

# Security Internals

Kelvin Chan

Microsoft

# WHO AM I

- Kelvin Chan
- Former Tencent Senior Security Researcher
- Microsoft Kernel Security Researcher

# Prerequisite

- x86 Assembly , C language
- PE file format
- Hooking
- Windows user mode application development

# Course Overview

- Windows Secure Driver Development
- Windows Internals for different parts
  - Input System
  - NTOS
  - Win32k
  - I/O
- Hardware based security solution
- Emulation based security solution

# OS overview

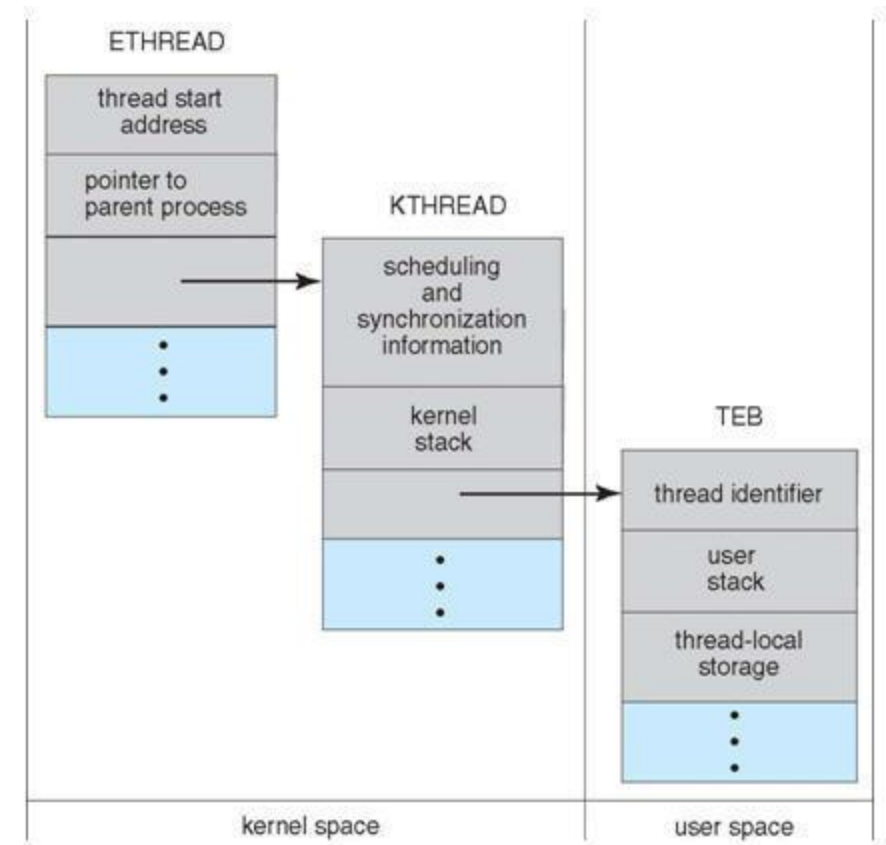
- Hardware
- Kernel Mode
- User Mode
- No matter what language you used, it turns on compile as platform (OS) application
- An OS application need to access OS API eventually
- NTOS is OS management service provider, it manage file, thread, process, etc.
- Win32k is GUI service provider.

# What is process

- Process is just a container, and set of different defined component, such as, thread
- Process contains library(DLL/LIB), threads, virtual memory structure(heap/stack), security access token, etc....

# What is thread

- In Windows ETHREAD structure represent a thread
- Conceptually, thread is a component of process
  - An execution unit represent to a process
- At any given time, only one thread can be executed on the processor core
- Thread has its own stack , data and code segment
- Thread can only be either in kernel mode or user mode at any given time.
- Include IRQL and CR3
- User mode delegate of ETHREAD is call TEB, which responsible for any user-mode thread operation



# What is memory space

- In x86 flat mode, each process thinks its has fully usage of physical memory
- Actually it's a lie from processor to process by switch it page table
- Each process contains a page table
- Only a thread of the process can access same memory space, except for the thread that attach the other page table(process space)



# Processor Mode – Real Mode

- Although we didn't introduce memory structure, however, we have to know, for legacy 8086 processor, we used segmentation for location memory
- For old processor we only have 16bit in address bus, however, we have to locate 20bit address
- Segmentation is the solution. Segment Base + offset
- Written Format: XXXX:YYYY, we locate every memory require segment (  $XXXX \ll 4$  ) + YYYY = 20bit address
- CS, DS is stand for code segment and data segment

# Processor Mode – Protected Mode

- Real Mode v.s. Protected Mode
- BIOS, part of UEFI, DOS, always run at Real Mode
- Processor Mode affect instruction behavior, memory access
- Cause processor mode change by :
  - SYSENTER / SYSCALL / INT 21H / Exception / Interrupt / FAR JMP or CALL
  - SYSRET / IRET / SYSEXIT / FAR RET
- Windows, Linux, Android, always run at Protected Mode
- Whenever memory access happen (RWX) , it consult the privilege level bit and compared to current thread's level, if it is low privilege, general exception (#GP) is throw

# Processor Mode – Protected Mode

- From user to kernel mode
- Processor automatically read MSR register
- A common system call via syscall / sysenter assembly instruction
- When we execute these instruction, current stack, code, and data will be changed
- The new value of stack pointer, instruction pointer will be loaded from MSR register
- Segment Privilege has to be changed as well

# Processor Mode – Protected Mode

- Whenever user-mode exception occurs, it will be captured by processor and dispatch by IDT (Interrupt descriptor table)
- For example
  - Virtual memory access cause Page Fault, IDT 0xE handler will assist swapping in the pages into system memory
  - Software breakpoint is assisted by processor as well, IDT 0x3 handler will assist to dispatch breakpoint exception to kernel mode debugger , user mode debugger, exception handler , and throw error eventually.
- INT 21 Instruction was also used for getting into kernel by Windows 2000

# Processor Mode Switch

- Segment is still usable in modern processor, however, in modern processor, the segment base are always zero, limit always MAX memory – 1
- We want every process used the page table and no other effect
- Therefore modern OS hide segmentation by zero base (flat memory model) but still need to added because it's still exist
- Segmentation can be ignored in Windows

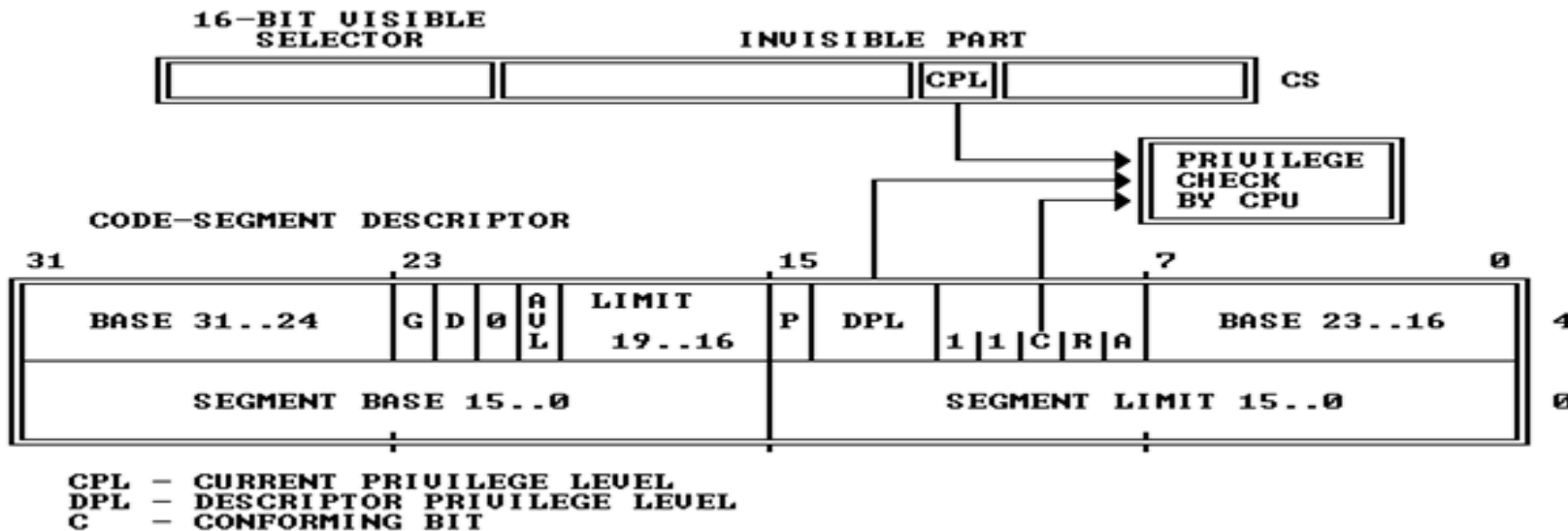
```
kd> dg 0 30
```

Sel	Base	Limit	Type	P	Si	Gr	Pr	Lo	Flags
----	-----	-----	-----	-	---	---	---	---	-----
0000	00000000	00000000	<Reserved>	0	Nb	By	Np	N1	00000000
0008	00000000	ffffffff	Code RE Ac	0	Bg	Pg	P	N1	00000c9b
0010	00000000	ffffffff	Data RW Ac	0	Bg	Pg	P	N1	00000c93
0018	00000000	ffffffff	Code RE Ac	3	Bg	Pg	P	N1	00000cfb
0020	00000000	ffffffff	Data RW Ac	3	Bg	Pg	P	N1	00000cf3
0028	80042000	000020ab	TSS32 Busy	0	Nb	By	P	N1	0000008b
0030	ffdf0000	00001fff	Data RW Ac	0	Bg	Pg	P	N1	00000c93

# Processor Mode Switch

- Although kernel and user mode share same segment
- With different segment selector, current privilege level still being different
- Whenever memory access occur, processor check it page's supervisor bit and CPL
- It explains why syscall need to change its segment, just for privilege change
- Processor make sure  $CPL < RPL$  (request level of destination segment/ page)

Figure 6-4. Privilege Check for Control Transfer without Gate



# System Call

- SYSCALL / SYSENTER instruction is executed
- First instruction that kernel executes is `nt!KiSystemCall64` after Windows 7
- It is purely assembly implemented function in kernel
- It mainly responsible for indexing the system call table and find the corresponding function and invoke it

# System Call

- ALL application want to use OS API always require a system call
- System call classify as NT system call / Win32 system call

```
typedef struct _KeShadowTable
{
    SYSTEM_SERVICE_TABLE NtKernel;
    SYSTEM_SERVICE_TABLE Win32k;
}TKeShadowTable, *PKeShadowTable;
```

```
typedef struct _SYSTEM_SERVICE_TABLE {
    PVOID      ServiceTableBase;
    PVOID      ServiceCounterTableBase;
    ULONGLONG  NumberOfServices;
    PVOID      ParamTableBase;
} SYSTEM_SERVICE_TABLE, *PSYSTEM_SERVICE_TABLE;
```



# System Call

- A Service Table store function pointer directly in 32bit system, in 64bit system is slightly different
- It is signed integer array, it has negative number
- Every element in array is offset to its table base
- Calculate a system routine address as:
  - $\text{Table} + \text{Table}[\text{Index}] \gg 4$  , last four bits describe the parameter count, we shift it
  - Sometime routine will be placed before table , so the offset will be negative number, for this case, we should extend it as:
    - $\text{Table} + (0xFFFFFFFF00000000 + (\text{Table}[\text{Index}] \gg 4))$

# System Call

- More detail can consult me or <https://github.com/Kelvinhack/PerfMon/blob/master/PerfMon/SSDT.cpp>

# Interrupt

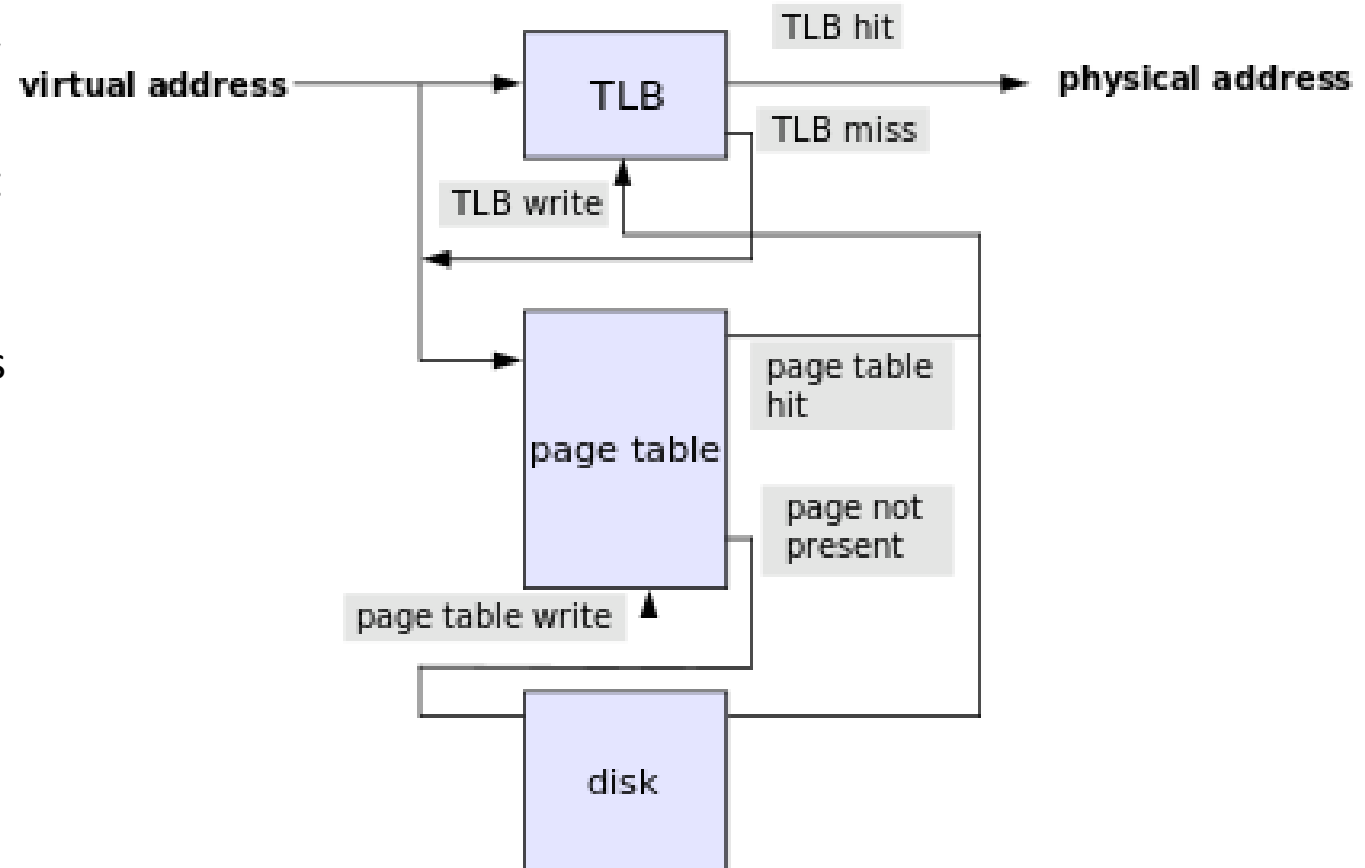
- Interrupt can be fired whenever processor executed special instruction
- “Interrupt” is general term that states event delivered by interrupt descriptor table, however, real interrupt is only for index 32~255
- Interrupt is dispatched from device-> APIC -> Processor
- Each interrupt has its priority used by APIC for prioritize interrupt event
- APIC scheduling, masking and deliver interrupt to processor, that's IRQ in Windows.
- INT 0~31 is software reserved for operating system use, for example, 0xE for page fault, 0x1 for hardware breakpoint, 0x3 software breakpoint..
- INT 32~255 is hardware reserved, external interrupt from different device, for example, PS/2 keyboard, mouse, disk controller, etc

# Memory

- Intel / AMD platform divided memory as a page
- Page size always is 4096 byte
- Each process has it own page table for describing current process memory usage
- We cannot access memory directly in protected mode
- Every time we access memory it consult Page Table for further operation, such as, privilege judge, physical address locating.. etc

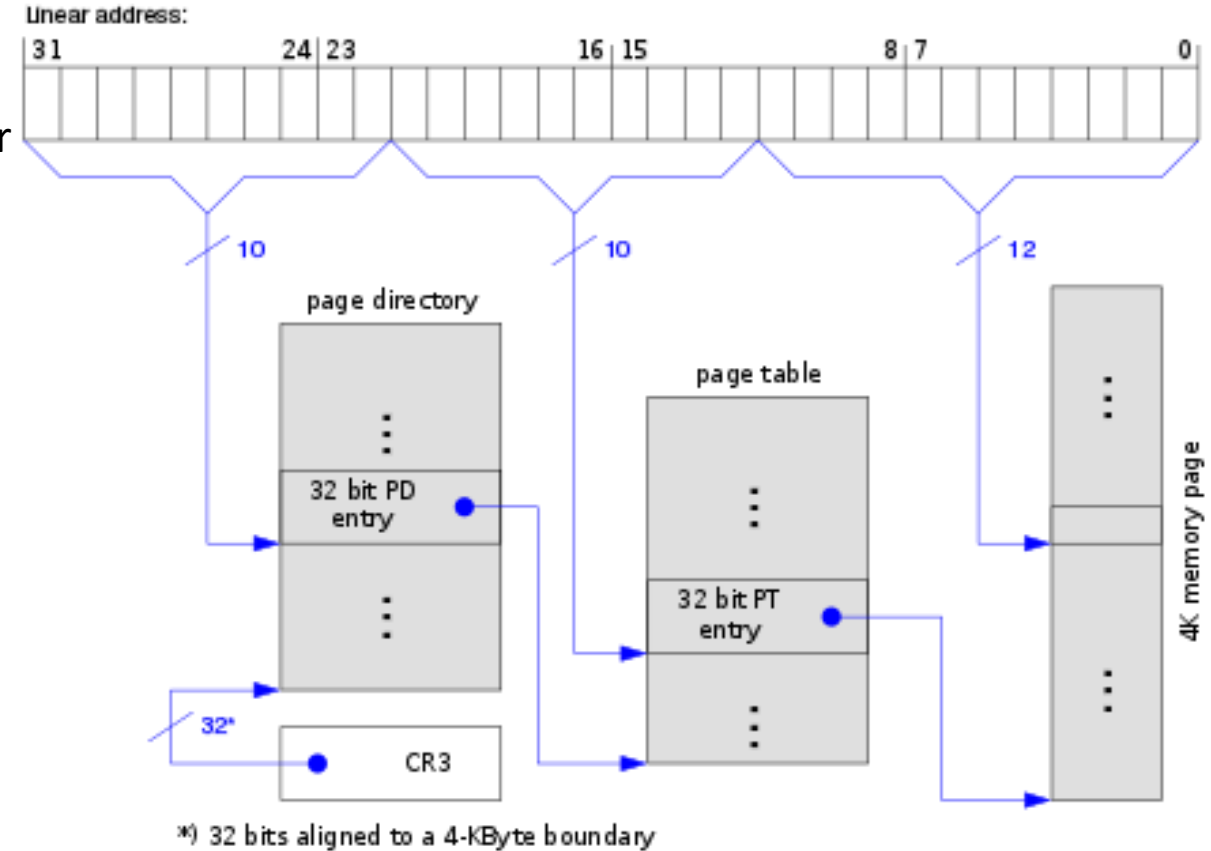
# Memory - access

- Address translation happen whenever memory access in protected mode
- First time processor and memory management unit(MMU) will consult translation buffer (Translation lookaside buffer)
- If there's no TLB entry exist, it consult process's page table
- If there's no valid Page Table Entry, it cause exception, we call it **Page Fault**, and require swap-in (if is does exist in disk)



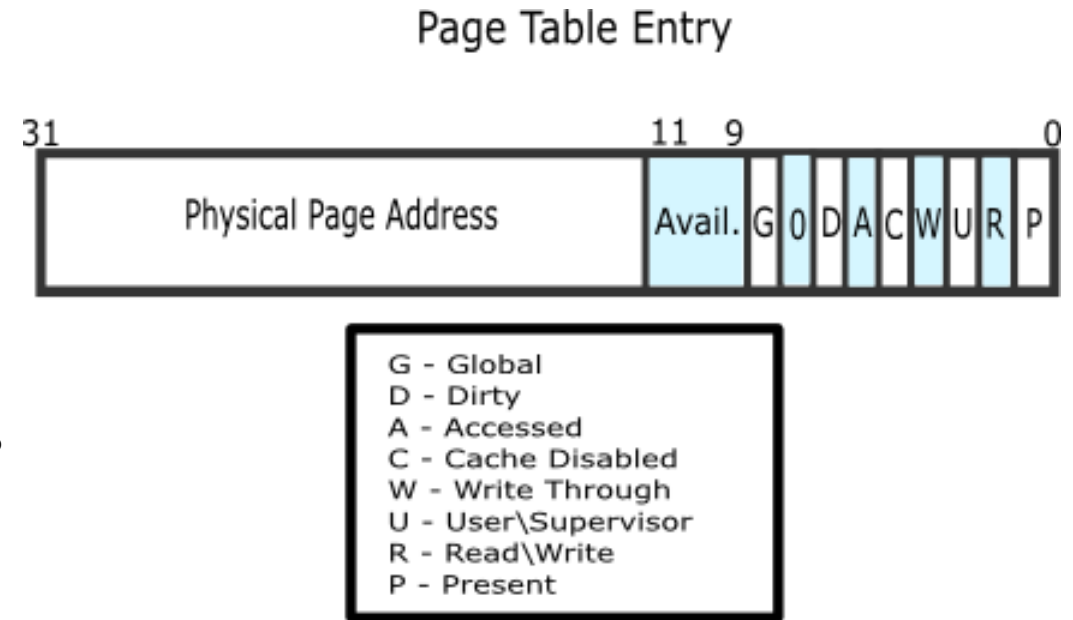
# Memory - Page Table Structure

- Page Table is memory mapped in kernel space
- Every Page Table base address is located in CR3 register
- Parsing memory translation is processor job
- OS only create the mapping to processor
- Last level – PTE is an interesting structure.



# Memory - PTE

- Hardware used structure
- User / Supervisor bit determine the corresponding memory page privilege level
- Present bit means the page is exist in memory
- R/W means read/writable
- Global bit means is it shared across process, if it does, TLB entry of this page flushes will be skipped.
- Dirty bit means is it already written by someone
- Access bit similar to dirty bit, but also count for read
- Some memory based attack based from this structure by rootkit 😏



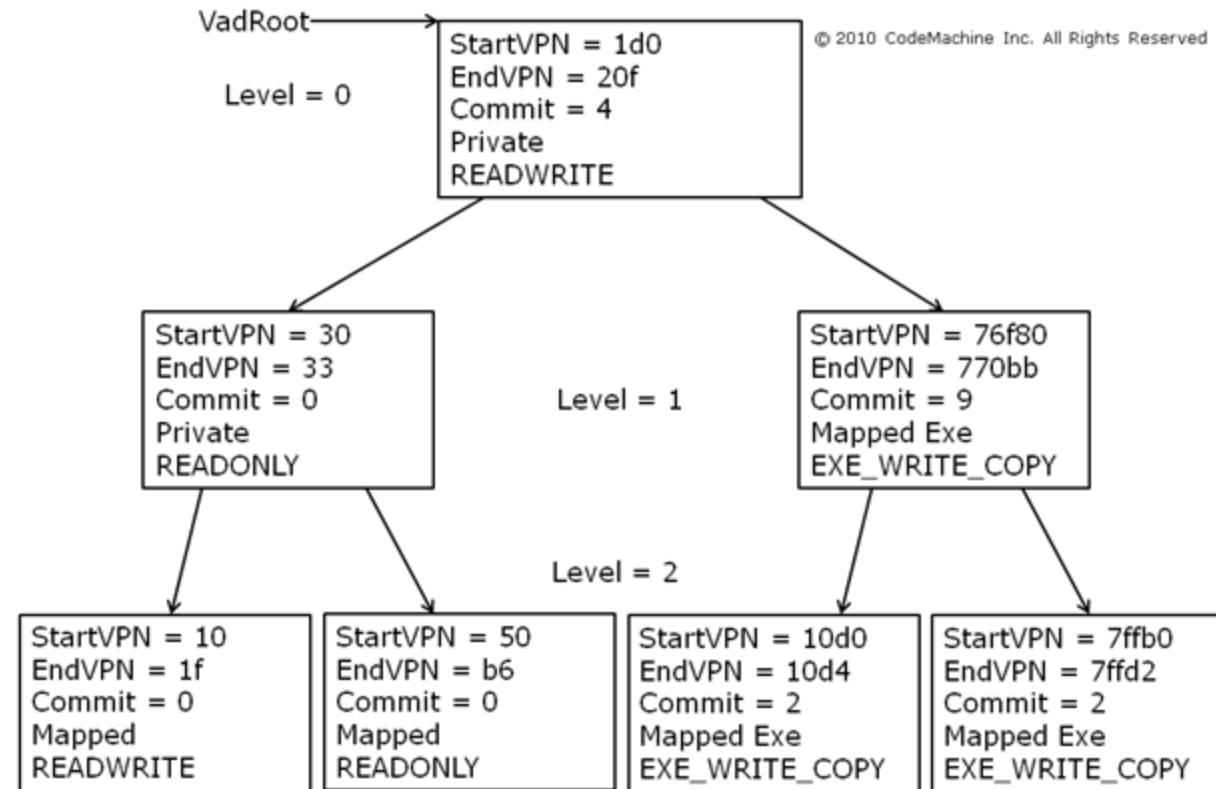
# Memory - Virtual Memory Management

- In order to optimize memory usage, every virtual memory request isn't instant allocate to the requestor.
- Every memory request will be bookkeeping into system-manage structure
  - Virtual Address Descriptor (VAD) in kernel
  - Software PTE
- Whenever memory accessed, PTE is empty, Page Fault occur, the handler consult VAD and filled PTE on-demand



# Virtual Memory Management - VAD

- VAD is AVL tree
- Record all virtual memory status and properties
- VAD is process dependent, stored in EPROCESS
- VAD is not open structure(symbol required)
- Enumerate VAD could find some malicious memory space, or even unlink DLL.



# Virtual Memory Management –Shared Memory

- For those memory shared between process
- Windows introduced SECTION object, which called file mapping in user mode
- Windows invalid the shared page's PTE , and manipulate the software PTE, for locating the prototype PTE.
- It should be very fast operation, other than first time, it already in main memory

# Virtual Memory Management – PFN database

- PFN Database which is an array that indexed by PFN number
- PFN database bookkeeping for all memory usage
- nt!MMPFN
- Latest structure can be found :
  - [https://www.vergiliusproject.com/kernels/x86/Windows%2010/1809%20Redstone%205%20\(October%20Update\)/\\_MMPFN](https://www.vergiliusproject.com/kernels/x86/Windows%2010/1809%20Redstone%205%20(October%20Update)/_MMPFN)
- It basically record the physical page reference count and shared count

# Kernel Object

- Kernel uses object to represent and maintain an abstract concept
  - ETHREAD, KTHREAD
  - EPROCESS, KPROCESS
  - VAD
  - IRP
- Each kernel object has callback for pre/post callback corresponding to different operation

```
NTSTATUS ObRegisterCallbacks(  
    POB_CALLBACK_REGISTRATION CallbackRegistration,  
    PVOID *RegistrationHandle  
);
```

# Debugging

- What we need debugging? For answer a question by yourself
  - We don't have source code
  - We don't know what the source compiled
  - We don't how does source work internally
- User / Kernel Mode debugging

# Debugging

- Debug is not actually 'debug' in security, but reversing engineering
- We could know all through debugging
- ReactOS is an example
- static debugging:
  - IDA
- dynamic debugging:
  - x32/x64dbg
  - Windbg
  - GDB
  - IDA... etc

# Debugging

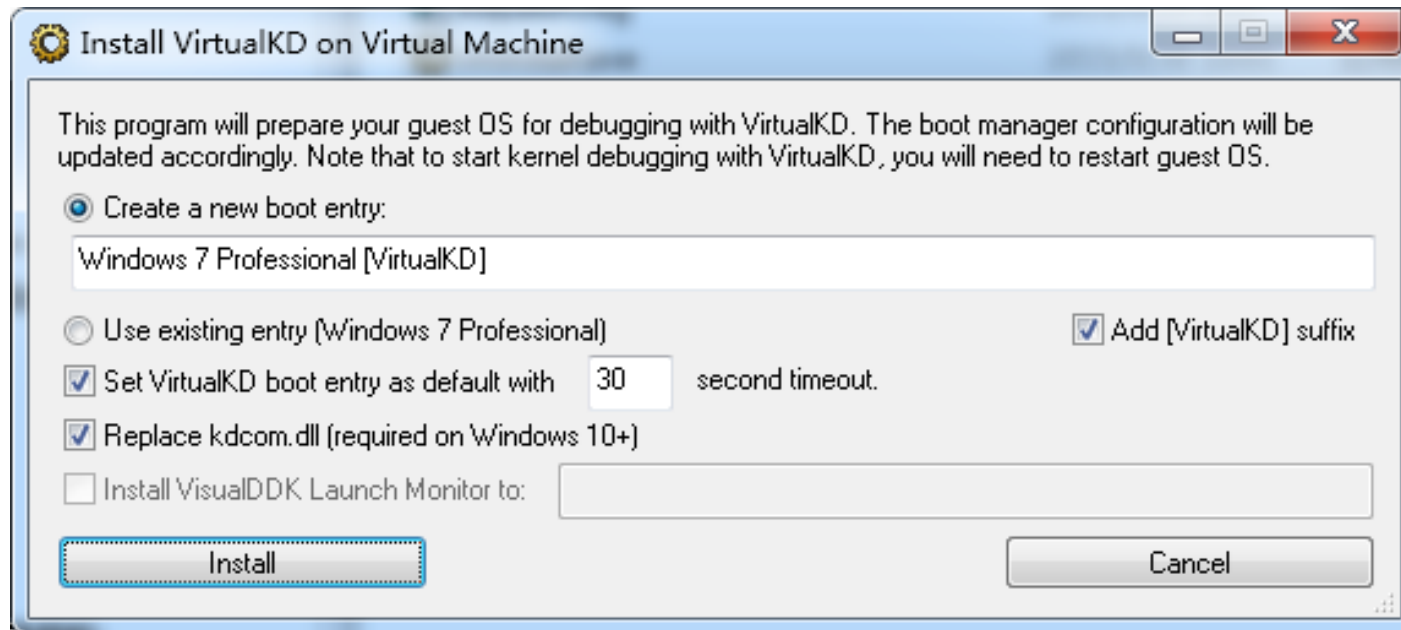
- We could do a easy exercise
- break your favorite function through windbg

The screenshot shows the WinDbg interface with the following components:

- Command Window:** Contains the command `!NTCreateFile` and the output of the `g` command, showing the loaded module `C:\WINDOWS\SYSTEM32\Ntinput.dll` at address `00007ffe79f1a980`.
- Disassembly Window:** Shows the assembly code for `ntdll!NtCreateFile`, including instructions like `mov r10,rcx` and `ntdll!NtCreateFile`.
- Hex Dump Window:** Displays the memory dump starting at address `0:008>`, showing registers `rax=000000ae2affbbb0`, `rbx=0000000000000000`, `rcx=000000ae2affbc50`, and other registers. It also shows the instruction `mov r10,rcx`.

# Kernel Debugging

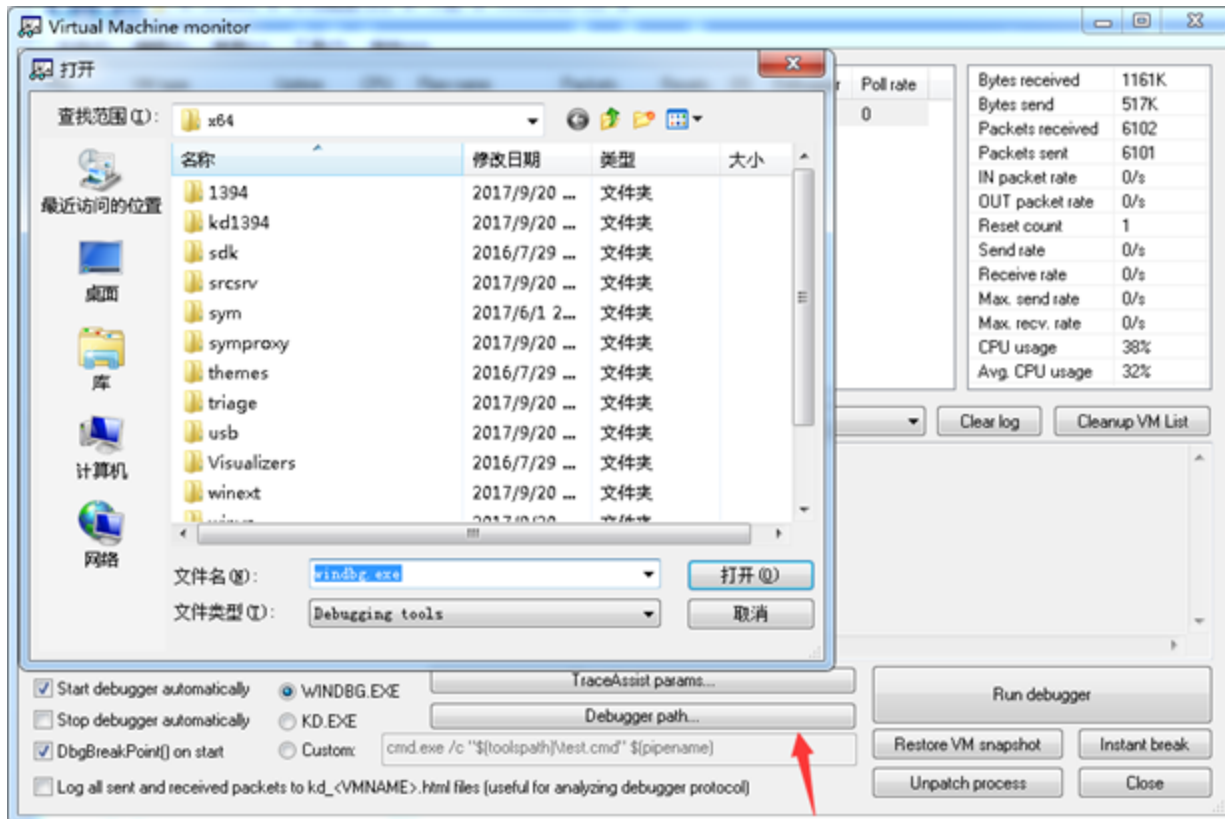
- 1) Setup your Windows 7 Virtual Machine
- 2) Download VirtualKD From the link: <http://virtualkd.sysprogs.org/download/>
- 3) Extracting the downloaded file , and put the “target” file into the VM
- 4) Execute vminstall.exe, after that, you should successfully config your VM as a debuggee





# Kernel Debugging

- You should install Windows Driver Kit, or download the Sysinternal tools set
- Then, Setting your Windbg path with VirtualKd
  - Start VirtualKD (vmmon64.exe)



# Kernel Debugging

- Start your VM with kernel debug mode
- Find a process that you play around with...
- I choose notepad.exe
  - Breakdown the debugger by press ctrl+break
  - Enter ' !prcoess 0 1 <process name> ' to find your process
- Switch the process memory space with ' .process <eprocess> command '
- Enter ' bp <modulename> ! <func name> to set software breakpoint
- Enter 'g' command

```
Command - Kernel 'com:pipe,reset=0,reconnect,port=\\.\pipe\kd_NestedVt-Test' - WinDbg:10.0.15063.468 AMD64

*****
nt!RtlpBreakWithStatusInstruction:
fffff800`032caf60 cc      int      3
2: kd> !process 0 1 notepad.exe
PROCESS fffffa8003d63b30
  SessionId: 1 Cid: 095c Peb: 7fffffd7000 ParentCid: 09ac
  DirBase: 10a124000 ObjectTable: fffff8a001850e10 HandleCount: 79.
  Image: notepad.exe
  VadRoot fffffa80071b8300 Vads 63 Clone 0 Private 405. Modified 2. Locked 0.
  DeviceMap fffff8a000fa4860
  Token fffff8a001a36a90
  ElapsedTime 00:00:03.369
  UserTime 00:00:00.000
  KernelTime 00:00:00.000
  QuotaPoolUsage [PagedPool] 154144
  QuotaPoolUsage [NonPagedPool] 7504
  Working Set Sizes (now,min,max) (1970, 50, 345) (7880KB, 200KB, 1380KB)
  PeakWorkingSetSize 1970
  VirtualSize 76 Mb
  PeakVirtualSize 79 Mb
  PageFaultCount 2049
  MemoryPriority FOREGROUND
  BasePriority 8
  CommitCharge 513

2: kd> .process fffffa8003d63b30
Implicit process is now fffffa80`03d63b30
WARNING: .cache forcedecodeuser is not enabled
2: kd> u ntdll!NtReadFile
ntdll!NtReadFile:
00000000`77a2ff10 4c8bd1 mov     r10,rcx
00000000`77a2ff13 b803000000 mov     eax,3
00000000`77a2ff18 0f05 syscall
00000000`77a2ff1a c3 ret
00000000`77a2ff1b 0f1f440000 nop     dword ptr [rax+rax]
ntdll!ZwDeviceIoControlFile:
00000000`77a2ff20 4c8bd1 mov     r10,rcx
00000000`77a2ff23 b804000000 mov     eax,4
00000000`77a2ff28 0f05 syscall
2: kd> bp ntdll!NtReadFile
2: kd> bp nt!NtReadFile

2: kd>
```

# Driver development

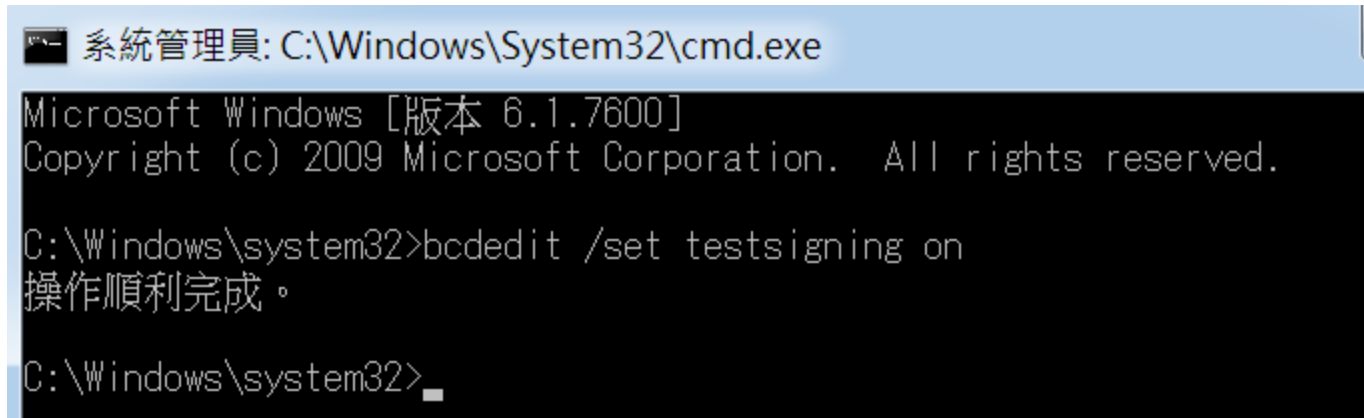
- DriverEntry is the main of every driver
- WDK is driver development kit like the other SDK does, including kernel library, object file, header file, tools, etc.

```
NTSTATUS DriverEntry(  
    _In_ PDRIVER_OBJECT DriverObject,  
    _In_ PUNICODE_STRING RegistryPath  
)
```

- Driver Object represent the driver instance in kernel, contains all information of a driver.
- Registry Path usually point to  
\\Registry\\Machine\\System\\CurrentControlSet\\Services<i>DriverName

# Enabling Testsigning

- Windows Kernel require the driver is signed , however , the testing signing mode could be enabled by following command in CLR



```
系統管理員: C:\Windows\System32\cmd.exe
Microsoft Windows [版本 6.1.7600]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Windows\system32>bcdedit /set testsigning on
操作順利完成。

C:\Windows\system32>_
```

# Device Object

- How driver communicate with user-mode application?
- Memory mapped
- IOCTL
  - NT Device Name , optional, only NT kernel can recognize the object via this name
  - Symbolic Device Name, optional, exposed driver interface to user mode application
  - Win32 Device Name, user-mode use it to find Symbolic Device Name to find NT Device at last

# Memory Usage

- `ExAllocatePoolWithTag` / `ExAllocatePool`
  - We should check its return value everytime
- `ExFreePoolWithTag` / `ExFreePool`
  - We should clear the original pointer on the stack
- In Modern processor, minimum granularity of page size is 4096 byte
- No way to allocate memory at lower than 4096 bytes
- High level language hide its detail and re-encapsulate the memory management from intel /amd processor, for example, `malloc(100)` or `VirtualAlloc(100)`
- Memory Type
  - `NonPagedPool`, always resides in main memory
  - `PagedPool`, can be swapped out of main memory , in storage / disk

# IRQL

- Interrupt request level (IRQL)
- User mode always at PASSIVE\_LEVEL, means, it can be interrupt or switch out whenever the thread execute, so once a thread running in kernel, it is always in very low priority
- When a thread get into kernel mode, it will be changed on demand
- APC\_LEVEL / DPC\_LEVEL .... To HIGH\_LEVEL
- Driver developer needs to care of current operation is permitted in current IRQL, otherwise, it cause BSOD /unexpected problem
- KeGetCurrentIrql () is your friend

# Security Implication of IRQL

- Driver developer should always pay attention on what are they doing.
- For example, PagedPool memory cannot be swap in when the current thread `IRQL >= DISPATCH_LEVEL`
- Because it require a APC lock for whole process, and perform I/O synchronous operation, from reading data from disk and copy onto memory, such behavior is totally inhibit in `DISPATCH_LEVEL`.
- In LoadImage system callback, we cannot read/write files, due to its disable APC sometime, it remains deadlock.



# DPC timer

- Queue a special timer that executes in desired timing.
- Every DPC execute for each time interrupt
- Every processors has its DPC queue
- Time interrupt occur per 15.625 ms, means execute 64 times in a second ( $1000/15.625$ )
- DPC callback routine should be do sensitive job
-

# Exercise

- Try to queue a DPC for checking driver function integrity

# APC Timer

- APC\_LEVEL timer
- Unexported functionality
- Undocumented functionality

# Workitem

- Usual way to schedule item from high IRQL , such as Interrupt handler
- Workitem is running at PASSIVE\_LEVEL

# Thread Synchronization

- Due to the differences between IRQL, high IRQL should be wait too long time
- When high IRQL thread wait too long, it probably cause deadlock / watchdog BSOD
- Consider a thread running at DISPATCH\_LEVEL , and it wait for the object completed without timeout , and no other processor free to accomplish.
- Thread scheduling on this processor core is stopped due to it high priority
- Eventually no one can finish this event

# Thread Synchronization

Suggested synchronization method

- PASSIVE\_LEVEL - **KeInitializeMutex**
- APC\_LEVEL - **ExInitializeFastMutex**
- DISPATCH\_LEVEL - **KeInitializeSpinLock**

# I/O Request Packet (IRP)

- A way that used for interact with NT subsystem or third party driver
- For example, ReadFile, to NTFS file system, SendPacket to NDIS network subsystem, Receive Mouse data from HID subsystem.
- IRP isn't included in system security boundary
- Every driver has to provide the corresponding IRP callback if they want to expose the functionality to upper level.

# System Callback

- PsSetCreateProcessNotifyRoutine
- PsSetCreateThreadNotifyRoutine
- PsSetLoadImageNotifyRoutine



# NTFS Internals

TBC....

# Exercise

- Enumerate IDT
- Enumerate SSDT
- Enumerate VAD
- Hook IDT
- Hook SSDT
- Hook IRP

# HID Device Internals

- Windows hides the complexity of USB packet, HID protocol standard, for mouse and keyboard manufacture, they only need to develop HID minidriver to collect their USB data and give more feasibility and functionality upon this support
- HidClass driver provide bunch of interface and helper for parsing your data
- Basically, HidMiniDriverRegister receive a driver object and hooking all IRP of your driver.

# Win32k

- The other entry of kernel
- User Interaction related subsystem
- Graphics
- Input SubSystem
- DirectX

# HID Device Internals

- HID Class driver create the device on your registering driver behalf, it named Client PDO, and used interact with upper driver, like kbdclass, mouclass, etc
- HID Class driver will implicitly create and binding with lower level named FDO, which will attached the USB Hub device list, and used for interact with USB Hub
- Your HID client driver's device object is standalone, it's logically lowest level to your device list, means, if someone traverse the list from upper kbd/mou driver, it cannot be directly access the lowest level(USB), but your driver only.

# HID Device Internals

- As we known, there is two part of a HID device
  - FDO and Client PDO
- Client PDO represent the high level , FDO represent the low level
- HIDClass uses this two device object to communication on behalf of ourselves, dealing every messy problem from USB and Industrial protocol stuff, for example, HID interfacing, HID device ID, etc

# HID Device Internals

For single function / interface device case

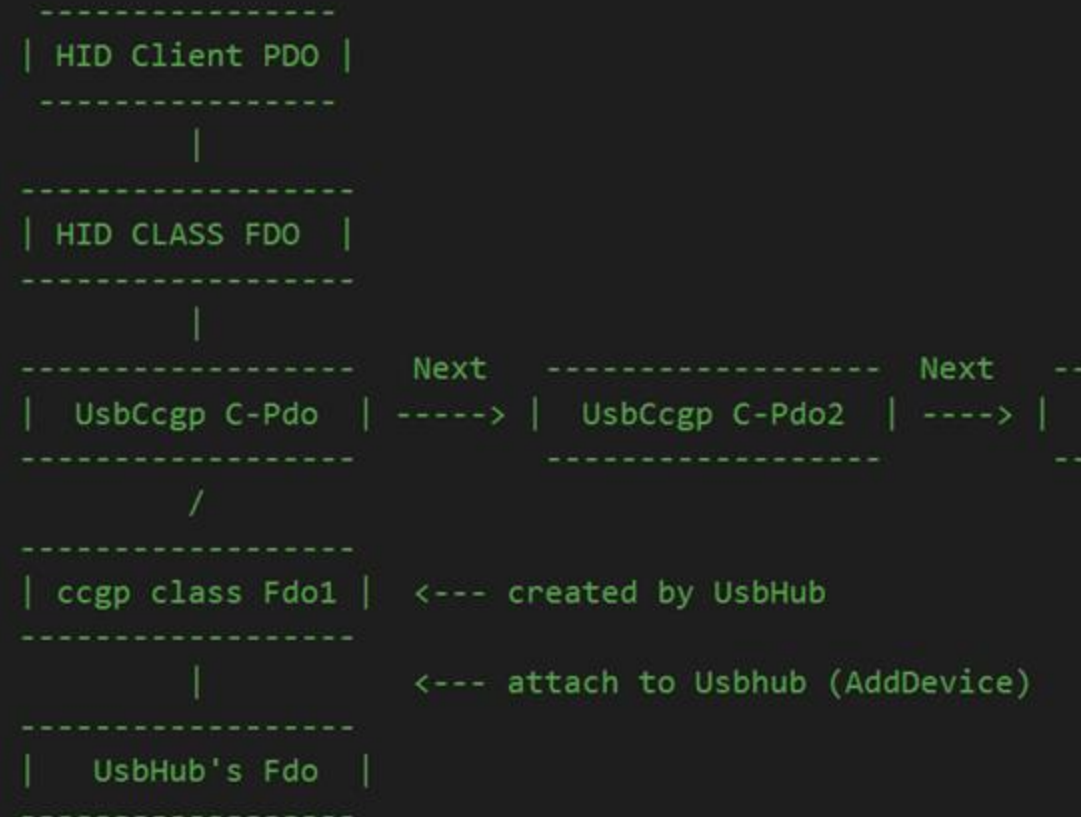
- It is directly attach on USBHUB device linkedlist



# HID Device Internals

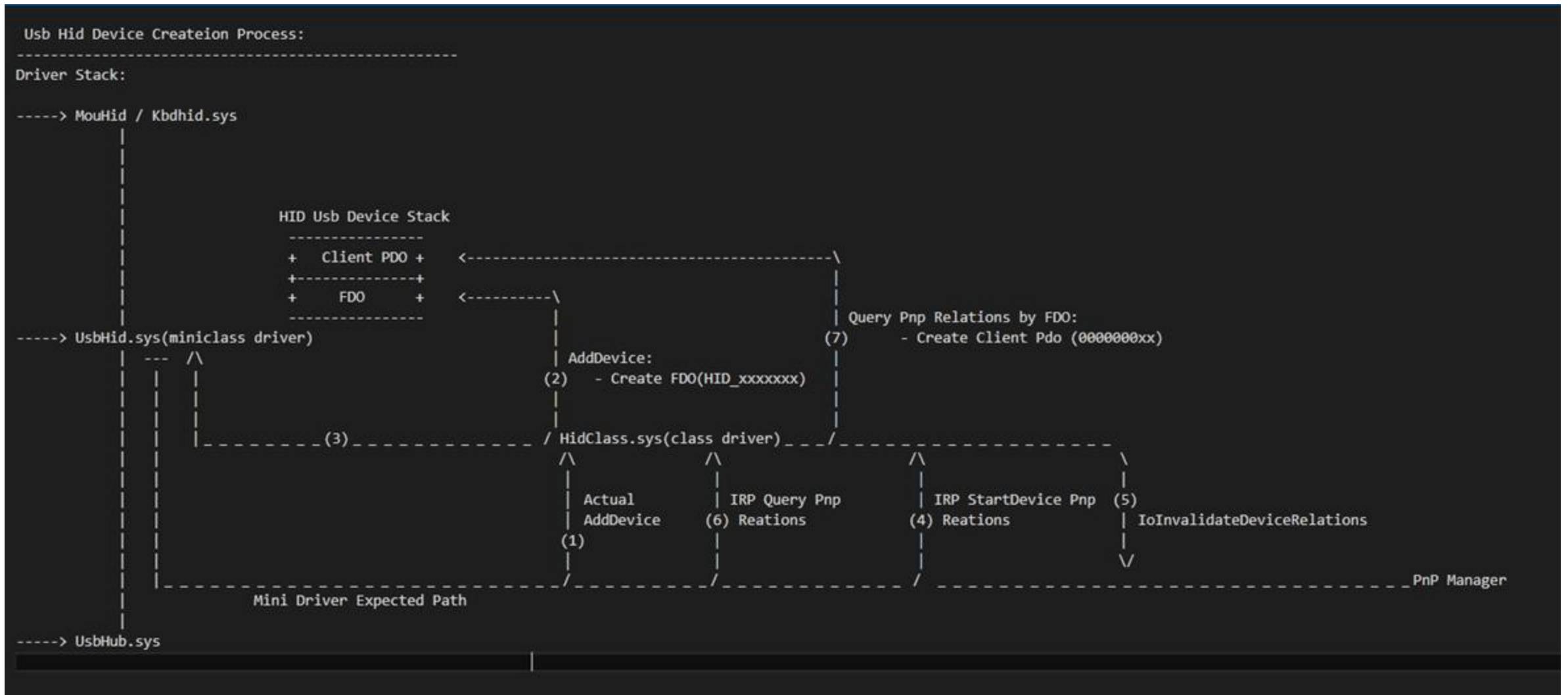
- Multi-functional device
- It will not attach on HUB directly
- Instead, it will attach on USBCCGP

Case 2 :





# HID Device Internals



# HID Device Internals

- The whole input system is polling process, there's thread call RawInputThread, is always pending for the next data
- When packet is sent from hardware to USBHUB, it will complete the pending IRP callback link from bottom to down and wake the thread up
- The packet called USB Request Block
- Upon that time, HID protocol is irrelevant

# Exercise

- Take some notes of Usbmon and make a tools that enumerate the HID devices tree
- Mice logger
- KeyLogger
- Usb Packet Analyzer

# Windows Desktop Internal

- Every program / application is able to modify the screen, so there's **Windows Desktop Manager (WDM)** to collect every manipulation from different process and combining them into framebuffer
- With DirectDraw enabled, WDM combine framebuffer by DirectX to enable the direct manipulation on GPU, and releasing the time of CPU processing on framebuffer from GDI legacy approach

# Win32k GDI

- GDI provide set of interface that supposed to be used for virtual desktop, to clone your current desktop on-the-fly
- With DirectDraw disabled, we can simply create isolated and virtual device that intercept every single instruction on the main surface, intercept all operation on your main surface by mirror driver
- Basically it create an invincible chance to intercept the drawing operation on-the-fly without modify the system code

# Hypervisor

- Virtualization expose more possibility to security research
- Monitoring without breaking your PatchGuard, your AV software:)
- Some monitoring can be assisted by hardware
- Critical Register switch
- Critical Event Happened
- Exception and Interrupt
- Memory access fault
- ...more

# Monitor Your Input Device

Intercept your device data which come from hardware

- IRP Hook based
- USB Hardware Packet Protocol Extract
- Keyboard/ Mice Protocol Extract
- Anti-Key input emulation, protect your process :)
- <https://github.com/Kelvinhack/UsbMon>

# Monitor your screen

Intercept your screen rendering without hook

- GDI Based drawing callback driver
- Kernel based attack model
- <https://github.com/Kelvinhack/ScreenCapAttack>



# Monitor your system call

Hardware based attack model

- Provide stealthy detect syscall hook
- Provide stealthy attack syscall hook
- <https://github.com/Kelvinhack/PerfMon>

# Monitor your memory

Hypervisor based protect user mode memory access

Defend:

- Provide Integrity enforcement to your product

Attack:

- Hiding your hook
- <https://github.com/Kelvinhack/NoTruth>

# Monitor your devices communication

## Hypervisor based MMIO detection

- Device Driver communicate with devices via MMIO
- SPI/LPC device is one of the PCI device
- BIOS flash image is stored in SPI flash mostly
- Protect your BIOS on-the-fly :)
- <https://github.com/Kelvinhack/DeviceMon>

# Hypervisor Undercover

## Nesting Hypervisor

- Too much hypervisor nowadays, we need some undercover :)
- Hypervisor-on-Hypervisor
- Nesting ability
- <https://github.com/Kelvinhack/kHypervisor>

# Firmware Security

- Firmware indicate the software that provided by OEMs, BIOS/UEFI, a.k.a. Flash
- Flash is stored in standalone chip, for example, SPI flash mostly, for legacy device, PCI or Firmware Hub(FWH)
- All firmware storage can be access via programming your LPC or SPI controller
- Controller is always on the southbridge, take some vision from following link
- <https://apprize.info/science/computer/8.html>

# Firmware Security

- Programming on device by MMIO
- According to PCI spec, Base Address Register(BAR) is assigned by firmware, each PCI Device has its own BAR.
- Whenever processor access to that physical memory range of BAR, it will be decode by PCI device, and it can be issue DEVSEL# for claiming that access.
- The actual memory range isn't used physically
- BAR just reserved the range and give a window between cpu and device

# SPI Controller

- SPI Controller programming is chipset specific
- By programming SPI BAR range
- Understanding of SPI protocol is needed , please consult your datasheet before any action :P

# Exercise

- Write a driver that dump your flash
- Try to monitor your dumper via hypervisor