Analyzing the Windows kernel *shadow stack* mitigation

Rémi Jullian and Alexandre Aulnette
remi.jullian@synacktiv.com
alexandre.aulnette@synacktiv.com

Synacktiv

Abstract. Intel and Microsoft worked together, with other players from the industry, to implement a mechanism named Intel CET, introducing a new mitigation, the shadow stack. Effective both in user mode and kernel mode, this mitigation has been designed to defeat exploit relying on control-flow hijacking, by overriding return addresses on the stack. In this paper, we will discuss the role of this mitigation. We will also deep dive into the implementation of this mitigation in the Windows kernel. We will explain how the Windows operating system leverage on virtualization technics to protect the shadow stack integrity, and to ensure this mitigation cannot be disabled on a live system, even if an attacker possess strong primitives such as a read/write in the kernel.

1 Introduction

1.1 A bit of history

When trying to exploit memory corruption vulnerabilities, attackers' final goal is to achieve code execution, generally by allocating and writing to an executable page, in order to execute an arbitrary payload. *Data Execution Prevention* (DEP) was introduced in the Windows operating system as a security feature to mitigate exploits that involve executing code from non-executable memory regions. On the Windows operating system, up to Windows 8, all pages of memory allocated by the kernel in the *non-paged pool* area were executable. With Windows 8 (64-bit), Microsoft introduced a new pool type, non-executable, the *NonPagedPoolNx* (0x200) [10]. For compatibility reasons, Windows still allows drivers to allocate executable memory from the pool *NonPagedPool.* However by implementing mitigation based on virtualization technology such as *Hypervisor-Based Code Integrity* (HVCI), a secure kernel is used to ensure that all pages of kernel executable code are signed, and that these pages, once marked as an executable, cannot be writable again (W$\oplus$X). In other words, attackers can't just allocate a new executable page of code, write a shellcode and execute an arbitrary payload. In order to bypass this mitigation, a common method is to rely on *Return-Oriented Programming* (ROP), *Jump-Oriented Programming* (JOP) or *Call-Oriented Programming* (COP) mechanisms. To mitigate these technics, Intel developped a new set of hardware based mitigations named Intel *Control-flow Enforcement Technology* (CET). These mitigations are used to prevent forward-edge (indirect call/jump) and backward-edge (ret) control-flow transfer. The shadow stack is a back-edge oriented mitigation, part of Intel CET, designed to defeat ROP. In this paper, we will focus on the implementation of the shadow stack in the Windows kernel. We will discuss how it prevents executing ROP attacks and how it relies on virtualization to protect against an attacker with read/write primitive against the kernel. Finally, we will discuss the limitations of this mitigation.

1.2 State of the art

In 2018, Microsoft stated they were planning to use Intel CET for backward edge protection [23]. In 2019, B. Sun et al described in depth the implementation of Intel CET for Windows 10 x64 (RS5), to support user mode shadow stack [1]. The internals of user mode shadow stack capabilities have also been covered by Y. Shafir and A. Inoescu in [24]. In 2020, both Intel [7] and Microsoft [21] released blogposts describing this mitigation from a high-level perspective. In 2023, Y. Shafir mentioned that the kernel mode shadow stack was relying on virtualization mechanisms [22]. Finally, in 2025, C. McGarr released the first paper describing in details the kernel mode shadow stack implementation on Windows [13]. In this paper, we will focus on the kernel mode shadow stack, which has been less reviewed than the user mode shadow stack.

At the time of writing, the kernel shadow stack mitigation is not enabled by default on Windows 11 (24H2), with a computer meeting appropriate requirements. This is probably for compatibility reasons. Indeed, Microsoft states that some (third-party) drivers are currently not compatible with this mitigation because they use return-address hijacking to perform code obfuscation [19]. Also, Intel CET *Indirect Branch Tracking* (IBT) mitigation is not yet supported in the Windows kernel. Briefly, IBT is a CPU feature which tracks indirect `jmp` and `call` instructions and generates a fault if the destination instruction is not a `ENDBRANCH`[1] instruction [1]. At the time of writing, *ntoskrnl.exe* (Windows 11, 24H2) does not contain

[1] `ENDBR64` / `ENDBR32`

any `ENDBR64` instruction. We have no information regarding Microsoft plan to support this mitigation in the future.

1.3 Environment

In order to analyze the implementation of the shadow stack in the Windows kernel, we used both static and dynamic analysis. For dynamic analysis, we used a kernel debug environment, on a physical computer running an *Intel Core i3-N305* processor (supporting Intel CET and the virtualization requirements). Reverse engineering was performed on a Windows 11 24H2 operating System. Some proof-of-concepts have also been implemented and are accessible on Synacktiv's github: `https://github.com/synacktiv/windows_kernel_shadow_stack`. As Microsoft provides public *PDB* files[2] for the Windows kernel (*ntoskrnl.exe*), most symbols referenced in the paper can be retrieved for further analysis. Unless explicitly specified, the symbols mentioned in this paper are always prefixed by `nt!`. In order to enable the kernel shadow stack, the following commands can be used to modify the Windows registry:

Listing 1: Kernel shadow stack activation

```
reg add HKLM\SYSTEM\CurrentControlSet\Control\DeviceGuard\\
  Scenarios\HypervisorEnforcedCodeIntegrity /v Enabled /t
  ↪  REG_DWORD /d 1 /f
reg add HKLM\SYSTEM\CurrentControlSet\Control\DeviceGuard\\
  Scenarios\KernelShadowStacks /v Enabled /t REG_DWORD /d 1 /f
```

To enable the audit mode, the following command can be used:

Listing 2: Kernel shadow stack audit mode activation

```
reg add HKLM\SYSTEM\CurrentControlSet\Control\DeviceGuard\\
  Scenarios\KernelShadowStacks /v AuditModeEnabled /t REG_DWORD /d
  ↪  1 /f
```

The differences between regular and audit mode are covered later in the paper.

2 The shadow stack mitigation

With the shadow stack mitigation, when a `call` instruction is executed, the return address is pushed on the stack, but also on a separated stack,

[2] PDB files are debugging files containing symbols such as function or global variables names

called the shadow stack. The shadow stack pointer is stored in a register called *SSP*. The operating system is responsible for allocating the memory-area behind the shadow stack, and for configuring the CPU properly to enable this mitigation. When a `ret` instruction is executed by the CPU, the return address is popped from the stack, and before executing the flow transfer to the return address, the CPU checks if it matches with the one at the top of the shadow stack. If not, it will generate a *control protection exception* (#CP) and let the operating system decide how the exception should be handled. The Intel manual [8] provides the following pseudo-code related to the shadow stack, for a `ret` (near return) instruction:

Listing 3: Return address verification on a ret instruction

```
...
    RIP := Pop();
    IF ShadowStackEnabled(CPL)
        tempSsEIP = ShadowStackPop8B();
        IF RIP != tempSsEIP
            THEN #CP(NEAR_RET); FI;
    FI;
...
```

This mitigation can work in both user mode, where the *current privilege level* (CPL) is 3, and in kernel mode where the *current privilege level* is 0. The register state used by Intel CET comprises two 64-bit MSRs: [8]:

Architectural MSR Name	Register Address	Description
IA32_U_CET	0x6a0	Configure User Mode CET (R/W)
IA32_S_CET	0x6a2	Configure Supervisor Mode CET (R/W)

Table 1. Intel CET state related MSR

Also, MSR registers listed in table 2 are used to store the address of the shadow stack into the SSP when performing privilege level transition [8]. For instance, when performing a transition from user mode (CPL3) to kernel mode (CPL0), the CPU saves the SSP into the *IA32_PL3_SSP* MSR, and load the new SSP from *IA32_PL0_SSP*. As the Windows operating System only uses 2 privilege level (CPL0 and CPL3), the MSR *IA32_PL1_SSP* and *IA32_PL2_SSP* are currently not used. The Intel manual [8] also documents CPU instructions used to manipulate the shadow stack. Some instructions will be briefly described as they will be

Architectural MSR Name	Register Address	Description
IA32_PL0_SSP	0x6a4	Linear address to be loaded into SSP on transition to privilege level 0 (R/W)
IA32_PL1_SSP	0x6a5	Linear address to be loaded into SSP on transition to privilege level 1 (R/W)
IA32_PL2_SSP	0x6a6	Linear address to be loaded into SSP on transition to privilege level 2 (R/W)
IA32_PL3_SSP	0x6a7	Linear address to be loaded into SSP on transition to privilege level 3 (R/W)

Table 2. Intel CET privilege level transition related MSR

referenced later in this paper, in order to describe the implementation of the shadow stack mitigation in the Windows kernel.

The instruction `rdsspq` can be used to read the value of the stack pointer. The destination register can then be dereferenced for reading a return address which was at the top of the shadow stack:

```
; Copies the current shadow stack pointer (SSP) register to the
; register destination
rdsspq rdx
; Read the value at the top of the shadow stack
mov rax, qword ptr [rdx]
```

The instruction `wrssq` can be used to write to the shadow stack. The destination register must contain an address pointing to the shadow stack. The `wrssq` instruction is one of the few instructions which are allowed to perform a write access to the shadow stack. Also, this instruction requires that the bit `WR_SHSTK_EN` (Bit 1) is set within the MSR `IA32_U_CET` (in user mode) or `IA32_S_CET` (kernel mode):

```
; Write the content of the rcx register to the shadow stack
; reference by rax
wrssq qword ptr [rax], rcx
```

3 Virtualization for kernel shadow stack protection

On Windows, VBS (*Virtualisation-Based Security*) is a set of security features which relies on virtualization techniques. This allows the operating system to run 2 different kernels, with isolation between each other implemented thanks to a mechanism named VTL (*Virtual Trust Level*):

— the "classical" Windows kernel (*ntoskrnl.exe*)

— the secure kernel (*securekernel.exe*)

The secure kernel is more privileged, and runs at VTL 1, while the regular kernel, less privileged, runs at VTL 0 [5]. On Windows, VTL isolation is available thanks to Hyper-V, the Windows Hypervisor. Hyper-V is a type 1 (bare-metal) hypervisor, and thus lies between the hardware layer and the operating system [3]. As mentioned earlier, HVCI allows to ensure that all pages of kernel executable code are signed, and that these pages, once marked as executable, cannot be writable again (W⊕X). HVCI is also used to protect read-only areas such as the Kernel CFG (*Control Flow Guard*) bitmap, or kernel shadow stack's pages. HVCI relies on VBS. Another mechanism named KDP (*Kernel-Data-Protection*), allows to prevent data corruption attacks by protecting parts of the Windows kernel [20]. Both HVCI and KDP rely on SLAT (*Second Layer Address Translation*). SLAT is a hardware mechanism allowing to add an additional layer when performing address translation, that is the process of converting a virtual address to a physical address.

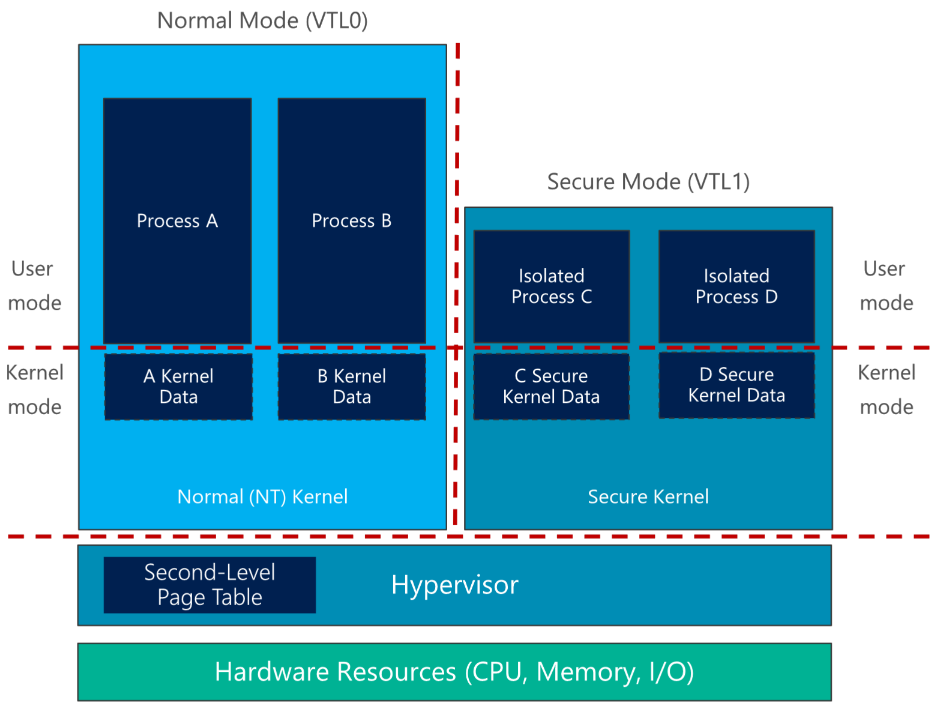


Fig. 1. Virtualisation-Based Security architecture (from [11])

Using SLAT, the hypervisor will be able to intercept a memory access made by the regular kernel, which is actually a GPA (*Guest Physical Address*) access, and to convert it to a SPA (*System Physical Address*) access [12]. Intel implements this conversion using another set of paging structure called EPT (*Extended Page Tables*).

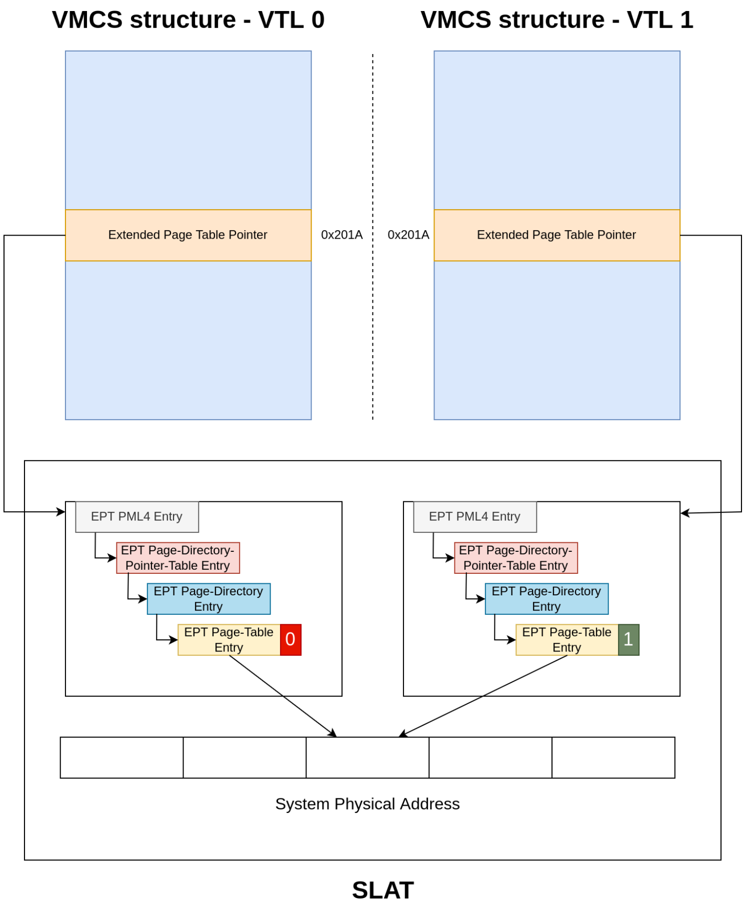


Fig. 2. SLAT implementation

The *Virtual Machine Control Structure* (VMCS) is a hardware-defined data structure used by Intel VT-x to manage the execution of a virtual machine. Each VTL in Hyper-V has its own VMCS, with separate EPTP (*Extended Page Table Pointer*) for memory isolation. As illustrated on figure 2, the EPTP can be retrieved from the VMCS structure, using a `vmread` instruction with 0x201a specified in the register operand. To configure EPT, the secure kernel uses hypercalls to create EPTE (*Extended Page-Table Entries*) for the classical kernel [12]. A hypercall is a calling mechanism used for guest to interface with the hypervisor [11]. In other words, the classical kernel has no way to interfere with the EPTE. The EPTP allows walking through the EPTs, and to implement EPTE. By implementing EPTE, managed *only* by the hypervisor, which are invisible to the guest (i.e regular kernel), it is possible to implement another level of trust. This allows, for instance, to mark a page as read-only in VTL 0, but writable in VTL 1. Like a regular PTE, the size of an EPTE is 64-bits, and some bits are used to specify access rights on a physical page. The format of an EPTE is described in the Intel developer manual [8]. The following bits are particularly useful for implementing HVCI and KDP:

— Bit 0: Read access; indicates whether reads are allowed from the 4-KByte page referenced by this entry
— Bit 1: Write access; indicates whether writes are allowed from the 4-KByte page referenced by this entry
— Bit 2: Execute access; indicates whether instruction fetches are allowed from the 4-KByte page controlled by this entry

PTEs associated with the shadow stack set their write access bit to 0, thus ensuring the shadow stack cannot be modified. In order to enforce that, the secure kernel will issue, through an hypercall, a request to the hypervisor, in order to set the page as read only in the EPTE for VTL 0. This is described with more details in section 4.7. The hypervisor will be able to catch and deny any write access tentative, by identifying that the write access bit is not set in the EPTE. Using EPT to prevent writes to the supervisor shadow-stack pages of a VM is detailed in section 9.5 (Supervisor Shadow-Stack Control) from Intel CET specifications [6].

4 Implementation in the Windows kernel

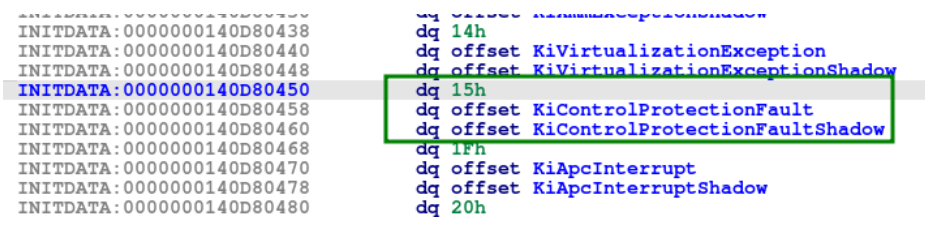
In order to enable the shadow stack mitigation, the operating system needs to perform several tasks such as:

— Initializing the exception handler, triggered by the CPU when a control protection exception (#CP) occurs
— Allocating a per-thread shadow stack and protecting it against a write operation implemented in software
— Enabling the mitigation at the CPU level

All these steps are explained in the following sections.

4.1 Exception handler initialization

During the boot process, the function `KiInitializeIdt` is responsible for setting up the *Interrupt Dispatch Table* (IDT), which manages how the processor handles interrupts and exceptions. Within the `INITDATA` section of *ntoskrnl*, the symbol `KiInterruptInitTable` defines the initial configuration of the interrupt handlers used to populate the IDT. Specifically, the entry at vector 0x15 in this table points to the `KiControlProtectionFault` handler:



```
INITDATA:0000000140D80438          dq 14h
INITDATA:0000000140D80440          dq offset KiVirtualizationException
INITDATA:0000000140D80448          dq offset KiVirtualizationExceptionShadow
INITDATA:0000000140D80450          dq 15h
INITDATA:0000000140D80458          dq offset KiControlProtectionFault
INITDATA:0000000140D80460          dq offset KiControlProtectionFaultShadow
INITDATA:0000000140D80468          dq 1Fh
INITDATA:0000000140D80470          dq offset KiApcInterrupt
INITDATA:0000000140D80478          dq offset KiApcInterruptShadow
INITDATA:0000000140D80480          dq 20h
```

Fig. 3. `KiControlProtectionFault` handler

This handler is invoked when the processor raises a Control Protection Exception (#CP), which is triggered by violations related to Intel CET, such as shadow stack corruption or indirect branch tracking violations. The vector entry index 0x15 is actually defined in the Intel manual [8] (Volume 1, Table 6-1. Exceptions and Interrupts).

4.2 Kernel shadow stack activation

In this section, we will describe how the Windows kernel enables the shadow stack mitigation.

Kernel initialization: During the kernel initialization, the function `KiInitializeKernel` calls `KiSetControlEnforcement`, to enable this mitigation if it is supported by the CPU. More precisely, a `cpuid` instruction (with `rax` set to 0x07 and `rcx` set to 0x0) is used to query the CPU structured Extended Feature Flag. If the shadow stack is supported, `cpuid` sets the bit labeled `CET_SS` [8], indicating it supports the shadow stack mitigation, as an output bit within the `rcx` register. This bit is test and used to set a global variable named `KiCetCapable`, if the field `CpuVendor` of the `KPRCB` structure (Kernel Processor Control Block) equals 1 (`WheaCpuVendorIntel`) or 2 (`WheaCpuVendorAmd`). Then, the bit

23 of `CR4` register is set. This bit is labeled as `CR4.CET` [8], and therefore enable Control-flow Enforcement Technology. Finally, the global variable `KiUserCetAllowed` is set to 1, meaning that user mode applications, running at CPL3, can benefit from the shadow stack mitigation.

Kernel shadow stack initialization: Since the shadow stack mitigation can also be activated when the CPU executes in kernel mode, the function `KiInitializeKernelShadowStacks` is called at boot time, by `KiSystemStartup`. This function is responsible for setting global variables related to Intel CET activation in kernel mode. It takes as a parameter a pointer to a structure of type `_LOADER_PARAMETER_BLOCK`. This structure is initialized by Windows Boot Loader (winload.exe), during the boot process, and passed to ntoskrnl later on [4]. This structure contains a field named `Extension`, which is a pointer to a structure of type `_LOADER_PARAMETER_EXTENSION`. The function `KiInitializeKernelShadowStacks` tests the value of the bit `CR4.CET` as well as 2 bits from an anonymous bitfield located in this structure:

Listing 4: LOADER_PARAMETER_EXTENSION bitfield

```
/* Bit 14 */
unsigned __int32 KernelCetEnabled : 1;
/* Bit 18 */
unsigned __int32 KernelCetAuditModeEnabled : 1;
```

These bits are configured regarding the registry keys mentioned earlier in listing 1 and listing 2 and are used to set 2 global variables:

— `KiKernelCetEnabled`: The shadow stack mitigation is globally enabled in kernel mode, a #CP fault generated in kernel is likely to cause a *BSOD*.

— `KiKernelCetAuditModeEnabled`: The shadow stack mitigation is set in audit mode, the fault handler may generate an *Event Tracing for Windows* (ETW) log, and try to fixup the shadow stack to avoid causing a *BSOD*.

The value of these 2 global variables can be queried from userland, using `NtQuerySystemInformation`, with an undocumented value, as shown in appendix A. To protect these variables from an arbitrary write in kernel, they are a stored in a section named `CFGRO`, which is read only (in the PTE) but also enforced as read only in the EPTE, thanks to the secure kernel and `SLAT`. The value of the variables impacts how the control protection exception handler will handle the fault (in kernel mode). This will be described later in section 5 (Fault Handling).

When `KiInitializeKernelShadowStacks` returns to `KiSystemStartup`, the MSR *IA32_S_CET* is written to either 1 (*SH_STK_EN*) or 3 (*SH_STK_EN* | *WR_SHSTK_EN*) if `KernelCetAuditModeEnabled` is set:

Listing 5: Writing MSR *IA32_S_CET*

```
  mov     eax, 1
  test    cs:KiKernelCetAuditModeEnabled, 1
  jz      short loc_140A8818E
  or      eax, 2
loc_140A8818E:
  xor     edx, edx
  mov     ecx, 0x6A2 ; Write MSR IA32_S_CET
  wrmsr
```

This allows to enable the shadow stack for the kernel, and also allows write access to the shadow stack if the audit mode is enabled. Indeed, the Intel manual [8] describes the MSR *IA32_S_CET* as following:

— Bit 0, *SH_STK_EN*: When set to 1, enable shadow stacks at CPL0

— Bit 1, *WR_SHSTK_EN*: When set to 1, enables the WRSSD/WRSSQ instructions.

Please note that if a kernel debugger is attached during the boot of the Windows kernel (`KdDebuggerEnabled && !KdDebuggerNotPresent`), a call to `KdInitSystem` will force the bit *WR_SHSTK_EN* to 1 in the MSR *IA32_S_CET*.

4.3 Shadow stack allocation

The Windows kernel uses a structure of type `_KTHREAD` for each created thread. The function `KeInitThread` is responsible for initializing newly created threads. If `KiKernelCetEnabled` is set, the bit 22, labeled as `CetKernelShadowStack`, is set in the bitfield `_KTHREAD.MiscFlags`. Then `KiCreateKernelShadowStack` is called in order to allocate one page used to implement the kernel shadow stack. Using `MiReservePtes`, 3 PTEs are reserved, however, only one page will be allocated (committed) to implement the shadow stack . This allows to reserve a virtual address space of 0x3000 bytes, composed of a first guard page, a page dedicated to the shadow stack and another guard page.

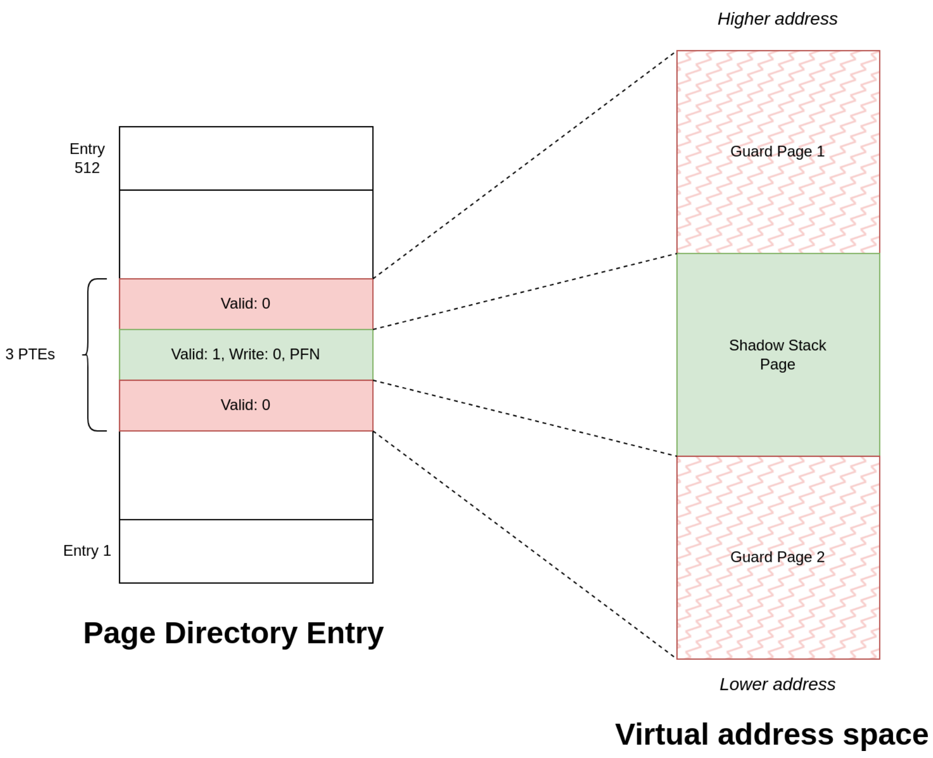


Fig. 4. Shadow stack virtual address space

Using WinDbg, dumping 3 PTEs returned by `MiAllocateKernelStackPages` gives an output like:

```
kd> dq ffffd0d2c5703eb8 L3
ffffd0d2c5703eb8  0000000000000000 8a000000041ff161
ffffd0d2c5703ec8  0000000000000000
```

The 2 PTEs used to implement the guard pages are set to 0, meaning no physical page is used. Any access to these virtual addresses would lead to a page fault. The PTE associated with the shadow stack page is backed by a physical page (`PageFrameNumber > 0`), and is marked as read-only (`Write = 0`):

```
kd> dt nt!_MMPTE_HARDWARE ffffd0d2c5703eb8+8
+0x000 Valid             : 0y1
+0x000 Dirty1            : 0y0
+0x000 Owner             : 0y0
+0x000 WriteThrough      : 0y0
+0x000 CacheDisable      : 0y0
+0x000 Accessed          : 0y1
+0x000 Dirty             : 0y1
+0x000 LargePage         : 0y0
+0x000 Global            : 0y1
+0x000 CopyOnWrite       : 0y0
+0x000 Unused            : 0y0
+0x000 Write             : 0y0
+0x000 PageFrameNumber   :
↪  0y000000000000000000000000100000111111111 (0x41ff)
+0x000 ReservedForSoftware : 0y0000
+0x000 WsleAge           : 0y1010
+0x000 WsleProtection    : 0y000
+0x000 NoExecute         : 0y1
```

The virtual address covering the range of 0x3000 bytes is then computed using the virtual address of the first PTE such as:

Listing 6: Virtual address from PTE computation

```
ResolvedVirtualAddress = (
  (PteAddress - SYSTEM_PTE_BASE_ADDRESS) << 9) |
  ↪  0xFFFF000000000000
)
```

The symbol `SYSTEM_PTE_BASE_ADDRESS` refers to the 512GB four-level page table map. Reading the assembly, this is hardcoded to the virtual address *0xFFFFF68000000000*, but its patched at runtime because of kASLR.

4.4 Secure kernel transition

After allocating the shadow stack, the NT kernel needs to request the secure kernel to perform 2 additional tasks:

— Initializing the shadow stack
— Protecting the integrity of the shadow stack

These 2 operations are discussed later in section 4.5 and 4.7, but can be realized thanks to a mechanism named *Secure Call*. A *Secure Call* allows the transition from the kernel to the secure kernel. With `VslAllocateKernelShadowStack`, the *Secure System Call Num-*

ber (SSCN), is set to the value 0xE6.[3] To perform the secure call, `VslpEnterIumSecureMode` is called and ends up in `HvlSwitchToVsmVtl1`, which executes a `vmcall` instruction with the SSCN specified in `rax` and `rcx` set to 0x11:

Listing 7: Transition from VTL 0 to VTL 1

```
kd> u poi(nt!HvlpVsmVtlCallVa)
fffff8032e3c000f 488bc1           mov     rax,rcx
fffff8032e3c0012 48c7c111000000   mov     rcx,11h
fffff8032e3c0019 0f01c1           vmcall
```

This allows to perform a *hypercall*, which causes a *VMEXIT* in the hypervisor [4], and thus allows transition from the kernel to the secure kernel. Therefore the *Virtual Secure Mode* is switched from VTL 0 to VTL 1. Hyper-V dispatchs the *hypercall* and the routine `IumInvokeSecureService` is executed in the secure kernel. Readers interested in more details regarding *hypercall* are invited to read the blogpost written by A. Chevalier [2].

In the secure kernel, the routine `IumInvokeSecureService` dispatches the SSCN value and calls the appropriate function. For the shadow stack, we identified the following secure call exposed by the secure kernel:

SSCN	kernel	secure kernel
0xE6	VslAllocateKernelShadowStack	SkmmCreateNtKernelShadowStack
0xE7	VslFreeKernelShadowStack	SkmmDestroyNtKernelShadowStack
0x112	VslKernelShadowStackAssist	SkmmNtKernelShadowStackAssist
0xE8	VslResetKernelShadowStack	SkmmResetNtKernelShadowStack

Table 3. Secure call related to the shadow stack (Windows 24H2)

4.5 Shadow stack initialization

Now that we introduced the notion of secure call, let's dive into the shadow stack initialization by the secure kernel. As we saw previously, the shadow stack is read only for the NT kernel. Thus, the NT kernel needs to request the secure kernel

[3] SSCN are likely to change (e.g for `VslAllocateKernelShadowStack` it is 0xE3 on a 23H2 build but it is 0xE6 on a 24H2 build)

for the shadow stack initialization. `VslAllocateKernelShadowStack` ends up in `SkmmCreateNtKernelShadowStack`. This function calls `SkmiInitializeNtKernelShadowStack` which is responsible for the shadow stack initialization. This function writes a shadow stack token as well as the address of a start thread routine. The goal is to allow a shadow-stack context switch, when the newly created thread is going to execute. According to the intel manual [8], a shadow token is a 64 bit value composed of:

— Bit 0: Mode bit. If 1, then this shadow stack restore token can be used with a `rstorssp` instruction in 64-bit mode
— Bit 1: Reserved. Must be zero.
— Bit 63:2: Value of shadow stack pointer when this restore point was created.

One can observe the impact of `SkmiInitializeNtKernelShadowStack`, by dumping the shadow-stack content in the kernel, once the `VslAllocateKernelShadowStack` routine returns. Please note that at the time, the new thread has not started its execution, and the type of kernel shadow stack was `KernelShadowStackTypeKernelThread` for this call:

Listing 8: Shadow stack content after `SkmiInitializeNtKernelShadowStack`

```
kd> dps ffffa58ae0bcb000+0x2000-0x20 L?5
ffffa58ae0bccfe0  0000000000000000 # not yet accessed
ffffa58ae0bccfe8  ffffa58ae0bccff1 # shadow stack token
ffffa58ae0bccff0  fffff8050b417670 nt!KiStartSystemThread
ffffa58ae0bccff8  0000000000000000
ffffa58ae0bcd000  ???????????????? # start of second guard page
```

The shadow-stack token, `0xffffa58ae0bccff1` has its mode bit set, thus it can be used by a `rstorssp` instruction. Once the lowest two bits cleared, it gives the address `0xffffa58ae0bccff0`. This will be the value of the SSP after execution of the `rstorssp` instruction, which consumes the shadow stack token and sets the new SSP value. At that time, the SSP will thus points on a 64 bits value which is the address of the function `nt!KiStartSystemThread`. According to the type of shadow stack requested, it can be another function. Indeed, the secure kernel gets the address of the start routines from a table named `SkmmNtFunctionTable`.

4.6 _KTHREAD structure initialization

In Windows 11 21H2 (2022 Update), the `_KTHREAD` structure has been updated in order to add the following fields, introducing the support of the kernel shadow stack:

Listing 9: New fields in the `_KTHREAD` structure

```c
// Offset 0x408
void *KernelShadowStack;
void *KernelShadowStackInitial;
void *KernelShadowStackBase;
_KERNEL_SHADOW_STACK_LIMIT KernelShadowStackLimit;
```

As explained previously, when returning from `VslAllocateKernelShadowStack`, the shadow stack of the new kernel thread has been initialized by the secure kernel. The `KeInitThread` function can then update the shadow stack related fields, as shown in the code snippet below. The field which we named as `pShadowStackUpdated` has been computed by the secure kernel, to decrement the SSP value, each time a value was written in the shadow stack, during its initialization:

```c
// Points on the Shadow stack token
pkthread->KernelShadowStack = pKernelShadowStackUpdated;

// Points on the start routine (such as nt!KiStartSystemThread)
pkthread->KernelShadowStackInitial = pKernelShadowStackUpdated + 8;

// Points at the top of the guard page above the shadow stack
pkthread->KernelShadowStackBase = pKernelShadowStackBase;

// _KERNEL_SHADOW_STACK_LIMIT
// Bits 0:2 (3 lowers bits) are used to set ShadowStackType (enum
↪  _KERNEL_SHADOW_STACK_TYPE)
// Bits 3:11 are unused
// Bits 12:63 are used to set ShadowStackLimitPfn (52 bits)
pkthread->KernelShadowStackLimit.AllFields = (unsigned int)ShadowStackType
    | pkthread->KernelShadowStackLimit.AllFields & 0xFF8
    | (pKernelShadowStackBase - 0x3000) & 0xFFFFFFFFFFFFF000uLL;
```

Figure 5 bellow illustrates the layout of the shadow stack once the `KeInitThread` function returns.

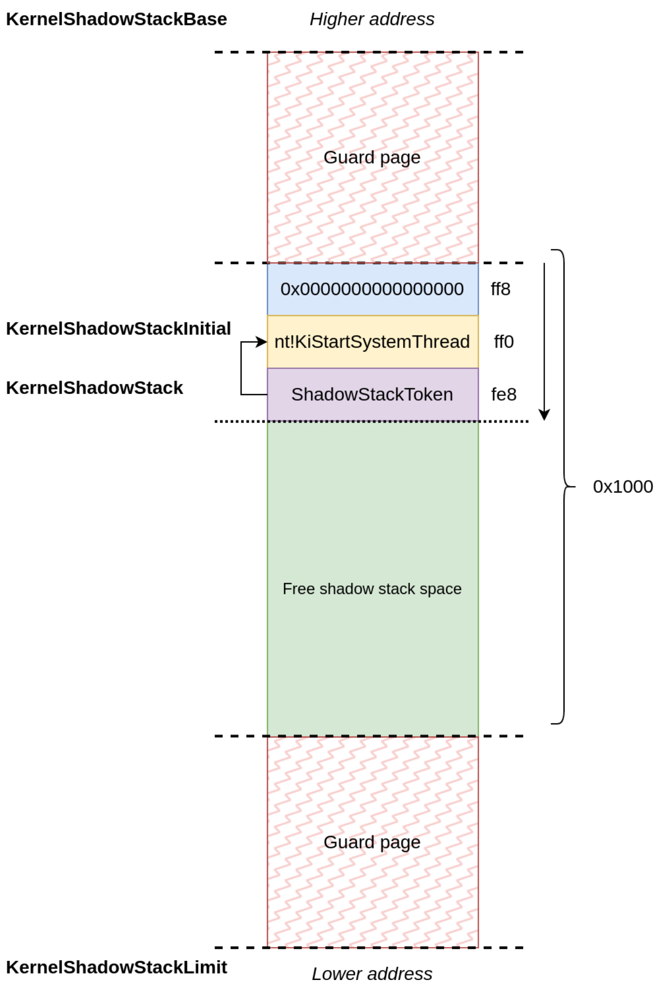


Fig. 5. Shadow stack layout after `KeInitThread` execution

4.7 Shadow stack protection

In section 4.5, we mentioned the role of `SkmmCreateNtKernelShadowStack`. This function is also responsible for protecting the shadow stack integrity. Indeed, once the shadow stack is allocated, the secure kernel is used to protect its integrity from a write primitive in VTL 0. Indeed, an attacker could try to make the shadow stack writable, by modifying the PTE to flip the `Write` bit. The goal here is not to protect the PTE itself (it is writable by design) but rather set the appropriate bits in the EPT, so that an EPT violation would occur on an illegal write tentative on the shadow stack (even if the

PTE would mark the page as writable). As explained previously, in VTL 0, the shadow stack is read only and only instructions such as `call` or `wrssq` (in audit mode) are allowed to write into the shadow stack.

To enforce this ready only permission, the secure kernel uses a function named `SkmiProtectSinglePage`. This function uses `ShvlpProtectPages` which initiates a hypercall number 0xC. This hypercall is mentioned as `HvCallModifyVtlProtectionMask` in the top-level functional specification (TLFS) of the Microsoft hypervisor [14], and is also officially documented by Microsoft:

Listing 10: Definition of `HvModifyVtlProtectionMask`

```
HV_STATUS
HvModifyVtlProtectionMask(
    _In_ HV_PARTITION_ID TargetPartitionId,
    _In_ HV_MAP_GPA_FLAGS MapFlags,
    _In_ HV_INPUT_VTL TargetVtl,
    _In_reads(PageCount) HV_GPA_PAGE_NUMBER GpaPageList
    );
```

Unlike for the NT kernel or the secure kernel, Microsoft does not release a PDB file regarding `hvix64.exe`.[4] In 2019, A. Ionescu released an (unofficial) Hyper-V Development Kit header file [9], based on a file named `HvGdk.h`. This file was shipped once, with the Windows Driver Kit for Windows 7. This allows to partially understand the permissions behind `HV_MAP_GPA_FLAGS`:

Listing 11: Access flag to a GPA

```
//
// Flags to describe the access a partition has to a GPA page.
//
typedef UINT32 HV_MAP_GPA_FLAGS;

#define HV_MAP_GPA_READABLE     (0x00000001)
#define HV_MAP_GPA_WRITABLE     (0x00000002)
#define HV_MAP_GPA_EXECUTABLE   (0x00000004)
```

When `SkmiProtectSinglePage` is called, the index 0x3 is used to retrieve the `MapFlags` from an array named `SkmiVtlPageAccess`. This results in the value 0x11, which could be decomposed as `HV_MAP_GPA_READABLE | 0x10`. First, one can notice that the page will

[4] The Hypervisor Interface for Intel (x64)

not be writable as the bit `HV_MAP_GPA_WRITABLE` is not set. Then the bit related to the value 0x10 is probably used to specify that this page is a shadow stack page. It is probably used, to configure the EPTE in order to set the bit 60 in the EPTE. Indeed, the bit 60 in an EPT Page-Table entry is described in the intel manual [8] as the supervisor shadow stack's bit. Instructions such as `rstorssp` use this bit to check whether the page backing the shadow stack is legitimate and not a simple writable page that an attacker could have allocated with abitrary execution in VTL 0. Please note that we did not verify this statement by debugging *hvix64.exe*, thus we may have missed some details. However, this is something that we would like to do in a near future.

5 Fault handling

In this section we will describe how the control protection exception (#CP) handler, `KiControlProtectionFault`, handles a shadow stack related fault. This handler is called if a fault is generated by a userland process (CPL3), or by a kernel thread (CPL0). As explained in the intel manual [8], when an interruption occurs, without privilege-level change (i.e when the fault is generated by a kernel thread) three values are pushed on the shadow stack (this is not specific to a #CP fault):

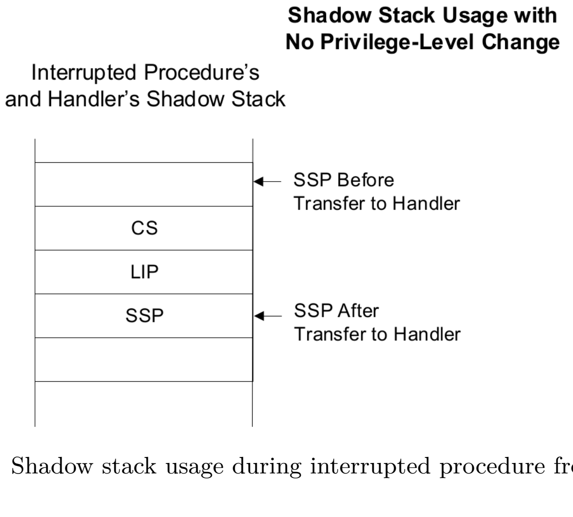


Fig. 6. Shadow stack usage during interrupted procedure from [8]

These value are documented in the Intel manual [8] such as:

— tempSsCS: Value of the code segment (i.e 0x10 if the fault was generated by a kernel thread)

— tempSsLIP: Value of the instruction pointer when the fault was generated
— tempSSP: Value of the SSP when the fault was generated

Figure 7 illustrates the layout of the stack and shadow-stack, when a #CP fault is generated. In this example, the call stack is composed of few functions labeled `functionA` to `functionE`. The return address consumed when functionE ends, has been overridden and does not matches with `functionD+0x10`. Thus, a #CP fault is raised by the CPU.

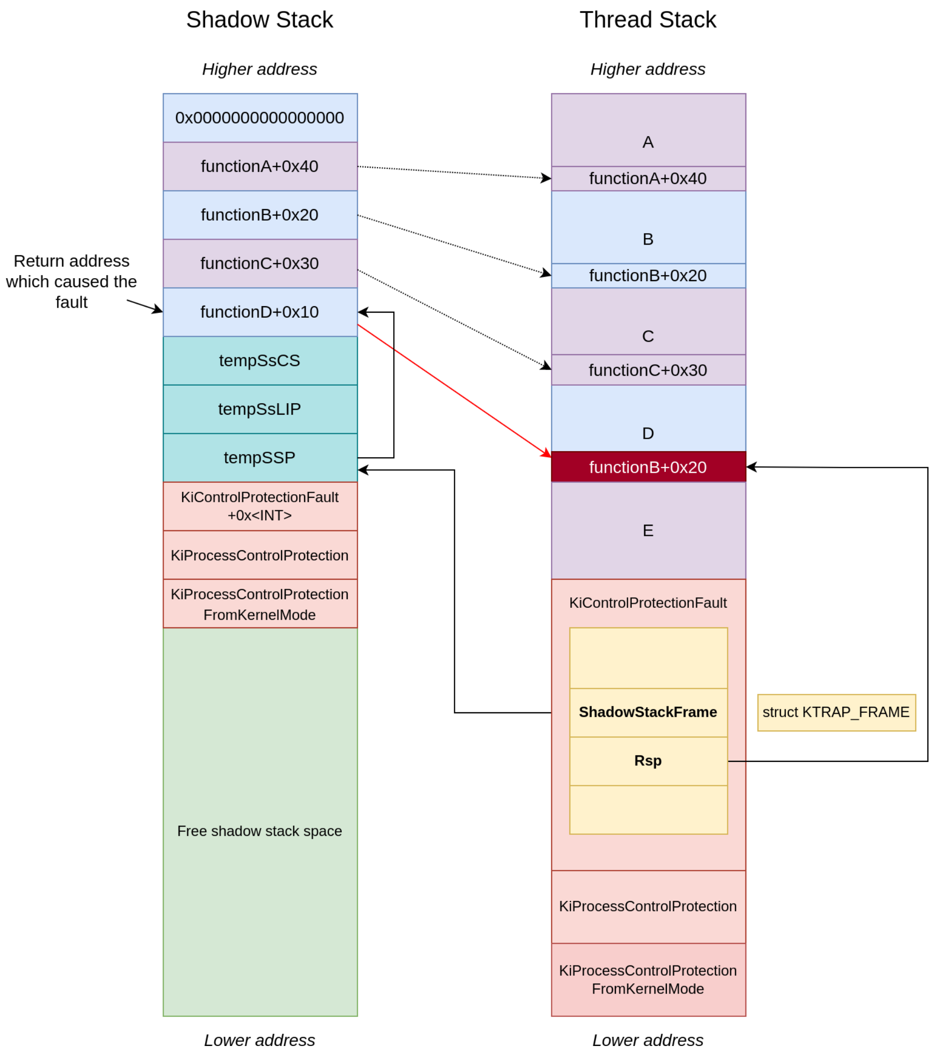


Fig. 7. Shadow stack layout after a #CP fault

In a similar way, the stack from the thread in which the interruption occurs, is used by the interruption handler. Therefore, the function `KiControlProtectionFault` setup a `KTRAP_FRAME` structure on its stack-frame. This is a kernel mode data structure used to save the processor's state during exceptions or interrupts. It captures the state of the CPU registers when a transition from user mode to kernel mode occurs due to events like system calls, page faults, or hardware interrupts. A pointer to this structure is passed to the function `KiProcessControlProtection`. This function checks the code segment field `KTRAP_FRAME.SegCs` in order to check whether the fault is generated by a userland process or a kernel thread. If the code segment field equals 0x10 (CPL is 0), the function `KiProcessControlProtectionFromKernelMode` is called. Otherwise, if the code segment field equals 0x33 (CPL is 3), the fault is handled directly within the `KiProcessControlProtection` function (may be a function like `KiProcessControlProtectionFromUserMode` exists and has been inlined). In [24], faults resulting from user mode have been reviewed. Here, we will focus on faults generated in kernel mode and therefore review the implementation of `KiProcessControlProtectionFromKernelMode`. In section 5.1, we will explain how the fault is handled if the shadow stack is in non-audit mode whereas in section 5.2 we will describe the fault handling in audit mode.

5.1 Non-Audit mode

First, the faulty return address is read by dereferencing the field `KTRAP_FRAME.Rsp`, which points on the top of the stack-frame which caused the fault. Then a call to `KiGetCurrentKernelShadowStackBounds` is made to get the shadow stack boundaries. A loop is then made, starting from the pointer `KTRAP_FRAME.ShadowStackFrame + 0x20` up to the bottom of the shadow stack pointer boundary. The value 0x20 allows to skip `tempSsCS, tempSsLIP, tempSSP` and the return address on the shadow stack which generated the fault. At each iteration, a test is made to check if the faulty return address matches. If the faulty return address is found, the shadow stack can be fixed. This is highlighted in figure 7 where the faulty return address is present in the shadow stack.

A call to `VslKernelShadowStackAssist` allows to perform a *Secure System Call* which writes into the shadow stack and thus allows to fix it. In that context, the secure kernel executes `SkmmNtKernelShadowStackAssist` with a parameter allowing to execute `SkmiNtKssAssistCpPopSsp`. Then, 2 callbacks are executed consecutively: `SkmiNtKssAssistCpPopSspValidationCallback` and SkmiNtKssAssistCpPopSspOperationCallback. The second one fixes the pointer tempSSP so that it points higher on the shadow stack, precisely at the location where the faulty return address was identified during the loop. It also nullifies on the shadow stack, the return address which caused the fault. This simply allows, when the interruption handler routine ends, to restore the SSP with a modified tempSSP which now points on 64-bit value which is a valid return address (it is the same as the one present at the thread's top of stack). Then, KiProcessControlProtectionFromKernelMode returns with the value 0x1 and the function KiControlProtectionFault ends its execution with a iretq instruction, to restore the processor's state after handling the exception. In this case, no BSOD is raised.

However, if the faulty return address has not been found in the shadow stack, this means the call stack has been corrupted. If the global variable KiKernelCetAuditModeEnabled is not set, then KiProcessControlProtectionFromKernelMode returns the value 2, and KiControlProtectionFault will call KiFastFailDispatch in order to trigger a BSOD. Finally, a call to KeBugCheckEx with the code 0x139 (KERNEL_SECURITY_CHECK_FAILURE) is performed. With a kernel debugger attached to the target computer, the following bugcheck analysis could be observed:

```
2: kd> !analyze -v
*******************************************************************
*                                                                 *
*                        Bugcheck Analysis                        *
*                                                                 *
*******************************************************************

KERNEL_SECURITY_CHECK_FAILURE (139)
A kernel component has corrupted a critical data structure. The
↪  corruption could potentially allow a malicious user to gain
↪  control of this machine.
Arguments:
Arg1: 0000000000000039, A shadow stack violation has occurred due
↪  to mismatched return addresses on the call stack vs the shadow
↪  stack.
Arg2: ffffd8840c3ef410, Address of the trap frame for the exception
↪  that caused the BugCheck
Arg3: ffffd8840c3ef368, Address of the exception record for the
↪  exception that caused the BugCheck
Arg4: 0000000000000000, Reserved
```

The subcode (0x39) has a self-explainatory message: *A shadow stack violation has occurred due to mismatched return addresses on the call stack vs the shadow stack.*

5.2 Audit mode

If the faulty return address has not been found on the shadow stack and the global variable `KiKernelCetAuditModeEnabled` is set, then another path is taken. The function `KiFixupControlProtectionKernelModeReturnMismatch` is called in order to try to fix the shadow stack using `wrssq` instructions. If this operation succeeds, the function `KiLogControlProtectionKernelModeReturnMismatch` is called in order to generate an `Event Tracing for Windows` (ETW) log. A call to `EtwTimLogControlProtectionKernelModeReturnMismatch` allows the generation of the log for a provider named `Microsoft-Windows-Security-Mitigations`:

```
result = EtwWriteEx(
  // {FAE10392-F0AF-4AC0-B8FF-9F4D920C3CDF}
  EtwSecurityMitigationsRegHandle,
  &MITIGATION_AUDIT_CONTROL_PROTECTION_KERNEL_MODE_RETURN_MISMATCH,
  0LL,
  0,
  0LL,
  0LL,
  0xDu,
  &UserData);
```

Then, the function `KiProcessControlProtectionFromKernelMode` ends with the return value 1, and no BSOD is raised.

6 Evaluation

As this mitigation is hardware-based, the overhead is very low for the operating system, because the #CP handler is supposed to be executed very rarely. Actually only when a try/except statement causes a mismatch between the return addresses on the call stack vs the shadow stack, or when a control flow corruption occurs.

The shadow stack mitigation is quite effective to catch exploit relying on ROP attacks. Indeed, stack-pivoting to control a set of arbitrary gadgets is not possible anymore, as the first `ret` instruction will cause a #CP fault.

However, the current implementation in the kernel allows returning to any address on the shadow stack. This was already mentioned by Y. Shafir in [22]. Theoretically, it is still possible to use gadgets who allow returning on another address within the shadow stack. In the same way, JOP gadgets can still be used. First, because they do not modify the stack and shadow stack, then, because indirect branch tracking mitigation is not yet supported in the Windows kernel.

Using virtualization mechanims allows implementing this mitigation in a secure way. For instance, it is not possible to disable Intel CET by simply switching the `CR4.CET` bit to 0, because this operation would immediately be caught by HyperGuard. Also, even with a kernel read/write primitive, it is not possible to change the PTE of a page related to the shadow stack, because of HVCI. Writing to the shadow stack is only limited to few instructions such as `call` or `wrssq`. Finally HVCI also protects the (read-only) `CFGRO` section, where the global variables `KiKernelCetEnabled` and `KiKernelCetAuditModeEnabled` live. So without a HVCI bypass, the shadow stack could theoretically not be disabled or switched to audit mode on a running kernel.

7 Proof of concepts

In order to demonstrate various aspects of the shadow stack mitigation, a driver has been developed along with a user client. The client and the driver communicate via IOCTL. These different aspects of the shadow stack will be illustrated through test cases and described in the current section, as detailed below:

- Writing to the shadow stack
- Writing to the MSR registers and the CR4 register
- Incrementing the return address
- Skipping the stack frame
- Try/Except path

It must be mentioned that the audit mode is disabled.

7.1 Writing to the shadow stack

This test case is implemented through `IOCTL_WRITE_CURRENT_SHADOW_STACK`. The driver function

`IoctlWriteShadowStack` handles this IOCTL. The function attempts to write a value to the shadow stack address, plus an offset. Altering the shadow stack in this way should result in a crash.

In this test, the value `0x4141414141414141` is written to the address `u8ShadowStack - 0x200`. The result is as follows in WinDbg:

```
Entering DispatchDeviceControl
Entering IoctlWriteShadowStack
Kernel Shadow Stack: FFFFF1807CBA5FB8
Reading -> FFFFF1807CBA5DB8 = 0000000000000000
Writing -> FFFFF1807CBA5DB8 = 4141414141414141

KDTARGET: Refreshing KD connection

*** Fatal System Error: 0x00000050
                        (0xFFFFF1807CBA5DB8,
                        0x0000000000000003,
                        0xFFFFF80268281ADD,
                        0x0000000000000002)

Driver at fault:
***  shadow_stack_driver.sys - Address FFFFF80268281ADD base at
↪  FFFFF80268280000, DateStamp 67e2e14f
```

Then, the bugcheck analysis:

```
2: kd> !analyze -v
*******************************************************************
*                                                                 *
*                        Bugcheck Analysis                        *
*                                                                 *
*******************************************************************

PAGE_FAULT_IN_NONPAGED_AREA (50)
Invalid system memory was referenced.  This cannot be protected by
↪  try-except.
Typically the address is just plain bad or it is pointing at freed
↪  memory.
Arguments:
Arg1: fffff1807cba5db8, memory referenced.
Arg2: 0000000000000003,
  bit 0 set if the fault was due to a not-present PTE.
  bit 1 is set if the fault was due to a write, clear if a read.
Arg3: fffff80268281add, If non-zero, the instruction address which
↪  referenced the bad memory
  address.
Arg4: 0000000000000002, (reserved)
```

As expected, a `PAGE_FAULT_IN_NONPAGED_AREA` error occurs when attempting a write operation to the memory reference address `0xFFFFF1807CBA5DB8`. This is caused by the read-only rights of the page, which can be illustrated with WinDbg and the address `0xFFFFF1807CBA5DB8`:

```
2: kd> !pte FFFFF1807CBA5DB8
                                           VA fffff1807cba5db8
PXE at FFFFB85C2E170F18    PPE at FFFFB85C2E1E3008    PDE at
↪  FFFFB85C3C601F28    PTE at FFFFB878C03E5D28
contains 0A0000088778F863  contains 0A00000887790863  contains
↪  0A000007AFB49863  contains 8A0000015F536161
pfn 88778f    ---DA--KWEV  pfn 887790    ---DA--KWEV  pfn 7afb49
↪ ---DA--KWEV  pfn 15f536    -G-DA--KR-V
```

Thus, this test case demonstrates that the shadow stack cannot be rewritten as mentioned in section 4.7.

7.2 Writing to the MSR registers and the CR4 register

This test case is implemented through `IOCTL_WRITE_MSR`. The driver function `IoctlWriteMsr` handles this IOCTL. The function attempts to write a value into an MSR register. As mentioned in [18], some registers are monitored for access or modifications. Depending of the register address, this should result in a crash triggered by HyperGuard.

In this test, the MSR register `0x6a2`, which is `IA32_S_CET_REGISTER`, is set. Since the targeted machine is currently being debugged, the value of this register is `3` instead of `1`. Therefore, the value set to this MSR register is `1`. The result is as follows in WinDbg:

```
Entering DispatchDeviceControl
Entering IoctlWriteMsr
Reading MSR[0x06a2] = 0x0000000000000003
Writing MSR[0x06a2] = 0x0000000000000001
KDTARGET: Refreshing KD connection

*** Fatal System Error: 0x0000003b
                       (0x00000000C0000096,
                        0xFFFFF800355318FB,
                        0xFFFFE40A43C6EB10,
                        0x0000000000000000)
```

As expected an exception occurs. According to the bugcheck analysis:

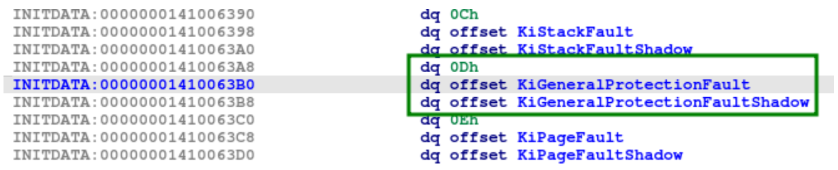
```
*******************************************************************
*                                                                 *
*                        Bugcheck Analysis                        *
*                                                                 *
*******************************************************************

SYSTEM_SERVICE_EXCEPTION (3b)
An exception happened while executing a system service routine.
Arguments:
Arg1: 00000000c0000096, Exception code that caused the BugCheck
Arg2: fffff8057c9218fb, Address of the instruction which caused the
↪  BugCheck
Arg3: ffffe400431c6b10, Address of the context record for the exception
↪  that caused the BugCheck
Arg4: 0000000000000000, zero.

cs=0010  ss=0018  ds=002b  es=002b  fs=0053  gs=002b
↪  efl=00040246
shadow_stack_driver+0x18fb:
fffff800`355318fb 0f30            wrmsr
```

This exception is due to a `SYSTEM_SERVICE_EXCEPTION` (0x3b), specifically caused by the execution of a privileged instruction, as indicated by the `STATUS_PRIVILEGED_INSTRUCTION` (0xc0000096) code. The culprit instruction is `wrmsr`, but the code is running in `CPL0` as the `cs` register shows, and according to [8], the `rdmsr` and `wrmsr` instructions are normally allowed.

With the call stack as follows:

```
2: kd> k
 # Child-SP          RetAddr               Call Site
00 ffffe400`431c5978 fffff805`d256c432     nt!DbgBreakPointWithStatus
01 ffffe400`431c5980 fffff805`d256b95c     nt!KiBugCheckDebugBreak+0x12
02 ffffe400`431c59e0 fffff805`d24b8c07     nt!KeBugCheck2+0xb2c
03 ffffe400`431c6170 fffff805`d2686fe9     nt!KeBugCheckEx+0x107
04 ffffe400`431c61b0 fffff805`d268603c     nt!KiBugCheckDispatch+0x69
05 ffffe400`431c62f0 fffff805`d267c69f     nt!KiSystemServiceHandler+0x7c
06 ffffe400`431c6330 fffff805`d239dc72
↪  nt!RtlpExecuteHandlerForException+0xf
07 ffffe400`431c6360 fffff805`d239edd9     nt!RtlDispatchException+0x2d2
08 ffffe400`431c6ae0 fffff805`d2687145     nt!KiDispatchException+0xac9
09 ffffe400`431c7210 fffff805`d2681e25     nt!KiExceptionDispatch+0x145
0a ffffe400`431c73f0 fffff805`7c9218fb
↪  nt!KiGeneralProtectionFault+0x365
0b ffffe400`431c7580 fffff805`7c921231     shadow_stack_driver+0x18fb
0c ffffe400`431c75b0 fffff805`d229697e     shadow_stack_driver+0x1231
0d ffffe400`431c75e0 fffff805`d288a568     nt!IofCallDriver+0xbe
0e ffffe400`431c7620 fffff805`d2889400
↪  nt!IopSynchronousServiceTail+0x1c8
0f ffffe400`431c76d0 fffff805`d2888aae     nt!IopXxxControlFile+0x940
10 ffffe400`431c7940 fffff805`d2686655     nt!NtDeviceIoControlFile+0x5e
11 ffffe400`431c79b0 00007ffe`94bdeee4     nt!KiSystemServiceCopyEnd+0x25
```

The function `nt!KiGeneralProtectionFault`, pointed to by stack frame number `0x0a`, is of interest. It is related to interrupt code `0xd` from the `nt!KiInterruptInitTable` of the kernel, as shown in figure 8.



```
INITDATA:0000000141006390                 dq 0Ch
INITDATA:0000000141006398                 dq offset KiStackFault
INITDATA:00000001410063A0                 dq offset KiStackFaultShadow
INITDATA:00000001410063A8                 dq 0Dh
INITDATA:00000001410063B0                 dq offset KiGeneralProtectionFault
INITDATA:00000001410063B8                 dq offset KiGeneralProtectionFaultShadow
INITDATA:00000001410063C0                 dq 0Eh
INITDATA:00000001410063C8                 dq offset KiPageFault
INITDATA:00000001410063D0                 dq offset KiPageFaultShadow
```

Fig. 8. `KiGeneralProtectionFault` handler

As mentioned earlier, the hypervisor monitors certain MSR registers and content modifications. When register `0x6a2` is accessed for writing in order to alter it, the hypervisor raises a general protection fault.

Thus, this test case demonstrates that MSR registers cannot be altered as discussed in section 6.

Note: The alteration of the `CR4` register will not be demonstrated, as its related mitigation is the same as the one in place for the MSR registers.

7.3 Incrementing the return address

This test case is implemented through `IOCTL_INC_RET_ADDR`. The driver function `IncRetAddr` is called by `IoctlIncRetAddr`, which handles this IOCTL. The function simulates an unaligned return address, similar to how a ROP chain operates. Returning to an address which is not present in the shadow stack may result in a crash.

The `IncRetAddr` function alters its return address by incrementing it by one. The caller function, `IoctlIncRetAddr`, which calls the `IncRetAddr` function in assembly, is shown below:

```
IoctlIncRetAddr proc near

sub     rsp, 28h
lea     rcx, aEnteringIoctli ; "Entering IoctlIncRetAddr\n"
call    DbgPrint
call    IncRetAddr
lea     rcx, aLeavingIoctlin ; "Leaving IoctlIncRetAddr\n"
call    DbgPrint
xor     eax, eax
add     rsp, 28h
retn

IoctlIncRetAddr endp
```

By incrementing its return address by one, the `IncRetAddr` function updates the return address to point from `lea rcx, aLeavingIoctinc` to `lea ecx, aLeavingIoctinc`, which is still a valid instruction. Going through this IOCTL results in the following in WinDbg:

```
Entering DispatchDeviceControl
Entering IoctIncRetAddr
Entering IncRetAddr
Legitimate return address:  FFFFF80362B916E5
Incremented return address: FFFFF80362B916E6
Leaving IncRetAddr
KDTARGET: Refreshing KD connection

*** Fatal System Error: 0x00000139
                       (0x0000000000000039,
                        0xFFFFF880F1E9F3C0,
                        0xFFFFF880F1E9F318,
                        0x0000000000000000)
```

As expected an exception occurs. The bugcheck analysis is as follows:

```
2: kd> !analyze -v
*******************************************************************
*                                                                 *
*                        Bugcheck Analysis                        *
*                                                                 *
*******************************************************************

KERNEL_SECURITY_CHECK_FAILURE (139)
A kernel component has corrupted a critical data structure.  The
↪  corruption
could potentially allow a malicious user to gain control of this
↪  machine.
Arguments:
Arg1: 0000000000000039, A shadow stack violation has occurred due
↪  to mismatched return addresses
        on the call stack vs the shadow stack.
Arg2: fffff880f1e9f3c0, Address of the trap frame for the exception
↪  that caused the BugCheck
Arg3: fffff880f1e9f318, Address of the exception record for the
↪  exception that caused the BugCheck
Arg4: 0000000000000000, Reserved
```

A `KERNEL_SECURITY_CHECK_FAILURE` (0x139) is triggered with the corruption code `0x39`. Which indicates that the return address on the stack mismatches the one from the shadow stack, as intended.

The call stack is as follows:

```
2: kd> k
 # Child-SP          RetAddr               Call Site
00 fffff880`f1e9e868 fffff803`b936c432     nt!DbgBreakPointWithStatus
01 fffff880`f1e9e870 fffff803`b936b95c     nt!KiBugCheckDebugBreak+0x12
02 fffff880`f1e9e8d0 fffff803`b92b8c07     nt!KeBugCheck2+0xb2c
03 fffff880`f1e9f060 fffff803`b9486fe9     nt!KeBugCheckEx+0x107
04 fffff880`f1e9f0a0 fffff803`b94875f2     nt!KiBugCheckDispatch+0x69
05 fffff880`f1e9f1e0 fffff803`b9484b1f     nt!KiFastFailDispatch+0xb2
06 fffff880`f1e9f3c0 fffff803`62b92787
↪  nt!KiControlProtectionFault+0x3df
07 fffff880`f1e9f558 fffff803`62b916e6     shadow_stack_driver+0x2787
08 fffff880`f1e9f560 fffff803`62b911a4     shadow_stack_driver+0x16e6
09 fffff880`f1e9f590 fffff803`b909697e     shadow_stack_driver+0x11a4
0a fffff880`f1e9f5e0 fffff803`b968a568     nt!IofCallDriver+0xbe
0b fffff880`f1e9f620 fffff803`b9689400
↪  nt!IopSynchronousServiceTail+0x1c8
0c fffff880`f1e9f6d0 fffff803`b9688aae     nt!IopXxxControlFile+0x940
0d fffff880`f1e9f940 fffff803`b9486655     nt!NtDeviceIoControlFile+0x5e
0e fffff880`f1e9f9b0 00007ffd`499beee4     nt!KiSystemServiceCopyEnd+0x25
```

The function `nt!KiControlProtectionFault`, pointed to by stack frame number `0x06`, is of interest. It is related to interrupt code `0x15` from the `nt!KiInterruptInitTable` of the kernel, as mentioned in section 4.1. This interrupt handler is called due to a CET fault as discussed in section 5.1.

Then, the related shadow stack:

```
2: kd> dps @ssp
ffffb102`f1e59f78  fffff803`b936c432 nt!KiBugCheckDebugBreak+0x12
ffffb102`f1e59f80  fffff803`b936b95c nt!KeBugCheck2+0xb2c
ffffb102`f1e59f88  fffff803`b92b8c07 nt!KeBugCheckEx+0x107
ffffb102`f1e59f90  fffff803`b9486fe9 nt!KiBugCheckDispatch+0x69
ffffb102`f1e59f98  fffff803`b94875f2 nt!KiFastFailDispatch+0xb2
ffffb102`f1e59fa0  fffff803`b9484b1f nt!KiControlProtectionFault+0x3df
ffffb102`f1e59fa8  ffffb102`f1e59fc0
ffffb102`f1e59fb0  fffff803`62b92787 shadow_stack_driver+0x2787
ffffb102`f1e59fb8  00000000`00000010
ffffb102`f1e59fc0  fffff803`62b916e5 shadow_stack_driver+0x16e5
ffffb102`f1e59fc8  fffff803`62b911a4 shadow_stack_driver+0x11a4
ffffb102`f1e59fd0  fffff803`b909697e nt!IofCallDriver+0xbe
ffffb102`f1e59fd8  fffff803`b968a568 nt!IopSynchronousServiceTail+0x1c8
ffffb102`f1e59fe0  fffff803`b9689400 nt!IopXxxControlFile+0x940
ffffb102`f1e59fe8  fffff803`b9688aae nt!NtDeviceIoControlFile+0x5e
ffffb102`f1e59ff0  fffff803`b9486655 nt!KiSystemServiceCopyEnd+0x25
```

The faulting return address is located at `0xFFFFB102F1E59FC0`, before the `cs` (0x10) register, as illustrated in figure 7 of section 5. During the call to the handler, the function `nt!KiProcessControlProtectionFromKernelMode` is reached through sub-calls, in order to know if the faulty address is located in the shadow stack. Because the addres is not present in the shadow stack, the control flow falls into the `nt!KiFastFailDispatch` function.

Thus, this test case demonstrates that the shadow stack mitigation prevents control flow redirection through ROP chain.

7.4 Skipping the stack frame

This test case is implemented through `IOCTL_SKIP_NEXT_FRAME`. The driver function `SkipNextFrame` is called by `IoctlSkipNextFrame`, which handles this IOCTL. The function retrieves the address of a previous stack frame and set the current `rsp` register to it. Returning to an address which is present in the shadow stack may not result in a crash.

The `SkipNextFrame` function locates the address of its return address on the stack. It then calls the `setRsp` assembly function, which updates the `rsp` register to point to the return address in `IoctlSkipNextFrame`.

The result is shown below in WinDbg:

```
Entering DispatchDeviceControl
Entering IoctlSkipNextFrame
Entering SkipNextFrame
Module found: FFFFF8000D770000
Leaving IoctlSkipNextFrame
Leaving DispatchDeviceControl
```

Everything seems to be fine. As in the previous test, the function `nt!KiProcessControlProtectionFromKernelMode` is reached through sub-calls, as shown in the following output from WinDbg with a breakpoint set on it:

```
Entering DispatchDeviceControl
Entering IoctlSkipNextFrame
Entering SkipNextFrame
Module found: FFFFF8027B930000
Breakpoint 0 hit
nt!KiProcessControlProtectionFromKernelMode:
fffff800`75a28b44 4c8bdc          mov     r11,rsp
```

With the following call stack:

```
2: kd> k
 # Child-SP          RetAddr               Call Site
00 ffff960f`fe02f2f8 fffff802`d1a28aa0
↪  nt!KiProcessControlProtectionFromKernelMode
01 ffff960f`fe02f300 fffff802`d1c84a96
↪  nt!KiProcessControlProtection+0x330
02 ffff960f`fe02f3c0 fffff802`7b933313
↪  nt!KiControlProtectionFault+0x356
03 ffff960f`fe02f558 fffff802`7b931ca5     shadow_stack_driver+0x3313
04 ffff960f`fe02f560 fffff802`7b9311cb     shadow_stack_driver+0x1ca5
05 ffff960f`fe02f590 fffff802`d189697e     shadow_stack_driver+0x11cb
06 ffff960f`fe02f5e0 fffff802`d1e8a568     nt!IofCallDriver+0xbe
07 ffff960f`fe02f620 fffff802`d1e89400
↪  nt!IopSynchronousServiceTail+0x1c8
08 ffff960f`fe02f6d0 fffff802`d1e88aae     nt!IopXxxControlFile+0x940
09 ffff960f`fe02f940 fffff802`d1c86655     nt!NtDeviceIoControlFile+0x5e
0a ffff960f`fe02f9b0 00007ff8`12ffeee4     nt!KiSystemServiceCopyEnd+0x25
```

And the shadow stack is as follows:

```
2: kd> dps @ssp
ffffcb0b`a80aff90  fffff802`d1a28aa0 nt!KiProcessControlProtection+0x330
ffffcb0b`a80aff98  fffff802`d1c84a96 nt!KiControlProtectionFault+0x356
ffffcb0b`a80affa0  ffffcb0b`a80affb8
ffffcb0b`a80affa8  fffff802`7b933313 shadow_stack_driver+0x3313
ffffcb0b`a80affb0  00000000`00000010
ffffcb0b`a80affb8  fffff802`7b932fa7 shadow_stack_driver+0x2fa7
ffffcb0b`a80affc0  fffff802`7b931ca5 shadow_stack_driver+0x1ca5
ffffcb0b`a80affc8  fffff802`7b9311cb shadow_stack_driver+0x11cb
ffffcb0b`a80affd0  fffff802`d189697e nt!IofCallDriver+0xbe
ffffcb0b`a80affd8  fffff802`d1e8a568 nt!IopSynchronousServiceTail+0x1c8
ffffcb0b`a80affe0  fffff802`d1e89400 nt!IopXxxControlFile+0x940
ffffcb0b`a80affe8  fffff802`d1e88aae nt!NtDeviceIoControlFile+0x5e
ffffcb0b`a80afff0  fffff802`d1c86655 nt!KiSystemServiceCopyEnd+0x25
```

The faulty instruction is located at `shadow_stack_driver+0x3313`, which corresponds to the `ret` instruction of `setRsp`. Normally, the `ret` instruction returns to `shadow_stack_driver+0x2fa7`, as expected by the shadow stack. However, since `rsp` has been updated to `0xFFFF960FFE02F560`, which points to `shadow_stack_driver+0x1ca5`, the stack no longer matches the shadow stack. Because the return address is still present in the shadow stack, as indicated in section 5.1, the `nt!VslKernelShadowStackAssist` function is invoked to correct the shadow stack and realign it with the current stack.

Thus, this test case demonstrates that it is possible to alter return addresses in a way that does not break the shadow stack mitigation.

7.5 Try/Except path

This test case is implemented through `IOCTL_DIV_INTEGER`. The driver function `DivInteger` is called by `IoctlDivInteger`, which handles this

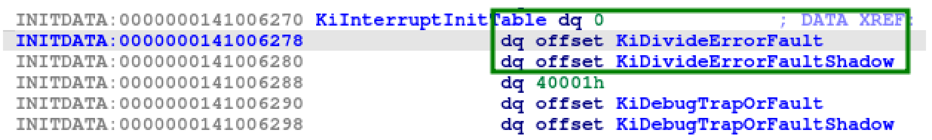
IOCTL. As the name suggests, the function performs integer divisions. Using the `try/except` mechanism may prevent a crash in the event that an exception occurs.

It is interesting to note that these keywords refer to `__try/__except` through macros. The usage of these keywords is presented by Microsoft in [16]. The internal mechanism is quite similar to the one in userland, as it uses the same structures under the hood in both the kernel and userland. The relevant structures are described in [17].

To trigger an exception, the operation `1 / 0` is performed, raising a division by zero exception. The result is shown below in WinDbg:

```
Entering DispatchDeviceControl
Entering IoctlDivInteger
Entering DivInteger
_DivZeroFilter
A division by zero occurred
Leaving IoctlDivInteger
IoctlDivInteger failed
Leaving DispatchDeviceControl
```

Due to the `try/except` block, the crash is avoided. It is important to note that the call stack during the exception does not match the shadow stack. To trace the exception caused by the division by zero, a breakpoint is set at `nt!KiDivideErrorFault`. This handler is responsible for managing division by zero exceptions. It is related to interrupt code `0` from the `nt!KiInterruptInitTable` of the kernel, as shown figure 9.



```
INITDATA:0000000141006270 KiInterruptInitTable dq 0                ; DATA XREF
INITDATA:0000000141006278                      dq offset KiDivideErrorFault
INITDATA:0000000141006280                      dq offset KiDivideErrorFaultShadow
INITDATA:0000000141006288                      dq 40001h
INITDATA:0000000141006290                      dq offset KiDebugTrapOrFault
INITDATA:0000000141006298                      dq offset KiDebugTrapOrFaultShadow
```

Fig. 9. `KiDivideErrorFault` handler

Once the first breakpoint is hit, a second breakpoint is set at `nt!KeKernelShadowStackRestoreContext+0x4c`. This address corresponds to the call to `nt!VslKernelShadowStackAssist` when the shadow stack context is restored.

This address represents the end of the `nt!KeKernelShadowStackRestoreContext` function, as illustrated in the snippet below:

```
nt!KeKernelShadowStackRestoreContext+0x4c   call
↪  VslKernelShadowStackAssist
nt!KeKernelShadowStackRestoreContext+0x51   add     rsp, 38h
nt!KeKernelShadowStackRestoreContext+0x55   retn
```

At this breakpoint, the shadow stack appears as follows:

```
fffffa89`ac9d2f60  fffff807`d8eb9ebb nt!RtlRestoreContext+0x21b
fffffa89`ac9d2f68  fffff807`d8c5fdb4 nt!RtlUnwindEx+0x374
fffffa89`ac9d2f70  fffff807`d8eb8992 nt!_C_specific_handler+0xe2
fffffa89`ac9d2f78  fffff807`d907c69f nt!RtlpExecuteHandlerForException+0xf
fffffa89`ac9d2f80  fffff807`d8d9dc72 nt!RtlDispatchException+0x2d2
fffffa89`ac9d2f88  fffff807`d8d9edd9 nt!KiDispatchException+0xac9
fffffa89`ac9d2f90  fffff807`d9087145 nt!KiExceptionDispatch+0x145
fffffa89`ac9d2f98  fffff807`d907e44f nt!KiDivideErrorFault+0x34f
fffffa89`ac9d2fa0  fffffa89`ac9d2fb8
fffffa89`ac9d2fa8  fffff807`82e524d5 shadow_stack_driver+0x24d5
fffffa89`ac9d2fb0  00000000`00000010
fffffa89`ac9d2fb8  fffff807`82e52549 shadow_stack_driver+0x2549
fffffa89`ac9d2fc0  fffff807`82e5167e shadow_stack_driver+0x167e
fffffa89`ac9d2fc8  fffff807`82e5140e shadow_stack_driver+0x140e
fffffa89`ac9d2fd0  fffff807`d8c9697e nt!IofCallDriver+0xbe
fffffa89`ac9d2fd8  fffff807`d928a568 nt!IopSynchronousServiceTail+0x1c8
fffffa89`ac9d2fe0  fffff807`d9289400 nt!IopXxxControlFile+0x940
fffffa89`ac9d2fe8  fffff807`d9288aae nt!NtDeviceIoControlFile+0x5e
fffffa89`ac9d2ff0  fffff807`d9086655 nt!KiSystemServiceCopyEnd+0x25
fffffa89`ac9d2ff8  00000000`00000000
```

Stepping over the call to `KeKernelShadowStackRestoreContext` results in the following shadow stack:

```
fffffa89`ac9d2f40  fffff807`d8eb9ebb nt!RtlRestoreContext+0x21b
fffffa89`ac9d2f48  fffffa89`ac9d2fc8
fffffa89`ac9d2f50  fffff807`82e5169a shadow_stack_driver+0x169a
fffffa89`ac9d2f58  00000000`00000010
fffffa89`ac9d2f60  00000000`00000000
fffffa89`ac9d2f68  fffff807`d8c5fdb4 nt!RtlUnwindEx+0x374
fffffa89`ac9d2f70  fffff807`d8eb8992 nt!_C_specific_handler+0xe2
fffffa89`ac9d2f78  fffff807`d907c69f nt!RtlpExecuteHandlerForException+0xf
fffffa89`ac9d2f80  fffff807`d8d9dc72 nt!RtlDispatchException+0x2d2
fffffa89`ac9d2f88  fffff807`d8d9edd9 nt!KiDispatchException+0xac9
fffffa89`ac9d2f90  fffff807`d9087145 nt!KiExceptionDispatch+0x145
fffffa89`ac9d2f98  fffff807`d907e44f nt!KiDivideErrorFault+0x34f
fffffa89`ac9d2fa0  fffffa89`ac9d2fb8
fffffa89`ac9d2fa8  fffff807`82e524d5 shadow_stack_driver+0x24d5
fffffa89`ac9d2fb0  00000000`00000010
fffffa89`ac9d2fb8  fffff807`82e52549 shadow_stack_driver+0x2549
fffffa89`ac9d2fc0  fffff807`82e5167e shadow_stack_driver+0x167e
fffffa89`ac9d2fc8  fffff807`82e5140e shadow_stack_driver+0x140e
fffffa89`ac9d2fd0  fffff807`d8c9697e nt!IofCallDriver+0xbe
fffffa89`ac9d2fd8  fffff807`d928a568 nt!IopSynchronousServiceTail+0x1c8
fffffa89`ac9d2fe0  fffff807`d9289400 nt!IopXxxControlFile+0x940
fffffa89`ac9d2fe8  fffff807`d9288aae nt!NtDeviceIoControlFile+0x5e
fffffa89`ac9d2ff0  fffff807`d9086655 nt!KiSystemServiceCopyEnd+0x25
fffffa89`ac9d2ff8  00000000`00000000
```

It is important to focus on the start of the shadow stack:

Shadow Stack Space	Value	Description
ffffa89‘ac9d2f48	ffffa89‘ac9d2fc8	New old SSP
ffffa89‘ac9d2f50	shadow_stack_driver+0x169a	Address of the except statement
ffffa89‘ac9d2f58	00000000‘00000010	CS
ffffa89‘ac9d2f60	00000000‘00000000	Act as the nullified return address

Table 4. Live shadow stack description in `KiDivideErrorFault` handler

Once the `iretq` instruction of the `RtlRestoreContext` function is executed, the shadow stack appears as follows:

```
fffffa89`ac9d2fc8  fffff807`82e5140e shadow_stack_driver+0x140e
fffffa89`ac9d2fd0  fffff807`d8c9697e nt!IofCallDriver+0xbe
fffffa89`ac9d2fd8  fffff807`d928a568 nt!IopSynchronousServiceTail+0x1c8
fffffa89`ac9d2fe0  fffff807`d9289400 nt!IopXxxControlFile+0x940
fffffa89`ac9d2fe8  fffff807`d9288aae nt!NtDeviceIoControlFile+0x5e
fffffa89`ac9d2ff0  fffff807`d9086655 nt!KiSystemServiceCopyEnd+0x25
fffffa89`ac9d2ff8  00000000`00000000
```

The execution flow then returns to normal execution.

Thus, the `try/except` mechanism is implemented alongside the shadow stack mitigation.

References

1. Chong Xu Bing Sun, Jin Liu. How to Survive the Hardware-assisted Control-flow Integrity Enforcement. *Blackhat Asia*, 2019. `https://i.blackhat.com/asia-19/Thu-March-28/bh-asia-Sun-How-to-Survive-the-Hardware-Assisted-Control-Flow-Integrity-Enforcement.pdf`.
2. Adrien Chevalier. Virtualization Based Security - Part 2: kernel communications. `https://www.amossys.fr/insights/blog-technique/virtualization-based-security-part2/`, 2017.
3. Diane Dubois. Hyntrospect: a fuzzer for Hyper-V devices. *SSTIC*, 2021. `https://www.sstic.org/media/SSTIC2021/SSTIC-actes/hyntrospect_a_fuzzer_for_hyper-v_devices/SSTIC2021-Article-hyntrospect_a_fuzzer_for_hyper-v_devices-dubois.pdf`.
4. Allievi et al. *Windows Internal 7, Part 2.* Microsoft Press, 2022.
5. Yosifovich et al. *Windows Internal 7, Part 1.* Microsoft Press, 2017.
6. Intel. Control-flow Enforcement Technology Specification. `https://kib.kiev.ua/x86docs/Intel/CET/334525-003.pdf`, 2019.

7. Intel. A Technical Look at Intel's Control-flow Enforcement Technology. `https://www.intel.com/content/www/us/en/developer/articles/technical/technical-look-control-flow-enforcement-technology.html`, 2020.
8. Intel. Intel 64 and IA-32 Architectures Software Developer's Manual, Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4. `https://www.intel.fr/content/www/fr/fr/content-details/782158/intel-64-and-ia-32-architectures-software-developer-s-manual-combined-volumes-1-2a-2b-2c-2d-3a-3b-3c-3d-and-4.html`, 2024.
9. Alex Ionescu. hdk – (unofficial) Hyper-V Development Kit. `https://github.com/ionescu007/hdk`, 2020.
10. Matt Miller Ken Johnson. Exploit Mitigation Improvements in Windows 8. *Black Hat USA*, 2012. `https://media.blackhat.com/bh-us-12/Briefings/M_Miller/BH_US_12_Miller_Exploit_Mitigation_Slides.pdf`.
11. Daniel King and Shawn Denbow. Growing Hypervisor 0day with Hyperseed. *OffensiveCon*, 2019. `https://github.com/microsoft/MSRC-Security-Research/blob/master/presentations/2019_02_OffensiveCon/2019_02%20-%20OffensiveCon%20-%20Growing%20Hypervisor%200day%20with%20Hyperseed.pdf`.
12. Connor McGarr. Exploit Development: No Code Execution? No Problem! Living The Age of VBS, HVCI, and Kernel CFG. `https://connormcgarr.github.io/2022/05/23/hvci.html`, 2022.
13. Connor McGarr. Exploit Development: Investigating Kernel Mode Shadow Stacks on Windows. `https://connormcgarr.github.io/2025/02/03/km-shadow-stacks.html`, 2025.
14. Microsoft. Hypervisor Top Level Functional Specification. `https://github.com/MicrosoftDocs/Virtualization-Documentation/blob/main/tlfs/Hypervisor%20Top%20Level%20Functional%20Specification%20v6.0b.pdf`, 2020.
15. Microsoft. NtQuerySystemInformation function (winternl.h). `https://learn.microsoft.com/en-us/windows/win32/api/winternl/nf-winternl-ntquerysysteminformation`, 2021.
16. Microsoft. try-except statement. `https://learn.microsoft.com/en-us/cpp/cpp/try-except-statement`, 2021.
17. Microsoft. x64 exception handling. `https://learn.microsoft.com/en-us/cpp/build/exception-handling-x64`, 2022.
18. Microsoft. Virtualization-based Security System Resource Protections. `https://learn.microsoft.com/en-us/windows-hardware/design/device-experiences/vbs-resource-protections`, 2023.
19. Microsoft. Kernel Mode Hardware-enforced Stack Protection. `https://learn.microsoft.com/en-us/windows-server/security/kernel-mode-hardware-stack-protection`, 2024.
20. Omri Misgav. Running Rootkits Like A Nation-State Hacker. *OffensiveCon*, 2022. `https://media.defcon.org/DEF%20CON%2030/DEF%20CON%2030%20presentations/Omri%20Misgav%20-%20Running%20Rootkits%20Like%20A%20Nation-State%20Hacker.pdf`.
21. Hari Pulapaka. Understanding Hardware-enforced Stack Protection. `https://techcommunity.microsoft.com/blog/windowsosplatform/understanding-hardware-enforced-stack-protection/1247815`, 2020.

22. Yarden Shafir. Your Mitigations Are My Opportunities. *OffensiveCon*, 2023. https://www.youtube.com/watch?v=YnxGW8Fvqvk.
23. Joe Bialek (MSCR Vulnerabilites & Mitigations Team). The Evolution Of CFI Attacks And Defenses. *OffensiveCon*, 2018. https://github.com/Microsoft/MSRC-Security-Research/blob/master/presentations/2018_02_OffensiveCon/The%20Evolution%20of%20CFI%20Attacks%20and%20Defenses.pdf.
24. Alex Ionescu Yarden Shafir. R.I.P ROP: CET Internals in Windows 20H1. https://windows-internals.com/cet-on-windows/, 2020.

A `SYSTEM_INFORMATION_CLASS` 0xDD

Using *system call* `NtQuerySystemInformation` with `SYSTEM_INFORMATION_CLASS` value set to 0xDD, the following globals variables from *ntoskrnl* can be retreived by a userland program:
- KiCetCapable
- KiUserCetAllowed
- KiKernelCetEnabled
- KiKernelCetAuditModeEnabled

Listing 12: Querying shadow stack status using `NtQuerySystemInformation`

```c
#include <Windows.h>
#include <winternl.h>
#include <cstdio>

int main()
{
    ULONG SystemInformation = 0;
    ULONG ReturnLength = 0;
    NTSTATUS status;

    status = NtQuerySystemInformation((SYSTEM_INFORMATION_CLASS) 0xDD,
↪   &SystemInformation, 4, &ReturnLength);

    if (NT_SUCCESS(status))
    {
        printf("KiCetCapable = %d\n",
          (SystemInformation & 1));
        printf("KiUserCetAllowed = %d\n",
          (SystemInformation >> 1) & 1);
        printf("KiKernelCetEnabled = %d\n",
          (SystemInformation >> 8) & 1);
        printf("KiKernelCetAuditModeEnabled = %d\n",
          (SystemInformation >> 9) & 1);
    }
}
```

At the time of writing, this operation (`SYSTEM_INFORMATION_CLASS` value 0xDD) is not documented by Microsoft [15].