How to Survive the Hardware-assisted Control-flow Integrity Enforcement

Bing Sun, Jin Liu, Chong Xu

McAfee

Abstract

- Control-flow hijacking is a crucial technique of modern vulnerability exploitation that converts a memory safety vulnerability into arbitrary code execution. The security industry has been striving to combat the control-flow hijacking. Since the software-only control-flow integrity solution (such as Microsoft's CFG) has been proven inadequate in defeating sophisticated control-flow hijacking attacks, hardware-assisted solutions are needed. Intel Control-flow Enforcement Technology (CET) is such a solution that aims at preventing the exploits from hijacking the control-flow transfer instructions for both forward-edge (indirect call/jmp) and back-edge transfer (ret). The latest Windows 10 RS5 has introduced some new mitigation changes to support Intel CET, which is a clear sign that Microsoft is taking serious steps to address the control-flow hijacking issue once for all. In this talk, we first give a deep dive into Intel CET and its implementation on the latest Windows 10 x64 operating system (RS5 and 19H1). We then discuss possible attacks that can still achieve the control-flow hijacking even when CET is enabled. We'll demonstrate such attack scenarios.

About the Speakers

- Bing Sun is a senior security researcher. He leads the IPS security research team of McAfee. He has extensive experiences in operating system low-level and information security technique R&D, with especially deep diving in advanced vulnerability exploitation and detection, rootkits detection, firmware security, and virtualization technology. Bing is a regular speaker at international security conference such as XCon, Black Hat, and CanSecWest.
- Jin Liu is a security researcher of Xfuture Security. Jin mainly focuses on vulnerability research. He specializes in vulnerability analysis and exploitation, particularly in browser vulnerability research on Windows platform.
- Chong Xu received his Ph.D. degree in networking and security from Duke University. His current focus includes research and innovation on intrusion and prevention techniques as well as threat intelligence. He is the head of security research the McAfee network security business unit, which leads McAfee vulnerability research, malware and APT detection, and botnet detection. Chong's team feeds security content and innovative protection solutions into McAfee's network IPS, host IPS, and sandbox products, as well as McAfee Global Threat Intelligence (GTI).

Agenda

1. Software-based vs Hardware-assisted Control-flow Integrity Enforcement
2. Intel Control-flow Enforcement Technology (CET)
3. Intel CET Implementation on Windows 10
4. Control-flow Hijacking and ACE on Windows 10 with CET enabled
5. Conclusion
6. Q&A

1. Software-based vs Hardware-assisted Control-flow Integrity Enforcement

Software-based vs Hardware-assisted Control-flow Integrity Enforcement

- Control-flow Integrity
  - A security measure to ensure the software execution stays on the path of pre-determined control flow graph.
- Software-based Control-flow Integrity Enforcement
  - What: implementing CFI enforcement in software only
  - Examples: Microsoft CFG, RFG, Google IFCC
  - Merits: faster to implement/productize, more flexible and adaptive to various application scenarios
- Hardware-assisted Control-flow Integrity Enforcement
  - What: enforcing CFI with the support of dedicated hardware (new ISA feature etc)
  - Examples: Intel CET
  - Merits: less performance degradation, more effective against attack/bypass

Software-based vs Hardware-assisted Control-flow Integrity Enforcement - Microsoft Control Flow Guard (CFG)

CFG implements **coarse-grained control-flow integrity** for indirect calls

Compile time

```
void Foo(...) {
   // SomeFunc is address-taken
   // and may be called indirectly
   Object->FuncPtr = SomeFunc;
}
```

Metadata is automatically added to the image which identifies functions that may be called indirectly

```
void Bar(...) {
   // Compiler-inserted check to
   // verify call target is valid
   _guard_check_icall(Object->FuncPtr);
   Object->FuncPtr(xyz);
}
```

A lightweight check is inserted prior to indirect calls which will verify that the call target is valid at runtime

Runtime

Process Start
- Map valid call target data

Image Load
- Update valid call target data with metadata from PE image

Indirect Call
- Perform O(1) validity check
- Terminate process if invalid target
- Jmp if target is valid

CFG is a deterministic mitigation, its security is not dependent on keeping secrets.

For C/C++ code, CFG requires no source code changes.

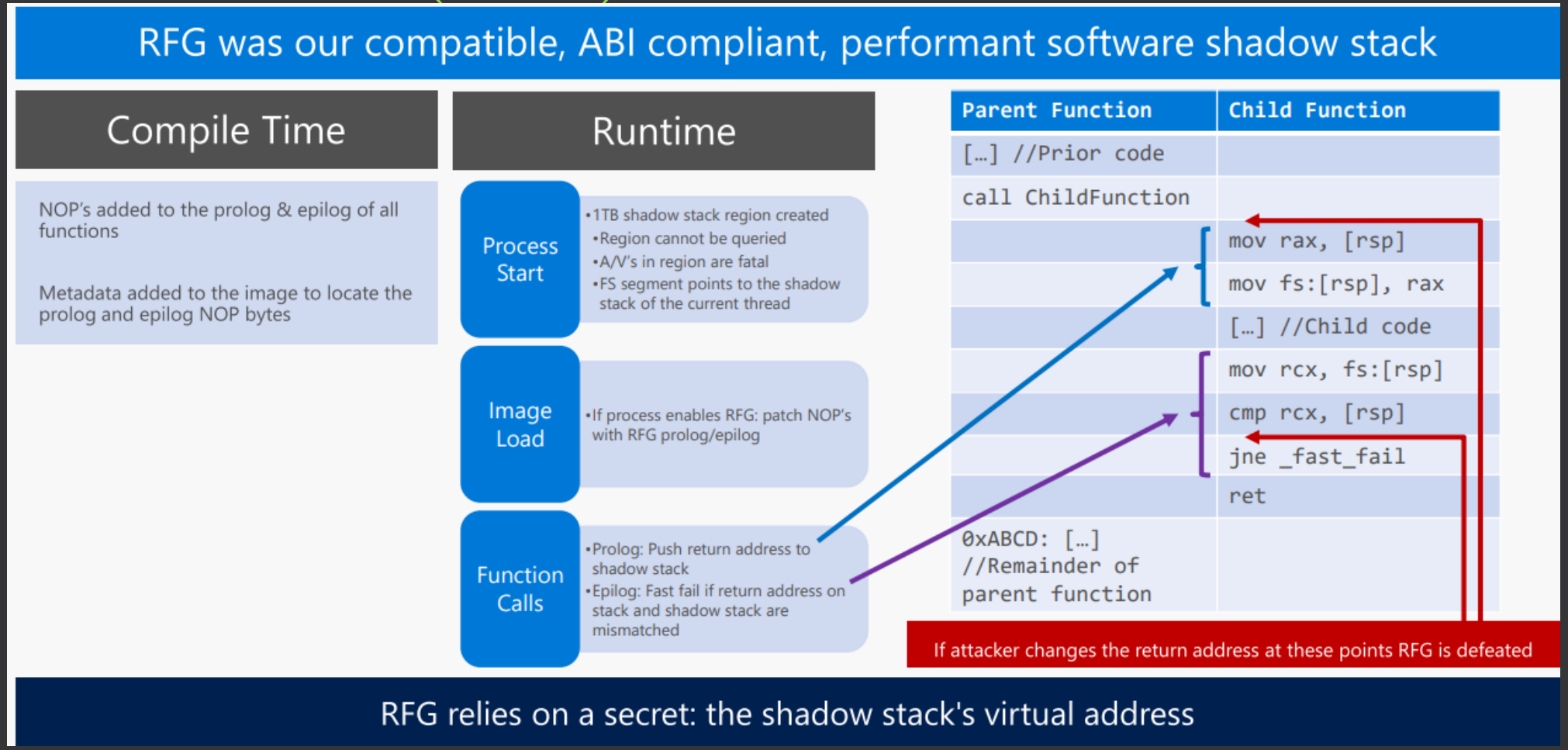
```
ntdll!LdrpDispatchUserCallTarget:
00007ffb`4e100e10 4c8b1d59e50d00  mov     r11,qword ptr
[ntdll!LdrSystemDllInitBlock+0xb0]
00007ffb`4e100e17 4c8bd0          mov     r10,rax
00007ffb`4e100e1a 49c1ea09        shr     r10,9
00007ffb`4e100e1e 4f8b1cd3        mov     r11,qword ptr [r11+r10*8]
00007ffb`4e100e22 4c8bd0          mov     r10,rax
00007ffb`4e100e25 49c1ea03        shr     r10,3
00007ffb`4e100e29 a80f            test    al,0Fh
00007ffb`4e100e2b 7509            jne     ntdll!LdrpDispatchUserCallTarget+0x26
ntdll!LdrpDispatchUserCallTarget+0x1d:
00007ffb`4e100e2d 4d0fa3d3        bt      r11,r10
00007ffb`4e100e31 7303            jae     ntdll!LdrpDispatchUserCallTarget+0x26
ntdll!LdrpDispatchUserCallTarget+0x23:
00007ffb`4e100e33 48ffe0          jmp     rax
```

Software-based vs Hardware-assisted Control-flow Integrity Enforcement - The Known Limitations of CFG

Mitigation	In scope	Out of scope
Control Flow Guard(CFG)	Techniques that make it possible to gain control of the instruction pointer through an indirect call in a process that has enabled CFG.	• Hijacking control flow viare turn address corruption • Bypasses related to limitations of coarse-grained CFI (e.g. calling functions out of context) • Leveraging non-CFG images • Bypasses that rely on modifying or corrupting read-only memory • Bypasses that rely on CONTEXT record corruption • Bypasses that rely on race conditions or exception handling • Bypasses that rely on thread suspension • Instances of missing CFG instrumentation prior to an indirect call

From Microsoft's Mitigation Bypass Bounty

Software-based vs Hardware-assisted Control-flow Integrity Enforcement - Microsoft Return Flow Guard (RFG)



From <<The Evolution of CFI Attacks and Defenses>>

Software-based vs Hardware-assisted Control-flow Integrity Enforcement - The Defects of RFG

- The reliable leakage of shadow stack address was demonstrated to be possible.
- RFG had a by-design race condition issue that was proved to be exploitable.

2. Intel Control-flow Enforcement Technology (CET)

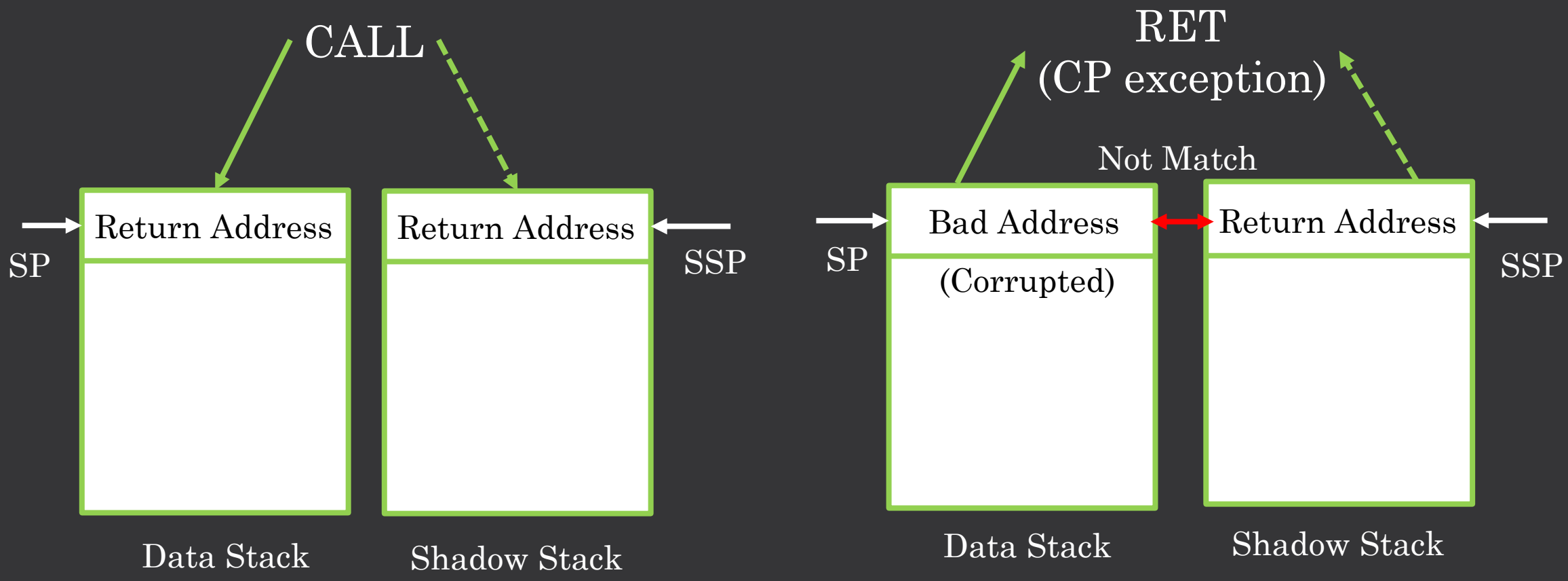
Intel Control-flow Enforcement Technology (CET)

- CET is an upcoming hardware feature of Intel® processor family targeting the control-flow hijacking attack prevention.
- CET provides two capabilities to defend against ROP/JOP style control-flow subversion attacks
  - Shadow Stack – return address protection to defend against Return Oriented Programming,
  - Indirect branch tracking – free branch protection to defend against Jump/Call Oriented Programming.

Intel Control-flow Enforcement Technology - Shadow Stack

- A shadow stack is a second stack exclusively used for control transfer operations. This second stack is separate from the data stack, and it holds only the return addresses (no parameters).
- The shadow stack is protected from being tampered through the page table protections (additional page attribute) such that regular store instructions cannot modify the contents of the shadow stack. Writes to the shadow stack are restricted to control transfer instructions and shadow stack management instructions.

Intel Control-flow Enforcement Technology - The Principle of Shadow Stack



From <<Control-flow Enforcement Technology>>

Intel Control-flow Enforcement Technology - Indirect Branch Tracking (IBT)

- The CPU implements a state machine that tracks indirect jmp and call instructions. The new ENDBRANCH instruction is used to mark valid indirect call/jmp targets in the program (NOP on legacy machines).
- “No-track” prefix (3EH) disables IBT for near indirect call/jmp instructions.
- The legacy compatibility treatment (legacy code page bitmap) disables IBT on legacy software.

Intel Control-flow Enforcement Technology - The Principle of IBT

```
main() {
 int (*f) ();
 f = test;
 f();
}
int test() {
return
}
```

Enter WAIT_FOR_END BRANCH

```
<main>:
ENDBR
…
movq $0x4004fb, -8(%rbp)
mov -8(%rbp), %rdx
call *%rdx
…
retq
<test>:
ENDBR
…
add rax, rbx
…
retq
```

Encounter ENDBR, IDLE

Hijacked

Miss ENDBR, #CP(ENDBRANCH)

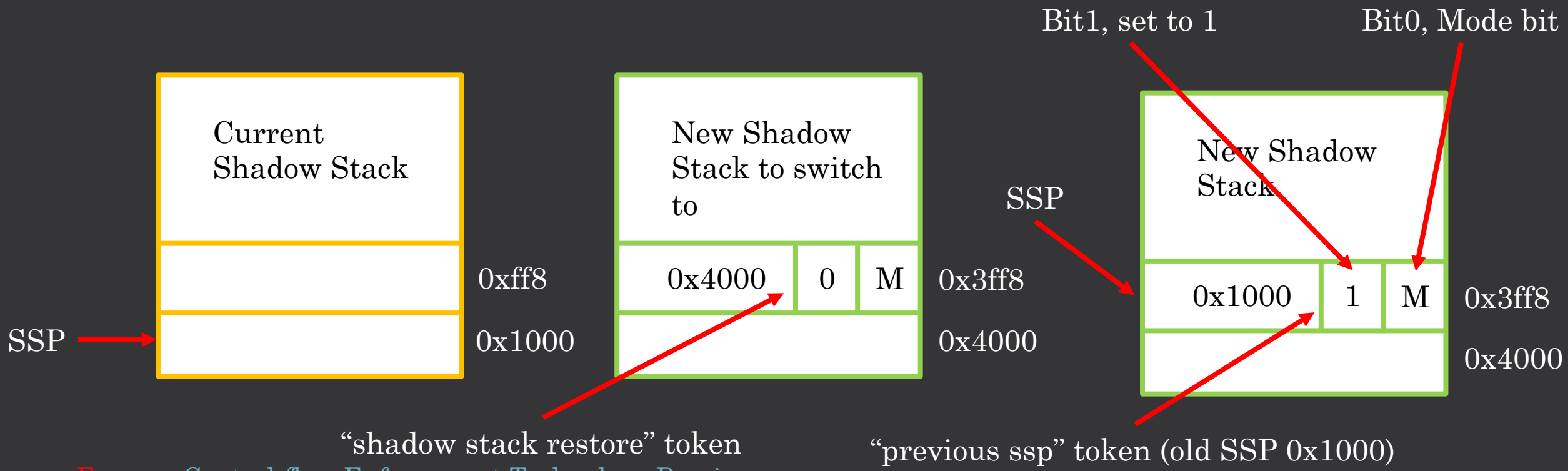
From <<Control-flow Enforcement Technology>>

Intel Control-flow Enforcement Technology - Shadow Stack Management Instructions

- INCSSP: Increment the shadow stack pointer
- RDSSP: Read the shadow stack pointer
- SAVEPREVSSP/RSTORSSP: Save the previous shadow stack pointer/ restore the saved shadow stack pointer(for shadow stack switching)
- WRSS/ WRUSS: write to the shadow stack
- SETSSBSY/CLRSSBSY: Mark the shadow stack busy/ clear the shadow stack busy flag (supervisor shadow stack token management)

Intel Control-flow Enforcement Technology - Shadow Stack Switch

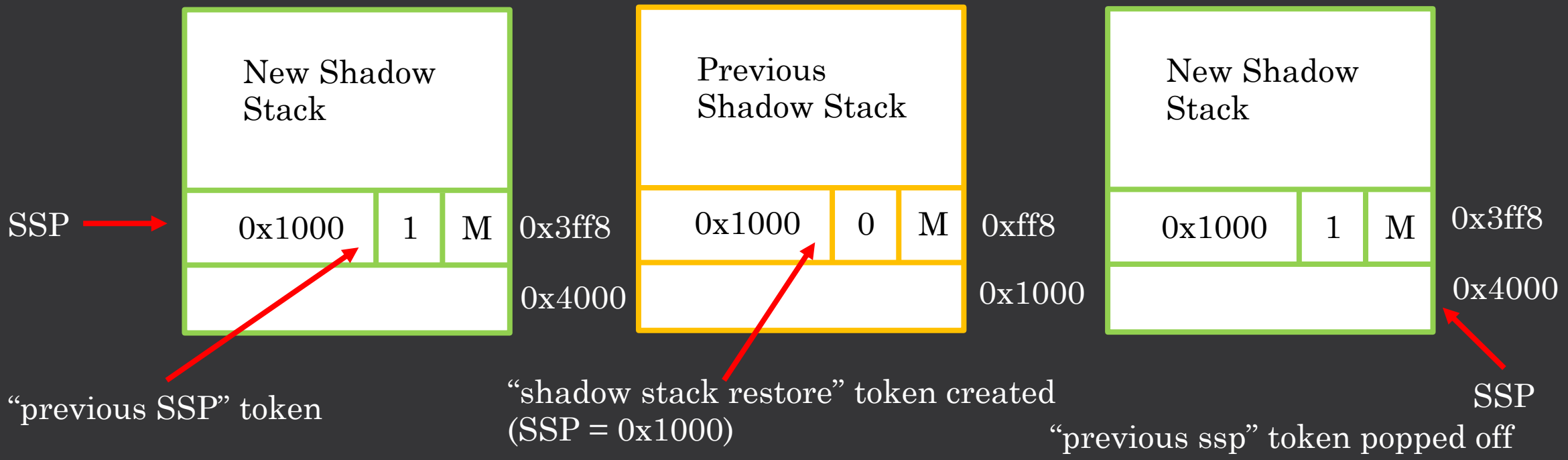
- The CET architecture provides a mechanism to switch shadow stacks using a pair of instructions; RSTORSSP and SAVEPREVSSP.
- RSTORSSP 0x3ff8



From <<Control-flow Enforcement Technology Preview >>

Intel Control-flow Enforcement Technology - Shadow Stack Switch

- SAVEPREVSSP (no operand)



From <<Control-flow Enforcement Technology Preview >>

Intel Control-flow Enforcement Technology – IBT Control Transfer Terminating Instructions

- ENDBR32
  - Terminate an indirect branch in 32 bit and compatibility mode.
- ENDBR64
  - Terminate an indirect branch in 64 bit mode.

Intel Control-flow Enforcement Technology - Control Protection Exception

- Interrupt 21 (new Control Protection Exception #CP)
  - Saved CS and EIP/RIP pointing to the violating instruction
- Exception Error Code:
  - NEAR-RET (value 1) – return addresses mismatch for a near RET instruction.
  - FAR-RET/IRET (value 2) – return addresses mismatch for a FAR RET or IRET instruction.
  - ENDBRANCH (value 3) – missing ENDBRANCH at target of an indirect call or jump instruction.
  - RSTORSSP (value 4) – token check failure in RSTORSSP instruction.
  - SETSSBSY (value 5) – token check failure in SETSSBSY instruction.

Intel Control-flow Enforcement Technology - CET Feature Enumeration

- Shadow Stack
  - If CPUID.(EAX=7, ECX=0):ECX.CET_SS[bit 7] is 1, the processor supports CET shadow stack feature
- Indirect Branch Tracking
  - If CPUID.(EAX=7, ECX=0):EDX.CET_IBT[bit 20] is 1, the processor supports CET indirect branch tracking

Intel Control-flow Enforcement Technology - CET Control Bit and MSRs

- Master enable
  - CR4.CET (bit 23)
- CET MSRs
  - IA32_U_CET (0x6a0): user mode CET configuration
  - IA32_S_CET (0x6a2): supervisor mode CET configuration
  - IA32_PL3_SSP (0x6a7): linear address of Ring3 shadow stack
  - IA32_PL2_SSP (0x6a6): linear address of Ring2 shadow stack
  - IA32_PL1_SSP (0x6a5): linear address of Ring1 shadow stack
  - IA32_PL0_SSP (0x6a4): linear address of Ring0 shadow stack
  - IA32_INTERRUPT_SSP_TABLE_ADDR (0x6a8): linear address of a table of 7 shadow stack pointers (IST)

Intel Control-flow Enforcement Technology - CET Extended State Management

- CET defines two sets of supervisory state that can be saved and restored with XSAVES/XRSTORS
- CET XState control bits
  - The CET_U: IA32_XSS.CET_U[bit 11]
  - The CET_S: IA32_XSS.CET_S[bit 12]
- CET XState feature enumeration
  - CPUID.(EAX=0DH, ECX=1): EBX – reports additional bytes for CET states
  - CPUID.(EAX = 0DH, ECX = 11): EAX – 16 bytes
  - CPUID.(EAX = 0DH, ECX = 12): EAX – 24 bytes
- CET XState buffer format
  - The CET_U state buffer:
    - Offset 0: IA32_U_CET
    - Offset 8: IA32_PL3_SSP
  - The CET_S state buffer:
    - Offset 0: IA32_PL0_SSP
    - Offset 8 : IA32_PL1_SSP
    - Offset 16: IA32_PL2_SSP

Intel Control-flow Enforcement Technology - Shadow Stack Paging

- Shadow Stack page attributes
  - The logical-AND of the R/W flags in the non-leaf paging structure entries is 1, and in the leaf paging structure entry has R/W flag set to 0 and the dirty flag is 1.
- Shadow Stack related page faults
  - Shadow stack page entry is not writeable (W=0) (enclave mode = 1)
  - Shadow stack page entry is writeable (W=1) or not dirty (D=0) (enclave mode = 0)
  - Shadow stack page entry is not writeable (W=0) in any non-leaf paging structure (enclave mode = 0)
  - Shadow stack page entry has user privilege (U=1) for a supervisor mode shadow stack access (except WRUSS)
- Shadow Stack related bit in page fault error code
  - SS flag (bit 6): This flag is 1 if (1) CR4.CET = 1; (2) the access causing the page-fault exception was a shadow-stack data access.

3. Intel CET Implementation on Windows 10

CET Implementation on Windows 10

- The latest Windows 10 insider preview (19H1) doesn't support IBT
- Changes to the following parts of operating system to support user-mode Shadow Stack:
  - Thread creation/termination
  - Fiber creation/deletion
  - NtContinue and get/set thread context (KeVerifyContextXStateCetU)
  - Exception unwinder (RtlpPopUserShadowStack)
  - Control protection fault handling (KiProcessControlProtection)
  - Page fault handling (MmAccessFault)
  - User mode call back (KeUserModeCallback)
  - …

Intel CET Implementation on Windows 10 - New Flag Added in EPROCESS

```
kd> dt nt!_EPROCESS MitigationFlags2.
   +0x82c MitigationFlags2  : Uint4B
   +0x82c MitigationFlags2Values :
      +0x000 EnableExportAddressFilter : Pos 0, 1 Bit
      +0x000 AuditExportAddressFilter : Pos 1, 1 Bit
      +0x000 EnableExportAddressFilterPlus : Pos 2, 1 Bit
      +0x000 AuditExportAddressFilterPlus : Pos 3, 1 Bit
      +0x000 EnableRopStackPivot : Pos 4, 1 Bit
      +0x000 AuditRopStackPivot : Pos 5, 1 Bit
      +0x000 EnableRopCallerCheck : Pos 6, 1 Bit
      +0x000 AuditRopCallerCheck : Pos 7, 1 Bit
      +0x000 EnableRopSimExec  : Pos 8, 1 Bit
      +0x000 AuditRopSimExec   : Pos 9, 1 Bit
      +0x000 EnableImportAddressFilter : Pos 10, 1 Bit
      +0x000 AuditImportAddressFilter : Pos 11, 1 Bit
      +0x000 DisablePageCombine : Pos 12, 1 Bit
      +0x000 SpeculativeStoreBypassDisable : Pos 13, 1 Bit
      +0x000 CetShadowStacks   : Pos 14, 1 Bit
```

Intel CET Implementation on Windows 10 - New Flags Added in KTHREAD

```
kd> dt nt!_KTHREAD
   …
   +0x074 UserStackWalkActive : Pos 5, 1 Bit
   +0x074 ApcInterruptRequest : Pos 6, 1 Bit
   +0x074 QuantumEndMigrate : Pos 7, 1 Bit
   +0x074 UmsDirectedSwitchEnable : Pos 8, 1 Bit
   +0x074 TimerActive      : Pos 9, 1 Bit
   +0x074 SystemThread     : Pos 10, 1 Bit
   +0x074 ProcessDetachActive : Pos 11, 1 Bit
   +0x074 CalloutActive    : Pos 12, 1 Bit
   +0x074 ScbReadyQueue    : Pos 13, 1 Bit
   +0x074 ApcQueueable     : Pos 14, 1 Bit
   +0x074 ReservedStackInUse : Pos 15, 1 Bit
   +0x074 UmsPerformingSyscall : Pos 16, 1 Bit
   +0x074 TimerSuspended   : Pos 17, 1 Bit
   +0x074 SuspendedWaitMode : Pos 18, 1 Bit
   +0x074 SuspendSchedulerApcWait : Pos 19, 1 Bit
   +0x074 CetShadowStack   : Pos 20, 1 Bit
```

Intel CET Implementation on Windows 10 - Thread Creation (NtCreateThreadEx)

- The logics related to the Shadow Stack allocation and setup
  1. When EPROCESS.MitigationFlags2.CetShadowStacks flag is on, nt!NtCreateThreadEx creates an extended Context structure that contains CET state (ContextFlags |= CONTEXT_XSTATE) for the new thread.
  2. When EPROCESS.MitigationFlags2.CetShadowStacks flag is on, nt!PspAllocateThread sets KTHREAD.CetShadowStack of the new thread to 1.
  3. If KTHREAD.CetShadowStack flag is on, nt!KiInitializeContextThread calls nt!KiSetSwitchingNpxState turns on CET state in KTHREAD.NpxState (| 0x800).
  4. If KTHREAD.CetShadowStack flag is on and CET is enabled in XSTATE_CONFIGURATION, nt!KiInitializeContextThread enables CET state in XSAVE header in extended Context structure (created in step 1), and copies the CET state from the extended Context to the XSAVE area on new thread's kernel stack (KTHREAD.StateSaveArea).
  5. When the new thread is scheduled to run, nt!SwapContext loads the CET state of new thread from its KTHREAD.StateSaveArea to CET MSRs using xrstors instruction (KTHREAD.NpxState used as instruction mask).
  6. When the new thread returns to user mode, SSP is automatically loaded from IA32_PL3_SSP MSR.

The shadow stack allocation seems to be missing in the thread creation.

Intel CET Implementation on Windows 10 - Thread Termination (NtTerminateThread)

- The logics related to the Shadow Stack deallocation
  1. When KTHREAD.CetShadowStack flag is on, PspExitThread calls the function PspFreeCurrentThreadUserShadowStack to free the user-mode shadow stack of the current thread.
  2. PspFreeCurrentThreadUserShadowStack obtains the shadow stack address of the current thread, which is accomplished by reading MSR of IA32_PL3_SSP (rdmsr).
  3. PspFreeCurrentThreadUserShadowStack retrieves the base address of shadow stack by calling ZwQueryVirtualMemory.
  4. PspFreeCurrentThreadUserShadowStack frees the shadow stack memory with MmFreeVirtualMemory.

Intel CET Implementation on Windows 10 - Fiber Creation (CreateFiberEx)

- The logics related to the Shadow Stack allocation and preparation
  1. When the shadow stack is enabled for the calling thread (obtained by rdssp), kernelbase!CreateFiberEx calls ntdll!RtlCreateUserFiberShadowStack to create shadow stack for an user fiber.
  2. ntdll!RtlCreateUserFiberShadowStack calls the system call ntdll!NtSetInformationProcess (ProcessInformationClass 0x62), providing the desired reserve size and initial commit size of shadow stack in the 3rd parameter of NtSetInformationProcess.
  3. The kernel-mode handler of ProcessInformationClass 0x62 in nt!NtSetInformationProcess verifies Shadow Stack feature are enabled in nt!KeFeatureBits and KTHREAD.CetShadowStack flags, then it calls nt!PspSetupUserFiberShadowStack.

Intel CET Implementation on Windows 10 - Fiber Creation (CreateFiberEx) (Cont.)

- The logics related to the Shadow Stack allocation and preparation

4. nt!PspSetupUserFiberShadowStack in turn calls nt!PspReserveAndCommitUserShadowStack, and the latter internally calls nt!MmAllocateUserStack and nt!ZwAllocateVirtualMemory to do the actual job of reserving and committing stack memory.
5. After the shadow stack is allocated, nt!PspSetupUserFiberShadowStack then prepares a return address (ntdll!RtlUserFiberStart) and creates a restore token on the shadow stack with the help of “wruss” instruction.
6. Returning from the system call, kernelbase!CreateFiberEx saves the address of created shadow stack somewhere in the fiber object. It also prepares a same return address on the fiber’s data stack in order to match that on shadow stack (in kernelbase!BaseInitializeFiberContext).

Intel CET Implementation on Windows 10 - Shadow Stack Setup in PspSetupUserFiberShadowStack

```
PAGE:00000001408AB824 PspSetupUserFiberShadowStack proc near  ; CODE XREF:
PAGE:00000001407AA487
...
PAGE:00000001408AB841                call    PspReserveAndCommitUserShadowStack
PAGE:00000001408AB846                mov     ebx, eax
PAGE:00000001408AB848                test    eax, eax
PAGE:00000001408AB84A                js      short loc_1408AB8A9
PAGE:00000001408AB84C                mov     rcx, [rsp+48h+var_18]
PAGE:00000001408AB851                sub     rcx, 8 // 1st qword on shadow stack bottom
PAGE:00000001408AB855                mov     [rsp+48h+var_18], rcx
PAGE:00000001408AB85A
PAGE:00000001408AB85A loc_1408AB85A:                         ; DATA XREF: .rdata:000000014040708Co
PAGE:00000001408AB85A                mov     rax, cs:PspUserFiberStart // ntdll!RtlUserFiberStart
PAGE:00000001408AB861                wruss   qword ptr [rcx], rax
PAGE:00000001408AB867                jmp     short loc_1408AB870
...
```

Intel CET Implementation on Windows 10 - Shadow Stack Setup in PspSetupUserFiberShadowStack (Cont.)

```
...
PAGE:00000001408AB870
PAGE:00000001408AB870 loc_1408AB870:                    ; CODE XREF:
PspSetupUserFiberShadowStack+43j
PAGE:00000001408AB870                test    ebx, ebx
PAGE:00000001408AB872                js      short loc_1408AB8A9
PAGE:00000001408AB874                mov     rax, rcx
PAGE:00000001408AB877                and     rax, 0FFFFFFFFFFFFFFFDh // Create a shadow
stack restore token (ptr to 1st return address on stack)
PAGE:00000001408AB87B                or      rax, 1 // L flag (create in 64-bit mode)
PAGE:00000001408AB87F                sub     rcx, 8 // 2nd dword on shadow stack bottom
PAGE:00000001408AB883                mov     [rsp+48h+var_18], rcx
PAGE:00000001408AB888                wruss   qword ptr [rcx], rax
PAGE:00000001408AB88E                jmp     short loc_1408AB897
...
PAGE:00000001408AB8EB PspSetupUserFiberShadowStack endp
```

Intel CET Implementation on Windows 10 - Shadow Stack Region

```
// guard page
0:000> !address 6098bfd000
Usage:                  <unknown>
Base Address:           00000060`98bfd000
End Address:            00000060`98bfe000
Region Size:            00000000`00001000 (   4.000 kB)
State:                  00001000        MEM_COMMIT
Protect:                00000102        <unknown>
Type:                   00020000        MEM_PRIVATE
Allocation Base:        00000060`98b00000
Allocation Protect:     00000002        PAGE_READONLY

// committed shadow stack page
0:000> !address 6098bfe000
Usage:                  <unknown>
Base Address:           00000060`98bfe000
End Address:            00000060`98bff000
Region Size:            00000000`00001000 (   4.000 kB)
State:                  00001000        MEM_COMMIT
Protect:                00000002        PAGE_READONLY
Type:                   00020000        MEM_PRIVATE
Allocation Base:        00000060`98b00000
Allocation Protect:     00000002        PAGE_READONLY
```

```
// reserved shadow stack region

0:000> !address 6098b00000
Usage:                  <unknown>
Base Address:           00000060`98b00000
End Address:            00000060`98bfd000
Region Size:            00000000`000fd000 (1012.000 kB)
State:                  00002000        MEM_RESERVE
Protect:                <info not present at the target>
Type:                   00020000        MEM_PRIVATE
Allocation Base:        00000060`98b00000
Allocation Protect:     00000002        PAGE_READONLY
```

Intel CET Implementation on Windows 10 - Fiber Execution (SwitchToFiber)

- The logics related to the Shadow Stack switching
  1. kernelbase!SwitchToFiber calls kernelbase!SwitchToFiberContext to perform the fiber context switching.
  2. When shadow stack is enabled for the new fiber (saved in fiber object), kernelbase!SwitchToFiberContext first saves the shadow stack address of current fiber by executing a “rdssp rdx” instruction.
  3. kernelbase!SwitchToFiberContext then performs the shadow stack switching by utilizing the new instruction pair rstorssp/saveprevssp.
  4. The shadow stack address of old fiber is decreased by 8 bytes (pointing to the restore token), then saved into the old fiber object.
  5. kernelbase!SwitchToFiberContext loads the data stack of new fiber then returns to the preset general fiber entry point on top of stack (ntdll!RtlUserFiberStart). Because a same return address is also prepared in shadow stack, this “ret” instruction doesn’t cause a CP fault.

Intel CET Implementation on Windows 10 - Shadow Stack Switching in SwitchToFiberContext

```
.text:0000000180093160 SwitchToFiberContext proc near          ; CODE XREF: SwitchToFiber+2Ap
.text:0000000180093160                 mov     rdx, gs:30h
...
.text:0000000180093295                 cmp     qword ptr [rcx+528h], 0 // shadow stack exists?
.text:000000018009329D                 jz      short loc_1800932C1
.text:000000018009329F                 xor     edx, edx
.text:00000001800932A1                 rdssp   rdx
.text:00000001800932A6                 mov     r9, [rcx+528h]
.text:00000001800932AD                 rstorssp qword ptr [r9]
.text:00000001800932B2                 saveprevssp ;
.text:00000001800932B6                 sub     rdx, 8
.text:00000001800932BA                 mov     [rax+528h], rdx
.text:00000001800932C1 loc_1800932C1:                          ; CODE XREF: SwitchToFiberContext+13Dj
...
.text:0000000180093359                 mov     rsp, [r8+98h]
.text:0000000180093360                 retn // return to ntdll!RtlUserFiberStart
.text:0000000180093360 SwitchToFiberContext endp
```

Intel CET Implementation on Windows 10 - Fiber Deletion (DeleteFiber)

- The logics related to the Shadow Stack deallocation
  1. kernelbase!DeleteFiber calls ntdll!RtlFreeUserFiberShadowStack to free the shadow stack of an user fiber.
  2. ntdll!RtlFreeUserFiberShadowStack is a wrapper function of system call ntdll!NtSetInformationProcess (ProcessInformationClass 0x63), the address of shadow stack is passed in as the 3rd parameter of NtSetInformationProcess.
  3. The kernel-mode handler of ProcessInformationClass 0x63 in nt!NtSetInformationProcess verifies Shadow Stack feature are enabled in nt!KeFeatureBits and KTHREAD.CetShadowStack flags, then it calls PspFreeUserFiberShadowStack.
  4. The following steps are pretty much the same as the step 3/4 of thread shadow stack deallocation.

Intel CET Implementation on Windows 10 - CET Context XState Verification (KeVerifyContextXStateCetU)

- The logic of the Shadow Stack context XState verification
  1. First, nt!KeVerifyContextXStateCetU tries to locate the user-mode CET state in the extended Context structure (relying on XSTATE_CONFIGURATION mapped at 0x0FFFFF780000003D8), exits if CET state can't be found.
  2. If KTHREAD.CetShadowStack flag is off, nt!KeVerifyContextXStateCetU verifies that both IA32_U_CET and IA32_PL3_SSP are set to 0, returns C000060Ah if otherwise.
  3. If KTHREAD.CetShadowStack flag is on and CET state is enabled in XSTATE_BV (state-component bitmap of XSAVE header), nt!KeVerifyContextXStateCetU verifies that the IA32_U_CET.SH_STK_EN is set to 1 and IA32_PL3_SSP is within the normal range of shadow stack, returns C000060Ah if otherwise.
  4. If KTHREAD.CetShadowStack flag is on but CET state is not enabled in XSTATE_BV, nt!KeVerifyContextXStateCetU enables CET state in XSTATE_BV (| 0x800), sets IA32_U_CET.SH_STK_EN to 1 and sets IA32_PL3_SSP to the current value of IA32_PL3_SSP MSR.

Intel CET Implementation on Windows 10 - Extended Context Structure

```
kd> dt nt!_CONTEXT // 0x4d0
   +0x000 P1Home           : Uint8B
   +0x008 P2Home           : Uint8B
   +0x010 P3Home           : Uint8B
   +0x018 P4Home           : Uint8B
   +0x020 P5Home           : Uint8B
   +0x028 P6Home           : Uint8B
   +0x030 ContextFlags     : Uint4B
   +0x034 MxCsr            : Uint4B
   +0x038 SegCs            : Uint2B
   +0x03a SegDs            : Uint2B
   +0x03c SegEs            : Uint2B
   +0x03e SegFs            : Uint2B
   +0x040 SegGs            : Uint2B
   +0x042 SegSs            : Uint2B
   +0x044 EFlags           : Uint4B
   ...
   +0x280 Xmm14            : _M128A
   +0x290 Xmm15            : _M128A
   +0x300 VectorRegister   : [26] _M128A
   +0x4a0 VectorControl    : Uint8B
   +0x4a8 DebugControl     : Uint8B
   +0x4b0 LastBranchToRip  : Uint8B
   +0x4b8 LastBranchFromRip : Uint8B
   +0x4c0 LastExceptionToRip : Uint8B
   +0x4c8 LastExceptionFromRip : Uint8B
```

```
typedef struct _CONTEXT_CHUNK {
    LONG Offset;
    ULONG Length;
} CONTEXT_CHUNK, *PCONTEXT_CHUNK;

typedef struct _CONTEXT_EX {
    // Offset and length of the entire extended context
    CONTEXT_CHUNK All;
    // Offset and length of the legacy context
    CONTEXT_CHUNK Legacy;
    // Offset and length of the extended state
    CONTEXT_CHUNK XState;
} CONTEXT_EX, *PCONTEXT_EX;
#define CONTEXT_AMD64   0x00100000L
#define CONTEXT_CONTROL         (CONTEXT_AMD64 | 0x00000001L)
#define CONTEXT_INTEGER         (CONTEXT_AMD64 | 0x00000002L)
#define CONTEXT_SEGMENTS        (CONTEXT_AMD64 | 0x00000004L)
#define CONTEXT_FLOATING_POINT  (CONTEXT_AMD64 | 0x00000008L)
#define CONTEXT_DEBUG_REGISTERS (CONTEXT_AMD64 | 0x00000010L)
#define CONTEXT_EXTENDED_REGISTERS (CONTEXT_AMD64 | 0x00000020L)
#define CONTEXT_FULL            (CONTEXT_CONTROL | CONTEXT_INTEGER |
CONTEXT_FLOATING_POINT)
#define CONTEXT_ALL             (CONTEXT_CONTROL | CONTEXT_INTEGER |
CONTEXT_SEGMENTS | CONTEXT_FLOATING_POINT |
CONTEXT_DEBUG_REGISTERS)
#define CONTEXT_XSTATE          (CONTEXT_AMD64 | 0x00000040L)
```

Intel CET Implementation on Windows 10 - XState Related Data Structures

XSTATE_CONFIGURATION is mapped at 0x0FFFFF780000003D8 on Windows 10 x64

```
kd> dt nt!_XSTATE_CONFIGURATION -b
   +0x000 EnabledFeatures  : Uint8B
   +0x008 EnabledVolatileFeatures : Uint8B
   +0x010 Size             : Uint4B
   +0x014 ControlFlags     : Uint4B
   +0x014 OptimizedSave    : Pos 0, 1 Bit
   +0x014 CompactionEnabled : Pos 1, 1 Bit
   +0x018 Features         : _XSTATE_FEATURE
      +0x000 Offset        : Uint4B
      +0x004 Size          : Uint4B
   +0x218 EnabledSupervisorFeatures : Uint8B
   +0x220 AlignedFeatures  : Uint8B
   +0x228 AllFeatureSize   : Uint4B
   +0x22c AllFeatures      : Uint4B
   +0x330 EnabledUserVisibleSupervisorFeatures :
Uint8B
```

```
kd> dt ntkrnlmp!_XSAVE_AREA
   +0x000 LegacyState      : _XSAVE_FORMAT // size of 0x200
   +0x200 Header           : _XSAVE_AREA_HEADER
kd> dt nt!_XSAVE_FORMAT
   +0x000 ControlWord      : Uint2B
   +0x002 StatusWord       : Uint2B
   +0x004 TagWord          : UChar
   +0x005 Reserved1        : UChar
   +0x006 ErrorOpcode      : Uint2B
   +0x008 ErrorOffset      : Uint4B
   +0x00c ErrorSelector    : Uint2B
   +0x00e Reserved2        : Uint2B
   +0x010 DataOffset       : Uint4B
   +0x014 DataSelector     : Uint2B
   +0x016 Reserved3        : Uint2B
   +0x018 MxCsr            : Uint4B
   +0x01c MxCsr_Mask       : Uint4B
   +0x020 FloatRegisters   : [8] _M128A
   +0x0a0 XmmRegisters     : [16] _M128A
   +0x1a0 Reserved4        : [96] UChar
kd> dt ntkrnlmp!_XSAVE_AREA_HEADER
   +0x000 Mask             : Uint8B //XSTATE_BV
   +0x008 CompactionMask   : Uint8B //XCOMP_BV
   +0x010 Reserved2        : [6] Uint8B
```

4. Control-flow Hijacking and ACE on Windows 10 with CET enabled

Control-flow Hijacking and ACE on Windows 10 with CET enabled - The Landscape for Vulnerability Exploitation With the Introduction of CET

- The Shadow Stack defeats the back-edge control-flow hijacking via return address overwrite
- The Shadow Stack + ACG + CIG makes it even harder to execute arbitrary code
  1. No ROP shellcode
  2. No executable page
  3. No 3rd party module
- Forward-edge control-flow hijacking is still possible (CFG bypass) as IBT support is not added
- The scripting engine can be leveraged to create a shellcode-free exploit/payload (<<Shellcodes are for the 99%>>)

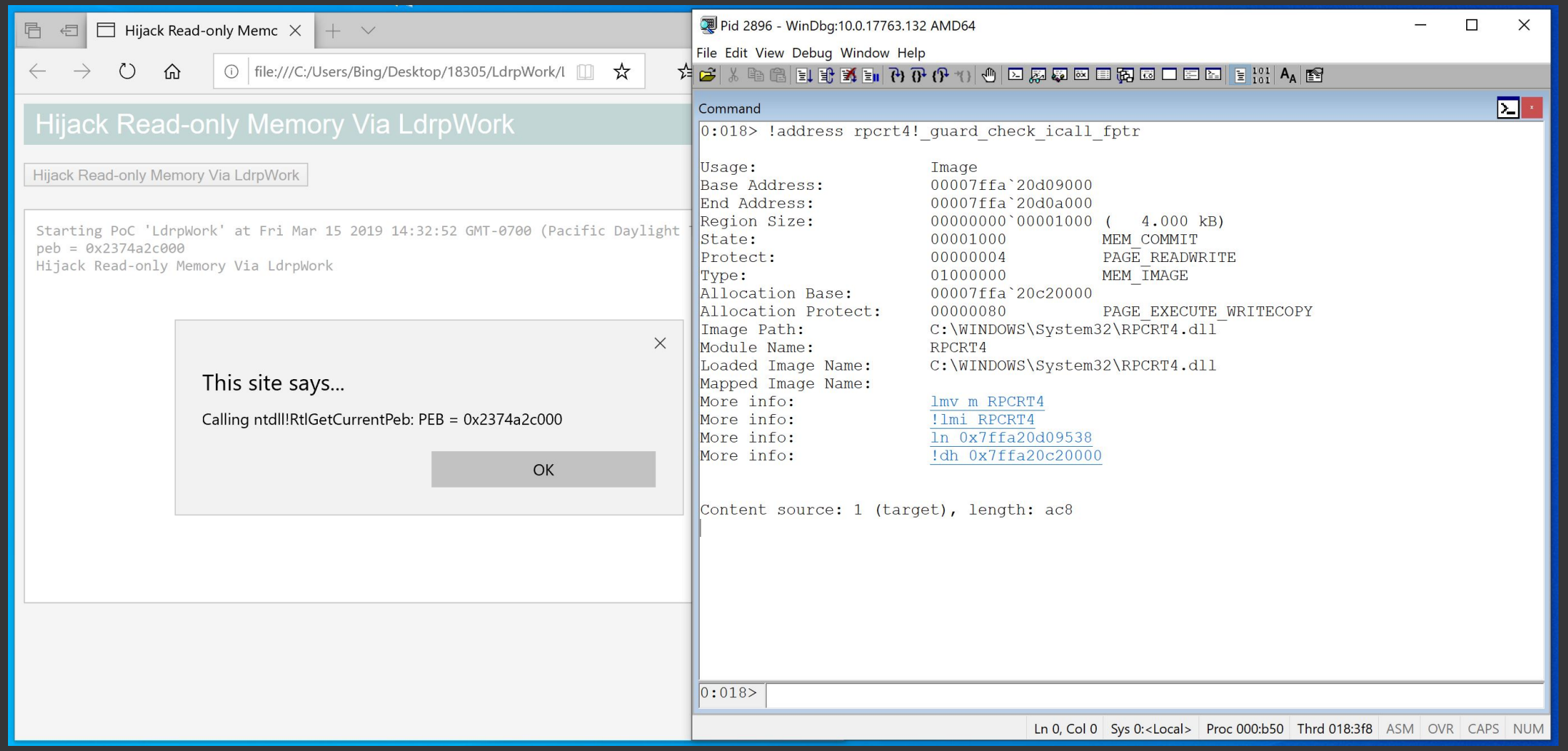
Control-flow Hijacking and ACE on Windows 10 with CET enabled - Possible Ways to Circumvent CET

- Code Replacement Attack (off topic)
- Counterfeit Object-Oriented Programming (COOP, off topic)
- Data-only corruption
- Function pointer hijacking through race condition attack
- Thread context hijacking by abusing NtContinue mechanism

Control-flow Hijacking and ACE on Windows 10 with CET enabled - Data-only Corruption

- CET and CFG protect only the integrity of control-flow (code), not the integrity of data. Therefore, corrupting a program's critical data can sometimes lead to control-flow hijacking
- Case study: CFG bypass by abusing ntdll Ldrpwork mechanism (issue still exists in Windows 10 19H1)

Control-flow Hijacking and ACE on Windows 10 with CET enabled - CFG Bypass by Abusing Ldrpwork Mechanism



Control-flow Hijacking and ACE on Windows 10 with CET enabled - Function Pointer Hijacking Through Race Condition Attack

- Due to lack of hardware based forward-edge control-flow enforcement, the in-memory target address of indirect call/jmp is still susceptible to race condition attack.
- Case study 1: Exception/Unwind handler hijacking through race condition attack.
- Case study 2: Frame consolidation unwind callback routine hijacking through race condition attack.

Control-flow Hijacking and ACE on Windows 10 with CET enabled - Exception/Unwind Handler Hijacking Through Race Condition Attack

- Race condition bugs were found in RtlDispatchException/ RtlpExecuteHandlerForException and RtlUnwindEx/ RtlpExecuteHandlerForUnwind functions of ntdll.dll, which can be exploited to achieve control-flow hijacking.
- The exception/unwind handler is first saved on stack before it gets executed. A small time window between the stack store and handler invocation makes it possible for a race condition attack.
- The exception/unwind handlers seem to come from certain trusted place, thus there is no CFG check against them; that makes a race condition attack easier

Control-flow Hijacking and ACE on Windows 10 with CET enabled - Vulnerable Code Analysis of Exception Handler Hijacking

```
ntdll!RtlDispatchException
...
.text:00000001800055AF                mov     [rbp+1E0h+var_140], rcx
// The exception handler is saved on stack
.text:00000001800055B6                mov     [rbp+1E0h+var_138], r9
.text:00000001800055BD                mov     [rbp+1E0h+var_130], rdx
.text:00000001800055C4                mov     [rbp+1E0h+var_128], eax
.text:00000001800055CA                cmp     [rbp+1E0h+var_1DF], bl
.text:00000001800055CD                jnz     loc_1800A6E5C
.text:00000001800055D3                mov     r8, [rbp+1E0h+var_178]
.text:00000001800055D7                lea     r9, [rbp+1E0h+ControlPc]
.text:00000001800055DB                mov     rdx, r14
.text:00000001800055DE                mov     rcx, r15
.text:00000001800055E1                call
RtlpExecuteHandlerForException
.text:00000001800055E6                mov     edx, eax
.text:00000001800055E8                test    rbx, rbx
.text:00000001800055EB                jnz     loc_1800A6E75
...
```

```
RtlpExecuteHandlerForException proc near
.text:00000001800A0D30                sub     rsp, 28h
.text:00000001800A0D34                mov
[rsp+28h+var_8], r9
.text:00000001800A0D39                mov     rax, [r9+30h]
// Call exception handler
.text:00000001800A0D3D                call    rax
.text:00000001800A0D3F                nop
.text:00000001800A0D40                add     rsp, 28h
.text:00000001800A0D44                retn
.text:00000001800A0D44
RtlpExecuteHandlerForException endp
```

The time window for race condition attack is marked in red

Control-flow Hijacking and ACE on Windows 10 with CET enabled - Vulnerable Code Analysis of Unwind Handler Hijacking

```
ntdll!RtlUnwindEx
...
.text:00000001800060EF          mov     rcx, [rbp+1C0h+var_158]
.text:00000001800060F3          mov     [r12+78h], rcx
.text:00000001800060F8          mov     [rbp+1C0h+var_110], rax // The
unwind handler is saved on stack
.text:00000001800060FF          mov     rax, [rbp+1C0h+var_188]
.text:0000000180006103          mov     [rbp+1C0h+var_108], rax
.text:000000018000610A          mov     rax, [rbp+1C0h+var_190]
.text:000000018000610E          mov     [rbp+1C0h+var_100], rax
...
.text:0000000180006155          lea     r9, [rbp+1C0h+ControlPc]
.text:000000018000615C          mov     r8, r12
.text:000000018000615F          mov     rdx, r14
.text:0000000180006162          mov     rcx, r10
.text:0000000180006165          call    RtlpExecuteHandlerForUnwind
.text:000000018000616A          mov     edx, eax
.text:000000018000616C          test    bl, bl
.text:000000018000616E          jnz     short loc_180006189
...
```

```
RtlpExecuteHandlerForUnwind proc
.text:00000001800A0DB0          sub     rsp, 28h
.text:00000001800A0DB4          mov     [rsp+28h+var_8], r9
.text:00000001800A0DB9          mov     rax, [r9+30h]
// Call unwind handler
.text:00000001800A0DBD          call    rax
.text:00000001800A0DBF          nop
.text:00000001800A0DC0          add     rsp, 28h
.text:00000001800A0DC4          retn
.text:00000001800A0DC4 RtlpExecuteHandlerForUnwind endp
```

The time window for race condition attack is marked in red

Control-flow Hijacking and ACE on Windows 10 with CET enabled - Frame Consolidation Unwind Callback Routine Hijacking Through Race Condition Attack

- Frame Consolidation Unwind
- RtlUnwindEx/RcFrameConsolidation function of ntdll.dll has a race condition bug, which can be exploited to achieve control-flow hijacking.
- The unwind callback routine (_EXCEPTION_RECORD.ExceptionInformation[0]) is validated by CFG before making the call. Nevertheless, there is a small window of time between the 1st exception code check for CFG (in RtlUnwindEx) and the 2nd exception code check for final execution (RcFrameConsolidation), during which the exception record on stack is exposed to a race condition attack.

Control-flow Hijacking and ACE on Windows 10 with CET enabled - Vulnerable Code Analysis of Frame Consolidation Unwind Callback Routine Hijacking

```
ntdll!RtlUnwindEx
...
.text:0000000180006384          mov     eax, [r14] // exception code
.text:0000000180006387          cmp     eax, 80000026h
.text:000000018000638C          jz      loc_1800066C2
// The 1st exception code check for CFG
.text:0000000180006392          cmp     eax, 80000029h  ;
.text:0000000180006397          jz      loc_18000672A
.text:000000018000639D          call    LdrControlFlowGuardEnforced
.text:00000001800063A2          test    eax, eax
.text:00000001800063A4          jnz     loc_18000648D
.text:00000001800063AA          mov     rdx, r14        // ExceptionRecord
.text:00000001800063AD          mov     rcx, r12        // ContextRecord
.text:00000001800063B0          call    RtlRestoreContext
...
.text:000000018000648D          mov     rcx, [r12+98h]
.text:0000000180006495          call    RtlGuardIsValidStackPointer
.text:000000018000649A          test    eax, eax
.text:000000018000649C          jnz     loc_1800063AA
...
.text:000000018000672A // CFG check
...
.text:0000000180006756          call    LdrpValidateUserCallTarget
.text:000000018000675B          jmp     loc_18000639
```

```
ntdll!RtlRestoreContext
...
.text:000000018009FE54          sub     rsp, 30h
.text:000000018009FE58          mov     rbp, rsp
.text:000000018009FE5B          test    rdx, rdx
.text:000000018009FE5E          jz      loc_18009FF96
// The 2nd exception code check for final execution.
.text:000000018009FE64          cmp     dword ptr [rdx], 80000029h
.text:000000018009FE6A          jnz     short loc_18009FE76
.text:000000018009FE6C          cmp     dword ptr [rdx+18h], 1
.text:000000018009FE70          jnb     loc_1800A0141
...
.text:00000001800A0141
...
.text:00000001800A0177          jmp     short RcFrameConsolidation
...
.text:00000001800A0180 RcFrameConsolidation proc near
// call _EXCEPTION_RECORD.ExceptionInformation[0]
.text:00000001800A0180          mov     rax, [rcx+20h]
.text:00000001800A0184          call    rax
```

The time window for race condition attack is marked in red

Control-flow Hijacking and ACE on Windows 10 with CET enabled - Thread Context Hijacking by Abusing NtContinue System Call

- NtContinue can change the current thread's context. Such a context change occurs in kernel space, thus all user-mode CFI enforcements become irrelevant.
- NtContinue takes its ThreadContext argument from memory, and the kernel-mode NtContinue syscall handler doesn't validate most members of context (except for Rsp and CET XState), both factors make thread context hijacking through race condition attack possible.
- Case study 1: Thread context hijacking in thread's user-mode initialization (ntdll!LdrInitializeThunk).
- Case study 2: Thread context hijacking in exception unwind process (ntdll!RtlRestoreContext).

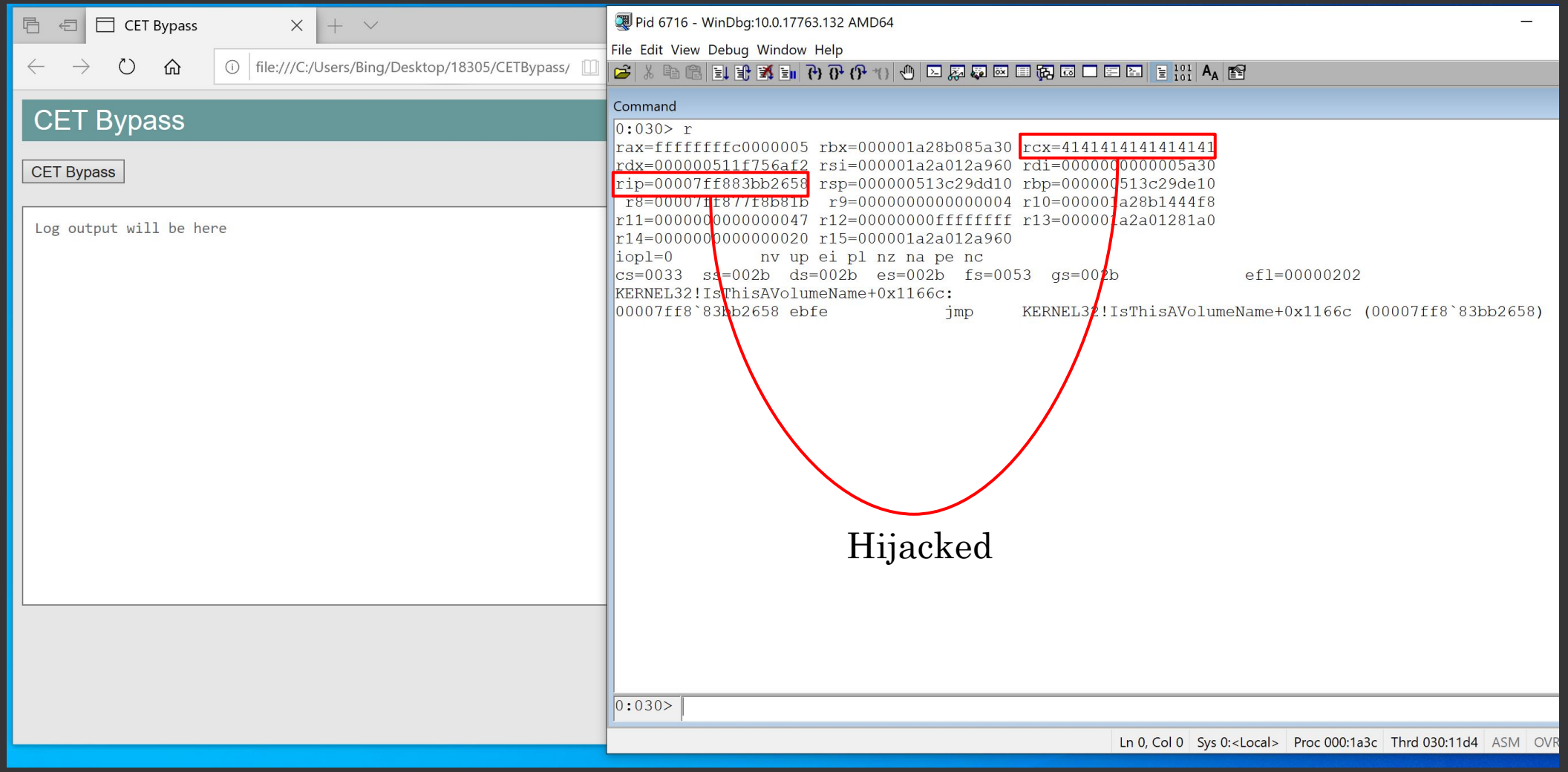
Control-flow Hijacking and ACE on Windows 10 with CET enabled - Vulnerable Code Analysis of Thread Context Hijacking in Exception Unwind Process

```
ntdll!RtlRestoreContext (Rcx: ContextRecord Rdx: ExceptionRecord)
...
.text:000000018009FE64          cmp     dword ptr [rdx], 80000029h
.text:000000018009FE6A          jnz     short loc_18009FE76
.text:000000018009FE6C          cmp     dword ptr [rdx+18h], 1
.text:000000018009FE70          jnb     loc_1800A0141
.text:000000018009FE76          cmp     dword ptr [rdx], 80000026h
.text:000000018009FE7C          jnz     loc_18009FF96
// Long jump
...
// normal context restore
// Select the way to restore context based on certain flags in ntdll's mrdata section and Context.ContextFlags
.text:000000018009FF96          lea     rax, xmmword_18017B370
.text:000000018009FF9D          mov     rax, [rax+8]
.text:000000018009FFA1          bt      rax, 3Ch
.text:000000018009FFA6          jb      short loc_18009FFC7
.text:000000018009FFA8          lea     rax, xmmword_18017B370
.text:000000018009FFAF          mov     rax, [rax+8]
.text:000000018009FFB3          bt      rax, 0Ch
.text:000000018009FFB8          jb      short loc_18009FFC7
.text:000000018009FFBA          mov     eax, [rcx+30h]
.text:000000018009FFBD          and     eax, 0FFFFFFBFh
.text:000000018009FFC0          cmp     eax, 10000Fh
.text:000000018009FFC5          jz      short loc_18009FFE4
```

```
ntdll!RtlRestoreContext
// Context restore via NtContinue syscall
.text:000000018009FFC7          xor     edx, edx
.text:000000018009FFC9          call    ZwContinue
...
// Do fast context restore in user-mode
.text:000000018009FFE4          mov     eax, [rcx+30h]
.text:000000018009FFE7          and     eax, 100040h
.text:000000018009FFEC          cmp     eax,
.text:000000018009FFF1          jnz     short loc_1800A0026
...
// Restore XSTATE when existed
.text:00000001800A001F          xrstor  byte ptr [rbx]
...
// Restore x87 FPU, MMX, SSE and general register states
.text:00000001800A0026          fxrstor dword ptr [rcx+100h]
...
// Restore SS:Rsp, CS:Rip and Eflags.
.text:00000001800A013F          iretq
.text:00000001800A0141
// Frame consolidation
...
.text:00000001800A0177          jmp     short RcFrameConsolidation
```

The context record on stack is subject to race condition attack during the context restore process

Control-flow Hijacking and ACE on Windows 10 with CET enabled - Thread Context Hijacking in Exception Unwind Process



Conclusion

- CET Shadow Stack is a good supplement to CFG, and it makes the control-flow hijacking and ACE more difficult. Shadow Stack can successfully block control-flow hijacking via return address overwrite and ROP-based shellcode as designed.
- However, even with the fully hardware-assisted CET in place, other types of control-flow hijacking (through data-only attack and NtContinue as discussed before) are still possible; but the subsequent ACE after the success control-flow hijacking might become extremely difficult.
- Compared with the fully hardware-based IBT, the current implementation of forward-edge control-flow hijacking prevention in Windows 10 still relies on the software-based CFG, which is subject to bypasses.
- Some critical system functionalities need to be moved to OOP, such as the management of mutable read-only memory and exception unwinder.

Q&A and Acknowledgements

- Send questions to
  - bing_sun@mcafee.com
  - mr.owens.nobody@gmail.com
  - chong_xu@mcafee.com
- Thanks to MSRC for helping analyze and confirm these issues in a timely manner
- Special thanks to McAfee IPS Security Research Team

About XFutureSecurity

- Security Services
  - professional security fields of vulnerability mining and penetration testing, security tools development and security services
- Security Products
  - AIRD--Artificial Intelligence Risk Decision
- Security Conference
  - XFocus Information Security Conference（XCon）
- Successfully introduced the world-class hacker event into China
  - DEF CON CHINA

- blackhat@xfuturesec.com

References

- https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf
- https://www.linuxplumbersconf.org/event/2/contributions/147/attachments/72/83/CET-LPC-2018.pdf
- https://github.com/Microsoft/MSRC-Security-Research/blob/master/presentations/2018_02_OffensiveCon/The%20Evolution%20of%20CFI%20Attacks%20and%20Defenses.pdf
- http://www.nynaeve.net/?p=110
- https://cansecwest.com/slides/2018/Shellcodes%20are%20for%20the%2099%25%20-%20Bing%20Sun,%20Stanley%20Zhu,%20and%20Chong%20Xu,%20McAfee%20and%20Didi%20Chuxing.pdf
- https://conference.hitb.org/hitbsecconf2017ams/sessions/bypassing-memory-mitigations-using-data-only-exploitation-techniques-part-ii/
- https://sites.google.com/site/bingsunsec/stackdatacorruption

Thanks