SYNACKTIV

Analyzing the Windows kernel shadow stack mitigation

June 5, 2025

Introduction

About us

SYNACKTIV

Rémi Jullian 𝕏 @netsecurity1

Alexandre Aulnette

- Working at @Synacktiv in the reverse engineering team
- Company specialized in offensive security: **penetration testing**, **reverse engineering**, software development, trainings, etc.)
- Around **190 experts** over 6 offices in France (Paris, Lyon, Rennes, Toulouse, Lille and Bordeaux)

Agenda

SYNACKTIV

- Windows Kernel 101
- Shadow stack introduction
- Windows kernel shadow stack implementation
- The usage of virtualization
- POC

Introduction

Windows kernel 101

SYNACKTIV

- The Windows kernel is a prime target for attackers
- It exposes a large attack surface for both LPE or RCE
- Numerous kernel vulnerabilities have been exploited in recent years
  - Pwn2Own
  - "In the wild" exploits
- Microsoft is adding various mitigations to limit the exploitation of vulnerabilities

Introduction

Exploit 101

SYNACKTIV

Typical scenario to achieve arbitrary code execution:

- Allocate a RWX (Read-Write-Execute) memory page
- Copy a shellcode
- Redirect the execution flow to the shellcode

Introduction

Windows kernel mitigation 101

SYNACKTIV

- Windows 8: Introduction of the *POOL_TYPE* `NonPagedPoolNx`
  - Pages allocated via `ExAllocatePool` with this *POOL_TYPE* are no longer executable
- Windows 10: Introduction of virtualization-based mitigations
  - HVCI: Hypervisor-Based Code Integrity
  - The hypervisor enforces the W⊕X property for pages allocated by the kernel
  - Arbitrary code execution from a `NonPagePool` page is no longer possible
  - Kernel Mode Code Integrity (KMCI)
- A lot of other kernel mitigations have been implemented (*KASLR*, *SMEP*, ...)

With an arbitrary read / write, it is still possible to achieve arbitrary code execution using ROP gadgets 😈

Shadow stack

Shadow stack

Introduction

SYNACKTIV

- Hardware-based mitigation using Intel *CET* (*Control-flow Enforcement Technology*)
- Designed by Microsoft, Intel, and other stakeholders
- Goal: Prevent control-flow hijacking by overwriting return addresses on the stack
- Uses a second "parallel" stack called the *Shadow Stack*
- Effective both in user mode (CPL 3) and kernel mode (CPL 0)
- Usermode shadow stack is a per-process mitigation (`/CETCOMPAT` [1])
- Kernel mode shadow stack configuration is global

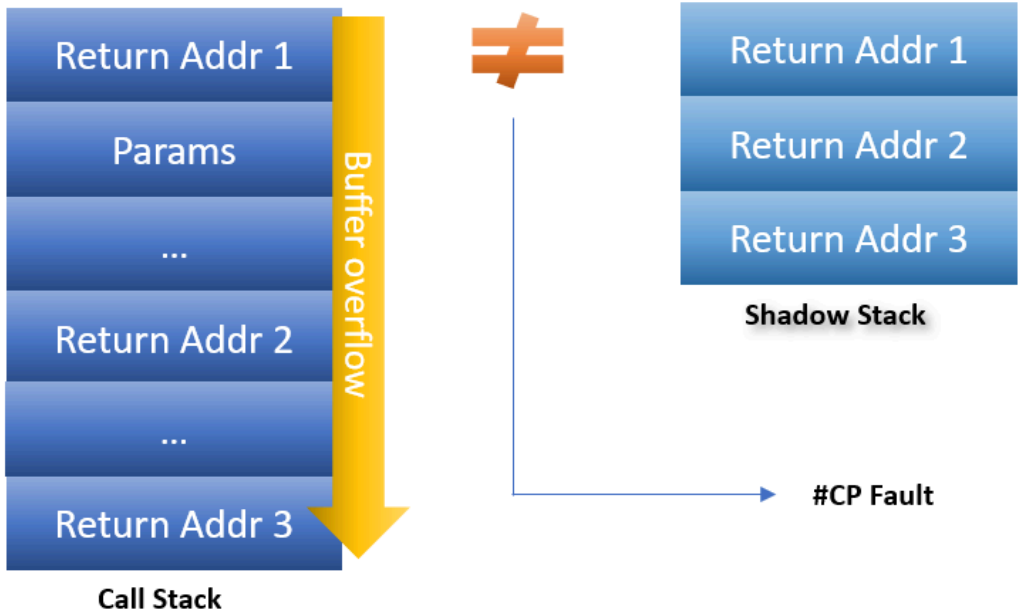
https://learn.microsoft.com/en-us/cpp/build/reference/cetcompat

Shadow stack

SYNACKTIV

Return address validation

- `call` or `ret` instructions modify both the stack and shadow stack
- `ret` may generate a *#CP* Fault generation if mismatch between the top return address




```
// Snippet from Intel Manual
// RET — Return From Procedure
...
RIP := Pop();
IF ShadowStackEnabled(CPL)
    tempSsEIP = ShadowStackPop8B();
    IF RIP != tempSsEIP
        THEN #CP(NEAR_RET); FI;
FI;
...
```

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

Shadow stack

CPU implementation

SYNACKTIV

- Intel manual[1] is useful for understanding shadow stack's CPU implementation
- The `cpuid` instruction allows to query *Structured Extended Feature Flags*
  - Bit 07 `CET_SS` in ECX -> CPU supports CET shadow stack features if 1
- Few MSR (Model Specific Register) have been added

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)

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

Shadow stack

CPU implementation

SYNACKTIV

- Intel CET is the combination of 2 hardware based mitigations
  - Shadow Stack
  - Indirect Branch Tracking (not yet supported in Windows)

Register Address: 6A0H, 1696	IA32_U_CET	
Configure User Mode CET (R/W)		Bits 1:0 are defined if CPUID.(EAX=07H, ECX=0H):ECX.CET_SS[07] = 1. Bits 5:2 and bits 63:10 are defined if CPUID.(EAX=07H, ECX=0H):EDX.CET_IBT[20] = 1.
0	SH_STK_EN: When set to 1, enable shadow stacks at CPL3.	
1	WR_SHSTK_EN: When set to 1, enables the WRSSD/WRSSQ instructions.	

- Only bits `0` and `1` from `IA32_U_CET` / `IA32_S_CET` are related to the shadow stack

Shadow stack

CPU implementation

SYNACKTIV

- Dedicated CPU instructions have been added to implement shadow stack support

Instruction	Description
CLRSSBSY	Clear busy bit in a supervisor shadow stack token
INCSSP	Increment the shadow stack pointer (SSP)
RDSSP	Read shadow stack pointer (SSP)
RSTORSSP	Restore a shadow stack pointer (SSP)
SAVEPREVSSP	Save previous shadow stack pointer (SSP)
SETSSBSY	Set busy bit in a supervisor shadow stack token
WRSS	Write to a shadow stack[1]
WRUSS	Write to a user mode shadow stack[1]

- Few instructions like `call`, `wrss` or `wruss` can write to the (read-only) shadow stack

[1] Only if bit `WR_SHSTK_EN` is set in `IA32_U_CET` / `IA32_S_CET`

Windows implementation

Windows implementation

Configuration

SYNACKTIV

- Kernel shadow stack protection is not (yet?) enabled by default (Windows 11 24H2)
- Can be enabled from *Core Isolation* menu
- Can also be enabled from the registry
- "Regular" mode vs Audit mode

```
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
# Only if audit mode is desired
reg add HKLM\SYSTEM\CurrentControlSet\Control\DeviceGuard\Scenarios\AuditModeEnabled /v Enabled /t REG_DWORD /d 1 /f
```

- Registry entries are read by Windows Boot Loader (*winload.exe*)

Windows implementation

Overview

SYNACKTIV

As the shadow stack is a *hardware-based* mitigation it needs supports from the operating system

The kernel is responsible for various tasks such as:

- Initializing the #CP exception handler
- Checking if the mitigation is supported by the CPU
- Enabling the mitigation at the CPU level
- Allocating a per-thread shadow stack
- Protecting the shadow stack against a write operation implemented in software
- Switching shadow stack when a thread context switch occurs
- Handling a #CP fault

Windows implementation

SYNACKTIV

Covered in this talk

As the shadow stack is a *hardware-based* mitigation it needs supports from the operating system

The kernel is responsible for various tasks such as:

- ~~Initializing the #CP exception handler~~
- Checking if the mitigation is supported by the CPU
- Enabling the mitigation at the CPU level
- Allocating a per-thread shadow stack
- Protecting the shadow stack against a write operation implemented in software
- ~~Switching shadow stack when a thread context switch occurs~~
- Handling a #CP fault

Windows implementation

Shadow stack support detection

SYNACKTIV

`nt!KiSetControlEnforcement` is called during kernel initialization

1. Checks if the CPU vendor is either `Intel` or `AMD`
2. Executes `cpuid` to check for shadow stack support (`CET_SS`)
3. Set the global variable `nt!KiCetCapable`
4. Set the bit 23 of `CR4` register (`CR4.CET`)

Windows implementation

Kernel shadow stack activation

SYNACKTIV

`nt!KiInitializeKernelShadowStacks` is called during system startup.

1. Check if the bit `CR4.CET` is set
2. Set global variables according to registry values
   - `nt!KiKernelCetEnabled`
   - `nt!KiKernelCetAuditModeEnabled`
3. Configure MSR `IA32_S_CET`
   - Bit `SH_STK_EN` is set (enable SS at CPL 0)
   - Bit `WR_SHSTK_EN` is set **only** in audit mode (allows `wrss` instructions)

From structure `LOADER_PARAMETER_BLOCK.Extension` of type `_LOADER_PARAMETER_EXTENSION`

Windows implementation

Kernel shadow stack allocation

SYNACKTIV

`nt!KeInitThread` is responsible for initializing newly created threads.

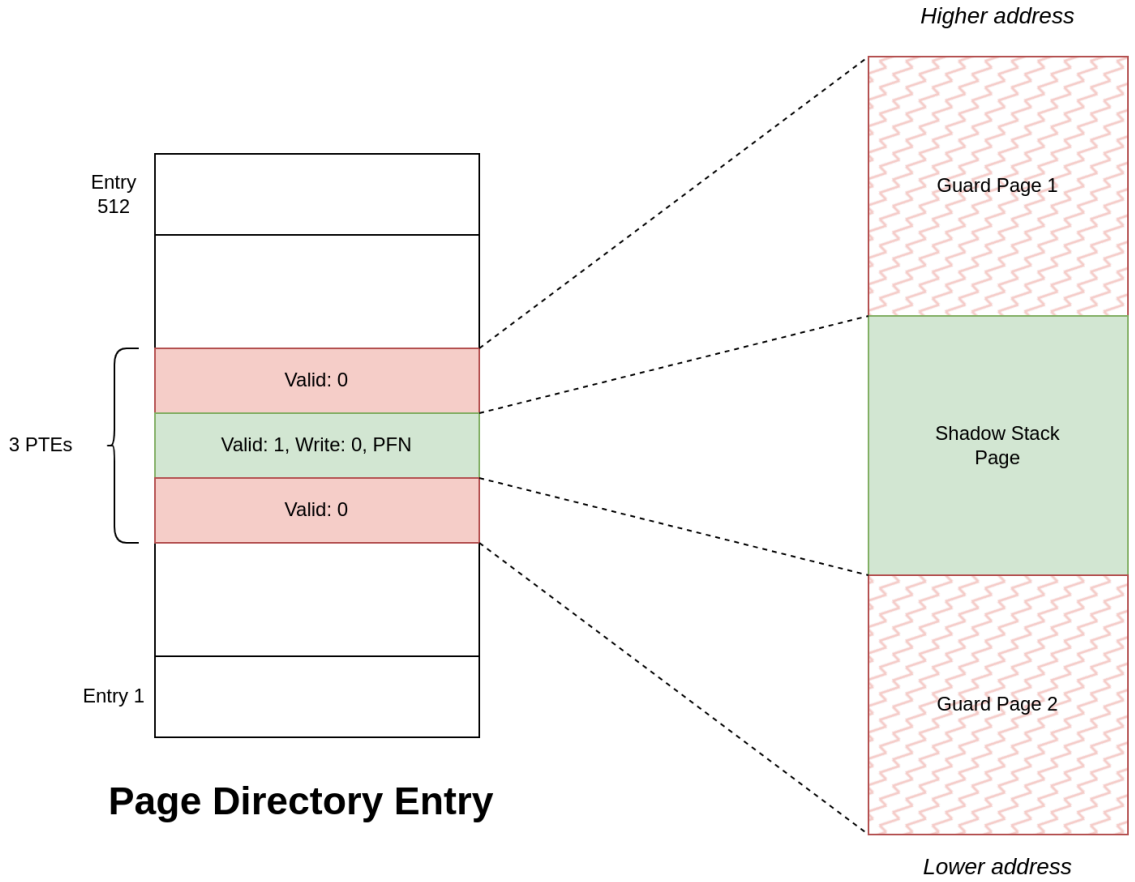
- Check if `nt!KiKernelCetEnabled` is set
- Call `nt!KiCreateKernelShadowStack`
  - 3 PTEs are reserved (`nt!MiReservePtes`)
  - Only one page is allocated (*commited*)

```
# Dumping the 3 PTEs when MiReservePtes returns
kd> dq ffffd0d2c5703eb8 L3
ffffd0d2c5703eb8 0000000000000000 8a000000041ff161
ffffd0d2c5703ec8 0000000000000000
```

Windows implementation

Kernel shadow stack allocation

- A virtual address space of 0x3000 bytes is reserved
- The shadow stack is marked as read-only in the PTE



Windows implementation

Kernel shadow stack protection

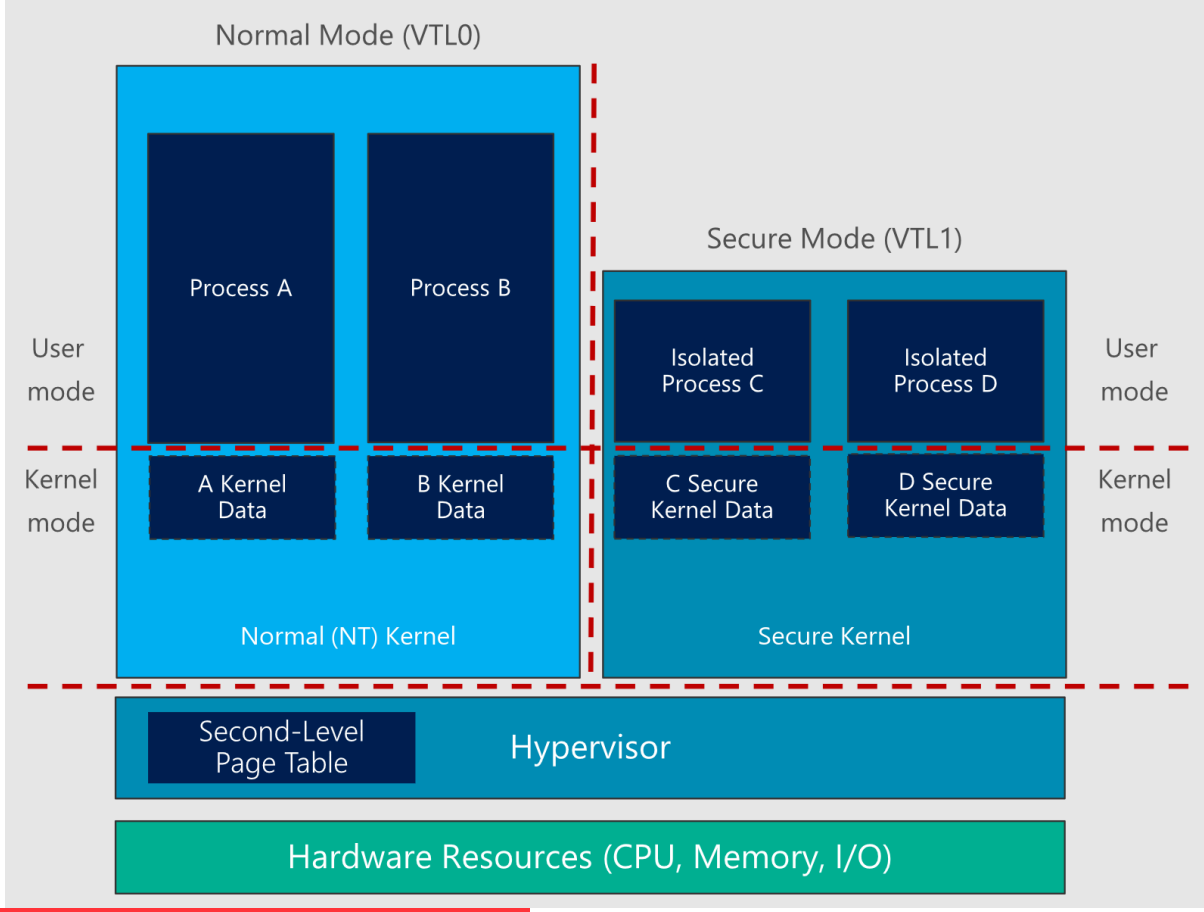
SYNACKTIV

- The shadow stack integrity must be protected from an attacker with a R/W primitive
- The protection is implemented using a virtualization-based mechanism
- Prevent a simple bypass like:
  1. Flip the write bit in the PTE -> Shadow stack page is writable
  2. Write ROP gadgets in the shadow stack page
  3. Trigger ROP chain execution

Windows implementation

Kernel shadow stack protection

- Hypervisor provides isolation between normal kernel and secure-kernel
- Both executes in the same *root partition* but with different VTL (Virtual Trust Level)



Windows implementation

Kernel shadow stack protection

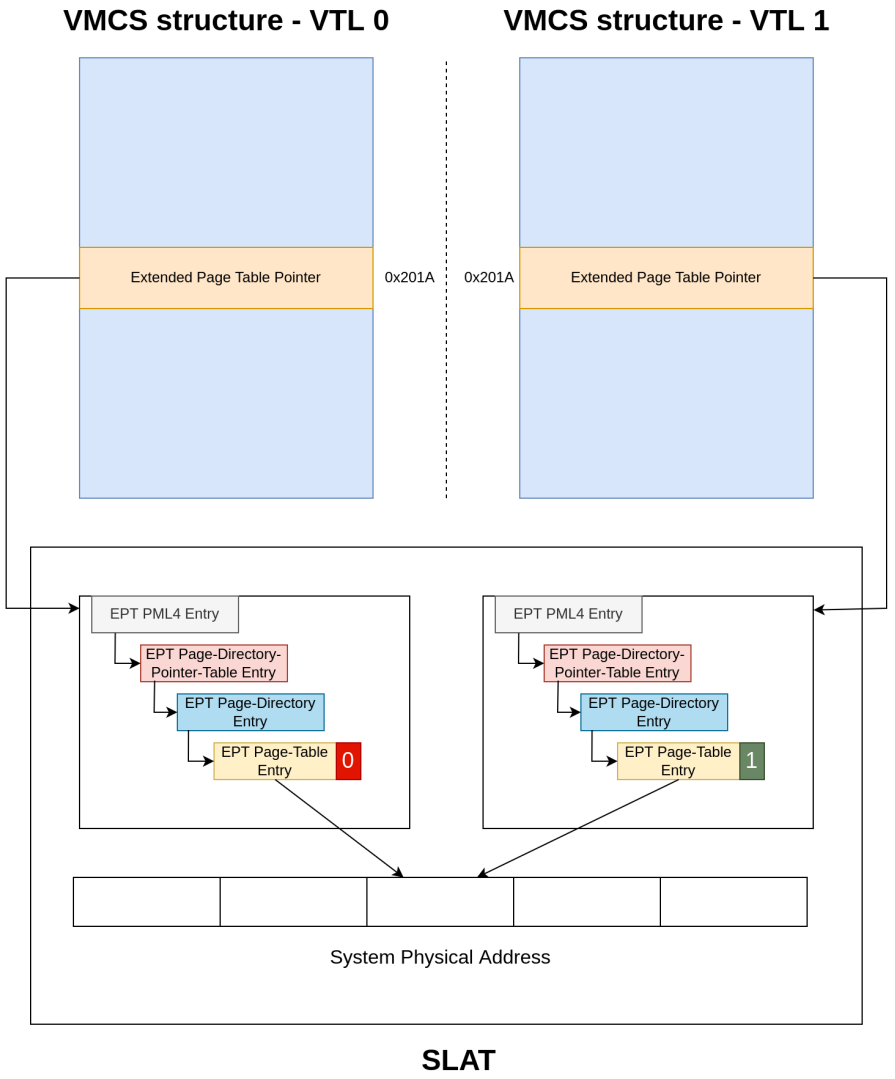
SYNACKTIV

- Shadow stack is protected by a mechanism similar to HVCI
- Second-Level-Page-Table allows to implement EPT (*Extended Page Tables*) managed by the Hypervisor
- EPT are used to convert GPA (Guest-Physical address) to SPA (System Physical address)
- EPTE (Extended Page Tables Entry) format is roughly similar to PTE format
  - Bits 0, 1, 2 are used to specify access rights on a physical page (Read, Write, Execute)
  - Bits 12 -> 51 (40 bits) are used to specify the SPA
  - Bit 60 indicates a supervisor shadow stack

Windows implementation

Kernel shadow stack protection

- VMCS (Virtual Machine Control Structure) is a hardware-defined data structure used by Intel VT-x
- Regular kernel (VTL 0) and secure kernel (VTL 1) have their own *VMCS* structure
- But are in the same *root partition*
- A GPA in both VTL 0 and VTL 1 after SLAT gives the same SPA
- Allows to implement different permissions in EPTEs referencing the same SPA



Windows implementation

Kernel shadow stack protection

SYNACKTIV

- The regular kernel **can't modify** EPTs
- But can ask the secure kernel to enforce read-only permissions in EPTs
- The regular kernel can issue a *secure call* to the secure kernel
  - `vmcall` instruction with `rcx = 0x11`
  - This causes a *VMEXIT* in the *hypervisor*
  - The hypervisor dispatches the secure call to the secure kernel
- The secure call is performed by `nt!VslAllocateKernelShadowStack`
- Ends up in `securekernel!SkmmCreateNtKernelShadowStack`

Windows implementation

Kernel shadow stack protection

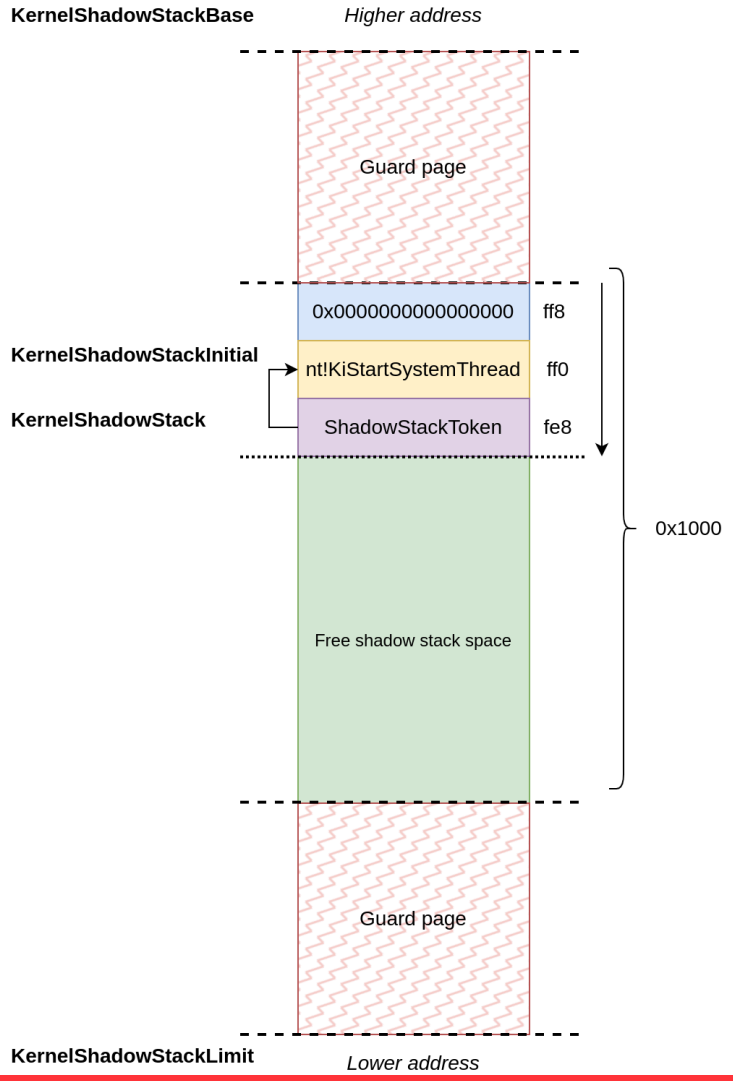
SYNACKTIV

- The secure kernel also can't directly modify EPTs but can request the hypervisor
- `securekernel!ShvlpProtectPages` perform an *hypercall* 0xC
  - `vmcall` instruction with `rcx = 0xc`
  - This causes a *VMEXIT* in the *hypervisor*
  - The hypervisor updates the EPTE
- *Hypercall* 0xC is mentioned as `HvModifyVtlProtectionMask` in Microsoft hypervisor's TLFS[1]

Hypervisor Top Level Functional Specification - https://github.com/MicrosoftDocs/Virtualization-Documentation/blob/main/tlfs/Hypervisor%20Top%20Level%20Functional%20Specification%20v6.0b.pdf

Windows implementation

Kernel shadow stack initialization



- New fields in Windows 11 21H2

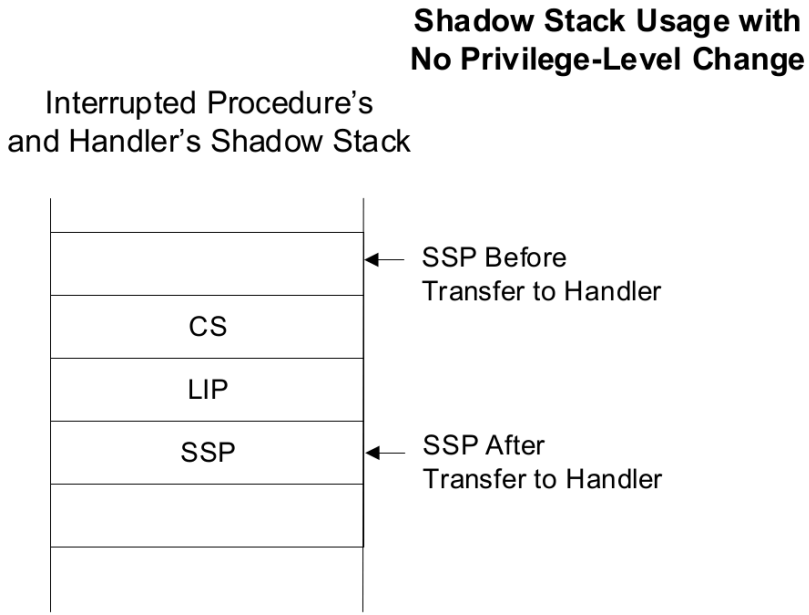
```
struct _KTHREAD {

  /* ... */

  void *KernelShadowStack;
  void *KernelShadowStackInitial;
  void *KernelShadowStackBase;
  _KERNEL_SHADOW_STACK_LIMIT KernelShadowStackLimit;

  /* ... */

};
```

Windows implementation

SYNACKTIV

Fault handling

- `nt!KiControlProtectionFault` handles a shadow stack related fault (#CP)
- Fault can be generated from a userland process (CPL 3) or kernel thread (CPL 0)
- If generated by a kernel thread there is no privilege level change

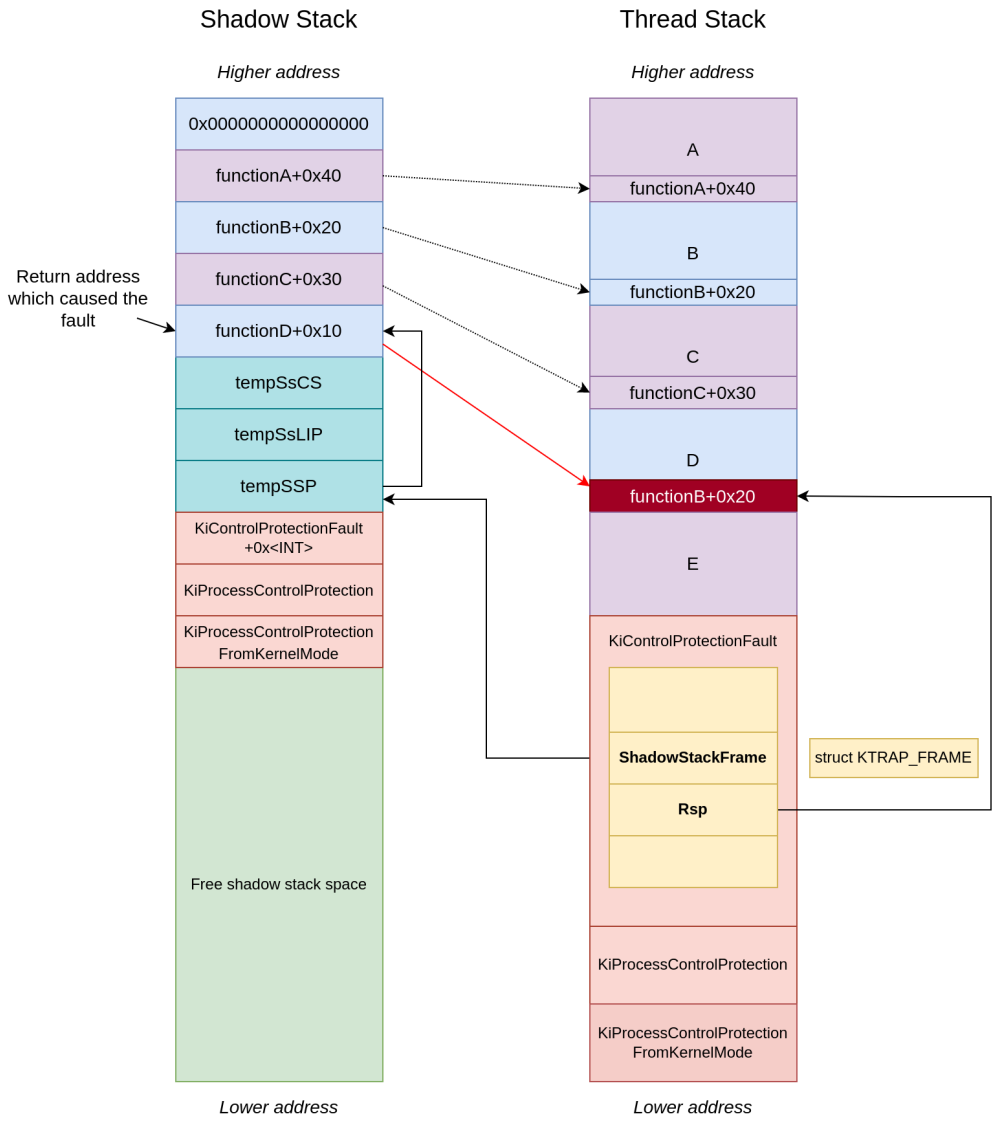


- *tempSsCS*: Value of the code segment
- *tempSsLIP*: Value of the instruction pointer when the fault was generated
- *tempSSP*: Value of the SSP when the fault was generated

Windows implementation

Fault handling (in kernel)

- Handler loops through shadow stack addresses to find if one matches the faulty one (bottom to top)



Windows implementation

SYNACKTIV

Fault handling (Non audit mode)

- If a valid return address is found (any position in shadow stack) -> No BSOD 😎
- The handler implementation is quite permissive!
- The shadow stack is patched by the secure kernel
  - Regular kernel (VTL 0) still can't write to the shadow stack
  - Secure call `nt!VslKernelShadowStackAssist`
  - Fixup `tempSSP` to point on return address matching the one at `RSP`
  - When the interrupt handler exits (`iretq`) the value at the top the stack and shadow stack matches
- If no valid return address is found -> BSOD 😱

Windows implementation

Fault handling (Non audit mode)

SYNACKTIV

```
*******************************************************************************
*                                                                             *
*                        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
```

Windows implementation

Fault handling (audit mode)

SYNACKTIV

- In audit mode even if no valid return address is found in the shadow stack -> No BSOD
- Shadow stack is fixed by

  `nt!KiFixupControlProtectionKernelModeReturnMismatch`

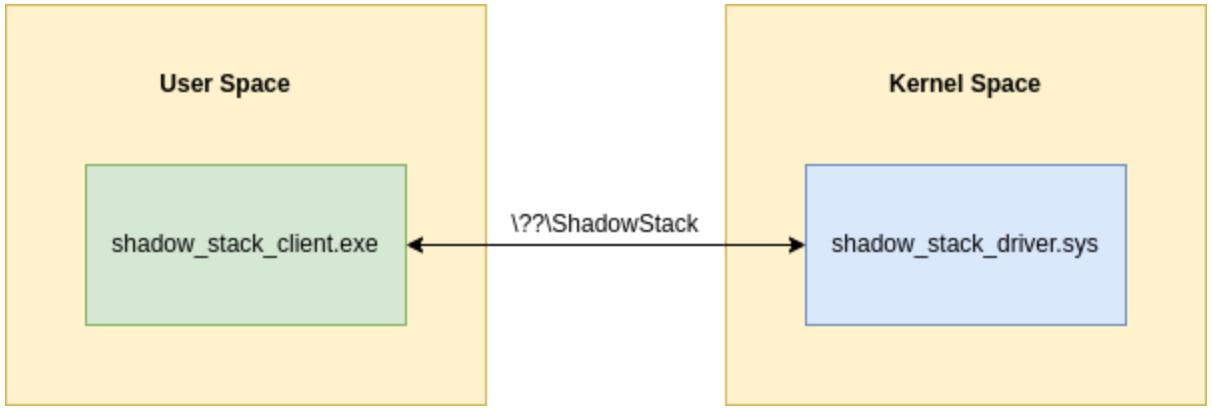
  - Write to shadow stack using `wrssq` instructions (regular kernel)
  - Only works because `WR_SHSTK_EN` is set in `IA32_S_CET` MSR
  - Would not work in non audit mode!
- An ETW (Event Tracing for Windows) log is generated

POC

POC

Architecture

- Github: https://github.com/synacktiv/windows_kernel_shadow_stack
- Userland client (shadow_stack_client.exe)
- Kernel driver (shadow_stack_driver.sys)
- Communication through IOCTL mechanism



POC

Summary

SYNACKTIV

- Writing to the MSR registers
- Incrementing the return address
- Skipping a stack frame
- Try/Except path

POC

Writing to the MSR registers

SYNACKTIV

- Rewrite the `IA32_S_CET` register (0x06A2)
- Reminders
  - Bit 0: Shadow Stack Enabled (`SH_STK_EN`)
  - Bit 1: Write Shadow Stack Enabled (`WR_SHSTK_EN`)
- Target in debug state, so both are set
- `STATUS_PRIVILEGED_INSTRUCTION` (0xC0000096)

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

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

```
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
```

POC

Writing to the MSR registers

- Running at `CPL0`
- So, `wrmsr` instruction is allowed
- **Hyper-V** is watching for MSR registers
  - Accesses
  - Content updates
- Then trigger a general protection fault

```
 # 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
```

```
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
```

POC

Incrementing the return address

- Caller

```
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
```

- Return to `lea ecx, ADDR` instead of `lea rcx, ADDR`
- Still a valid instruction

- Callee

```
#define RET_ADDR_OFF (3)

void IncRetAddr()
{
  uintptr_t* placeholder = NULL;

  DbgPrintEnter();

  placeholder = (uintptr_t*)&placeholder;

  DbgPrint("Legitimate return address:  %p\n",
                                  placeholder[RET_ADDR_OFF]);
  placeholder[RET_ADDR_OFF]++;
  DbgPrint("Incremented return address: %p\n",
                                  placeholder[RET_ADDR_OFF]);

  DbgPrintLeave();
}
```

- **But...**

POC

Incrementing the return address

SYNACKTIV

- … an exception occurs as intended

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

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

- Shadow stack violation (0x39)

```
*******************************************************************************
*                                                                             *
*                        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
```

POC

Incrementing the return address

SYNACKTIV

- Triggered by the kernel through a control protection fault

```
INITDATA:0000000000D81438        dq 14h
INITDATA:0000000000D81440        dq offset KiVirtualizationException
INITDATA:0000000000D81448        dq offset KiVirtualizationExceptionShadow
INITDATA:0000000000D81450        dq 15h
INITDATA:0000000000D81458        dq offset KiControlProtectionFault
INITDATA:0000000000D81460        dq offset KiControlProtectionFaultShadow
INITDATA:0000000000D81468        dq 1Fh
INITDATA:0000000000D81470        dq offset KiApcInterrupt
INITDATA:0000000000D81478        dq offset KiApcInterruptShadow
```

```
 # Call Site
00 nt!DbgBreakPointWithStatus
01 nt!KiBugCheckDebugBreak+0x12
02 nt!KeBugCheck2+0xb2c
03 nt!KeBugCheckEx+0x107
04 nt!KiBugCheckDispatch+0x69
05 nt!KiFastFailDispatch+0xb2
06 nt!KiControlProtectionFault+0x3df
07 shadow_stack_driver+0x2787
08 shadow_stack_driver+0x16e6
09 shadow_stack_driver+0x11a4
0a nt!IofCallDriver+0xbe
0b nt!IopSynchronousServiceTail+0x1c8
0c nt!IopXxxControlFile+0x940
0d nt!NtDeviceIoControlFile+0x5e
0e nt!KiSystemServiceCopyEnd+0x25
```

POC

Skipping a stack frame

SYNACKTIV

- `SkipNextFrame` called by `IoctlSkipNextFrame`
- Look for a valid return address location
- Update `rsp` with the caller stack frame
- Go through `setRsp` and `IoctlSkipNextFrame` epilogs

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

```
void SkipNextFrame()
{
  uintptr_t  module  = 0;
  uintptr_t* new_rsp = NULL;

  DbgPrintEnter();

  module  = (uintptr_t)GetModuleBase((void*)SkipNextFrame);
  new_rsp = (uintptr_t*)&new_rsp;

  // Looks for a valid return address in our own module
  for (new_rsp += 2;; ++new_rsp)
  {
    if (MASK_KERNEL_ADDR(module) == MASK_KERNEL_ADDR(*new_rsp))
    {
      break;
    }
  }

  setRsp(new_rsp);

  DbgPrintLeave();
}
```

POC

Skipping a stack frame

SYNACKTIV

- A kind of magic under the hood
- Trigger a control protection fault
- Check the return address VS the shadow stack
- Call `nt!VslKernelShadowStackAssist`
  - Fix the shadow stack

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

```
 # Call Site
00 nt!KiProcessControlProtectionFromKernelMode
01 nt!KiProcessControlProtection+0x330
02 nt!KiControlProtectionFault+0x356
03 shadow_stack_driver+0x3313
04 shadow_stack_driver+0x1ca5
05 shadow_stack_driver+0x11cb
06 nt!IofCallDriver+0xbe
07 nt!IopSynchronousServiceTail+0x1c8
08 nt!IopXxxControlFile+0x940
09 nt!NtDeviceIoControlFile+0x5e
0a nt!KiSystemServiceCopyEnd+0x25
```

POC

Try/Except path

- Keywords are quite similar to the userland
- Update return address on exception
- Let's trigger a division by zero!

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

```
NTSTATUS IoctlDivInteger(void* Buffer, uint32_t Length)
{
  IoctlDivInteger_t* di     = NULL;
  NTSTATUS           Status = STATUS_INVALID_PARAMETER;

  DbgPrintEnter();

  [...]

  di = (IoctlDivInteger_t*)Buffer;

  try
  {
    Status = DivInteger(di->Dividend, di->Divisor);

    [...]
  }
  except(_DivZeroFilter())
  {
    DbgPrint("A division by zero occurred\n");
    Status = STATUS_INTEGER_DIVIDE_BY_ZERO;
    goto EXIT;
  }

  Status = STATUS_SUCCESS;

EXIT:
  DbgPrintLeave();

  return Status;
}
```

POC

Try/Except path

- Unwind the exception
- Fix the thread shadow stack
- Restore the thread context

- `nt!KeKernelShadowStackRestoreContext` function end

```
call    VslKernelShadowStackAssist
add     rsp, 38h
retn
```

- Call stack from `nt!KeKernelShadowStackRestoreContext`

```
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
[...]
```

Conclusion

SYNACKTIV

- The kernel shadow stack mitigation is quite effective to prevent ROP attacks
- Relying on virtualization makes this mitigation strong against an attacker with R/W primitive
- The mitigation might be enabled by default in following Windows releases
- Doesn't mean that arbitrary code execution in Windows kernel is dead
  - JOP gadgets can still be used (they do not modify the stack)
  - At least while IBT (Indirect Branch Tracking) is not supported

Questions?

SYNACKTIV

https://www.linkedin.com/company/synacktiv

https://x.com/synacktiv

https://synacktiv.com