Exposing Bootkits with BIOS Emulation

Lars Haukli

Sr. Security Researcher
lars.haukli@bluecoat.com
Twitter: @zutle

Black Hat USA 2014

Security Empowers Business

Bootkits

- New security features raise the bar for kernel mode rootkits
  - Driver Signature Enforcement
  - Patch Guard
  - Secure Boot

- Why are techniques from the 1980s still a threat today?
  - Secure Boot is a UEFI feature
  - Legacy BIOS systems boot from unsigned sectors
  - Malware may run code before security features kick in

- Perhaps not a good idea to rely on technology from the 1970s

Roadmap

- Manipulating the BIOS boot sequence
- Overcoming rootkit hooks to read true disk contents
- Emulating the boot code and the BIOS
- Demo – Typical bootkit behaviour
- Heuristic detection based on boot code behavior
- Disabling bootkits
- Challenges with non-standard boot loaders

Bootkits: Common Denominators

- Aims to load an unsigned kernel mode driver
  - Manipulating boot sectors is just a way to achieve this
  - Bypass security features by running code early in the boot process

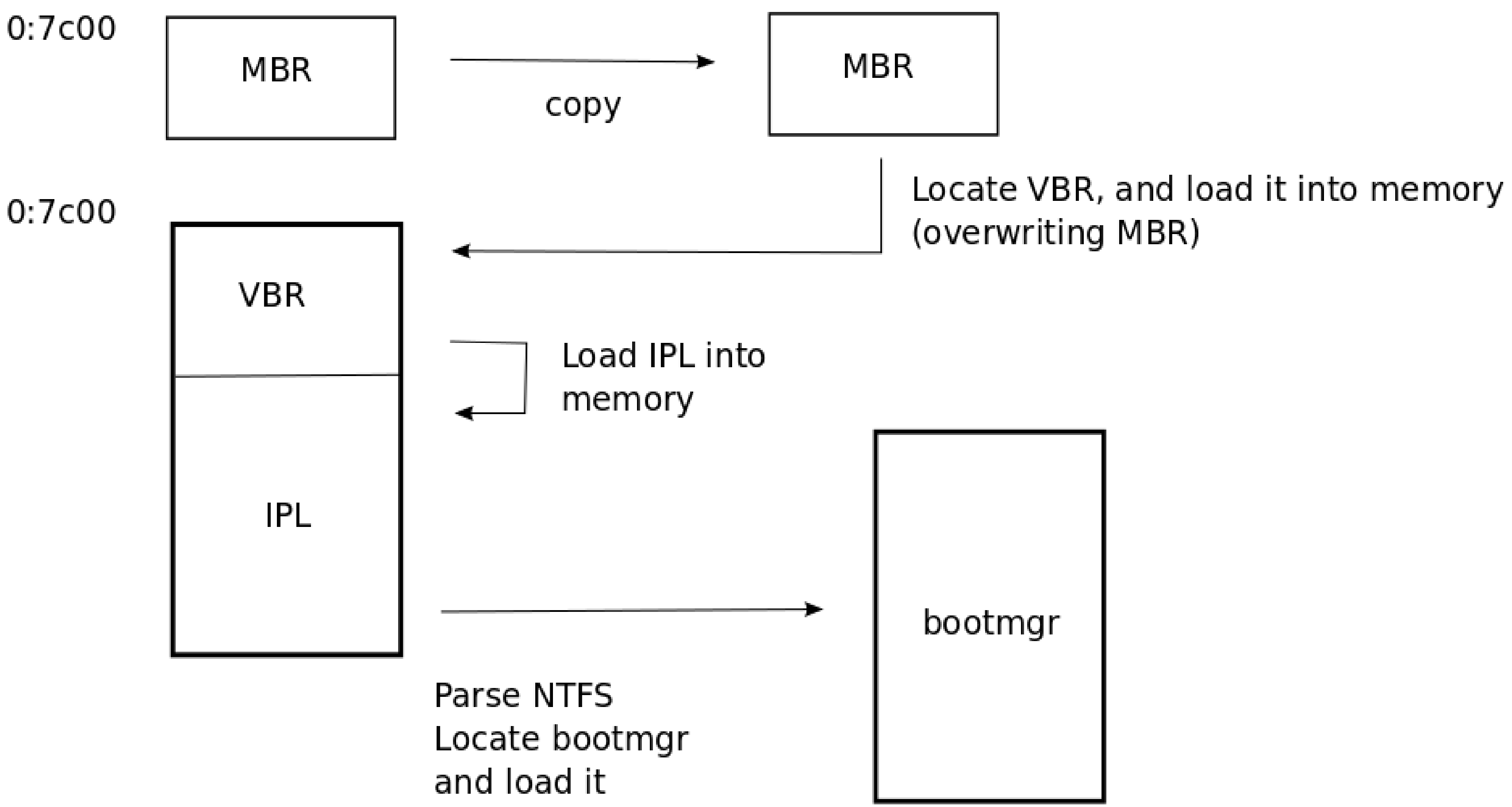
- Attack surface
  - ~17 unsigned sectors on disk (the boot sectors)
    - MBR, VBR, IPL
  - Cannot load driver this early – kernel is not yet loaded

- Load chains may be complex
  - TDL4 – replaces kdcom.dll in memory
  - Rovnix – patches bootmgr in memory
  - Boot sector modifications make this possible

Modifying BIOS Boot Sequence

- BIOS interrupt 19h loads first sector on disk into 0:7C00
  - 16-bit code running in real mode
  - Loads sectors on disk into memory using interrupt 13h
- MBR loads VBR into 0:7C00
  - Overwriting itself
- VBR loads IPL
  - Parses NTFS to locate bootmgr
- Bootkits replace contents
  - Still needs OS to load – resume normal boot after modification

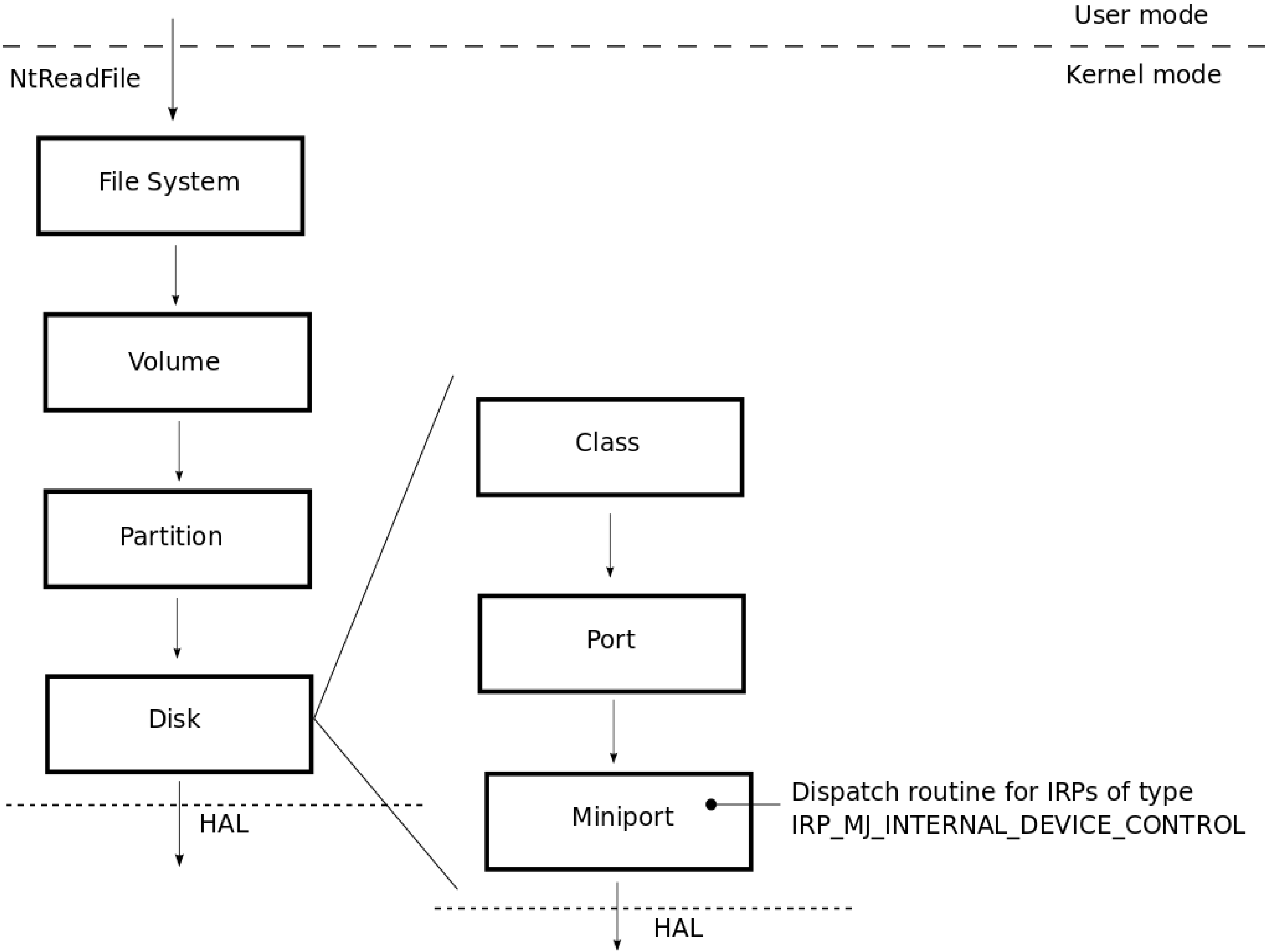
```
mov     ax, 201h
mov     bx, 7C00h
mov     cx, [bp+2]
mov     dx, [bp+0]
int     13h
```



0:7c00
MBR
copy
MBR
Locate VBR, and load it into memory
(overwriting MBR)
0:7c00
VBR
Load IPL into
memory
IPL
bootmgr
Parse NTFS
Locate bootmgr
and load it

Overcoming hooks

Using anti-rootkit techniques to read true disk content


User mode
NtReadFile
Kernel mode
File System
Volume
Partition
Disk
HAL
Class
Port
Miniport
Dispatch routine for IRPs of type
IRP_MJ_INTERNAL_DEVICE_CONTROL
HAL

Using Miniport’s Dispatch Routine

- Miniport’s DriverEntry sets up its Driver Object
  - MajorFunction array holds dispatch routines
- Obtain minport’s Driver Object to extract function pointer to a routine that implements reading and writing to raw sectors
  - No need to worry about hooks at higher levels
  - No need to implement hardware-specific logic
- See whitepaper for an alternative approach using PIO
  - Communicate directly with disk controller

The Challenge of Hooks

- This is a powerful routine
  - Great place for rootkits to install hooks

- Rootkits may manipulate Driver Object in memory
  - Install function pointer hook by replacing dispatch routine in MajorFunction array
  - Install inline hook by modifying the contents of the routine in memory

- We need to obtain the original function pointer

Overcoming Function Pointer Hooks

- Cannot trust memory contents
  - Need to find a trustworthy source of information

- Signed executable on disk cannot be modified

- Analyze miniport driver on disk
  - Retrieve RVA from disk image
  - Retrieve base address of loaded image

DriverEntry Initializes Dispatch Routines

```
NTSTATUS DriverEntry(__in DRIVER_OBJECT *pDriverObject, __in UNICODE_STRING *pRegistryPath)
{
    // ...

    // Set dispatch routines
    pDriverObject->MajorFunction[IRP_MJ_CREATE]                   = Dispatch_Dummy;
    pDriverObject->MajorFunction[IRP_MJ_CLOSE]                    = Dispatch_Dummy;
    pDriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL]           = Dispatch_DeviceControl;
    pDriverObject->MajorFunction[IRP_MJ_INTERNAL_DEVICE_CONTROL]  = Dispatch_InternalDeviceControl;
    pDriverObject->MajorFunction[IRP_MJ_PNP]                      = Dispatch_PnP;
    pDriverObject->MajorFunction[IRP_MJ_SYSTEM_CONTROL]           = Dispatch_SystemControl;
    pDriverObject->MajorFunction[IRP_MJ_POWER]                    = Dispatch_Power;

    // ...

    return STATUS_SUCCESS;
}
```

Obtaining RVA of Dispatch Routine

- Find the instructions that initialize the MajorFunction array
  - Retrieve the RVA of the dispatch routine responsible for handling IRPs of type IRP_MJ_INTERNAL_DEVICE_CONTROL
- Recursively disassemble driver on disk
  - Recursive approach to include subroutines (local functions)
  - Look for instructions that modify memory
  - There are some common logic that should always be present

Disassembly of DriverEntry

```
lea     rax, DriverUnload
mov     [rsi+68h], rax
lea     rax, Dispatch_InternalDeviceControl
xor     ecx, ecx
mov     [rsi+0E8h], rax ; Set IRP_MJ_INTERNAL_DEVICE_CONTROL
lea     rax, Dispatch_Dummy
mov     r8d, 'PedI'
mov     [rsi+70h], rax  ; Set IRP_MJ_CREATE
mov     [rsi+80h], rax  ; Set IRP_MJ_WRITE
lea     rax, Dispatch_DeviceControl
mov     [rsi+0E0h], rax ; Set IRP_MJ_DEVICE_CONTROL
lea     rax, Dispatch_Power
mov     [rsi+120h], rax ; Set IRP_MJ_POWER
lea     rax, Dispatch_PnP
mov     [rsi+148h], rax ; Set IRP_MJ_PNP
lea     rax, Dispatch_SystemControl
mov     [rsi+128h], rax ; Set IRP_MJ_SYSTEM_CONTROL
```

Searching for the Dispatch Routine

- Analyze entire routines, looking for:
  - mov [reg + offset], routine
- Keep register values
  - lea rax, routine
  - mov [rsi + E8h] , rax
- Critical observation – Some routines are always present
  - Power, PnP, DeviceControl, InternalDeviceControl, DriverUnload
  - All have fixed offsets within driver object
- Search for all offsets within a single routine
  - Extract RVA of InternalDeviceControl routine if all 5 are found

Overcoming inline hooks

- Knowing the expected contents of a routine enables us to detect and bypass inline hooks
  - Compare disk contents with memory contents

- Construct trampoline consisting of original instructions + branch
  - Execute original instructions, then pass control to the rest of the routine
  - Use disassembly to ensure we are stealing whole instructions
  - Pass control to the next whole instruction following the patch

Interfacing with the Dispatch Routine

- Imitate the next higher driver
- Create an IRP
  - Miniport will pass it back up when request has completed
  - Set an IoCompletion routine that will simply destroy the IRP
- Data on request goes into I/O Stack Location
  - Command Descriptor Block
  - SCSI commands
    - READ (10), READ (16)
  - Boils down to specifying sector numbers (LBA)

- Whitepaper has more details on this

Emulating the boot sequence

In order to emulate the boot code we also need to emulate the BIOS

Emulating the Boot Sequence

- Custom BIOS written in 16-bit assembly
  - Implements the functionality we expect boot loaders make use of

- Emulator provides a separate memory space
  - Only accessible the emulated code and the emulator itself

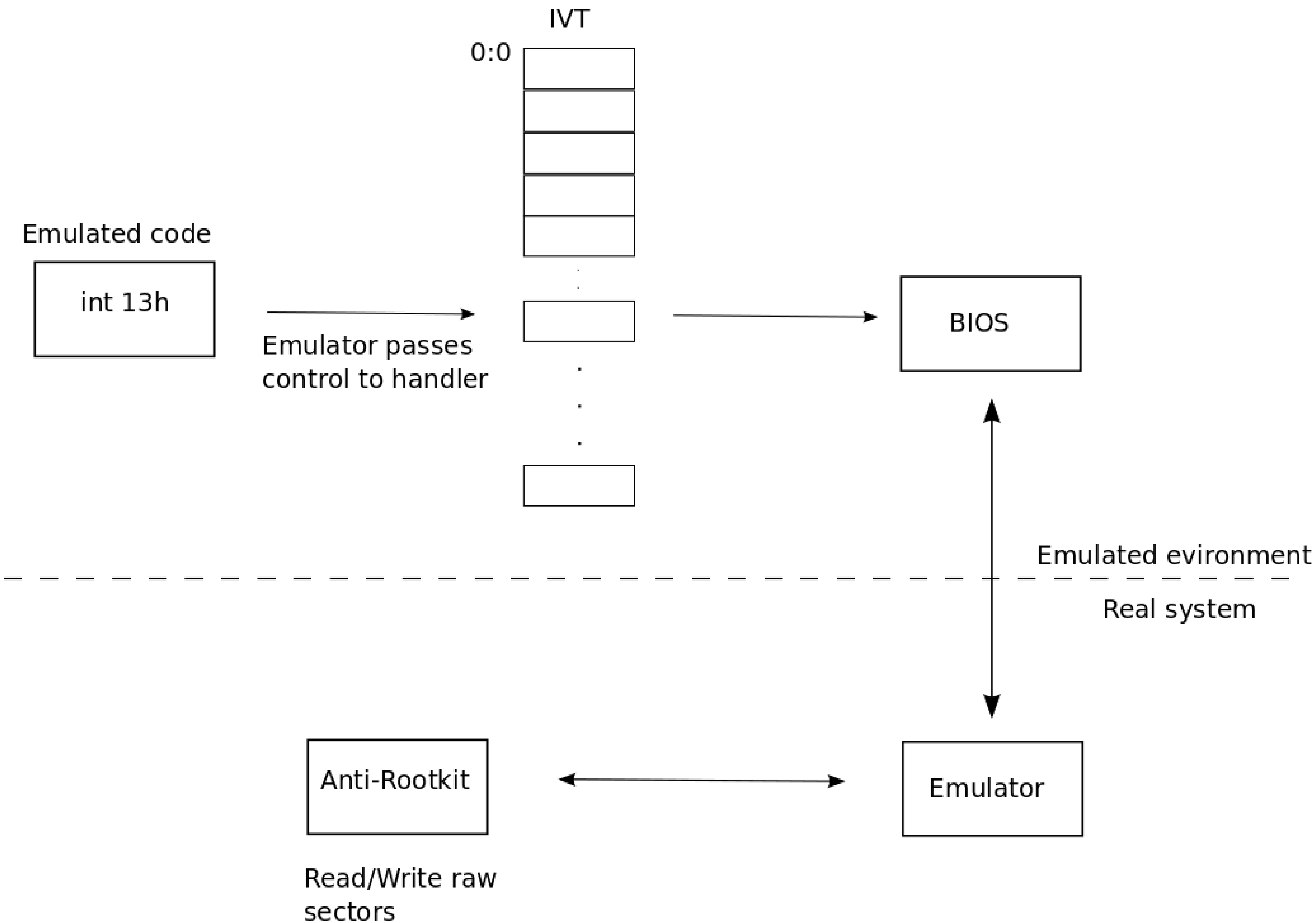
- Load MBR into emulator memory at 0:7C00

- Load custom BIOS into emulator memory at F000:FC00

- Emulation starts at BIOS entry point
  - We will emulate the initialization code
  - Once complete, transfer control to first instruction of MBR

BIOS Implementation

- Set up Interrupt Vector Table (IVT)
  - Located at 0:0
- Register interrupt vectors for:
  - interrupt 10h - Video
  - interrupt 13h – Disk I/O
  - interrupt 16h – Keyboard
  - Dummy routines for the rest
- When we emulate an interrupt, our BIOS will handle it
  - Break out of emulation loop for interrupt 13h, as we need to incorporate anti-rootkit techniques for disk I/O
  - Emulation resumes when contents has been written to memory


IVT
0:0
Emulated code
int 13h
Emulator passes control to handler
BIOS
Emulated evironment
Real system
Anti-Rootkit
Emulator
Read/Write raw sectors

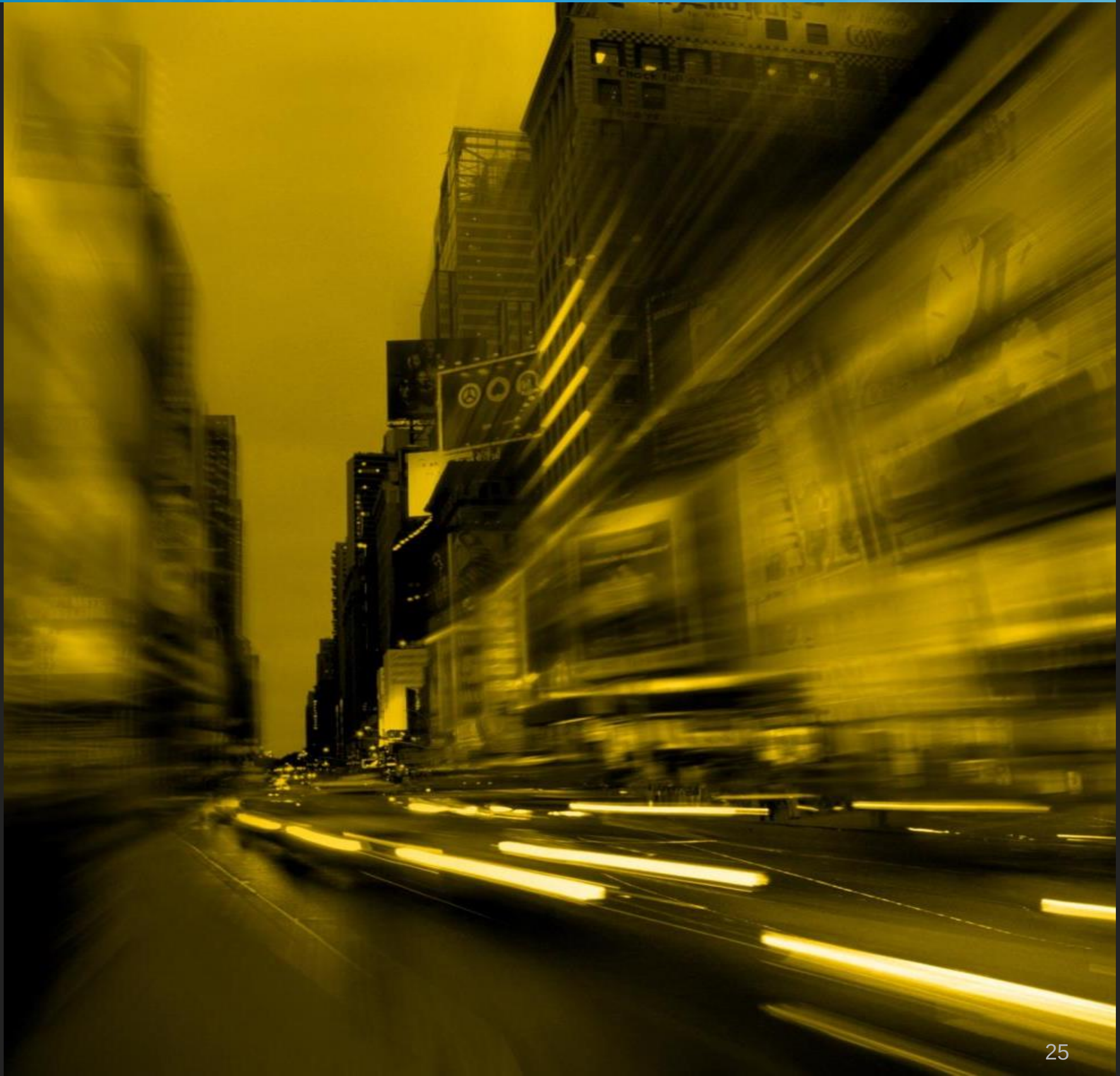
Demo

- Typical behavior of MBR boot process when compromised
- Debugger UI on top of our emulator ftw

Detecting Anomalies

Emulating the boot code reveals anomalies in its behavior

No baseline required



Interrupt 13h Hooks

- Boot code seeks to patch modules not yet loaded
  - Hooking interrupt 13h enables intercepting all disk i/o
  - Enables patching memory contents on-the-fly

- Needs to regain control later in boot process
  - Cannot load its kernel mode driver before kernel itself has loaded
  - Modify memory in some way to achieve this
  - May wait for a certain byte pattern or use other indicators

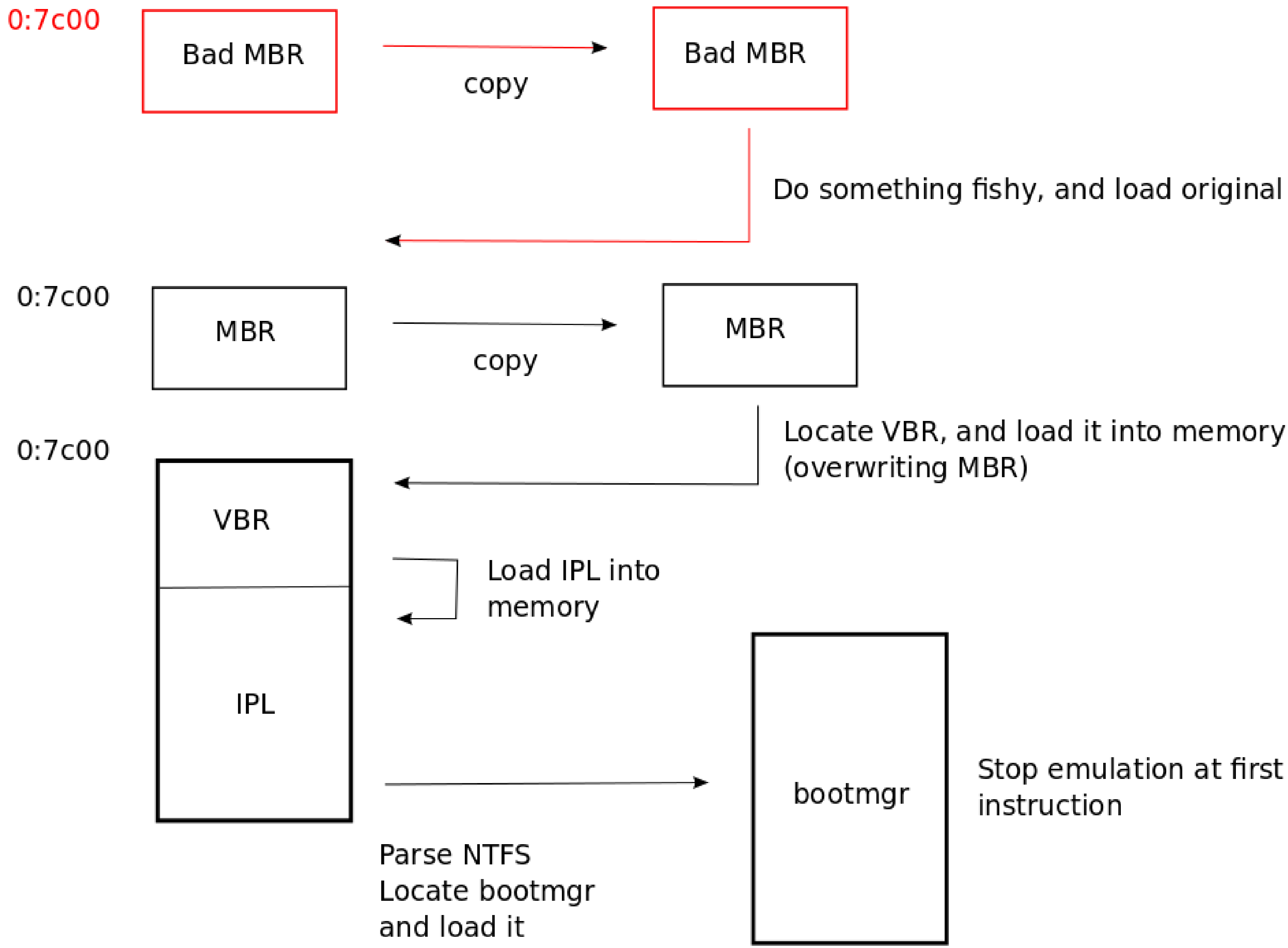
- Emulated code will interact with our custom BIOS
  - Will modify our interrupt 13h handler in our IVT
  - Check if it is still intact once emulation completes

Patching bootmgr

- bootmgr is signed for a reason
- When emulation reaches the point where control is passed to it, its entire contents resides in memory
- bootmgr is a special executable
  - disk image = memory image
- Comparing contents on disk with memory reveals anomalies
  - Normally, bootmgr will be patched using an interrupt 13h hook

MBR Replacement Anomaly

- Bootkits need the OS to boot
  - Make changes, then let normal boot sequence continue
- Retrieve original MBR, and load it back to 0:7C00h
  - This is where the original MBR expects to be loaded
- This results in an anomaly in the behavior of the boot code


0:7c00
Bad MBR
copy
Bad MBR
Do something fishy, and load original
0:7c00
MBR
copy
MBR
Locate VBR, and load it into memory
(overwriting MBR)
0:7c00
VBR
IPL
Load IPL into
memory
bootmgr
Stop emulation at first
instruction
Parse NTFS
Locate bootmgr
and load it

Breaking load chains

Retrieving Original Boot Sector Contents

- Key is to determine what has been changed
  - Count number of times 0:7C00 is executed
  - MBR case – Stop emulation at second execution of 0:7C00
  - VBR/IPL case – Let emulation complete
- Retrieve original contents from emulator memory
  - Encrypted on disk? No problem!
- Replace modified parts with original
  - Breaks the load chain
  - Reboot system to finish it off

Challenges

Non-standard boot loaders complicate detection

Challenges

- Non-standard boot loaders that load multiple OSes
  - e.g. GRUB requires user input
- Full disk encryption solutions
  - May require user to enter a password during boot
  - Also, often hook interrupt 13h in order to decrypt disk contents
- Hard or impossible for our BIOS to make decisions
- Detect whenever the boot loader ask for user input
  - Boot code will poll for keyboard input using interrupt 16h
  - Abort emulation and report that we cannot decide if it is good or bad

Conclusion

- Anomalies in boot sectors are detectable by emulation
  - Must incorporate anti-rootkit techniques when reading disk
  - Counters obfuscation and encryption
  - Challenges with non-standard boot loaders
- Break rootkit's load chain to defeat it
  - Emulation approach effective at retrieving original contents
- UEFI systems are more secure than BIOS systems
  - Booting from signed firmware is more secure than relying on technology from the 1970s

Thank you for your attention!

- Special thanks to my co-researcher Leif Arne Søderholm
- High five to the rest of the R&D team
- Also big thanks to the guys at kernelmode.info
  - Great source for rootkit samples!

BLUE COAT

Security Empowers Business

Questions?

Lars Haukli

Sr. Security Researcher

lars.haukli@bluecoat.com

Twitter: @zutle

BLUE COAT®

Security
Empowers
Business