SANS

OFFENSIVE OPERATIONS

HACKFEST SUMMIT 2023

November 16 - 17

Hollywood, CA & FREE Live Online

Just Pocket Change? Examining The Cost Of “Nickel and Dime” Exploit Mitigations

Connor McGarr

HACKFEST
SUMMIT 2023

About Me

- Software Engineer @ CrowdStrike
  - EDR features, detection content, etc.
- Blog
  - https://connormcgarr.github.io
- I like low-level Windows stuff!

What Are “Nickel and Dime” Mitigations?

- We all have that “spare change” jar
  - …well maybe you don’t, but my family does
- Although the spare change jar is full of coins that, by themselves, are seemingly insignificant one day you wake up and realize you have quite a lot of money in the jar!

What Are “Nickel and Dime” Mitigations?

- Enter “nickel and dime” mitigations
  - These are the “smaller” mitigations we may not give a second look
  - These mitigations also may not mean a lot “by themselves”
    - But when combined they can significantly raise the cost of exploitation, just like coins in a spare change jar!

Exploitation Trends

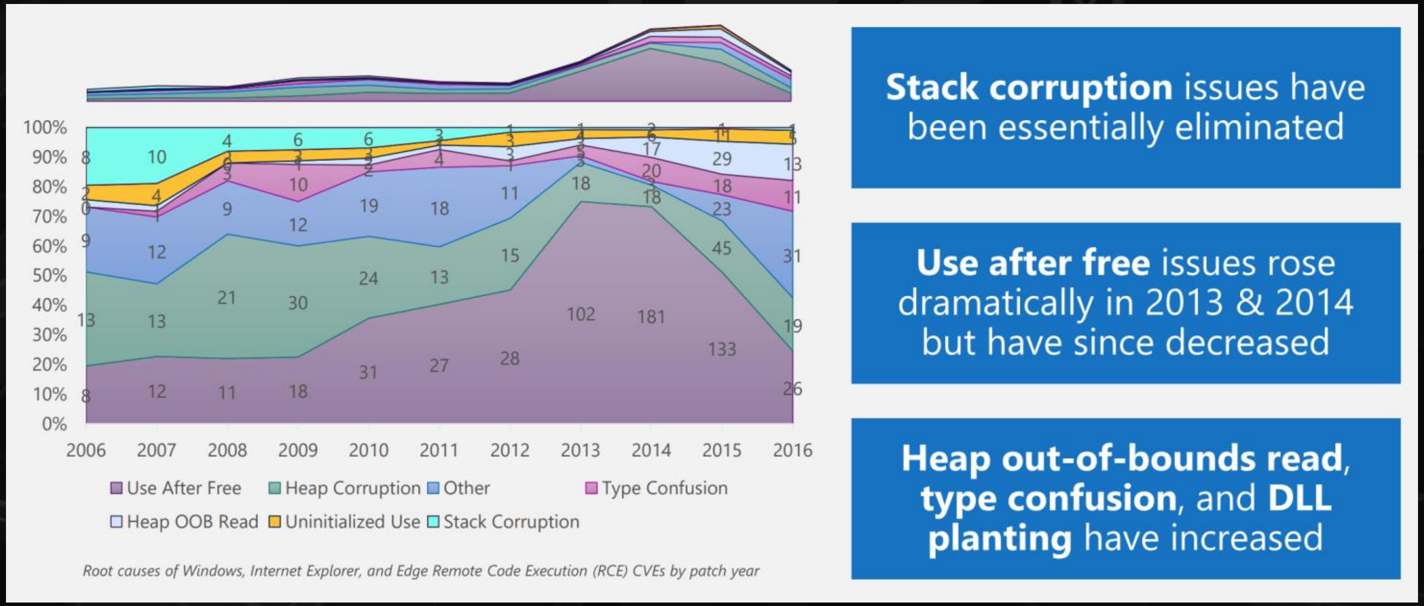
- Before we get into the specific mitigations themselves, it’s worth considering how exploitation has changed over time
- Exploit mitigations have been around for awhile
  - DEP/ASLR have been supported since Windows XP/Vista
  - Control Flow Integrity (CFG in the case of Microsoft) since Windows 8.1 as an optional update

Exploitation Trends

- The origin of mitigations…
  - Most exploits in “old” days are usually (but not always) stack overflows or use-after-free vulnerabilities
  - Both require making memory executable (mitigated by DEP)
  - Both require knowledge of the address space (mitigated by ASLR)
  - Use-after-frees (in many cases) specifically require crafting a fake object to overwrite a function pointer (mitigated by CFI)

Exploitation Trends

- DEP/ASLR/CFI, outlined by Microsoft in 2016, made a significant impact regarding stack corruption and use-after-free vulnerabilities
  - This success was used as an opportunity to invest heavily into “modern” mitigations around this time



Exploitation Trends

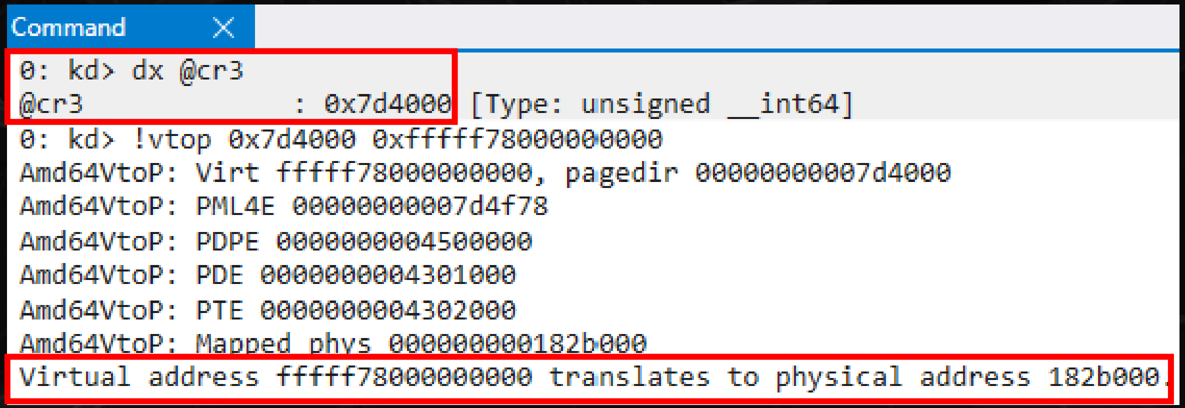
- This investment led to several well-known mitigations:
  - Arbitrary Code Guard (ACG)
    - Memory is either RW or RX, but never both
  - Code Integrity Guard
    - Only Microsoft signed DLLs can be loaded
  - Intel CET and eXtended Control Flow Guard (XFG)
    - Hijacking control flow is essentially impossible/very limited
  - Supervisor Mode Execution (and Access) Prevention (SMEP/SMAP)
    - Prevents redirecting execution/data access into user-mode, from kernel-mode (with some caveats not mentioned here)
  - Hypervisor Protected Code Integrity (HVCI)
    - Memory permissions are managed by hypervisor, can't create RWX memory dynamically in kernel-mode (the "ACG" of kernel-mode)
- ...and many more not mentioned!

Exploitation Trends

- Even with the heavy investment in mitigations, we know exploitation is still occurring. Why?
    1. Mitigations may still be disabled
    2. There are still other issues that need to be addressed
        - Generic memory leaks
        - Full implementation of things like ASLR in kernel-mode
        - Data-only corruption techniques
        - Other techniques which don’t rely on executable code creation or control-flow hijacking
- Today we will look at how some of the “smaller”, but often overlooked, mitigations affect exploitation cost and how they directly address some of these lingering issues

“Nickel and Dime” Mitigations

- Full ASLR Example 1: Page Table Randomization
  - Page tables maintain entries known as page table entries (PTEs)
    - PTEs are used to determine what physical frames correspond to what physical addresses, and to enforce virtual memory permissions (readable/writable/etc.)



“Nickel and Dime” Mitigations

- Full ASLR Example 1: Page Table Randomization
  - Page tables reside in physical memory
    - However, CPU instructions can only directly access virtual memory
      - This is a problem as paging structures need to constantly be updated/flushed/etc.
        - How can we access them if we are limited to accessing virtual memory only?
    - To compensate, the OS maps all the PTEs for each of the paging structures into virtual memory in the form of a “per-structure array”

Command

```
0: kd> !pte 0xfffff78000000000
                                           VA fffff78000000000
PXE at FFFFF8FC7E3F1F78    PPE at FFFFF8FC7E3EF000    PDE at FFFFF8FC7DE00000    PTE at FFFFF8FBC0000000
contains 0000000004500063  contains 0000000004301063  contains 0000000004302063  contains 800000000182B963
pfn 4500      ---DA--KWEV  pfn 4301      ---DA--KWEV  pfn 4302      ---DA--KWEV  pfn 182b      -G-DA--KW-V
```

  - Because each of these PTE arrays is needed early in the OS loading processes, they were stored at a static address

Address	Function	Instruction
INIT:0000000140BC3F79	MiInitSystem	mov rax, 0FFFFF68000000000h

Static PTE array for 4KB-aligned pages

“Nickel and Dime” Mitigations

- Why is this bad?
  - An adversary, **without** a read primitive, can correlate a memory address to its PTE
    1. Obtain the virtual page number (VPN) by dividing the virtual address by sizeof(PAGE_SIZE)
    2. Obtain the index into the array (multiply the VPN by sizeof(PTE))
    3. Index the array (add the index to the address of the PTE array)
  - With the address of the PTE, and a write primitive, an attacker can corrupt the PTE metadata, which enforces virtual memory permissions (RWX, RW, RX, copy-on-write, etc.)

nt!MiGetPteAddress
performs this with compiler
optimizations

```
; __int64 __fastcall MiGetPteAddress(unsigned __int64)
MiGetPteAddress proc near
shr     rcx, 9
mov     rax, 7FFFFFFFF8h
and     rcx, rax
mov     rax, 0FFFFF68000000000h
add     rax, rcx
retn
MiGetPteAddress endp
```

“Nickel and Dime” Mitigations

- With the array being randomized, the process for locating a page table entry doesn’t change, but it *requires* a read primitive

1. Fetch virtual page number
2. Grab the index (VPN * 0x8) and add the PTE base

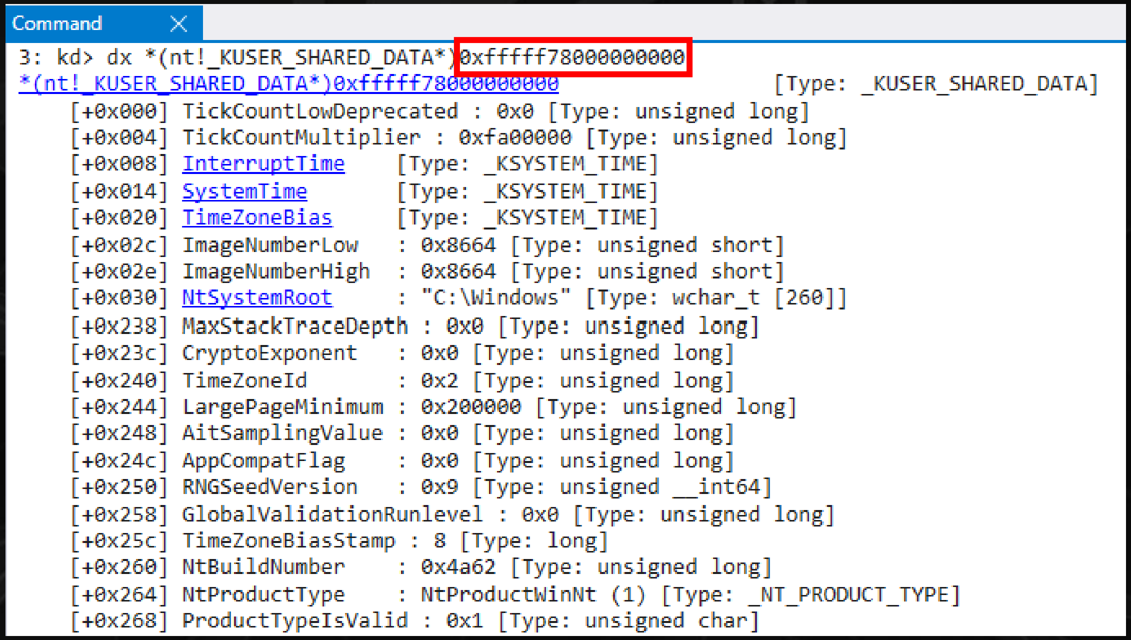
@$pteBase is now randomized and needs to **dynamically** be found

Physical frames match (this is the PTE for **0x7ff65e543600** (this is the correct PTE)

```
Command
3: kd> u 0x7ff65e543600 L3
csrss!HotPatchSpareGlobal7+0x8:
00007ff6`5e543600 90              nop
00007ff6`5e543601 90              nop
00007ff6`5e543602 90              nop
3: kd> dx @$vpn = (0x7ff65e543600 / 0x1000)
@$vpn = (0x7ff65e543600 / 0x1000) : 34349638979
3: kd> dx @$pteAddr = (@$vpn * 0x8) + @$pteBase
@$pteAddr = (@$vpn * 0x8) + @$pteBase : -7971540096488
3: kd> dx *(nt!_MMPTE_HARDWARE*)@$pteAddr
*(nt!_MMPTE_HARDWARE*)@$pteAddr                 [Type: _MMPTE_HARDWARE]
    [+0x000 ( 0: 0)] Valid            : 0x1 [Type: unsigned __int64]
    [+0x000 ( 1: 1)] Dirty1           : 0x1 [Type: unsigned __int64]
    [+0x000 ( 2: 2)] Owner            : 0x1 [Type: unsigned __int64]
    [+0x000 ( 3: 3)] WriteThrough     : 0x0 [Type: unsigned __int64]
    [+0x000 ( 4: 4)] CacheDisable     : 0x0 [Type: unsigned __int64]
    [+0x000 ( 5: 5)] Accessed         : 0x1 [Type: unsigned __int64]
    [+0x000 ( 6: 6)] Dirty            : 0x1 [Type: unsigned __int64]
    [+0x000 ( 7: 7)] LargePage        : 0x0 [Type: unsigned __int64]
    [+0x000 ( 8: 8)] Global           : 0x0 [Type: unsigned __int64]
    [+0x000 ( 9: 9)] CopyOnWrite      : 0x0 [Type: unsigned __int64]
    [+0x000 (10:10)] Unused           : 0x0 [Type: unsigned __int64]
    [+0x000 (11:11)] Write            : 0x1 [Type: unsigned __int64]
    [+0x000 (47:12)] PageFrameNumber  : 0x289c8 [Type: unsigned __int64]
    [+0x000 (51:48)] ReservedForHardware : 0x0 [Type: unsigned __int64]
    [+0x000 (55:52)] ReservedForSoftware : 0x0 [Type: unsigned __int64]
    [+0x000 (59:56)] WsleAge          : 0x2 [Type: unsigned __int64]
    [+0x000 (62:60)] WsleProtection   : 0x0 [Type: unsigned __int64]
    [+0x000 (63:63)] NoExecute        : 0x1 [Type: unsigned __int64]
3: kd> !pte 0x7ff65e543600
                                           VA 00007ff65e543600
PXE at FFFFF8FC7E3F17F8    PPE at FFFFF8FC7E2FFEC8    PDE at FFFFF8FC5FFD9790    PTE at FFFFF8BFFB2F2A18
contains 0A0000002848A867  contains 0A0000002848B867  contains 0A0000002848C867  contains 82000000289C8867
pfn 2848a     ---DA--UWEV  pfn 2848b     ---DA--UWEV  pfn 2848c     ---DA--UWEV  pfn 289c8     ---DA--UW-V
```

“Nickel and Dime” Mitigations

- Full ASLR Example 2: KUSER_SHARED_DATA
  - Attackers need to write malicious code somewhere into memory
  - The Windows kernel offers a commonly-abused technique in the form a static writable cave to store shellcode with a write primitive
    - KUSER_SHARED_DATA is located at 0xfffff78000000000



```
Command
3: kd> dx *(nt!_KUSER_SHARED_DATA*)0xfffff78000000000
*(nt!_KUSER_SHARED_DATA*)0xfffff78000000000                 [Type: _KUSER_SHARED_DATA]
    [+0x000] TickCountLowDeprecated : 0x0 [Type: unsigned long]
    [+0x004] TickCountMultiplier : 0xfa00000 [Type: unsigned long]
    [+0x008] InterruptTime       [Type: _KSYSTEM_TIME]
    [+0x014] SystemTime          [Type: _KSYSTEM_TIME]
    [+0x020] TimeZoneBias        [Type: _KSYSTEM_TIME]
    [+0x02c] ImageNumberLow     : 0x8664 [Type: unsigned short]
    [+0x02e] ImageNumberHigh    : 0x8664 [Type: unsigned short]
    [+0x030] NtSystemRoot       : "C:\Windows" [Type: wchar_t [260]]
    [+0x238] MaxStackTraceDepth : 0x0 [Type: unsigned long]
    [+0x23c] CryptoExponent     : 0x0 [Type: unsigned long]
    [+0x240] TimeZoneId         : 0x2 [Type: unsigned long]
    [+0x244] LargePageMinimum   : 0x200000 [Type: unsigned long]
    [+0x248] AitSamplingValue   : 0x0 [Type: unsigned long]
    [+0x24c] AppCompatFlag      : 0x0 [Type: unsigned long]
    [+0x250] RNGSeedVersion     : 0x9 [Type: unsigned __int64]
    [+0x258] GlobalValidationRunlevel : 0x0 [Type: unsigned long]
    [+0x25c] TimeZoneBiasStamp : 8 [Type: long]
    [+0x260] NtBuildNumber      : 0x4a62 [Type: unsigned long]
    [+0x264] NtProductType      : NtProductWinNt (1) [Type: _NT_PRODUCT_TYPE]
    [+0x268] ProductTypeIsValid : 0x1 [Type: unsigned char]
```

“Nickel and Dime” Mitigations

- Full ASLR Example 2: KUSER_SHARED_DATA
  - Attackers only need a write primitive
    - This is especially true/useful for remote exploits, as KASLR is an issue (KASLR is a non-issue for local attacks, but that is changing with one of the upcoming mitigations outlined in this talk)
  - Since most memory regions/operations are page-aligned on Windows, and since KUSER_SHARED_DATA is ~ 0x700 bytes, that leaves ~0x300 bytes of free memory

```
Command
3: kd> ?? sizeof(nt!_KUSER_SHARED_DATA)
unsigned int64 0x720
```

```
3: kd> dps 0xfffff78000000000+0x720
fffff780`00000720  00000000`00000000
fffff780`00000728  00000000`00000000
fffff780`00000730  00000000`00000000
fffff780`00000738  00000000`00000000
```

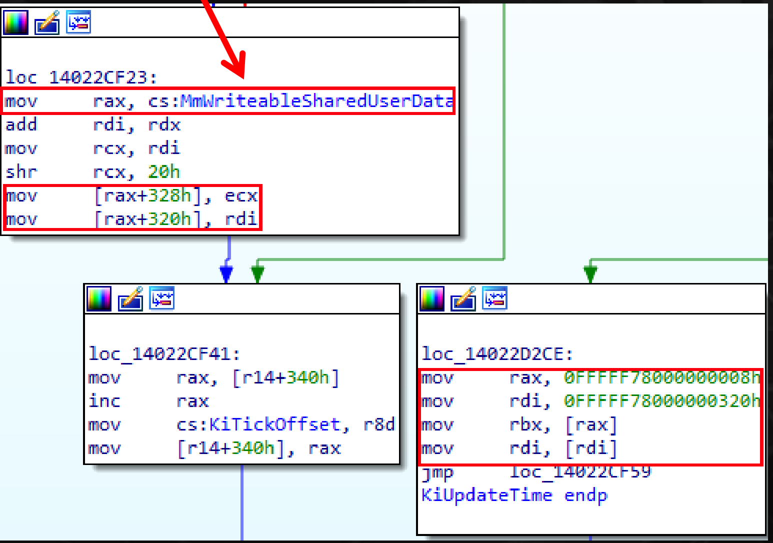
```
3: kd> !pte 0xfffff78000000000
                                           VA fffff78000000000
PXE at FFFFF8FC7E3F1F78    PPE at FFFFF8FC7E3EF000    PDE at FFFFF8FC7DE00000    PTE at FFFFF8FBC0000000
contains 0000000004500063  contains 0000000004301063  contains 0000000004302063  contains 800000000182B963
pfn 4500      ---DA--KWEV  pfn 4301      ---DA--KWEV  pfn 4302      ---DA--KWEV  pfn 182b      -G-DA--KW-V
```

“Nickel and Dime” Mitigations

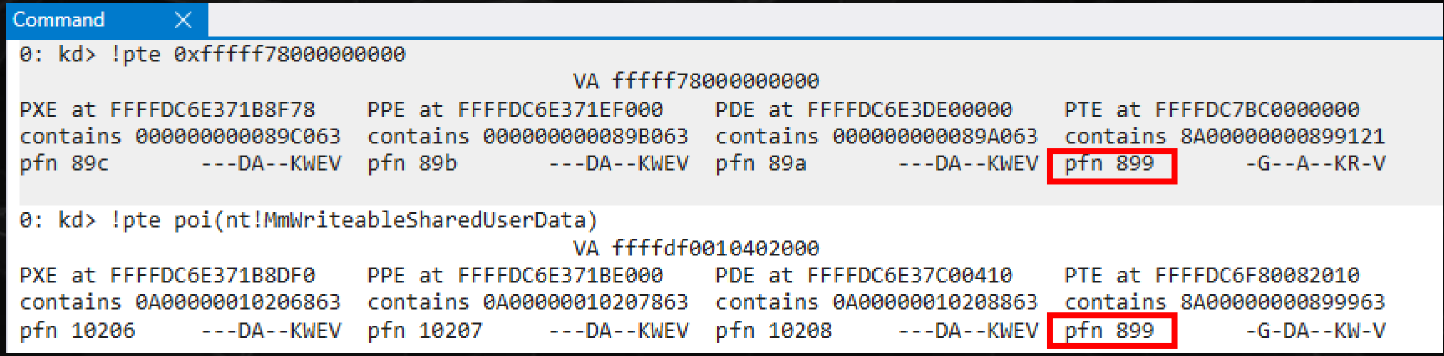
- Full ASLR Example 2: KUSER_SHARED_DATA
  - In the Windows Insider Preview builds, KUSER_SHARED_DATA has a new randomized write-only mapping
    - For read operations, the “old” KUSER_SHARED_DATA is used, as it is read-only
    - For write operations to any of the fields in KUSER_SHARED_DATA (like updating SystemTime), a new mapping of KUSER_SHARED_DATA (nt!MmWriteableSharedUserData) is used

Write operation

Read operation



New mapping is backed by the same physical page. Any write to the new mapping is automatically updated in the read-only mapping



"Nickel and Dime" Mitigations

- Restricted Caller Improvements*
  - ExIsRestrictedCaller was introduced in Windows 8.1 to, for example, block low integrity processes from invoking sensitive APIs that leak kernel addresses by design

* Not the official name of the mitigation

“Nickel and Dime” Mitigations

- ExIsRestrictedCaller now uses SeDebugPrivilege to determine if kernel addresses can be returned to any caller – not just low integrity processes
  - Effectively this means only admins should be allowed to leak kernel addresses (Microsoft doesn’t recognize an admin-to-kernel boundary)

```
bool __fastcall ExIsRestrictedCaller(_KPROCESSOR_MODE PreviousMode, bool *DisallowKernelAddresses)
{
  bool v4; // bl
  _SECURITY_SUBJECT_CONTEXT secSubjectContext; // [rsp+50h] [rbp-28h] BYREF
  int status; // [rsp+80h] [rbp+8h] BYREF
  int grantedAccessMask; // [rsp+88h] [rbp+10h] BYREF

  status = 0;
  grantedAccessMask = 0;
  memset(&secSubjectContext, 0, sizeof(secSubjectContext));
  if ( DisallowKernelAddresses )
    *DisallowKernelAddresses = 0;
  if ( PreviousMode == KernelMode )
    return 0;
  if ( DisallowKernelAddresses )
  {
    Feature_RestrictKernelAddressLeaks__private_IsEnabledPreCheck();
    *DisallowKernelAddresses = !SeSinglePrivilegeCheck(SeDebugPrivilege, PreviousMode);
  }
```

"Nickel and Dime" Mitigations

- How does one get kernel addresses from user mode?
  - ntdll!NtQuerySystemInformation -> nt!NtQuerySystemInformation -> nt!ExpQuerySystemInformation is the code path taken to leak kernel object addresses to user-mode via the SystemHandleInformation class
  - Even if nt!ExpQuerySystemInformation doesn't identify a low integrity process, ExIsRestrictedCaller can still override if kernel addresses are allowed
    - Meaning we can restrict even "normal" medium-integrity processes

```
IsRestrictedCaller = ExIsRestrictedCaller(PreviousMode, &disallowKernelAddresses);
if ( !*&IsRestrictedCaller )
{
  SystemBasicInformation = ExpGetHandleInformation(disallowKernelAddresses, userBuffer, v158, &v157);
  EtwTiLogSyscallUsage(SystemBasicInformation, 4u);
  goto LABEL_40;
}
return 0xC0000022i64;                    // STATUS_ACCESS_DENIED
```

“Nickel and Dime” Mitigations

- How are addresses removed?
  - Example: ObpCaptureHandleInformation
    - Eventually called when a caller requests all system handle information (usually to leak an EPROCESS or KTHREAD object in terms of exploitation)

```
__int64 __fastcall ObpCaptureHandleInformation(
        bool DisallowKernelAddresses,
        _SYSTEM_HANDLE_TABLE_ENTRY_INFO **HandleEntryInfoBuffer,
        __int16 ProcessId,
        void *HandleTableEntry,
        unsigned __int16 HandleIndex,
        unsigned int Length,
        unsigned int *RequiredLength)
{
  unsigned int v11; // eax
  unsigned int status; // r9d
  unsigned __int64 HandlePointer; // r10
  void *actualKernelObjectAddress; // rdx
  char handleTableEntry; // r8

  v11 = *RequiredLength + 24;
  *RequiredLength = v11;
  if ( v11 >= 0x18 )
  {
    if ( Length >= v11 )
    {
      HandlePointer = ExGetHandlePointer(HandleTableEntry);
      LOWORD((*HandleEntryInfoBuffer)->ProcessId) = ProcessId;
      status = 0;
      actualKernelObjectAddress = 0i64;
      handleTableEntry = (*HandleTableEntry >> 17) & 7 | 8;
      if ( (*(HandleTableEntry + 2) & 0x2000000) == 0 )
        handleTableEntry = (*HandleTableEntry >> 17) & 7;
      (*HandleEntryInfoBuffer)->Flags = handleTableEntry & 7;
      (*HandleEntryInfoBuffer)->ObjectTypeNumber = *(ObTypeIndexTable[ObHeaderCookie ^ *(HandlePointer + 24) ^ BYTE1(HandlePointer)]
                                                   + 40);
      (*HandleEntryInfoBuffer)->Handle = HandleIndex;
      if ( !DisallowKernelAddresses )
        actualKernelObjectAddress = (HandlePointer + 0x30);// Compensate for the OBJECT_HEADER (0x30 bytes)
      (*HandleEntryInfoBuffer)->Object = actualKernelObjectAddress;// This is NULL if kernel address leaks are not allowed
      HIWORD((*HandleEntryInfoBuffer)->ProcessId) = 0;
      (*HandleEntryInfoBuffer)[1].ProcessId = *(HandleTableEntry + 2) & 0x1FFFFFF;
      *HandleEntryInfoBuffer = (*HandleEntryInfoBuffer + 24);
    }
```

Default kernel address returned in 0 (NULL)

Only if kernel addresses are allowed are kernel addresses returned to the caller

“Nickel and Dime” Mitigations

- Why should we care?
  - Previously, adversaries could use NtQuerySystemInformation from a medium integrity processes (which are what processes spawn as by default) to easily grab specific EPROCESS or KTHREAD objects
  - With the new improvements, you must have SeDebugPrivilege – which effectively means you need to be admin to leak kernel addresses
    - Admin -> kernel is not a security boundary! If you have admin, you have achieved privilege escalation already

"Nickel and Dime" Mitigations

- PreviousMode Corruption Mitigation
  - Each thread object managed by the kernel (KTHREAD) has a member called PreviousMode
    - Value comes from the KPROCESSOR_MODE enum
    - KernelMode = 0
    - UserMode = 1
  - KernelMode
    - Caller originated from a kernel thread
  - UserMode
    - Caller originated from a user thread

C++

```
typedef enum _KPROCESSOR_MODE {
    KernelMode,
    UserMode,
    MaximumMode
} KPROCESSOR_MODE;
```

“Nickel and Dime” Mitigations

- Why is this important?
  - PreviousMode is used in a variety of important security checks when the OS does things like process user-input from a system call, requests/provides a handle, etc.
- Example: NtTerminateProcess
  - Process object lookup involves PreviousMode

```
if ( processHandle )
{
  result = ObpReferenceObjectByHandleWithTag(
             processHandle,
             1,
             PsProcessType,
             CurrentThread->PreviousMode,
             0x65547350,
             &processObject,
             0i64,
             0i64);
```

“Nickel and Dime” Mitigations

- (Example cont’d) Handles are stored in a per-process handle table. If PreviousMode == KernelMode, the lookup happens in the System process handle table
  - To prevent abuse, the OS checks if the process handle resides within the “kernel handle range” and the calling thread’s PreviousMode
    - If a user-mode thread (PreviouMode == UserMode) is detected, it means a user-mode client is attempting to use a kernel-level handle (for example, a privileged object which a kernel-mode component requested) and the OS returns access denied to the user-mode caller

```
if ( ObjectType == PsThreadType || !ObjectType )
{
  if ( (ProcessHandle & 0xFFE00000) != 0 && PreviousMode )
  {
    return 0xC0000022;  STATUS_ACCESS_DENIED
  }
```

“Nickel and Dime” Mitigations

- With a kernel primitive such as an arbitrary decrement, or arbitrary write, it is possible to “trick” the OS into thinking a user-mode thread is a kernel-mode caller
  1. Prior to invoking a system call, locate the KTHREAD object associated with the thread the system call will be invoked on
  2. Decrement KTHREAD->PreviousMode to 0 (KernelMode)
  3. Invoke the system call
  4. The thread is now a “kernel-mode caller” and can access privileged objects, write/read to kernel memory, etc.

“Nickel and Dime” Mitigations

- With the latest Insider Preview builds, this is no longer the case
  - Firstly, as part of the system call dispatching logic, PreviousMode is explicitly set to UserMode, from the kernel

```
KiSystemServiceUser:
mov       byte ptr [rbp-55h], 2
mov       rbx, gs:188h
prefetchw byte ptr [rbx+90h]
stmxcsr dword ptr [rbp-54h]
ldmxcsr dword ptr gs:180h
cmp       byte ptr [rbx+3], 0
mov       word ptr [rbp+80h], 0
jz        loc_140405704
```

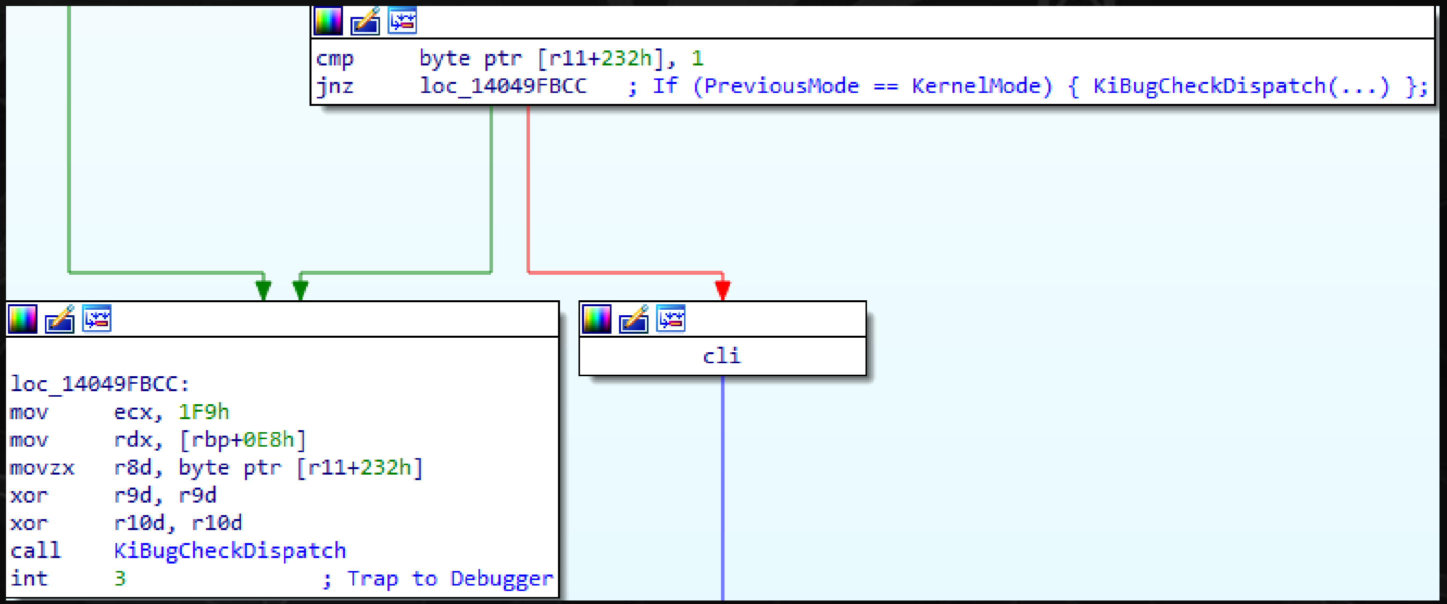
KTHREAD is fetched into RBX only (pre-mitigation)

```
KiSystemServiceUser:
mov       byte ptr [rbp-55h], 2
mov       byte ptr [rbp-58h], 1
mov       rbx, gs:188h
mov       byte ptr [rbx+232h], 1
prefetchw byte ptr [rbx+90h]
stmxcsr dword ptr [rbp-54h]
```

KTHREAD is fetched and PreviousMode (offset 0x232) is explicitly set to UserMode after the system call is invoked (mitigation enabled)

“Nickel and Dime” Mitigations

- In addition, as part of the sysret logic, PreviousMode is verified again
  - PreviousMode cannot be corrupted at the time of the system call invocation, and if it is, it must be restored before the sysret – effectively mitigating this technique



“Nickel and Dime” Mitigations

- Now, let’s look at some practical ways this thwarts exploitation

Demo #1

- In order to corrupt PreviousMode, we need a KTHREAD object
  - To get a KTHREAD object, without an already existing KASLR/arbitrary read primitive we would normally use NtQuerySystemInformation

```
do
{
    //
    // Free the last allocation
    //
    free(handleInformation);

    //
    // Increment the size.
    //
    size = size * 2;

    //
    // Re-alloc.
    //
    handleInformation = reinterpret_cast<PSYSTEM_HANDLE_INFORMATION>(malloc(size));

    if (handleInformation == nullptr)
    {
        goto Exit;
    }

    //
    // This may or may not be successful.
    //
    status = NtQuerySystemInformation(SystemHandleInformation,
                                      handleInformation,
                                      size,
                                      &outSize);
} while (status == STATUS_INFO_LENGTH_MISMATCH);
```

```
if (!NT_SUCCESS(status))
{
    wprintf(L"[-] Error in NtQuerySystemInformation call (0x%lx)!\n", status);
    goto Exit;
}

for (ULONG i = 0; i < handleInformation->NumberOfHandles; i++)
{
    if ((handleInformation->Handles[i].ProcessId != GetCurrentProcessId()) ||
        (handleInformation->Handles[i].Handle != reinterpret_cast<USHORT>(TargetThread)))
    {
        continue;
    }

    //
    // We found our target thread.
    //
    *ThreadObject = reinterpret_cast<ULONG64>(handleInformation->Handles[i].Object);

    result = true;
    break;
}
```

Demo #2

- Let's say you have a memory disclosure vulnerability, or you have some other primitive and you still manage to leak a KTHREAD object, as an example
  - Attempting to corrupt PreviousMode then invoke a "privileged system call" is effectively impossible

Conclusion

- Not every effective mitigation has to be a “huge”
- You may not even be aware of a given “nickel/dime” mitigation exists until an exploit starts to break on a new version of Windows
  - Take advantage of Insider Preview, Insider Preview SDK, etc. to look for potential upcoming changes and to look for trends!
- Even smaller mitigations have the potential, depending on your primitives as always, in significantly increase exploitation cost
- Happy exploit writing!

References

- https://www.alex-ionescu.com/kaslr-bypass-mitigations-in-windows-8-1/
- https://twitter.com/yarden_shafir/status/1644766239137357829
- https://twitter.com/GabrielLandau/status/1597001955909697536/
- https://github.com/hacksysteam/HackSysExtremeVulnerableDriver