black hat BRIEFINGS

AUGUST 6-7, 2025
MANDALAY BAY / LAS VEGAS

Out Of Control: How KCFG and KCET Redefine Control Flow Integrity in the Windows Kernel

Connor McGarr [@33y0re]

Software Engineer, Prelude Security

About

- Software Engineer at Prelude Security
  - Previously Software Engineer at CrowdStrike on the Windows Sensor Team
- Blog: connormcgarr.github.io
  - Windows OS internals, exploit mitigations, browser and kernel exploitation, malware, reverse engineering articles
- I like C, Assembly, Operating Systems, and Hypervisors!

[P] | Prelude

Introduction To Control Flow Integrity

- Most exploits require two things:
  1. Ability to hijack the legitimate execution (control flow) of an application/operating system
  2. ~~Use the above primitive to execute some malicious code~~
- Control Flow Integrity (CFI) attempts to address the first problem by verifying and mitigating attempts to alter the target of a control-flow transfer
  - Calls/jmps are forwards-edge control-flow transfers
  - Returns are backwards-edge control-flow transfers

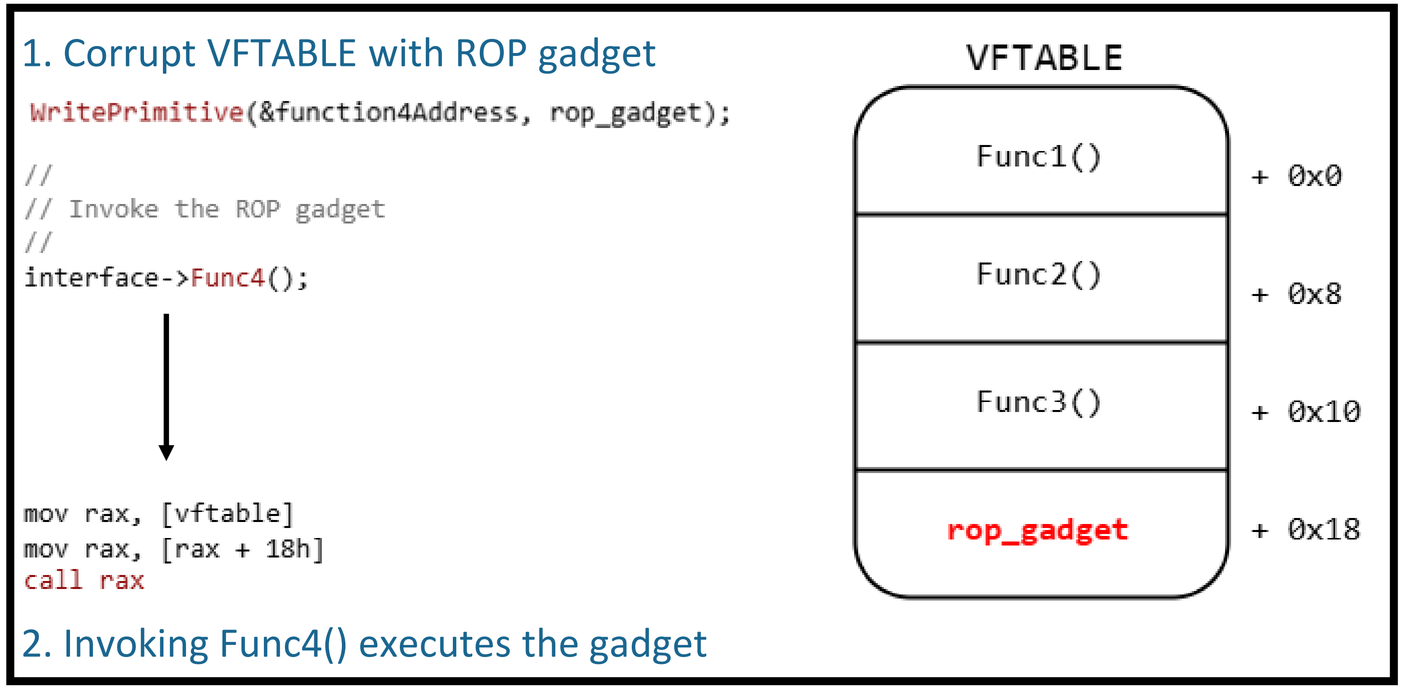
Control Flow Guard

- Control Flow Guard is Window's version of forwards-edge CFI
  - Present in user-mode since Windows 8.1 (as an optional update)
- All indirect call targets which are known at compile-time are stored in a read-only, kernel-protected (and per-process) "CFG bitmap"
  - User-mode address space is 128 TB on 64-bit Windows
  - …there are 128 TB of possible call targets (in theory), but the compiler *should* generate call targets at 16-byte (0x10) boundaries
    - 128 TB / 16 bytes = 8 TB of potential targets
    - 8 TB * 2 bits (denotes the "state" for every 16 bytes) = **2 TB CFG bitmap size**
    - Memory manager performs some optimizations…
  - Indirect call/jmps are replaced with "thunks" that first check the CFG bitmap for bits related to the call target before transferring execution

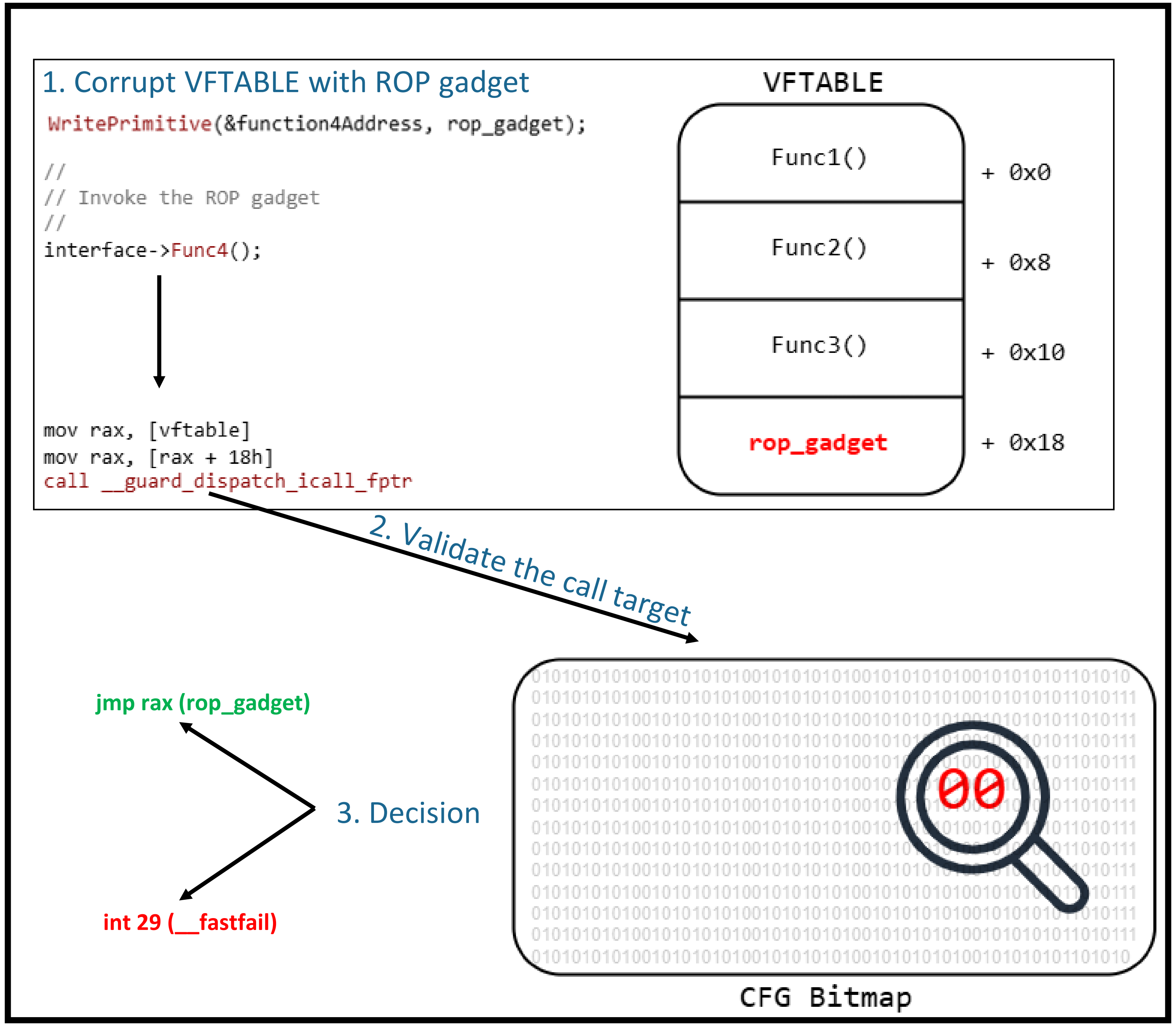
Control Flow Guard

- CFG bitmap states
  - 0, 0 -> No valid function present in these 16 bytes
  - 1, 0 -> A valid function (16-byte aligned)
  - 1, 1 -> A valid function (not 16-byte aligned)
  - 0, 1 -> This target is explicitly suppressed (special “export suppression” feature)
- We need 2 bits instead of just 1
  - Compilers should generate functions at 16-byte boundaries there is no guarantee
  - Instead of 1 bit for true/false, 2 bits allows us to encapsulate more information (such as 16-byte alignment validity)

Control Flow Guard



Before CFG



After CFG

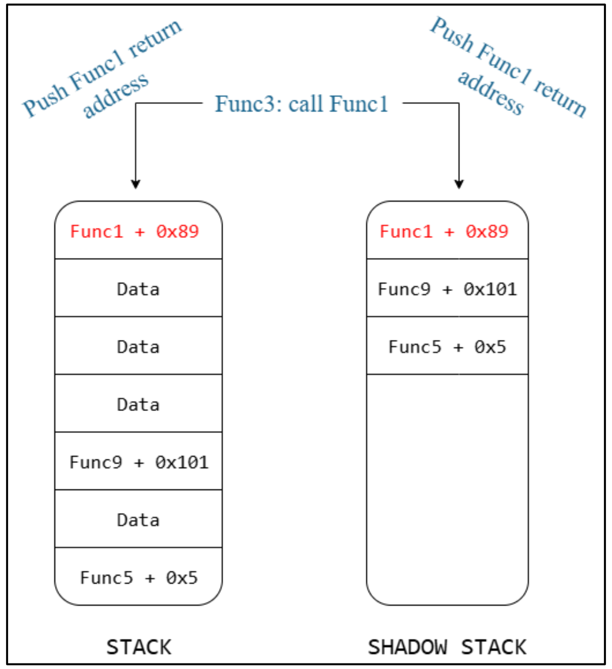
Backwards-Edge CFI On Windows

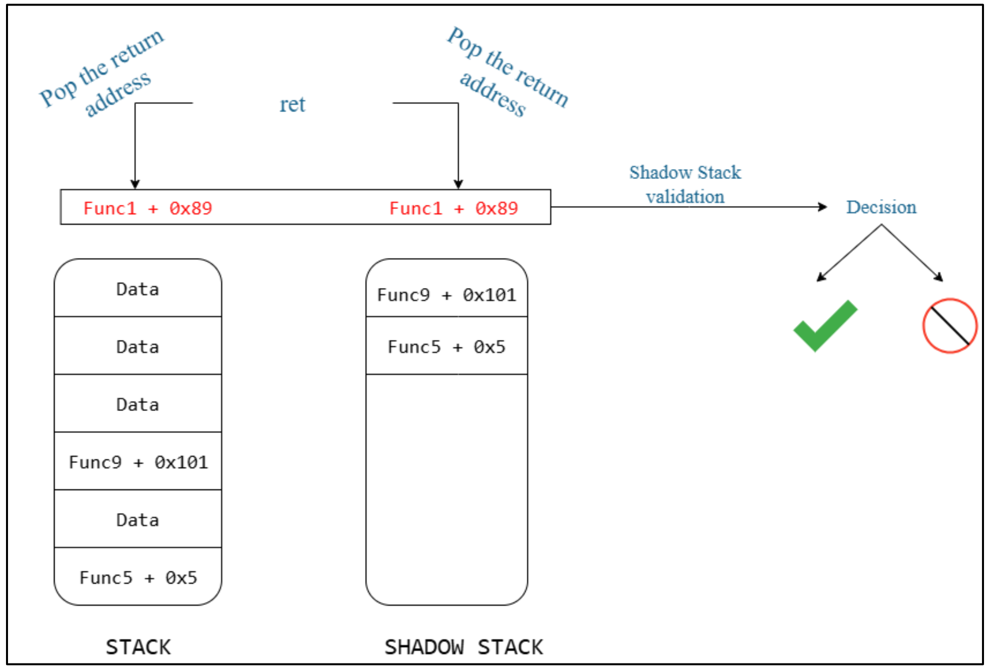
- Although CFG did have an impact on exploitation attackers started to just avoid CFG entirely
  - One example of this was reverting to return address corruption (not using a stack overflow primitive) – indicating that a comprehensive CFI solution requires protection of both forwards-edge ***AND*** backwards-edge control-flow
  - Microsoft began by attempting a software-based implementation of backwards-edge CFI called Return Flow Guard (RFG) but deprecated it due to discoveries by their internal red team
- Intel Control-Flow Enforcement Technology (CET) is a hardware solution used by Windows to provide a backwards-edge CFI solution (Windows also supports AMD Shadow Stack)
  - Present in user-mode since Windows 10 19H1 (1903)
  - Windows only uses the Shadow Stack feature of CET

Intel Control-Flow Enforcement Technology

- Intel CET maintains a “shadow stack” containing only return addresses
  - Protected by the kernel, “immutable” to a user-mode attacker
- “call” instructions now also push a return address onto the shadow stack
  - “ret” instructions pops the return address off the shadow stack and compares it with the “traditional” stack’s in-scope return address
  - Mismatch causes a control flow protection fault interrupt (int #21)

call





ret

CFG/CET – Kernel-Mode Counterparts

- You may have noticed a few themes so far…
  1. Both CFG and CET are based on a particular “source of truth”
     - CFG bitmap, CET shadow stack
  2. Both sources of truth are protected by the kernel
     - If an attacker wants to modify these sources of truth, they need to ask the kernel to do so (VirtualProtect system call, etc.)
       - There is a user <-> kernel security boundary
- …but what if we wanted to implement CFG and CET in the kernel?
  - If the kernel is the most privileged part of the OS there is no higher “boundary” to ask, an attacker with a kernel-mode read/write primitive can first just corrupt the source of truth and THEN detonate their exploit!

```
//
// 1. Get the KCFG bitmap's PTE
//
kCfgBitmapPteAddress = LeakPteAddressWithExploit(&leakedKCfgBimap);

//
// 2. Make the KCFG bitmap writable
//
WritePrimitive(&kCfgBitmapPteAddress, writablePteMask);

//
// 3. Mark our ROP gadget as a valid call target
//
WritePrimitive(&leakedKcfgBitmapRopGadgetPosition, validCallTargetBitState);

//
// 4. Get g_FptrArray's KM address
//
g_FptrArray = ReadPrimitive(&g_FptrLeakedAddress);

//
// 5. Corrupt the array and invoke the ROP gadget
//
CorruptFptrArrayAndTriggerCall();
```

kernel_exploit.exe

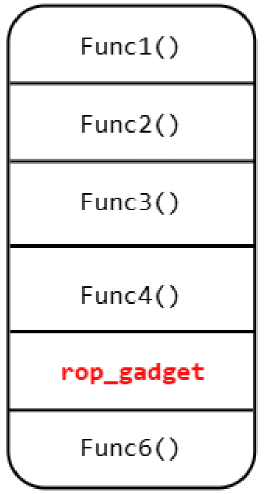
1. Make KCFG bitmap page(s) writable

2. Mark the ROP gadget as a valid call target

3. Leak the g_FptrArray, corrupt it with the ROP gadget, and coherce the kernel to invoke g_FptrArray[5]()

User mode

Kernel Mode



Function Pointer Array

g_FptrArray

Kernel CFG Bitmap

PTE:-G-DA-KW-V

A Higher Security Boundary – Hyper-V

- Luckily for us there IS a higher security boundary on Windows – Microsoft's hypervisor!
- About a decade ago now (hard to believe!) Microsoft implemented Virtualization-Based Security, or VBS, which is a suite of hypervisor-provided security features
  - "Secured-Core" PCs from Microsoft have many VBS features enabled by default
  - Clean installs of Windows 11 do as well!
- With the implementation of VBS we finally can provide CFG and CET mitigations in the Windows kernel to defend against kernel attackers with a read/write primitive!

Virtualization-Based Security

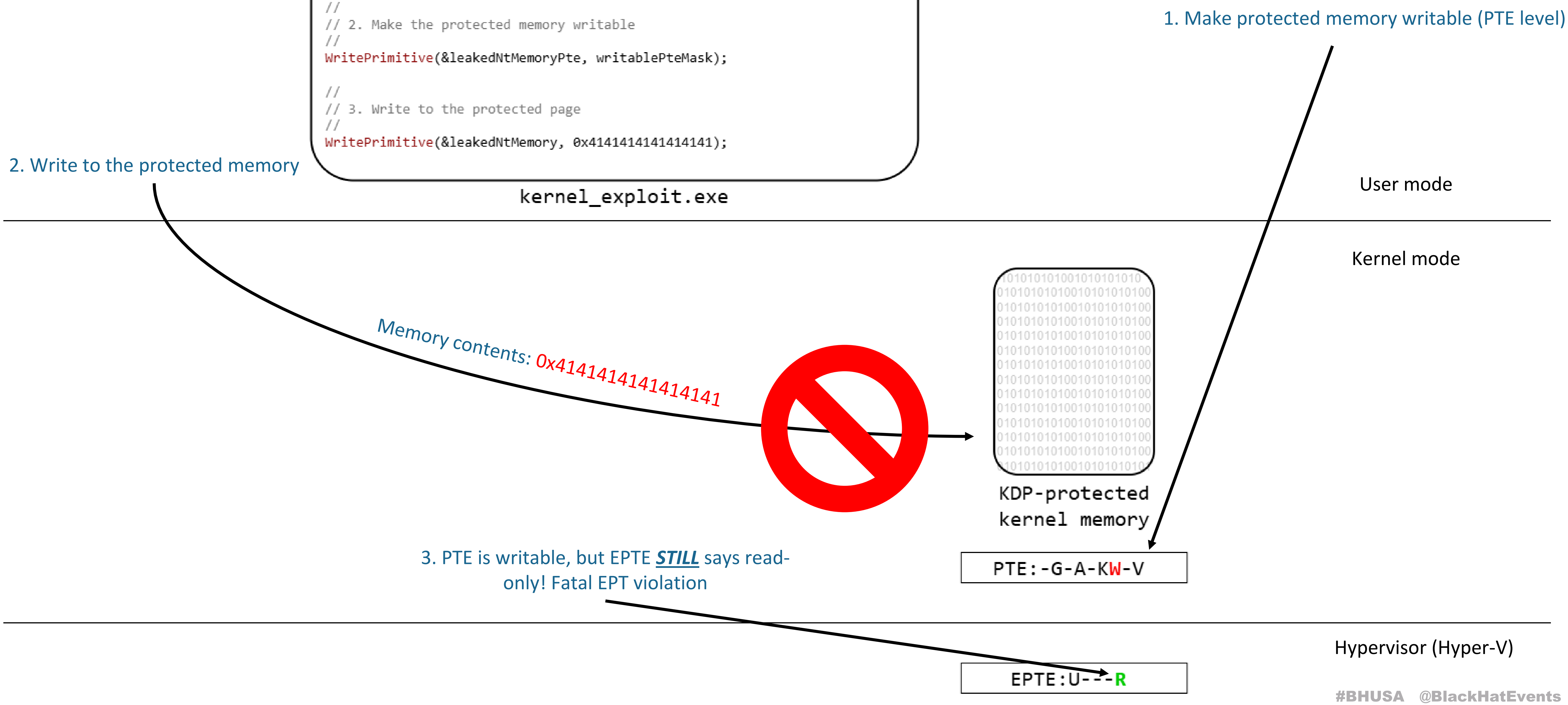
- VBS leverages Second Level Address Translation (SLAT) to enforce various policies/permissions which cannot be altered even by an attacker with kernel mode exploitation primitives
  - Does this by constructing the concept of “Virtual Trust Levels” which are an isolated region of physical memory (like a VM*)
    - VTL 0 – “Normal world” – What a user interfaces with
    - VTL 1 – “Secure world” – Configures VTL 0 security
- Example – Kernel Data Protection (KDP)
  - Sets a read-only Extended Page Table Entry (EPTE) on a target region(s) of memory
  - An attacker even with a kernel-mode read/write primitive cannot make the page(s) writable because the hypervisor manages the true source of truth for the permission of the target page(s) (EPTEs) (which is *not* accessible by the NT kernel!)

```
//
// 1. Get the protected memory's PTE
//
leakedNtMemoryPte = LeakPteAddressWithExploit(&leakedNtMemory);

//
// 2. Make the protected memory writable
//
WritePrimitive(&leakedNtMemoryPte, writablePteMask);

//
// 3. Write to the protected page
//
WritePrimitive(&leakedNtMemory, 0x4141414141414141);
```

kernel_exploit.exe



Virtualization-Based Security

- We can now guarantee the sources of truth for KCFG and KCET are immutable!
  - However, it is not as simple as “just shove CFG and CET in kernel-mode” (which is a primary reason for later adoption than their user-mode counterparts)
- Example – when certain actions occur in the context of a “guest” (VM), a VM Exit may occur to allow the hypervisor to inspect the operation
  - VM Exit is like a context switch but instead of switching into a new thread it involves a switch of execution from “guest” mode to “hypervisor” mode (such as an EPT violation or VMCALL)
    - This is not a free operation – engineers need to consider many such scenarios (this is why the KM implementation of these mitigations is complex in many cases)
- With this in mind, let’s now examine how CFG and CET are implemented in the Windows kernel!

Kernel Control Flow Guard

- Kernel Control Flow Guard (KCFG)
  - Present since Windows 10 1703 (RS2)
  - Fully enabled under Hypervisor-Protected Code Integrity

```
Command

2: kd> dx *(nt!_MI_FLAGS*)&nt!MiFlags
*(nt!_MI_FLAGS*)&nt!MiFlags                 [Type: _MI_FLAGS]
    [+0x000 ( 0: 0)] VerifierEnabled  : 0x0 [Type: unsigned long]
    [+0x000 ( 1: 1)] KernelVerifierEnabled : 0x0 [Type: unsigned long]
    [+0x000 ( 2: 2)] LargePageKernel  : 0x1 [Type: unsigned long]
    [+0x000 ( 3: 3)] StopOn4d         : 0x1 [Type: unsigned long]
    [+0x000 ( 5: 4)] InitializationPhase : 0x2 [Type: unsigned long]
    [+0x000 ( 6: 6)] PageKernelStacks : 0x1 [Type: unsigned long]
    [+0x000 ( 7: 7)] CheckZeroPages   : 0x0 [Type: unsigned long]
    [+0x000 ( 8: 8)] ProcessorPrewalks : 0x0 [Type: unsigned long]
    [+0x000 ( 9: 9)] ProcessorPostwalks : 0x1 [Type: unsigned long]
    [+0x000 (10:10)] CoverageBuild    : 0x0 [Type: unsigned long]
    [+0x000 (11:11)] CheckExecute     : 0x1 [Type: unsigned long]
    [+0x000 (12:12)] ProtectedPagesEnabled : 0x1 [Type: unsigned long]
    [+0x000 (13:13)] SecureRelocations : 0x1 [Type: unsigned long]
    [+0x000 (14:14)] StrongPageIdentity : 0x1 [Type: unsigned long]
    [+0x000 (15:15)] StrongCodeGuarantees : 0x1 [Type: unsigned long]
    [+0x000 (16:16)] HardCodeGuarantees : 0x0 [Type: unsigned long]
    [+0x000 (17:17)] ExecutePagePrivilegeRequired : 0x0 [Type: unsigned long]
    [+0x000 (18:18)] SecureKernelCfgEnabled : 0x1 [Type: unsigned long]
    [+0x000 (19:19)] FullHvci         : 0x0 [Type: unsigned long]
    [+0x000 (20:20)] BootDebuggerActive : 0x0 [Type: unsigned long]
    [+0x000 (21:21)] ExceptionHandlingReady : 0x0 [Type: unsigned long]
```

```
// Phase -1 = "Hypervisor-related"
switch ( Phase )
{
  case 0xFFFFFFFF:
    dword_140E136E0 = 0x800;
    qword_140E136E8 = &unk_140E136F0;
    word_140E13852 |= 1u;
    extendedPageProtectionFlags = VslGetNestedPageProtectionFlags();
    if ( extendedPageProtectionFlags )
    {
      // _MI_FLAGS.ProtectedPagesEnabled = 1
      // _MI_FLAGS.SecureRelocations = 1
      // _MI_FLAGS.StrongPageIdentity = 1
      miFlags = MiFlags | 0x7000;
      // _MI_FLAGS.ProtectedPagesEnabled = 1
      if ( (extendedPageProtectionFlags & 4) == 0 )
        miFlags = MiFlags | 0x1000;
      if ( (extendedPageProtectionFlags & 1) != 0 )
      {
        // _MI_FLAGS.StrongCodeGuarantees = 1
        // _MI_FLAGS.ExecutePagePrivilegeRequired = 1
        miFlags |= 0x28000u;
      }
      else if ( (extendedPageProtectionFlags & 2) != 0 )
      {
        // _MI_FLAGS.StrongCodeGuarantees = 1
        miFlags |= 0x8000u;
      }
      // _MI_FLAGS.HardCodeGuarantees = 1
      miFlags |= 0x10000u;
      MiFlags = miFlags;
      miFlags |= 0x80000u;
      isFullHvciActive = extendedPageProtectionFlags & 0x40;
      if ( (extendedPageProtectionFlags & 0x40) != 0 )
        // _MI_FLAGS.FullHvci = 1
        MiFlags = miFlags | 0x80000;
      if ( (extendedPageProtectionFlags & 0x80u) != 0 )
        // _MI_FLAGS.SecureKernelCfgEnabled = 1
        MiFlags = miFlags | 0x40000;
    }
    if ( KasaniEnabled )
      *(&MiFlags + 1) |= 8u;
    MiInitializeSystemVa(LoaderBlock, isFullHvciActive, miFlags, extendedPageProtectionFlags);
    break;
```

Kernel Control Flow Guard

- The NT kernel is responsible for asking the Secure Kernel to initialize KCFG as part of system initialization (**nt!VslInitializeSecureKernelCfg** -> **securekernel!SkmmInitializeNtKernelCfg**) via secure system call

```
Command

lkd> dx ((nt!_MI_SYSTEM_INFORMATION*)&nt!MiState)->Vs.SystemVaRegions[nt!_MI_ASSIGNED_REGION_TYPES.AssignedRegionCfg]
((nt!_MI_SYSTEM_INFORMATION*)&nt!MiState)->Vs.SystemVaRegions[nt!_MI_ASSIGNED_REGION_TYPES.AssignedRegionCfg]
    [+0x000] BaseAddress      : 0xfffff31d041bd8e8 [Type: void *]
    [+0x008] NumberOfBytes    : 0x28000000000 [Type: unsigned __int64]
```

```
//
// Is KCFG enabled?
//
if ((MiFlags & 0x40000) != 0)
{
    kCfgPtes = MiReservePtes(...);
    if (kCfgPtes == NULL)
    {
        goto Exit;
    }

    if (VslInitializeSecureKernelCfg(SystemVaRegionBaseAddressKernelCfg,
                                     PteToVa(kCfgPtes)) == FALSE)
    {
        goto Exit;
    }
}
```

Kernel Control Flow Guard

- On Kernel CFG initialization the Secure Kernel tracks the region of memory associated with the KCFG bitmap through a structure known as a Normal Address Range (NAR)
  - SK maintains a two kinds of NARs, “normal” NARs (associated with a KM virtual address executable range) and “static” NARs
  - KCFG bitmap, shadow stacks, and a few other regions of memory are static NARs because they are not associated with an image but require management by the Secure Kernel

```
// Is KCFG Enabled?
if ( (SkmiFlags & 0x1000) == 0
  || KernelCfgBaseRangeFromVtl0 < 0xFFFF800000000000uLL
  || (KernelCfgBaseRangeFromVtl0 & 7) != 0
  || KernelCfgBaseRangeFromVtl0 > 0xFFFFFE0000000000uLL )
{
  return STATUS_INVALID_PARAMETER;
}
if ( _InterlockedCompareExchange64(&SkmiKernelCfgBitmapBase, KernelCfgBaseRangeFromVtl0, 0LL) )
  return STATUS_CONFLICTING_ADDRESSES;
// Create KCFG Bitmap NAR
status = SkmiReserveNar(KernelCfgBaseRangeFromVtl0, 0x20000000000LL, 0x10000, 0, 0LL, 0LL, &kcfgBitmapNar);
```

Kernel Control Flow Guard

- After the Secure Kernel is enlightened with the KCFG bitmap range each kernel image load will result in (generally) these steps:
  1. Allocate and map memory in the KCFG bitmap range
  2. Mark the new mapping as read-only in the EPTEs (Bitmap cannot be corrupted from VTL 0)
  3. Update the KCFG bitmap with the appropriate bit states for all KCFG-protected call targets provided by the image (**securekernel!RtlSetBits**)

Hypercall to set read-only SLAT entry for VTL 0

Command

```
0: kd> k
 # Child-SP          RetAddr               Call Site
00 ffffeb00`4f51f0b8 fffff800`80233b36     securekernel!ShvlpInitiateFastHypercall
01 ffffeb00`4f51f0c0 fffff800`8026f228     securekernel!SkmiProtectPlaceholderPages+0x626
02 ffffeb00`4f51f1f0 fffff800`8026a488     securekernel!SkmiSecurePhysicalPage+0x1ec
03 ffffeb00`4f51f250 fffff800`80240b9d     securekernel!SkmiFillAllocatedBundle+0x2dc
04 ffffeb00`4f51f2c0 fffff800`8028a083     securekernel!SkmiAllocatePhysicalPage+0x8ed
05 ffffeb00`4f51f380 fffff800`8028cb1a     securekernel!SkmiAllocateNtKernelCfgBitmapPage+0x93
06 ffffeb00`4f51f480 fffff800`8028e06b     securekernel!SkmiMarkNtKernelCfgBits+0x18a
07 ffffeb00`4f51f520 fffff800`8028c53e     securekernel!SkmiPrepareDriverState+0x437
08 ffffeb00`4f51f630 fffff800`80231d6b     securekernel!SkmiLoadNormalDriver+0x15e
09 ffffeb00`4f51f660 fffff800`8024a770     securekernel!SkmiOperateOnLockedNar+0xbb
0a ffffeb00`4f51f690 fffff800`80304046     securekernel!IumInvokeSecureService+0x62c0
0b ffffeb00`4f51fe40 00000000`00000000     securekernel!SkpReturnFromNormalModeRaxSet+0x162
0: kd> dx @rcx,d
@rcx,d           : 12 [Type: unsigned __int64]
0: kd> dps @rdx L3
ffffeb00`4f51f130  ffffffff`ffffffff
ffffeb00`4f51f138  00000000`00000009
ffffeb00`4f51f140  00000000`0011d844
```

HV_MAP_GPA_READABLE

Kernel Control Flow Guard

- In addition to load image operations there are also special circumstances where the KCFG bitmap may need to be updated
  - Example – calling **nt!MmGetSystemRoutine** marks the target function as a valid call target

```
NTSTATUS __fastcall MiMarkKernelCfgTarget(__int64 TargetFunction)
{
  // Is Kernel CFG enabled?
  if ( (MiFlags & 0x40000) != 0 )
    VslEnableKernelCfgTarget(TargetFunction);
  return STATUS_SUCCESS;
```

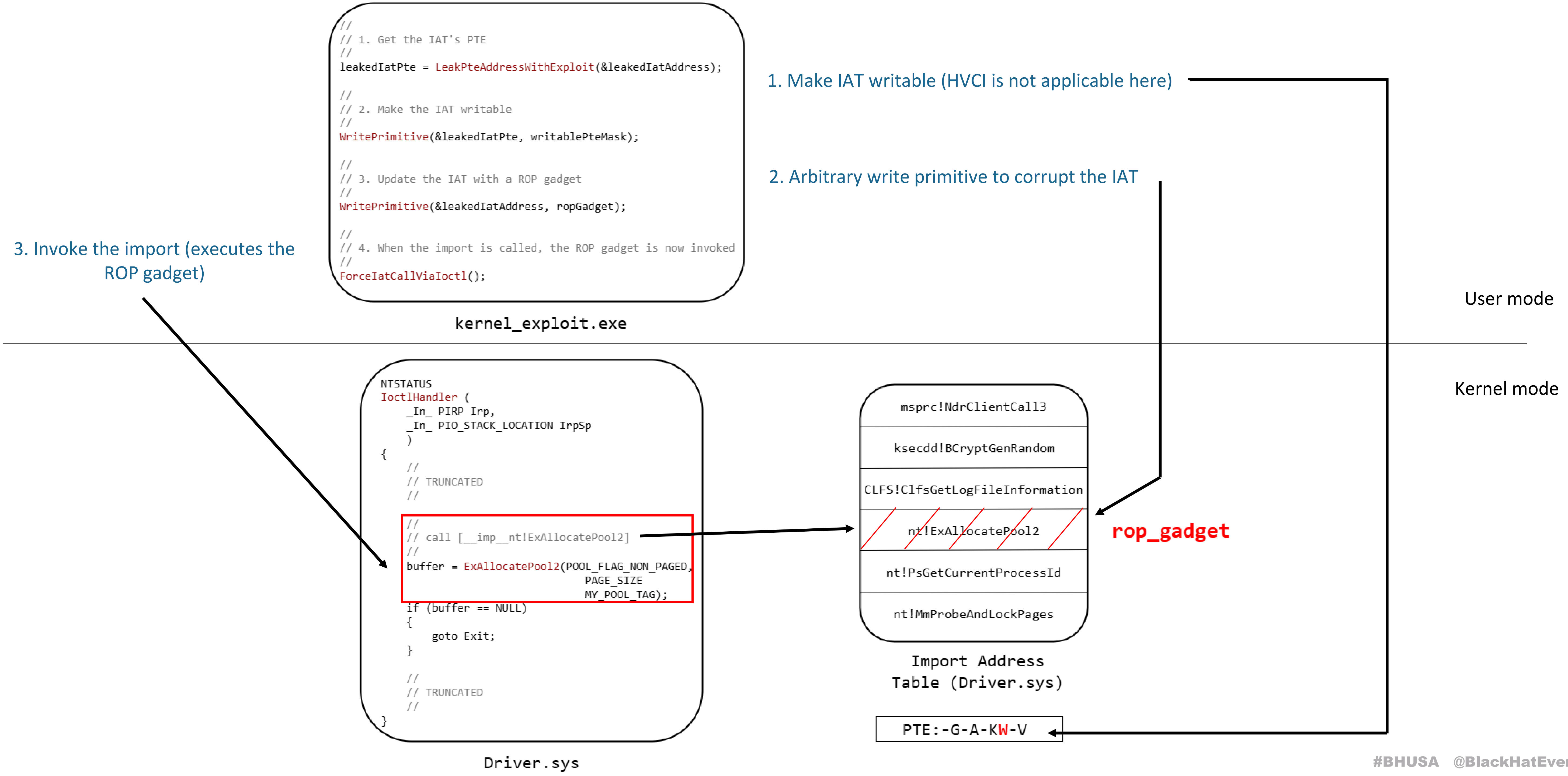
Kernel eXtended Control Flow Guard (KXFG)

- One of the ***known*** limitations of CFG is that it only validates a target exists anywhere in the bitmap, not that the target is the intended one (coarse-grained CFI)
  - Example – Call targets in Win32k can be corrupted with a valid NT call target
- eXtended Control Flow Guard (XFG) was an attempt to address this (fine-grained CFI)
  - Each indirect call has an additional check (the hash of its prototype). The intent was to limit valid call targets from anything in the bitmap to only developer-intended functions
  - XFG was never fully instrumented (UM/KM) and is now deprecated ☹

```
fffff806`689b5051 c744242800000000    mov     dword ptr [status (rsp+28h)], 0
fffff806`689b5059 e8c2000000          call    NotAVulnerableDriver!SetupFunctionArray (fffff806689b5120)
fffff806`689b505e b808000000          mov     eax, 8
fffff806`689b5063 486bc001            imul    rax, rax, 1
fffff806`689b5067 488d0d920f0000      lea     rcx, [NotAVulnerableDriver!g_FunctionArray{[0]} (fffff806689b6000)]
fffff806`689b506e 488b0401            mov     rax, qword ptr [rcx+rax]
fffff806`689b5072 4889442430          mov     qword ptr [rsp+30h], rax
fffff806`689b5077 49ba7048da56963ef185 mov    r10, 85F13E9656DA4870h
fffff806`689b5081 488b442430          mov     rax, qword ptr [rsp+30h]
fffff806`689b5086 ff159ccfffff        call    qword ptr [NotAVulnerableDriver!__guard_xfg_dispatch_icall_fptr (fffff806689b2028)]
fffff806`689b508c 8b442428            mov     eax, dword ptr [status (rsp+28h)]
fffff806`689b5090 4883c448            add     rsp, 48h
fffff806`689b5094 c3                  ret
```

Kernel Control Flow Guard

- KCFG in its current state (no XFG) works just like “traditional” CFG, but recent changes (since 24H2) due to a feature called “hot patching” have slightly altered mechanics
  - **nt!KscpCfgDispatchUserCallTargetEs[No]Smep** is the new dispatch function, and it is now made through a *direct* call (no longer called indirectly via IAT)
- Other interesting notes
  - KCFG acts as a “software SMEP” – meaning even when HVCI is **DISABLED** (which means KCFG is also not fully enabled) KCFG will *still* validate that kernel-mode indirect calls never invoke a user-mode address (even with U/S bit set to supervisor in the PTE!)
  - Import Address Table (IAT) indirect calls are explicitly documented as not protected by (K)CFG – and this has been abused by attackers! Since this is the case, not even XFG could help…


//
// 1. Get the IAT's PTE
//
leakedIatPte = LeakPteAddressWithExploit(&leakedIatAddress);
//
// 2. Make the IAT writable
//
WritePrimitive(&leakedIatPte, writablePteMask);
//
// 3. Update the IAT with a ROP gadget
//
WritePrimitive(&leakedIatAddress, ropGadget);
//
// 4. When the import is called, the ROP gadget is now invoked
//
ForceIatCallViaIoctl();
kernel_exploit.exe
1. Make IAT writable (HVCI is not applicable here)
2. Arbitrary write primitive to corrupt the IAT
3. Invoke the import (executes the ROP gadget)
User mode
Kernel mode
NTSTATUS
IoctlHandler (
_In_ PIRP Irp,
_In_ PIO_STACK_LOCATION IrpSp
)
{
//
// TRUNCATED
//
//
// call [__imp__nt!ExAllocatePool2]
//
buffer = ExAllocatePool2(POOL_FLAG_NON_PAGED,
PAGE_SIZE
MY_POOL_TAG);
if (buffer == NULL)
{
goto Exit;
}
//
// TRUNCATED
//
}
Driver.sys
msprc!NdrClientCall3
ksecdd!BCryptGenRandom
CLFS!ClfsGetLogFileInformation
nt!ExAllocatePool2
nt!PsGetCurrentProcessId
nt!MmProbeAndLockPages
rop_gadget
Import Address Table (Driver.sys)
PTE:-G-A-KW-V

Kernel Control Flow Guard

- In the case of IAT abuse KCFG can be “combined” with a mitigation known as Retpoline (developed by Google) which mitigates Specter Type 2 (CVE-2017-5715)
  - KM images can use Retpoline with undocumented **/guard:retpoline** and **/d2guardretpoline** linker and compiler flags
- Retpoline does many things, but importantly for us it replaces indirect IAT calls with *direct* calls to a special Retpoline dispatch function (which in 99% of cases, via “import optimization”, just calls the target directly)
  - Even though newer CPUs do not use Retpoline (Indirect Branch Restricted Speculation, IBRS), import optimization is still always available and Windows images still use it, even when Retpoline is not enabled!

```
Ntfs!TxfCloseHandlesForTransaction+0x34:
fffff803`311bb5a8 488bcf          mov     rcx,rdi
fffff803`311bb5ab e8008fe5ff      call    Ntfs!NtfsPurgeFileRecordCache (fffff803`310144b0)
fffff803`311bb5b0 0fba6f0409      bts     dword ptr [rdi+4],9
fffff803`311bb5b5 488b4e08        mov     rcx,qword ptr [rsi+8]
fffff803`311bb5b9 4c8b1550eeeeff  mov     r10,qword ptr [Ntfs!_imp_ZwClose (fffff803`310aa410)]
fffff803`311bb5c0 e88b59ce6c      call    nt!ZwClose (fffff803`9dea0f50)
```

Kernel Control Flow Guard

- No more reading call targets from the IAT!

```
fffff805`20625074 41b8fded7702        mov     r8d, 277EDFDh
fffff805`2062507a ba00100000          mov     edx, 1000h
fffff805`2062507f b900010000          mov     ecx, 100h
fffff805`20625084 4c8b1575cfffff      mov     r10, qword ptr [NotAVulnerableDriver!__imp_ExAllocatePool2 (fffff80520622000)]
fffff805`2062508b e860201469          call    ntkrnlmp!ExAllocatePool2 (fffff805897670f0)
fffff805`20625090 4889442430          mov     qword ptr [buffer (rsp+30h)], rax
fffff805`20625095 48837c243000        cmp     qword ptr [buffer (rsp+30h)], 0
fffff805`2062509b 7417                je      NotAVulnerableDriver!DriverEntry+0xb4 (fffff805206250b4)
fffff805`2062509d bafded7702          mov     edx, 277EDFDh
fffff805`206250a2 488b4c2430          mov     rcx, qword ptr [buffer (rsp+30h)]
fffff805`206250a7 4c8b155acfffff      mov     r10, qword ptr [NotAVulnerableDriver!__imp_ExFreePoolWithTag (fffff80520622008)]
fffff805`206250ae e81d2c1469          call    ntkrnlmp!ExFreePoolWithTag (fffff80589767cd0)

Command
3: kd> u @rip L1
NotAVulnerableDriver!DriverEntry+0x8b [C:\Users\conno\source\repos\NotAVulnerableDriver\Main.cpp @ 69]:
fffff805`2062508b e860201469      call    nt!ExAllocatePool2 (fffff805`897670f0)
3: kd> r r10
r10=4141414141414141
3: kd> p
NotAVulnerableDriver!DriverEntry+0x90:
fffff805`20625090 4889442430      mov     qword ptr [rsp+30h],rax
2: kd> !pool @rax
Pool page ffff8188427fc000 region is Paged pool
```

- You must be “eligible” for import optimization
  - Both the caller and callee must be from images compiled with **/guard:retpoline** and **/d2guardretpoline**
  - Caller and callee must be within 2 GB of each other

Kernel Control Flow Guard

- But what if attackers wanted to use return address corruption to circumvent KCFG?
  - Example: An attacker-controlled thread is suspended, the stack is corrupted, and on thread resume a ROP gadget is invoked

Command

```
0: kd> dx -r2 @$curprocess.Threads.Where(t => (nt!_KWAIT_REASON)t.KernelObject.Tcb.WaitReason == nt!_KWAIT_REASON::Suspended).Select(t => t.Stack.Frames)
@$curprocess.Threads.Where(t => (nt!_KWAIT_REASON)t.KernelObject.Tcb.WaitReason == nt!_KWAIT_REASON::Suspended).Select(t => t.Stack.Frames)
    [0x29dc]
        [0x0]            : nt!KiSwapContext + 0x76 [Switch To]
        [0x1]            : nt!KiSwapThread + 0x6a0 [Switch To]
        [0x2]            : nt!KiCommitThreadWait + 0x271 [Switch To]
        [0x3]            : nt!KeWaitForSingleObject + 0x773 [Switch To]
        [0x4]            : nt!KiSchedulerApc + 0xf7 [Switch To]
        [0x5]            : nt!KiDeliverApc + 0x22d [Switch To]
        [0x6]            : nt!KiApcInterrupt + 0x3ab [Switch To]
        [0x7]            : nt!PspUserThreadStartup + 0x26 [Switch To]
        [0x8]            : nt!KiStartUserThread + 0x28 [Switch To]
        [0x9]            : nt!KiStartUserThreadReturn [Switch To]
        [0xa]            : ntdll!RtlUserThreadStart [Switch To]
```

Before stack corruption

```
0: kd> dx -r2 @$curprocess.Threads.Where(t => (nt!_KWAIT_REASON)t.KernelObject.Tcb.WaitReason == nt!_KWAIT_REASON::Suspended).Select(t => t.Stack.Frames)
@$curprocess.Threads.Where(t => (nt!_KWAIT_REASON)t.KernelObject.Tcb.WaitReason == nt!_KWAIT_REASON::Suspended).Select(t => t.Stack.Frames)
    [0x29dc]
        [0x0]            : nt!KiSwapContext + 0x76 [Switch To]
        [0x1]            : nt!KiSwapThread + 0x6a0 [Switch To]
        [0x2]            : nt!KiCommitThreadWait + 0x271 [Switch To]
        [0x3]            : nt!KeWaitForSingleObject + 0x773 [Switch To]
        [0x4]            : nt!KiSchedulerApc + 0xf7 [Switch To]
        [0x5]            : nt!KiDeliverApc + 0x22d [Switch To]
        [0x6]            : nt!DbgUserBreakPoint + 0xf [Switch To]
0: kd> g
Break instruction exception - code 80000003 (first chance)
nt!DbgBreakPointWithStatus:
fffff805`bb1025e0 cc              int     3
0: kd> k
 # Child-SP          RetAddr               Call Site
00 fffff30c`33147130 00000000`00000000     nt!DbgBreakPointWithStatus
0: kd> dx @$curthread.Id
@$curthread.Id                 : 0x29dc
```

After stack corruption (breakpoint ROP gadget reached)

Kernel CET

- That's where KCET comes in! KM return addresses are now protected!
  - Available since Windows 11 22H2 and, as is the case with KCFG, HVCI is required

```
Command
lkd> dx *(nt!_MI_FLAGS*)&nt!MiFlags
*(nt!_MI_FLAGS*)&nt!MiFlags                 [Type: _MI_FLAGS]
    [+0x000 ( 0: 0)] VerifierEnabled   : 0x0 [Type: unsigned __int64]
    [+0x000 ( 1: 1)] KernelVerifierEnabled : 0x0 [Type: unsigned __int64]
    [+0x000 ( 2: 2)] LargePageKernel   : 0x1 [Type: unsigned __int64]
    [+0x000 ( 3: 3)] StopOn4d          : 0x1 [Type: unsigned __int64]
    [+0x000 ( 5: 4)] InitializationPhase : 0x2 [Type: unsigned __int64]
    [+0x000 ( 6: 6)] PageKernelStacks  : 0x1 [Type: unsigned __int64]
    [+0x000 ( 7: 7)] CheckZeroPages    : 0x0 [Type: unsigned __int64]
    [+0x000 ( 8: 8)] ProcessorPrewalks : 0x0 [Type: unsigned __int64]
    [+0x000 ( 9: 9)] ProcessorPostwalks : 0x1 [Type: unsigned __int64]
    [+0x000 (10:10)] CoverageBuild     : 0x0 [Type: unsigned __int64]
    [+0x000 (11:11)] CheckExecute      : 0x1 [Type: unsigned __int64]
    [+0x000 (12:12)] ProtectedPagesEnabled : 0x1 [Type: unsigned __int64]
    [+0x000 (13:13)] SecureRelocations : 0x1 [Type: unsigned __int64]
    [+0x000 (14:14)] StrongPageIdentity : 0x1 [Type: unsigned __int64]
    [+0x000 (15:15)] StrongCodeGuarantees : 0x1 [Type: unsigned __int64]
    [+0x000 (16:16)] HardCodeGuarantees : 0x0 [Type: unsigned __int64]
    [+0x000 (17:17)] ExecutePagePrivilegeRequired : 0x0 [Type: unsigned __int64]
    [+0x000 (18:18)] SecureKernelCfgEnabled : 0x1 [Type: unsigned __int64]
    [+0x000 (19:19)] FullHvci          : 0x0 [Type: unsigned __int64]
    [+0x000 (20:20)] BootDebuggerActive : 0x0 [Type: unsigned __int64]
    [+0x000 (22:21)] KvaShadow         : 0x0 [Type: unsigned __int64]
    [+0x000 (23:23)] ExceptionHandlingReady : 0x1 [Type: unsigned __int64]
    [+0x000 (24:24)] ShadowStacksSupported : 0x1 [Type: unsigned __int64]
    [+0x000 (25:25)] AccessBitFenceRequired : 0x0 [Type: unsigned __int64]
```

```
//
// Shadow stack policy
//
if ( isHvciEnabled )
  OslGetEffectiveKernelShadowStacksConfiguration(regHiveHandle, &isKernelCetEnabled, &isKernelCetAuditModeEnabled);
```

black hat BRIEFINGS

Kernel CET

- The "kernel shadow stack" region of memory (**MiVaKernelShadowStacks**) is maintained by NT
  - Secure system call is made to SK to mark the shadow stack pages as read-only (plus a special "supervisor shadow stack bit" leveraged by an Intel feature called "Supervisor Shadow-Stack Control") in the EPTEs
    - We will talk about Shadow-Stack Control later!

```
NTSTATUS __fastcall VslAllocateKernelShadowStack(
        unsigned __int64 TargetShadowStackBaseAddressFromPte,
        unsigned int NumberOfBytes,
        unsigned int StackType,
        __int64 ShadowStackPfns,
        unsigned int NumberOfPfns,
        unsigned __int64 *ShadowStack)
{
  NTSTATUS status; // eax
  _SHADOW_STACK_SECURE_CALL_ARGS secureSystemCallArgs; // [rsp+20h] [rbp-A8h] BYREF

  memset_0(&secureSystemCallArgs, 0LL, sizeof(secureSystemCallArgs));
  if ( NumberOfPfns > 9 )
    return STATUS_INVALID_PARAMETER;
  secureSystemCallArgs.ShadowStackBase = TargetShadowStackBaseAddressFromPte;
  secureSystemCallArgs.NumberOfBytes = NumberOfBytes;
  secureSystemCallArgs.StackType = StackType;
  secureSystemCallArgs.NumberOfPfns = NumberOfPfns;
  memmove(secureSystemCallArgs.PfnArray, ShadowStackPfns, 8LL * NumberOfPfns);
  status = VslpEnterIumSecureMode(2, 230LL, 0LL, &secureSystemCallArgs);
  if ( status >= STATUS_SUCCESS )

    // VTL 1 uses the 64-bit address where NumberOfBytes and
    // NumberOfPfns is stored to return the starting address
    // of the shadow stack
    *ShadowStack = *&secureSystemCallArgs.NumberOfBytes;
  return status;
```

Kernel CET

- Each shadow stack (also referred to as SS) receives a static NAR, but there is caching/re-use logic to not incur the cost of SK interaction on every stack creation
  - Two caches: A per-processor (PRCB) and a per-NUMA node cache (both are managed by NT). If “ideal”, on stack deletion, shadow stacks are sent to one of these caches
  - “Slow” path results in calling into SK

```
if ( (Flags & 1) == 0 )
{
  if ( (Flags & 0x10) != 0 && targetPartition == &MiSystemPartition )
  {
    shadowStack = MiCreateKernelStackFromPrcbCache(&shadowStackAllocationArgs);
    if ( shadowStack )
      goto Exit;
  }
  shadowStack = MiCreateKernelStackFromNodeCache(&shadowStackAllocationArgs);
  if ( shadowStack )
    goto Exit;
  Flags = shadowStackAllocationArgs.StackFlags;
}
if ( (Flags & 0x40) != 0 )
  return STATUS_CANT_WAIT;
shadowStack = MiCreateKernelStackSlow(&shadowStackAllocationArgs);
if ( !shadowStack )
  return STATUS_INSUFFICIENT_RESOURCES;
```

Kernel CET

- On both “re-use” from the cache and the slow path the KTHREAD object is updated with the target shadow stack (including other values like shadow stack type)
  - In the “cached” path the backing PFN structure also has its “shadow stack owner data” updated (for debugging/info purposes)

Command

```
lkd> dx -g @$curprocess.Threads.Select(t => new { ShadowStack = t.KernelObject.Tcb.KernelShadowStack, KernelSSType = (nt!_KERNEL_SHADOW_STACK_TYPE)t.KernelObject.
```

	ShadowStack	KernelSSType
[0xc]	0xffffb983fc404fb8	KernelShadowStackTypeKernelThread (1)
[0x10]	0xffffb983fc407fb8	KernelShadowStackTypeKernelThread (1)
[0x14]	0xffffb983fc40afb8	KernelShadowStackTypeKernelThread (1)
[0x18]	0xffffb983fc40dfb0	KernelShadowStackTypeKernelThread (1)
[0x1c]	0xffffb983fc410fb0	KernelShadowStackTypeKernelThread (1)
[0x60]	0xffffb983fc4c1fb8	KernelShadowStackTypeKernelThread (1)

Command

```
lkd> dt nt!_KERNEL_SHADOW_STACK_TYPE
   KernelShadowStackTypeUserThread = 0n0
   KernelShadowStackTypeKernelThread = 0n1
   KernelShadowStackTypeRstorssp = 0n2
   KernelShadowStackTypeSetssbsy = 0n3
   KernelShadowStackTypeSetssbsyForSystemStartup = 0n4
   KernelShadowStackTypeMax = 0n5
```

```
3: kd> dx ((nt!_MMPFN*)0xffffc4000317eda0)->u1.EntireField
((nt!_MMPFN*)0xffffc4000317eda0)->u1.EntireField : 0x4909f1f6410 [Type: unsigned __int64]
3: kd> pt
nt!MiUpdateKernelShadowStackOwnerData+0x97:
fffff800`93aa577f c3              ret
0: kd> dx ((nt!_MMPFN*)0xffffc4000317eda0)->u1.EntireField
((nt!_MMPFN*)0xffffc4000317eda0)->u1.EntireField : 0x4909f395418 [Type: unsigned __int64]
0: kd> ? (0x4909f395418 << 0x3) + 0xFFFF800000000000
Evaluate expression: -100583943266112 = ffffa484`f9caa0c0
0: kd> !object 0xffffa484f9caa0c0
Object: ffffa484f9caa0c0  Type: (ffffa484f04a19c0) Thread
    ObjectHeader: ffffa484f9caa090 (new version)
    HandleCount: 3  PointerCount: 65501
```

Kernel CET

- …but before we update the thread object the Secure Kernel is responsible for first doing a few things (through the previously-mentioned secure system call), depending on the type of SSS which needs to be created:

1. Marking the region as read-only (and as supervisor shadow stack, also referred to as SSS) in the EPTEs
2. Configuring the "shadow stack token"
   - The token is used to validate the shadow stack and denote the state (busy/free)
     - A "restore" token can also be used in conjunction w/ rstorssp/saveprevssp instructions
     - Context switch pattern for SSS: After the Secure Kernel makes the SSS immutable in VTL 0, the restore token from the SSS associated with the new thread's thread object is used to update the SSP in VTL 0 (note that winnt.h specifies CET_S XSAVE area is not used by NT!)
3. Initializing the return address

Kernel CET

- One of the main engineering hurdles with KCET is writes to the shadow stack (SS)
  - "Ordinary" data-writes (like mov) are not a problem (without SLAT/hypervisor) – WRSSQ is also only enabled in audit and/or debug mode (undefined instruction otherwise!)
  - PML4 -> PD paging structures are writable, PTE is read-only + dirty bit set
    - "Special" combination of PTE states which denote this is a SS page (thus disallowing data writes)
  - But what about with SLAT? KCET is only enabled when HVCI is!
    - How does the hypervisor know what pages are SSS pages (Hypervisor uses EPTEs!)
    - If the SS is read-only in the VTL 0 EPTEs this would result in an EPT violation (incurring a VM exit)
  - We can use our "special" SSS bit we talked about earlier! Supervisor Shadow-Stack Control to the rescue!

Kernel CET

- The Supervisor Shadow-Stack Control feature uses a special “SSS bit” in the EPTEs to address our problems (and provide more security!)
  - Allows the hypervisor to denote SSS pages
    - As we know VTL 0, via secure system call, tells VTL 1 about pages that need to be marked as SSS
  - EPT PML4 -> EPT PT (all extended paging structures) have the read bit set, EPT PML4 -> EPT PD have the write bit set, and the EPT PT (EPT PTE) has the SSS bit set
    - Just like in the “without the hypervisor” scenario, this combination of PTE states allows legitimate shadow-stack writes but not ordinary writes (mov, etc. – no EPT violation!)
  - Enforces that shadow-stack accesses cannot occur to a non-shadow stack page (prevents attackers re-mapping SSS pages to arbitrary non-SSS pages via PTEs to construct fake SSs)
- Windows is the only platform leveraging this today!

Kernel CET

- KCET needs to handle other legitimate SSS updates (like returning from an exception)
  - But we don't want to let VTL 0 handle this. The solution is to let the Secure Kernel do it!
    - CET MSRs allow the CPU to load SS values when a privilege level switch occurs (IA32_PLX_SSP) *and* when a hypervisor <-> guest context switch happens (VMX_GUEST_SSP)!
    - VM Entry controls defined by VTL 0's VMCS allows us to use the VMX_GUEST_SSP MSR value to populate the SSS on context switch (VMENTRY) back into VTL 0 (after the Secure Kernel returns)
  - During "assist", opportunistic checks are done as well, such as validating that RIP exists in an executable code region within a known kernel image tracked through a NAR in SK

```
if ( status < 0
  || (vtl0SspRegValue[0] = targetShadowStack,
      status = ShvlSetVpRegister(0xFFFFFFFE, 0, VMX_GUEST_SSP, vtl0SspRegValue),
      status < 0) )
{
  SKMI_SECURITY();
}
return status;
```

```
if ( (ExceptionType & 0xFFFFFFFD) != 0 )
{
  SKMI_SECURITY();
  return STATUS_INVALID_PARAMETER;
}
memset_0(kssAssistDispatchParameters, 0LL, sizeof(kssAssistDispatchParameters));
status = SkmiValidateRipWithLoadConfigTable(RipFromCtxRecord, ExceptionType);
```

```
// 0x10080 = CONTEXT_KERNEL_CET (CONTEXT_AMD64 | 0x80).
// This is part of CONTEXT_EX, which is undocumented, and
// is the latest addition (to the already existing CONTEXT +
// XSTATE context)
if ( (Context->ContextFlags & 0x100080) == 0x100080 )

  // The first member of the KERNEL_CET_CONTEXT is the shadow stack
  shadowStackPointerFromKernelCetContext = *(&Context->EndOfContextStructure
                                           + Context->ContextExOffsetToKernelCetContextShadowCtx);
else
  shadowStackPointerFromKernelCetContext = ShadowStackPointerFromRdsspq;
exceptionType = 0;
if ( ExceptionRecord && ExceptionRecord->ExceptionCode == 0x80000026 )
  // longjmp
  exceptionType = 2;
return VslKernelShadowStackAssist(
         1,
         0LL,
         ShadowStackPointerFromRdsspq,
         shadowStackPointerFromKernelCetContext,
         Context->Rip,
         exceptionType);
```

Kernel CET

- Other interesting notes
  - The CPU will fault (obviously) if the return addresses mismatch, but the interrupt handler on Windows will still allow a return into a different return address – so long as that return address is also on the shadow stack! (VTL 1 handles updating the SSP if this occurs)

```
returnAddress = *FaultingTrapFrame->Rsp;
if ( returnAddress >= 0xFFFF800000000000uLL )
{
  shadowStackBase = KiGetCurrentKernelShadowStackBounds(&shadowStackLimit);
  // Includes CS, RIP, and SSP
  shadowStackFrame = FaultingTrapFrame->ShadowStackFrame;
  errorCode = FaultingTrapFrame->ErrorCode;
  // Jump over CPU-issued SS trap frame information
  nextShadowStackAddress = (shadowStackFrame + 0x18);
  while ( 1 )
  {
    ++nextShadowStackAddress;
    if ( errorCode != 1 || nextShadowStackAddress >= shadowStackLimit )
      break;
    // Loop until we (hopefully) find a return address
    // on the shadow stack which matches the offending return address
    if ( *nextShadowStackAddress >= 0x10000uLL && *nextShadowStackAddress == returnAddress )
    {
      status = VslKernelShadowStackAssist(0, shadowStackFrame, 0LL, nextShadowStackAddress, 0LL, 4);
```

Conclusions

- KCET has only been available for a short time on Windows –execution research w/ both KCFG and KCET enabled is still limited
  - Most public research still revolves around known limitations in KCFG (JOP, COOP, calling other valid call targets) because IBT is not leveraged by Windows
- KCET seems to be the stronger of the two (hardware-enforced)
  - Out-of-context calls (calling into other valid SSP values) is an interesting vector for research!
- Remapping attacks are still possible
  - HLAT mitigates this (and is now available in 24H2!)
- The presence of HVCI, KCFG and KCET raises the bar for attackers, while also outright mitigating some primitives!
- It will be fun to see the “cat-and-mouse” game which follows!

Thank You!

- Greetz & shout-outs!
  - Alan Sguigna, Alex Ionescu, Andrea Allievi (special shout-out!), Satoshi Tanda, Yarden Shafir
- Additional resources
  - Yarden Shafir – OffensiveCon 23: https://www.youtube.com/watch?v=YnxGW8Fvqvk&t=751s
  - https://tandasat.github.io/blog/2025/04/02/sss.html
  - https://cdrdv2-public.intel.com/782161/326019-sdm-vol-3c.pdf
  - https://www.sstic.org/media/SSTIC2025/SSTIC-actes/windows_kernel_shadow_stack_mitigation/SSTIC2025-Article-windows_kernel_shadow_stack_mitigation-aulnette_jullian.pdf

[P] | Prelude