up – CD-ROM

- Differences from disks and diskettes
  - Sector size is 800h bytes (2KB)
  - Data format is more complicated (ECMA-119 / ISO 9660)
  - Bootable CD format dictated by “El Torito” Specification

- Boot sector (only first 200h bytes) loads at 07C0h:0000h
  - Executes in real mode
  - SS:SP = 0000h:0400h, DS = 0040h (BIOS data area)

- Additional disc contents are accessed via INT 13h
  - Boot catalog entry indicates “emulation mode” (floppy or HD)


digital self defense
Booting Up – Bootable CD Layout (2)

8800 BYTE Volume Descriptor Type = 0
8801 [5] Standard Identifier = "CD001"
8806 BYTE Volume Descriptor Version = 1
8807 [20h] Boot System Identifier = "EL TORITO SPECIFICATION", [0]
8847 DWORD Pointer to First Sector of Boot Catalog

9800 [20h] Validation Entry
9800 BYTE Header ID = 1
9801 BYTE Platform ID = 0
981C WORD Checksum = 55AAh
981E WORD Key = AA55h
9820 [20h] Initial/Default Entry
9820 BYTE Boot Indicator = 88h
9821 BYTE Boot Media Type = 2 (1.44MB floppy)
9822 WORD Load Segment = 0
9824 BYTE System Type = 0
9826 WORD Sector Count (virtual sectors) = 1
9828 DWORD Load RVA (sector) = 14h


digital self defense
Booting Up – Network Boot

- **PXE: Preboot eXecution Environment**
  - Network boot via BOOTP (basis for DHCP) and TFTP
  - BIOS PXE boot agent requests configuration over BOOTP
    - Requires an IP address, server's IP address, and boot file name
    - BOOTP server receives on UDP/67, client on UDP/68
  - Downloads boot file from TFTP service on server
    - TFTP server receives on UDP/69

- **Executes boot file in real mode at 0000h:7C00h**
  - Up to ~500KB of data will be downloaded and stored there
  - Register values should be considered undefined

Booting Up – Network Boot Traffic Example

<table>
<thead>
<tr>
<th>Client IP</th>
<th>Port</th>
<th>Packet</th>
<th>Server IP</th>
<th>Port</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0.0.0</td>
<td>68</td>
<td>DHCP Discovery</td>
<td>=&gt; 255.255.255.255</td>
<td>67</td>
</tr>
<tr>
<td>255.255.255.255</td>
<td>68</td>
<td>&lt;- DHCP Offer</td>
<td>(server IP)</td>
<td>67</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[Server Identifier = (server IP); Boot File Name = &quot;...&quot;]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.0.0.0</td>
<td>68</td>
<td>DHCP Request</td>
<td>=&gt; 255.255.255.255</td>
<td>67</td>
</tr>
<tr>
<td>255.255.255.255</td>
<td>68</td>
<td>&lt;- DHCP Ack</td>
<td>(server IP)</td>
<td>67</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[Server Identifier = (server IP); Boot File Name = &quot;...&quot;]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(client ID) (var) TFTP Read Req => (server ID) 69
[File: (boot file name); Mode: "octet"; "tsize" = 0; "blksize" = (block size)]

(client ID) (var) <- TFTP Option ACK (server ID) 69
["tsize" = (size of boot file); "blksize" = (supported block size)]

(client ID) (var) TFTP ACK => (server ID) 69
[Block: 0]

(client ID) (var) <- TFTP Data (server ID) 69
[Block: 1; file data]

(client ID) (var) TFTP ACK => (server ID) 69
[Block: 1]

...
Windows Startup

Windows Boot Sector to NTOSKRNL Execution

Windows Startup – Boot Loader

- Windows partition boot sector
  - Loads first 16 sectors (itself is first) at 0D00h:0000h
  - Uses IBM/MS INT 13h Extensions if available
  - Passes execution to next stage of Windows boot loader

- Windows boot loader
  - Loads and executes NTLDR at 2000h:0000h in real mode
  - Does not export any functionality to NTLDR
  - Only uses ~40% of its allotted 8KB (room for our code?)
### Windows Startup – NTLDR

- **Enters 16-bit protected mode**
  - Creates GDT and IDT for use throughout Windows startup
  - Wraps real mode BIOS interrupt functionality that subsequent protected mode startup code will invoke:
    - INT 10h: Video
    - INT 13h: Disk
    - INT 14h: Serial
    - INT 15h: System Configuration, Power Management

- **Maps OSLOADER.EXE at its preferred image base**
  - OSLOADER.EXE is a PE image embedded in NTLDR
  - No MZ header or PE signature prior to Windows 2003
  - NTLDR executes its entry point in 32-bit protected mode

<table>
<thead>
<tr>
<th>Entry Point</th>
<th>Limit</th>
<th>Base</th>
<th>DPL</th>
<th>P</th>
<th>A</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>#0008</td>
<td>FFFFFFFF</td>
<td>00000000</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>Code32</td>
</tr>
<tr>
<td>#0010</td>
<td>FFFFFFFF</td>
<td>00000000</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>Data32</td>
</tr>
<tr>
<td>#0018</td>
<td>FFFFFFFF</td>
<td>00000000</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>Code32</td>
</tr>
<tr>
<td>#0020</td>
<td>FFFFFFFF</td>
<td>00000000</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>Data32</td>
</tr>
<tr>
<td>#0028</td>
<td>00000000</td>
<td>00024460</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>Task Gate</td>
</tr>
<tr>
<td>#0030</td>
<td>00000100</td>
<td>00000000</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>Data32</td>
</tr>
<tr>
<td>#0038</td>
<td>000000FF</td>
<td>00000000</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>Data32</td>
</tr>
<tr>
<td>#0040</td>
<td>000000FF</td>
<td>00000400</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>Data16</td>
</tr>
<tr>
<td>#0048</td>
<td>(reserved)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>#0050</td>
<td>0000006F</td>
<td>00023B7E</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>Task Gate</td>
</tr>
<tr>
<td>#0058</td>
<td>000000FF</td>
<td>00020000</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>Code16</td>
</tr>
<tr>
<td>#0060</td>
<td>000000FF</td>
<td>00022F30</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>Data16</td>
</tr>
<tr>
<td>#0068</td>
<td>000003FF</td>
<td>000B8000</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>Data16</td>
</tr>
<tr>
<td>#0070</td>
<td>000003FF</td>
<td>00007000</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>Data16</td>
</tr>
<tr>
<td>#0078</td>
<td>000000FF</td>
<td>00400000</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>Data16</td>
</tr>
<tr>
<td>#0080</td>
<td>000000FF</td>
<td>00400000</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>Data16</td>
</tr>
<tr>
<td>#0088</td>
<td>00000000</td>
<td>00000000</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>Data16</td>
</tr>
</tbody>
</table>

---

digital self defense
Windows Startup – OSLOADER.EXE (1)

- OSLOADER.EXE loads the operating system
  - Processes \BOOT.INI
  - Executes NTDETECT.COM in real mode at 1000h:0000h
  - Enables paging
    - Applies /3GB BOOT.INI option
    - Sets typical virtual addresses for GDT, IDT, and page tables
  - Loads HAL.DLL and NTOSKRNL.EXE, and any import dependencies (BOOTVID.DLL), at their preferred virtual addresses, and applies relocations
  - Loads the registry (system32\config\system)
  - Loads NLS code pages and required fonts

Windows Startup – OSLOADER.EXE (2)

- OSLOADER.EXE loads boot drivers
  - Loads drivers with a Start type of Boot (0)
    - Creates a PsLoadedModuleList-format list (*.BLoaderBlock)
    - Does not realign image sections prior to Windows 2003: in-memory image is the raw file contents!
  - Drivers do not execute at this stage

- Transfers execution to NTOSKRNL.EXE entry point
Windows Startup – NTOSKRNL.EXE

- NTOSKRNL and HAL.DLL finish initializing machine state
  - NTOSKRNL assumes control of TSS, IDT, and GDT
  - Initializes processor(s) and ABIOS support

- Kernel subsystems initialize in two passes or “phases”
  - Phase 0 initialization
    - KiSystemStartup calls KiInitializeKernel, which calls ExpInitializeExecutive
  - Phase 1 initialization
    - Phase1Initialization executes as a separate system thread
    - Boot drivers execute during this phase
    - Finishes kernel initialization and starts user-mode SMSS.EXE
  - “Phase 2” mostly deals with licensing (ExInitSystemPhase2)

Windows Startup – Phase 0 Initialization

- NTOSKRNL.EXE!KiSystemStartup
  - HAL.DLL!HalInitializeProcessor
  - KiInitializeKernel
    - KiInitSystem (initializes _KeServiceDescriptorTable and _KeServiceDescriptorTableShadow)
    - KeInitializeProcess (_KiIdleProcess), KeInitializeThread (P0BootThread)
    - ExpInitializeExecutive
      - HAL.DLL!HalInitSystem
      - ExInitSystem
      - MmInitSystem (0)
      - ObInitSystem
      - SeInitSystem
      - PsInitSystem (creates _PsInitialSystemProcess and Phase1Initialization thread)
      - PpInitSystem
Windows Startup – Phase 1 Initialization

- NTOSKRNL.EXE\Phase1Initialization
  - HAL.DLL\HalInitSystem
  - PolntSystem (0)
  - ObInitSystem
  - ExInitSystem
  - KelntSystem
  - SelInitSystem
  - MmInitSystem (1)
  - CmlInitSystem
  - FsRtlInitSystem
  - PpInitSystem
  - LpcInitSystem
  - ExInitSystemPhase2
  - IoInitSystem (loInitializeSystemDrivers runs boot drivers, PsLocateSystemDll loads NTDLL.DLL)
  - MmInitSystem (2) (makes executive pageable)
  - PoInitSystem (1)
  - PsInitSystem (locates certain NTDLL exports)

---

eEye Digital Security

---

eEye BootRoot

Technology for Windows Kernel Pre-Subversion

digital self defense
eEye BootRoot – The Problem

• We execute after the BIOS but before the operating system

• Advantages
  – Our code is privileged – real mode is “ring 0”
  – We can control all subsequent code execution

• Disadvantages
  – No part of the operating system is loaded yet
  – We need the system to function normally, except with a few of our own “adjustments”
  – OS startup will bring about dramatic machine state changes

eEye BootRoot – Playing Field

• Real mode environment features
  – Interrupt Vector Table (100h doublewords at 0000h:0000h)
    • Hooking BIOS interrupt services is like hooking APIs
  – BIOS data area (100h bytes at 0040h:0000h)
    • See Ralf Brown’s MEMORY.LST for more information
  – 640KB conventional memory

• CPU and hardware settings
  – CRn, DRn, GDTR, IDTR, MSRs, etc.
  – Chipset: e.g., Programmable Interrupt Controller
  – Any hardware device
  – Other processors...?
eEye BootRoot – Game Plan

- Windows startup will assume exclusive control over almost every facet of machine state...
  - CPU state, IRQs, chipset, eventually most hardware
  - Eventually all other CPUs in a multiprocessor system
  - Unused memory

- ...But its weakness is reliance upon the BIOS
  - It uses BIOS interrupts, so IVT is mostly preserved
  - It has to respect memory ranges reserved by BIOS

We can exploit this trust to function like a BIOS “hook”

eEye BootRoot – Our Solution

- “Go resident” – reserve memory for a copy of our code
  - Reduce conventional memory KB reported by 0040h:0013h
    - Boot virii have used this technique forever

- Hook INT 13h (Disk) to “patch” OS files as they load
  - Scan for a code signature in OSLOADER and patch there
  - Must handle INT 13h/AH=02h (Read Sectors) and INT 13h/AH=42h (IBM/MS Extensions – Extended Read)

- OSLOADER patch gives us an intermediate point to regain execution and modify OS further (i.e., patch boot drivers)
### eEye BootRoot – Other Possibilities

- **Modify system files on disk before Windows startup**
  - Intrusive; requires code to navigate FAT and NTFS
  - Could we piggyback off Windows boot loader code?

- **Hook INT 15h to reserve any amount of extended memory**
  - OSLOADER calls INT 15h/AX=E820h to get memory map

- **Regain execution by hooking an interrupt called late in Windows startup**
  - More of OS is loaded – more available to modify
  - Our hook runs in real mode, so we must re-enter protected mode to modify OS memory above 1MB

---

### eEye BootRoot – System Memory Map Example

<table>
<thead>
<tr>
<th>Base Address</th>
<th>Length</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000000000009F800</td>
<td>00000000000F800</td>
<td>Available</td>
</tr>
<tr>
<td>0000000000009F800</td>
<td>000000000000800</td>
<td>(Reserved)</td>
</tr>
<tr>
<td>00000000000000CA000</td>
<td>000000000002000</td>
<td>(Reserved)</td>
</tr>
<tr>
<td>00000000000000DC000</td>
<td>000000000004000</td>
<td>(Reserved)</td>
</tr>
<tr>
<td>0000000000000024000</td>
<td>000000000001C000</td>
<td>(Reserved)</td>
</tr>
<tr>
<td>0000000000000010000</td>
<td>000000000007DF000</td>
<td>Available</td>
</tr>
<tr>
<td>000000000000007EF0000</td>
<td>0000000000000C000</td>
<td>(ACPI Reclaimable)</td>
</tr>
<tr>
<td>000000000000007EFC0000</td>
<td>0000000000004000</td>
<td>(ACPI NVS)</td>
</tr>
<tr>
<td>000000000000007F00000</td>
<td>00000000000100000</td>
<td>Available</td>
</tr>
<tr>
<td>000000000000000FEC0000</td>
<td>0000000000010000</td>
<td>(Reserved)</td>
</tr>
<tr>
<td>000000000000000FEE0000</td>
<td>0000000000001000</td>
<td>(Reserved)</td>
</tr>
<tr>
<td>000000000000000FFE0000</td>
<td>0000000000002000</td>
<td>(Reserved)</td>
</tr>
</tbody>
</table>

System memory map generated using INT 15h/AX=E820h on a VMWare 4.5 system with 128MB RAM.
eEye BootRootKit

“Finally, someone implemented it.”

eEye BootRootKit – Overview

• Proof-of-concept for eEye BootRoot technology
  – Loads from many bootable media
  – Installs INT 13h hook to "patch" OSLOADER on load
  – OSLOADER patch locates module list, hooks NDIS.SYS
  – NDIS backdoor inspects incoming packets for code to run

• Features
  – Works on Windows 2000 and later
  – Fits into 512 bytes!

• The idea is simple, but there are always hidden complexities
eEye BootRootKit – INT 13h Hook

- Move to reserved conventional memory and hook INT 13h
  - Warning: Don’t assume value of CS!
    • Executed from disk – CS:IP = 0000h:7C00h
    • Executed from CD – CS:IP = 07C0h:0000h

- INT 13h hook scans read sectors for a code signature
  - INT 13h hook must be able to handle INT 13h/AH=02h and
    INT 13h/AH=42h extended reads (required for large disks)
  - Signature should be unique, cross-version, and must not be
    split across a read boundary (i.e., across two sectors)
  - Warning: OSLOADER verifies PE checksums (except itself)
    • Could disable checksum checking code... ("CMP reg1, [reg2+58h]")

We patch 6 bytes executed after boot driver load:

- Hook must be absolute – we don’t know where code will load
  • “CALL seg:ofs32” is 7 bytes
  • “CALL DWORD PTR [ofs32]” is 6 bytes – perfect for this patch site
- We use “CALL DWORD PTR [addr1]”, where [addr1] = addr2, and both addr1 and addr2 are addresses in our resident code
- Paging is not a concern – OSLOADER will map low 16MB virtual memory to low 16MB physical memory
eEye BootRootKit – OSLOADER Patch Function

- Scan OSLOADER for address of _BlLoaderBlock
  - Assume OSLOADER begins on a 1MB boundary
    - 00300000h for 2000 and XP, 00400000h for 2003
  - Use CALL hook return address as pointer into OSLOADER
  - Scan for the following code signature:
    00301888 C7 46 34 00 40 00 00    MOV  DWORD PTR [ESI+34h], 4000h
    ...
    00301895 A1 xx xx xx xx          MOV  EAX, [ _BlLoaderBlock]
  - [ _BlLoaderBlock]+0 points to base of module list

- Search module list for NDIS.SYS
  - Name is usually uppercase, but don’t assume

eEye BootRootKit – OSLOADER Module List

+00h LIST_ENTRY module list links
+08h [ 10h] ???
+18h PTR image base address
+1Ch PTR module entry point
+20h DWORD size of loaded module in memory
+24h UNICODE_STRING full module path and file name
+2Ch UNICODE_STRING module file name

Format of loaded module list nodes used by OSLOADER and based at [ _BlLoaderBlock]+0.
Structure is identical to that used by NTOSKRNL in PsLoadedModuleList.
**eEye BootRootKit – Hooking NDIS (1)**

- **Scan NDIS.SYS for code signature**
  - This signature within ndisMLoopbackPacketX:
    ```
    BFCCEE7E 50                      PUSH EAX
    BFCCEE7F 53                      PUSH ECX
    BFCCEE80 C7 46 10 0E 00 00 00    MOV  DWORD PTR [ESI+10h], 0Eh
    BFCCEE87 E8 xx xx xx xx          CALL ethFilterDprIndicateReceivePacket
    ```
  - Leads to ethFilterDprIndicateReceivePacket, which we hook
  - Note: These two functions are in different PE sections

- **In 2000 and XP, boot drivers’ sections aren’t aligned yet!**
  - We must translate raw offsets into Relative Virtual Addresses, and vice versa, to find actual CALL destination and then store our own relative JMP hook there
  - If listed module size is 64KB multiple, sections are aligned(?)

**eEye BootRootKit – Hooking NDIS (2)**

- **Hook ethFilterDprIndicateReceivePacket**
  - Store a relative JMP at function entry point, or two bytes afterward if first instruction is “MOV EDI, EDI”
  - Assume overwritten instructions will always be:
    ```
    PUSH EBP / MOV EBP, ESP / SUB ESP, imm (exact value is irrelevant, we just subtract a lot)
    ```
  - Write protection not enabled yet, modify away!
  - Code is not pageable so it will never be reloaded from disk

- **Store hook function code**
  - We overwrite DOS “MZ” code at (image base + 40h)
  - This hook function provides a remote kernel backdoor
eEye BootRootKit – NDIS Backdoor

- Hook function checks received packets for signature
  - ethFilterDprIndicateReceivePacket sees all incoming frames
  - arg_4 0 8 0C is pointer to Ethernet frame data
  - arg_4 0 8 14 is frame size
  - Check offset 55h within frame for ‘eBR\xEE’ signature
    - Should be beyond IP and TCP/UDP headers, even with options
    - If present, execute code directly from frame at offset 59h

- For large payloads, send “mini-payloads” to construct code
  - SharedUserData (FFDF0000h) is universal and writable, and visible in user-land at 7FFE0000h

Demonstration

- From a floppy disk
- From a CD-RW
- Via network boot

Look for the blue smiley!
To-Do

- Adapt for more traditional rootkit functionality
- Explore other methods of retaining execution potential besides INT 13h hook-based patching
- Investigate bootable USB storage and other bootable media

Bonus Material!

- A little something extra for those who thought this talk would be entirely boring... (you may still be right)
- Did you know:
  - You can perform raw disk operations without entering the kernel?
  - It’s not an NT kernel vulnerability!
  - It’s...
IOPL Technique

It's a Feature, Not a Vulnerability

IOPL Technique – Overview

- EFlags contains an IOPL (I/O Privilege Level) field
  - CPL <= IOPL (numerically less; greater privileges) can use:
    - IN / INSB / INSW / INSD
    - OUT / OUTSB / OUTSW / OUTSD
    - CLI / STI
  - Only ring 0 can modify IOPL

- Some CSRSS threads run with IOPL=3 (prior to 2003)
  - These threads can be hijacked, or
  - NtSetInformationProcess(ProcessUserModeIOPL)
    - Must have SeTcbPrivilege
  - Either way requires SYSTEM-equivalent privileges
IOPL Technique – Disk Access

- Allows low-level disk access using port I/O
  - Possible on IDE drives with only port I/O
    - No DMA required, no IRQs generated, etc.
    - For sample code, see [Kaze] in References
  - So what?
    - Evade anti-virus boot sector protection?
    - Evade system integrity assurance software?
    - Defeat machine state preservation software?
    - Fun way to install eEye BootRootKit on a hard drive

IOPL Technique – Local Kernel Backdoor?

- Could it allow kernel subversion?
  - DMA has provisions for memory-to-memory transfers
  - PIC can be reprogrammed
    - Could a spurious software interrupt / exception / IRQ get EIP to an address < MmUserProbeAddress?
    - Some fault handlers expect CPU to push error code after EIP
      - Fault: Error, EIP, CS, EFlags, ESP, SS
      - IRQ: EIP, CS, EFlags, ESP, SS
  - Arbitrary disk contents can be modified...
    - ...And we can violently reboot ("MOV AL, 0FEh / OUT 64h, AL")
  - Much harder to monitor than "\Device\PhysicalMemory", ZwSystemDebugControl [randnut], or loading a driver
References


Kaze <Kaze_0mx@yahoo.fr>. “FATdoc#1.txt: Lire le Fat via les Ports.” http://fat.lyua.org/fat/data/fatdoc1.txt


Questions?

Questions? Comments? E-mail me! dsoeder@eeye.com

digital self defense