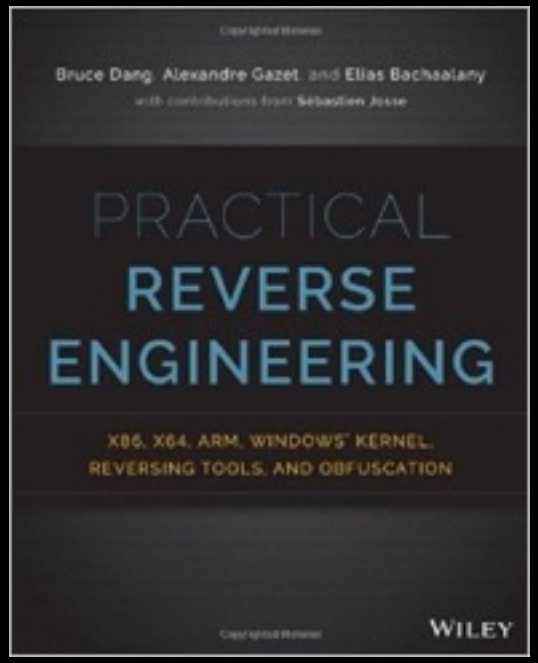
Intro to Windows Kernel

Aaron Sedlacek

Agenda

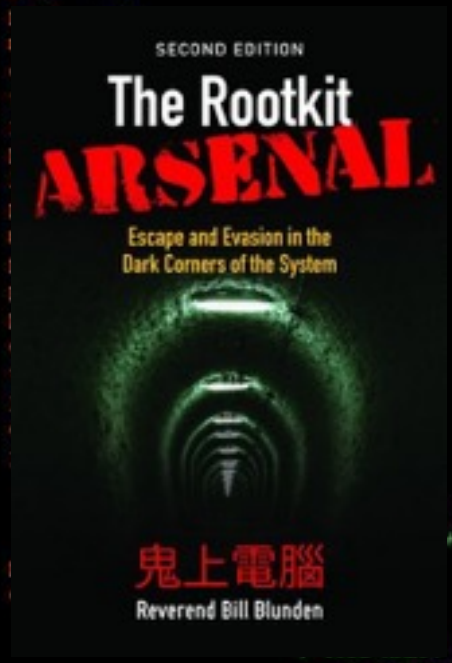
- Kernel Basics
- Windows API
- Windows Drivers & Devices
- Kernel Debugging

Changing Textbooks!



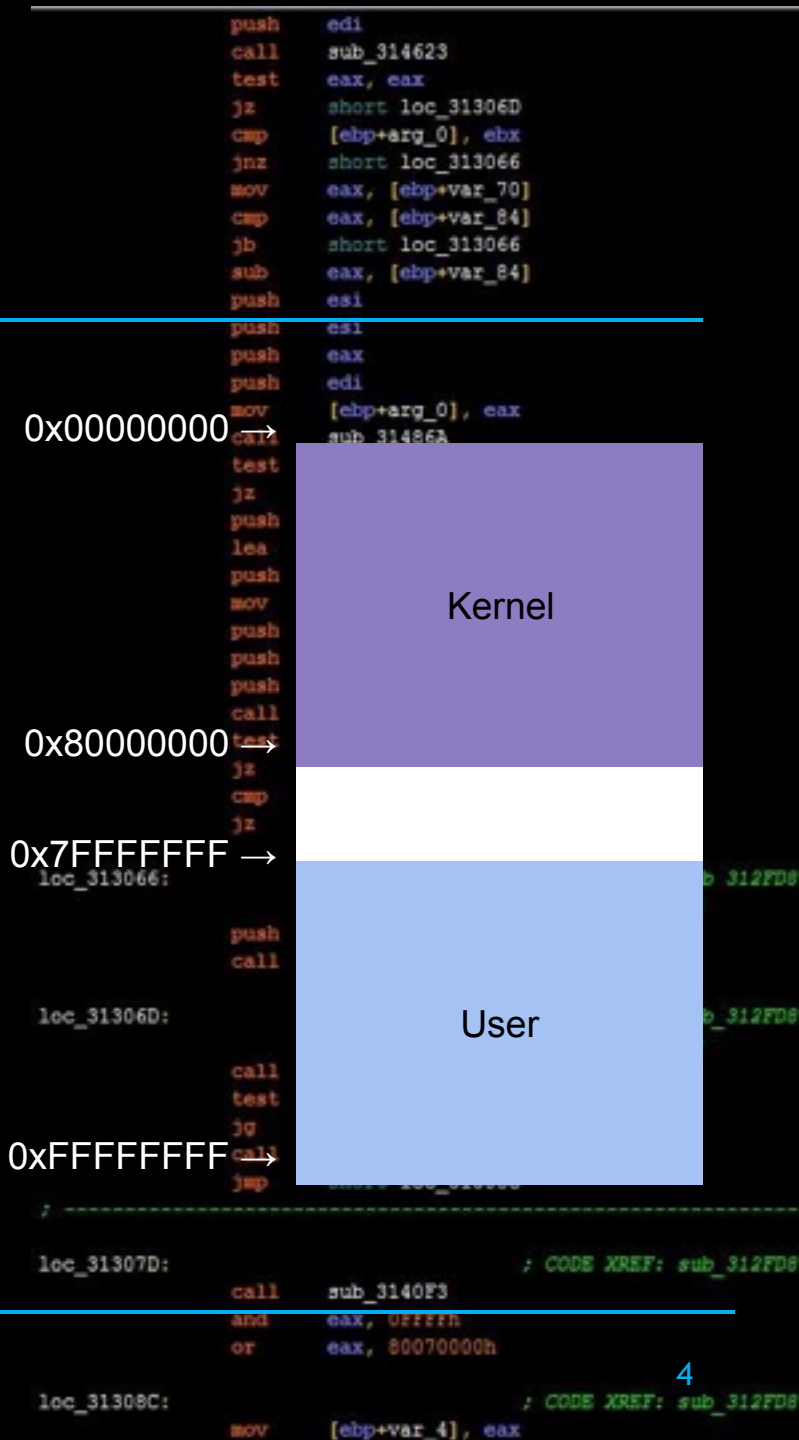
← Your kernel land field guide
← Highly recommended
← Ch. 3+4 are the focus of this lecture

The most common techniques →
Focuses primarily on ring 0 →
Also talks about ring -1,-2, and -3 →



Memory Layout

- Virtual Address Space is divided into two portions
  - Kernel memory space is mostly the same in all processes
    - Kernel mode can access both spaces
    - User mode can only access user space

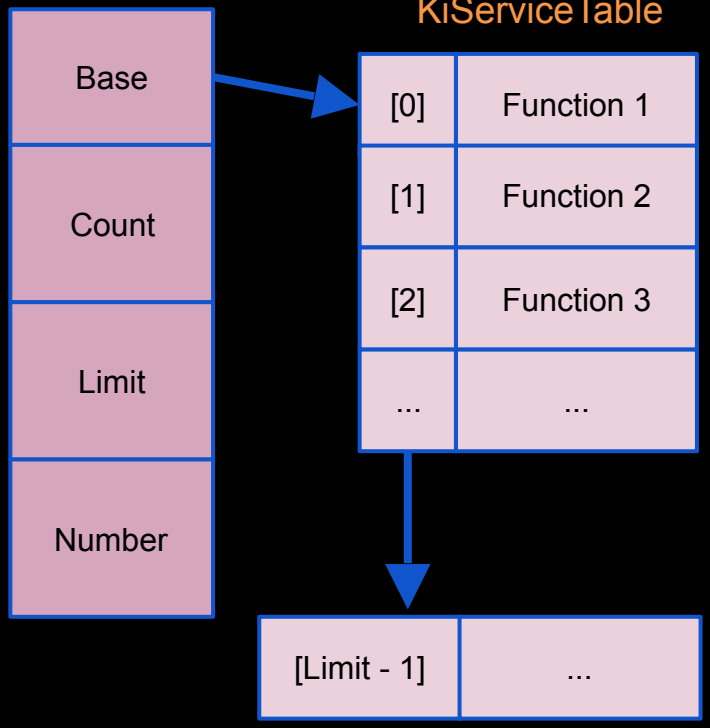


System Calls

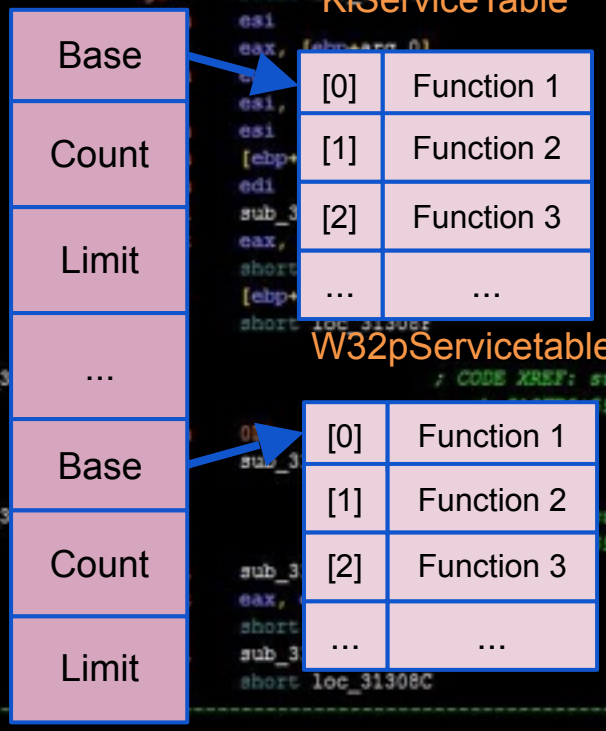
- Kernel function that services I/O requests from users
- Stored in two structures:
  - (System)ServiceDescriptorTable (SDT, SSDT)
  - (System)ServiceDescriptorTableShadow
    - The shadow table has the same information as the descriptor table
    - It also has an additional syscall table for GUI threads

SSDT

KeServiceDescriptorTable



KeServiceDescriptorTableShadow

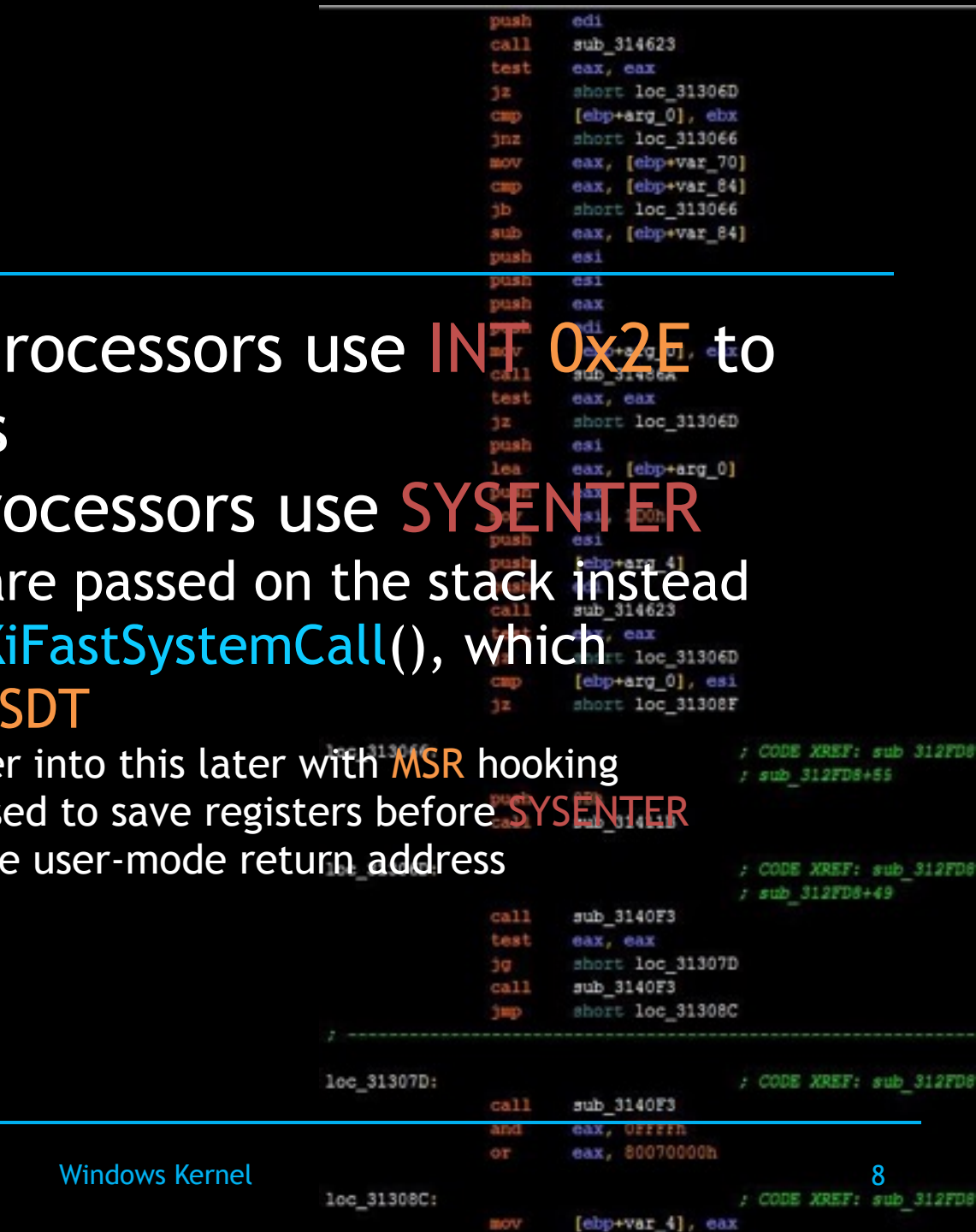


Interrupts

- Interrupt Descriptor Table →
  - AKA the IDT
  - 256 entries
    - Each interrupt is associated with an index in this table
- Each _KIDT_ENTRY defines the interrupt handler
- The base address of the IDT is stored in a special register called the IDTR

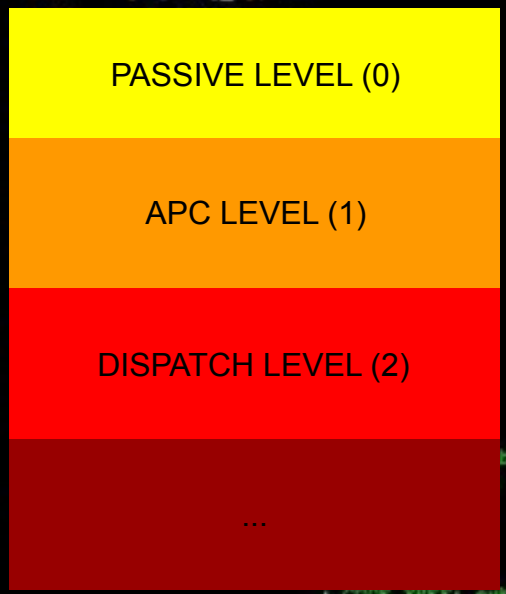
Traps

- Pre pentium 2 processors use INT 0x2E to perform syscalls
- More modern processors use SYSENTER
  - The arguments are passed on the stack instead
  - SYSENTER uses KiFastSystemCall(), which references the SSDT
    - We will dig deeper into this later with MSR hooking
    - A trap frame is used to save registers before SYSENTER
    - This also saves the user-mode return address



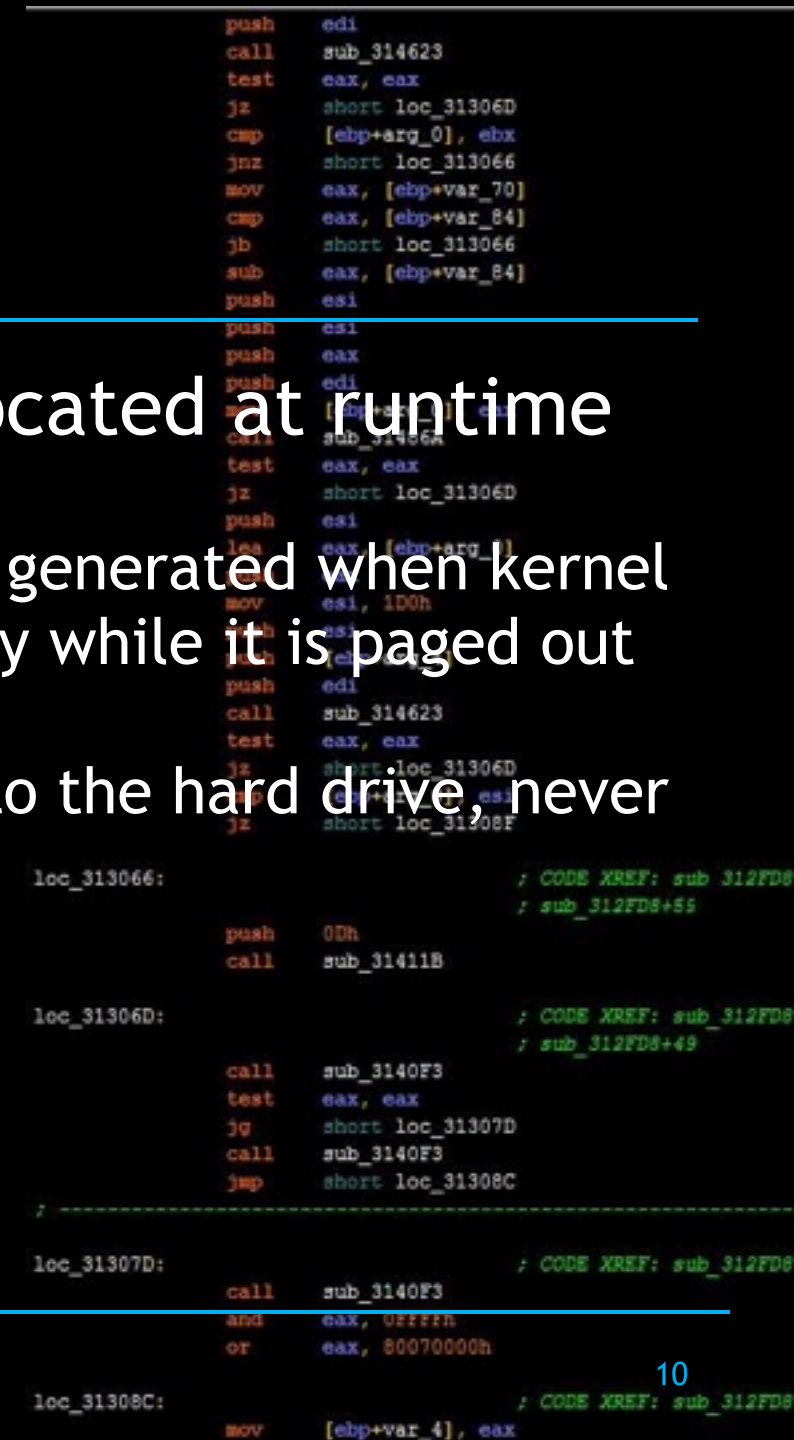
Interrupt Request Levels (IRQL)

- Just a number defined by the KIRQL struct
- (0) - All user-mode code and most kernel code executes here
- (1) - APC's are executed at this level
- (2) - Highest in the system
  - The thread dispatcher and DPC's run at this IRQL
- Higher IRQL's exist, but are typically associated with hardware



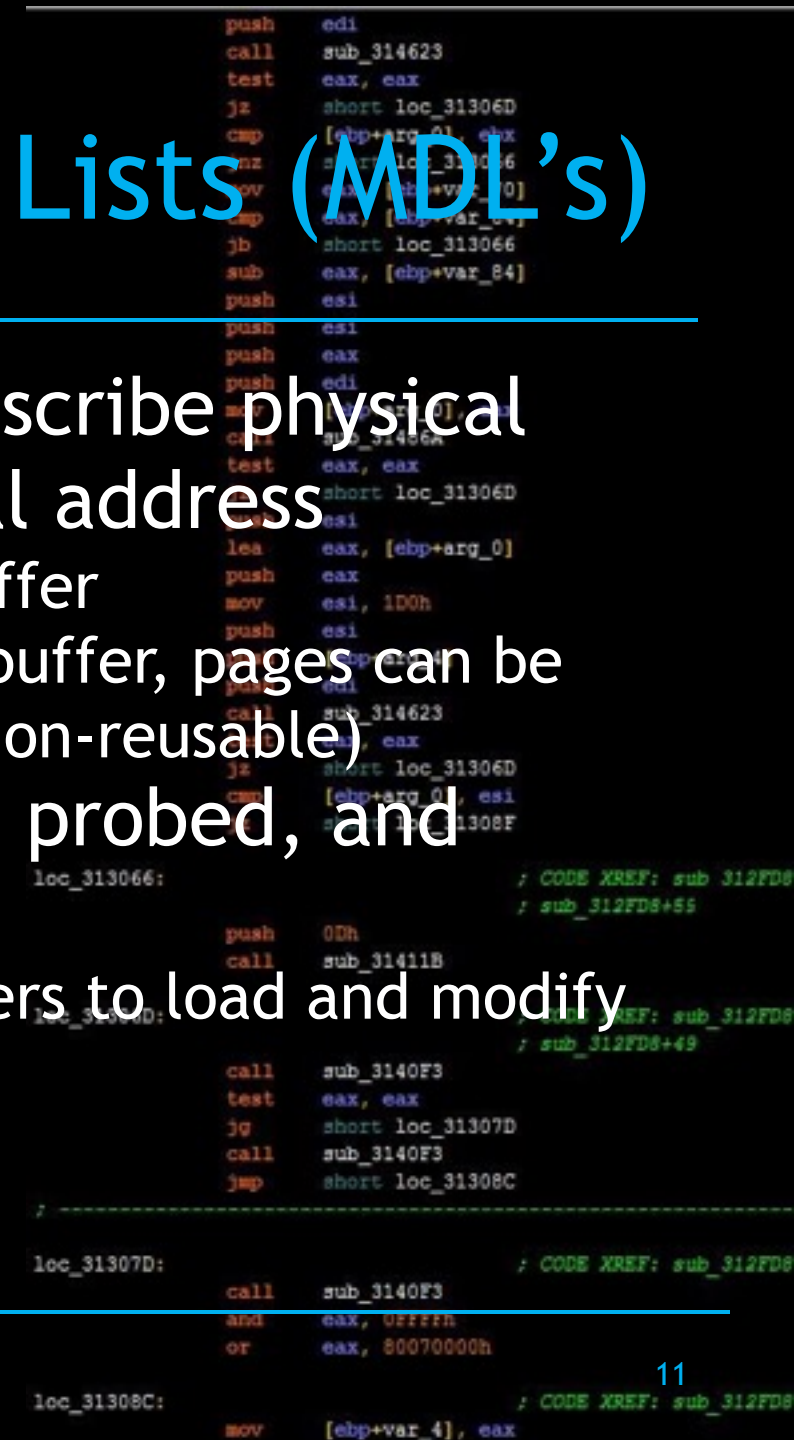
Pools

- Kernel mode memory allocated at runtime
  - Paged pool
    - A page fault exception is generated when kernel code touches this memory while it is paged out
  - Non-paged pool
    - Can never be paged out to the hard drive, never triggers a page fault



Memory Descriptor Lists (MDL's)

- Data structure used to describe physical pages mapped by a virtual address
  - Describes one contiguous buffer
  - Once an MDL is built over a buffer, pages can be locked in memory (marked non-reusable)
- MDL's must be initialized, probed, and locked
  - MDL's are used to allow drivers to load and modify memory in kernel space

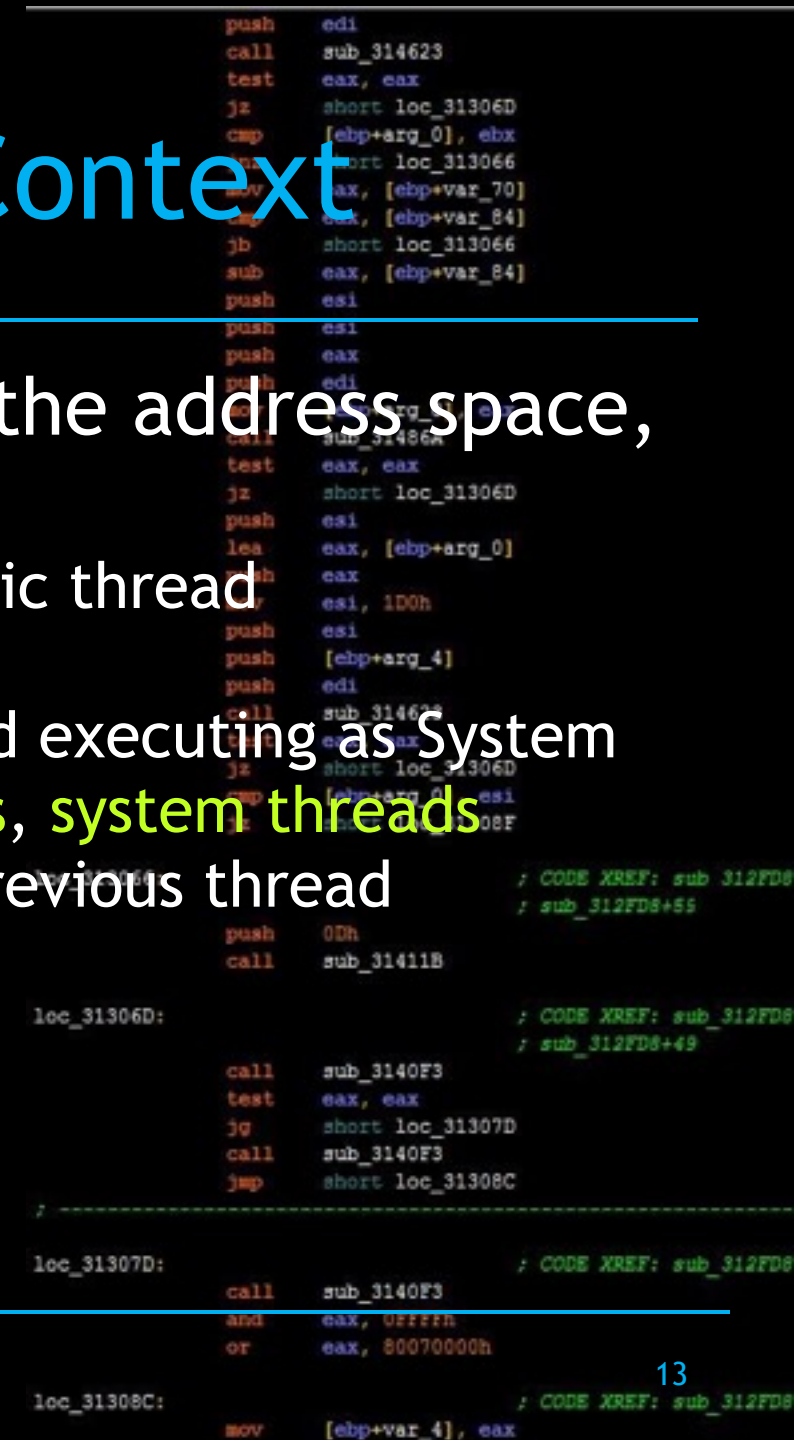


Processes and Threads

- Threads and processes are defined by the ETHREAD and EPROCESS structures, respectively
  - ETHREAD and EPROCESS contain basic housekeeping info
    - ETHREAD contains a KTHREAD structure
    - EPROCESS contains a KPROCESS structure
  - KTHREAD and KPROCESS contain scheduling information
  - The PEB and TEB are analogous user-mode data structures

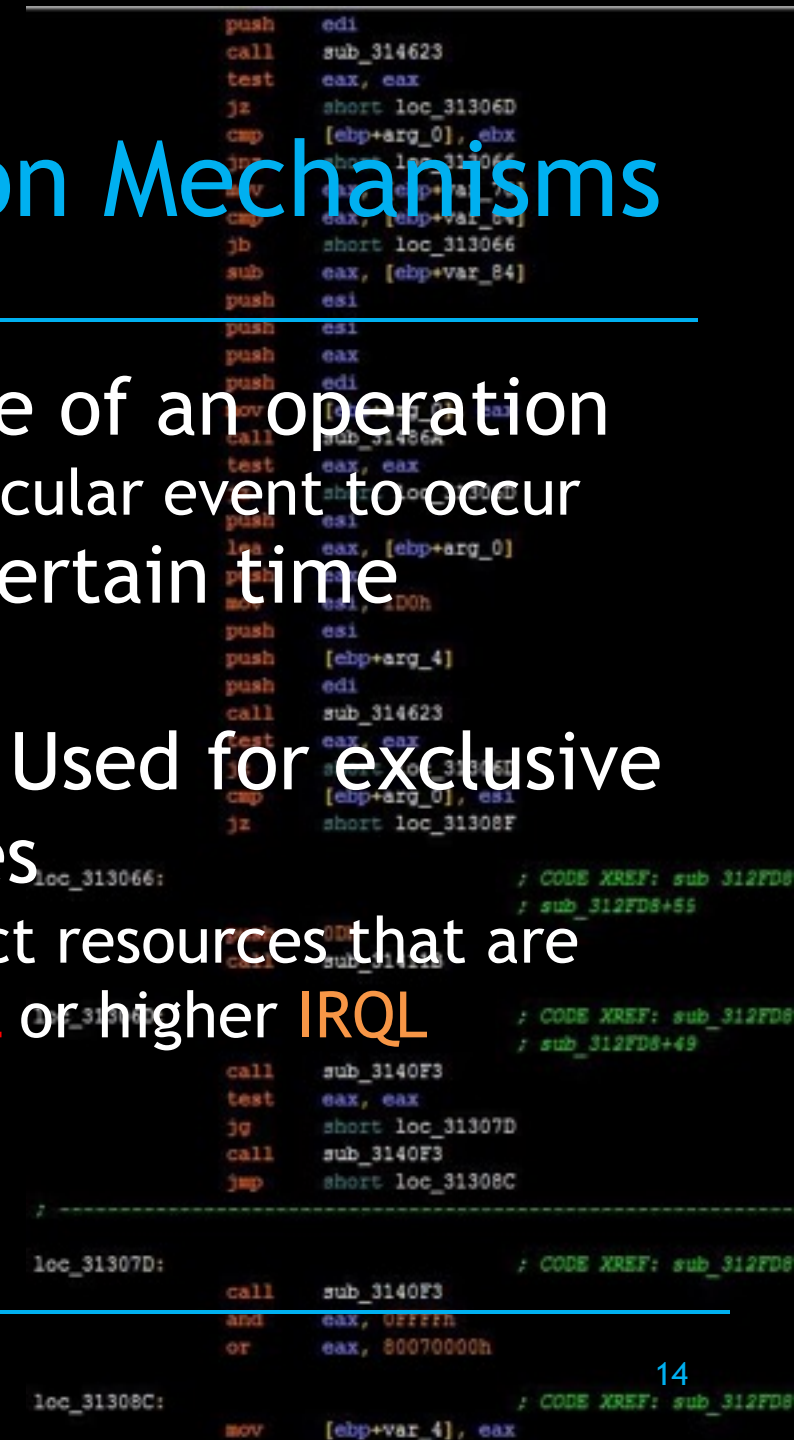
Thread Execution Context

- Thread Contexts contain the address space, security token, etc.
  a. Thread - Context of a specific thread
     - IOCTL handler, APC's
  b. System - Context of a thread executing as System
     - DriverEntry(), work items, system threads
  c. Arbitrary - Context of the previous thread
     - DPC's, timers



Kernel Synchronization Mechanisms

- Events - Indicate the state of an operation
  - A thread can wait for a particular event to occur
- Timers - Indicate that a certain time interval has passed
- Spin locks and Mutexes - Used for exclusive access to shared resources
  - Spin locks are used to protect resources that are accessed at DISPATCH_LEVEL or higher IRQL

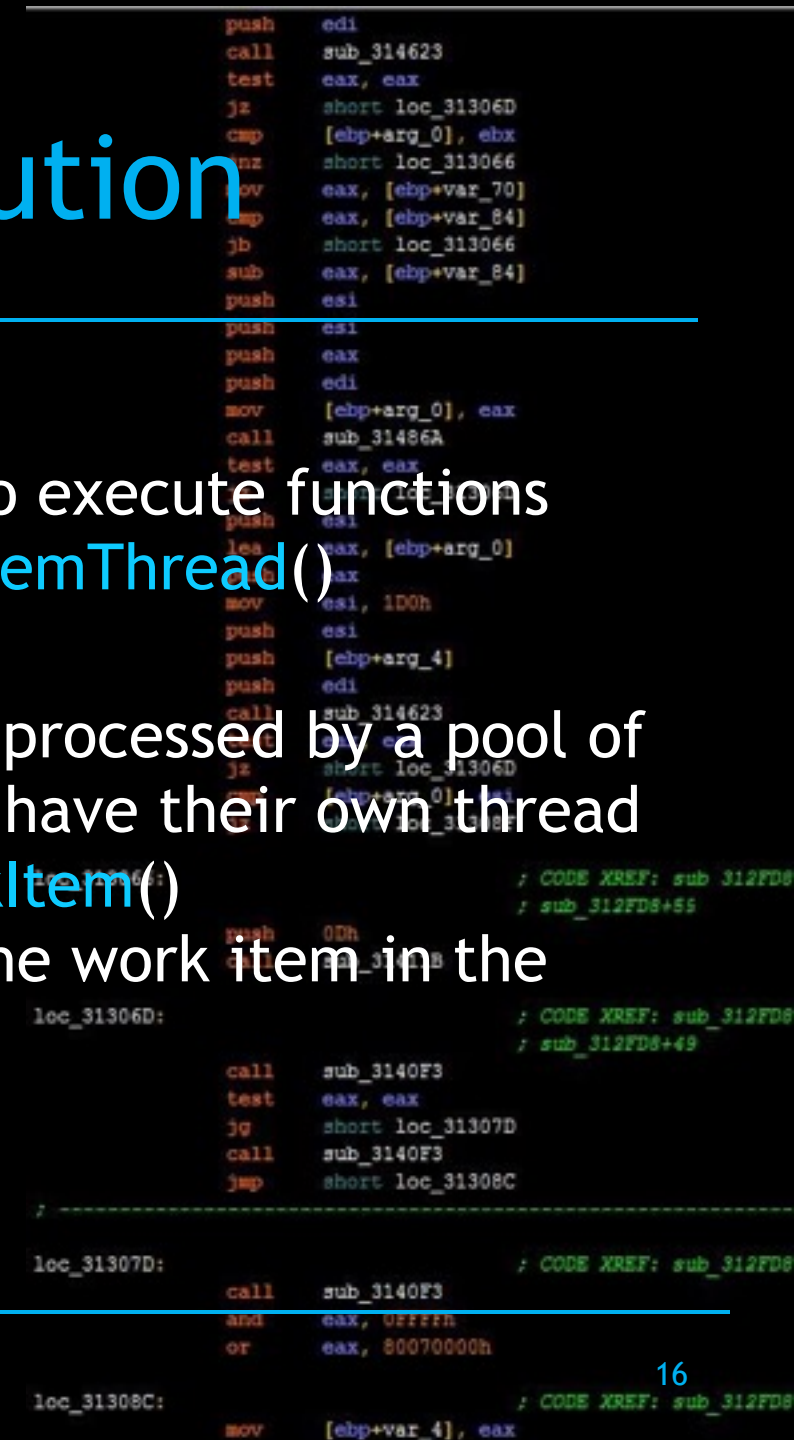


Agenda

- Kernel Basics
- Windows API
- Windows Drivers & Devices
- Kernel Debugging

Asynchronous Execution

- System Threads
  - Drivers can create threads to execute functions
  - Initialized with PsCreateSystemThread()
- Work Items
  - They are objects in a queue processed by a pool of system threads, they do not have their own thread
  - Initialized by IoAllocateWorkItem()
  - IoQueueWorkItem() places the work item in the PRCB

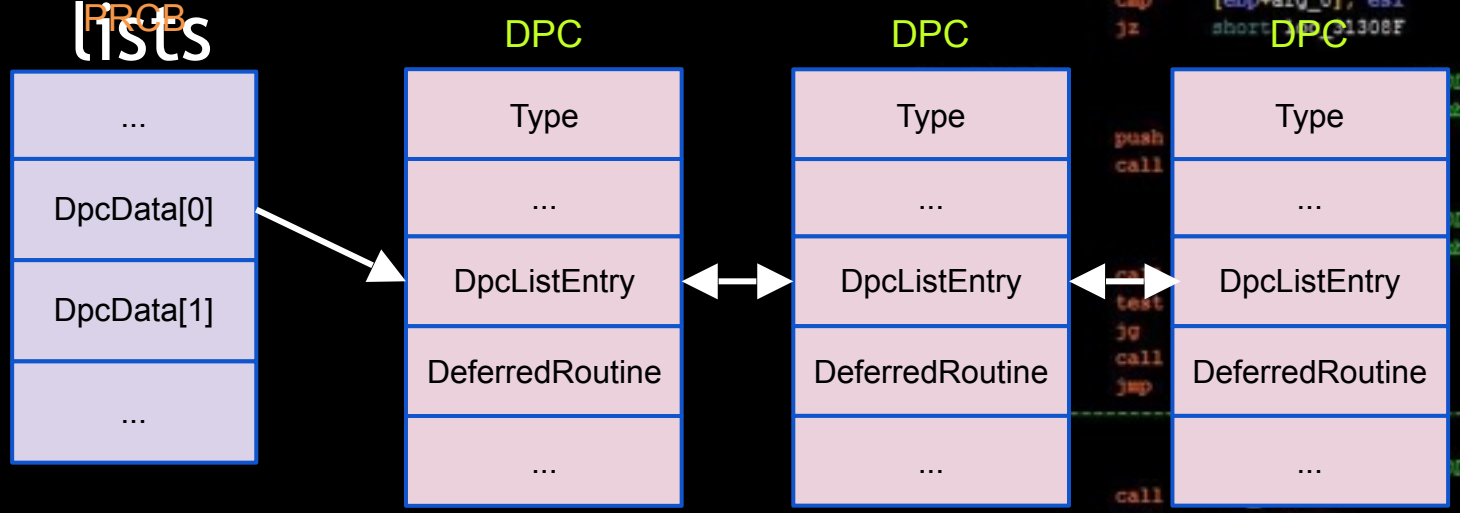


Asynchronous Procedure Calls (APC)

- The functions that are executed in a particular thread context
  - Defined by the KAPC structure
  - Executes at either PASSIVE_LEVEL or APC_LEVEL
  - Initialized by KeInitializeApc() and inserted into a KTHREAD's queue with KeInsertQueueApc()
- APC's are frequently used by rootkits
  - Clean way to inject code into user-land from kernel-land

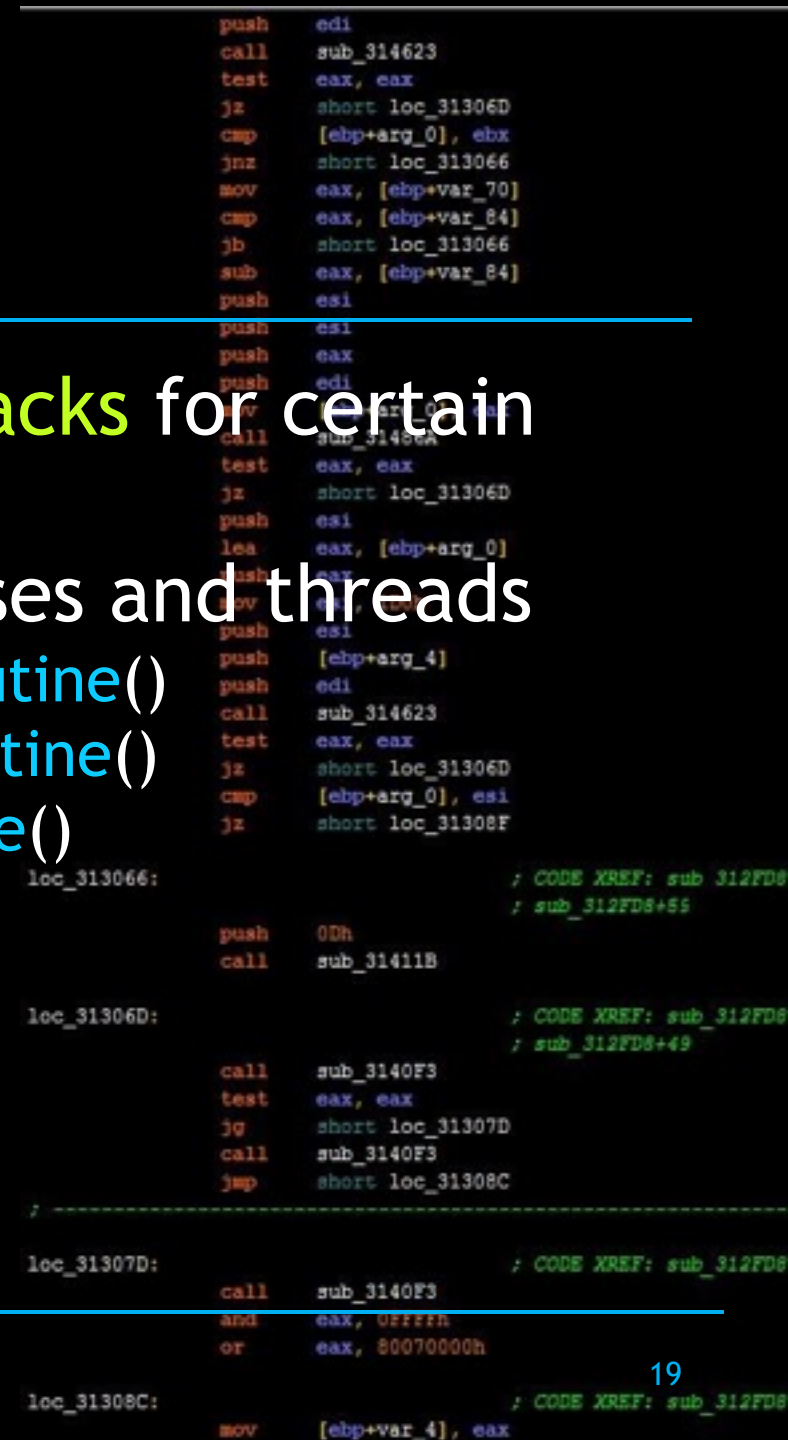
Deferred Procedure Calls (DPC)

- Routines executed at DISPATCH_LEVEL
  - Used to quickly execute short tasks
  - KeInitializeDpc() and KeInsertQueueDpc()
  - Rootkits use DPC's to access global linked lists



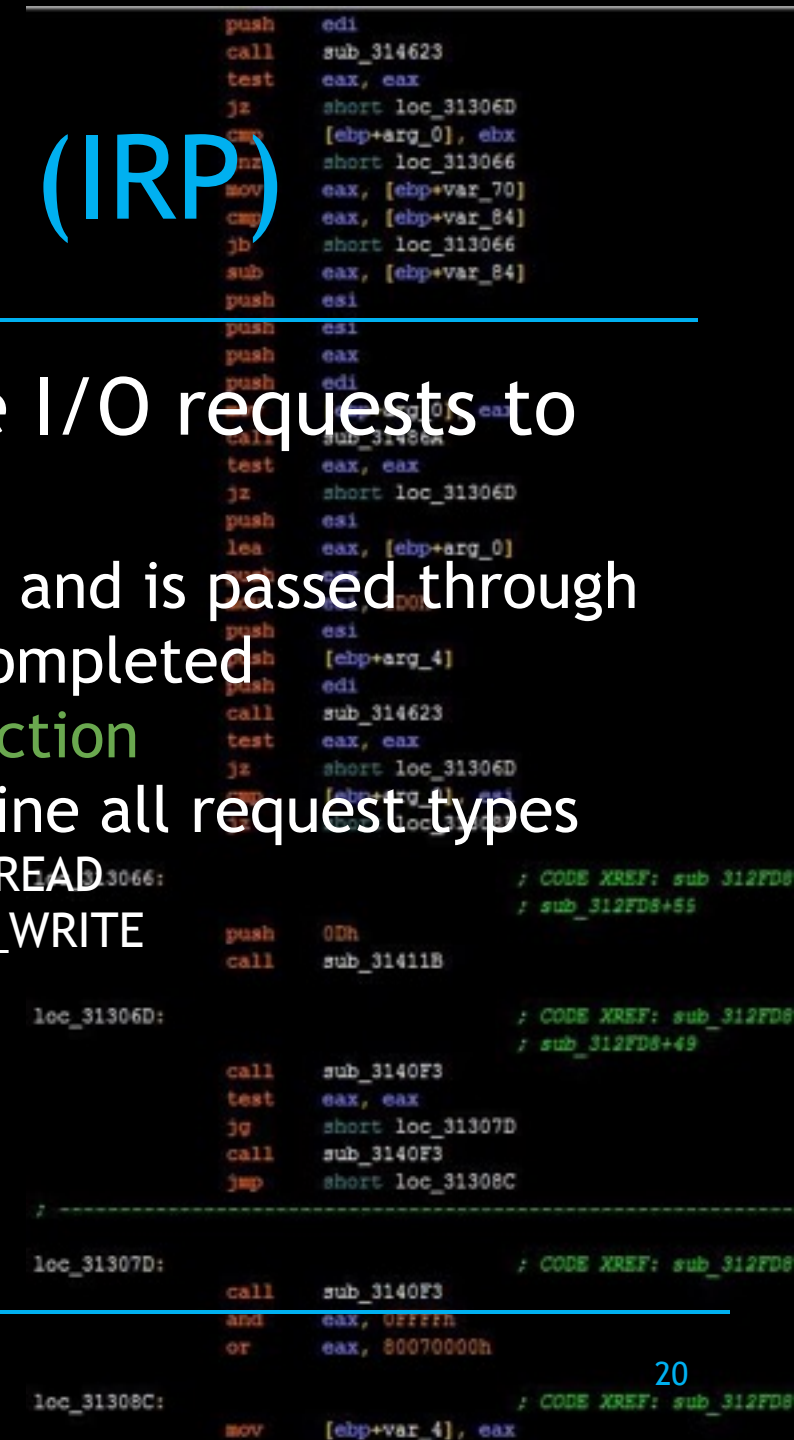
Callback Functions

- Drivers can register callbacks for certain system events
- Usually related to processes and threads
  - PsSetCreateProcessNotifyRoutine()
  - PsSetCreateThreadNotifyRoutine()
  - PsSetLoadImageNotifyRoutine()



I/O Request Packet (IRP)

- IRP's are used to describe I/O requests to kernel components
  - Each IRP describes a request and is passed through multiple devices until it is completed
  - MajorFunction and MinorFunction
    - Predefined lists that define all request types
      - File read request → IRP_MJ_READ
      - File write request → IRP_MJ_WRITE

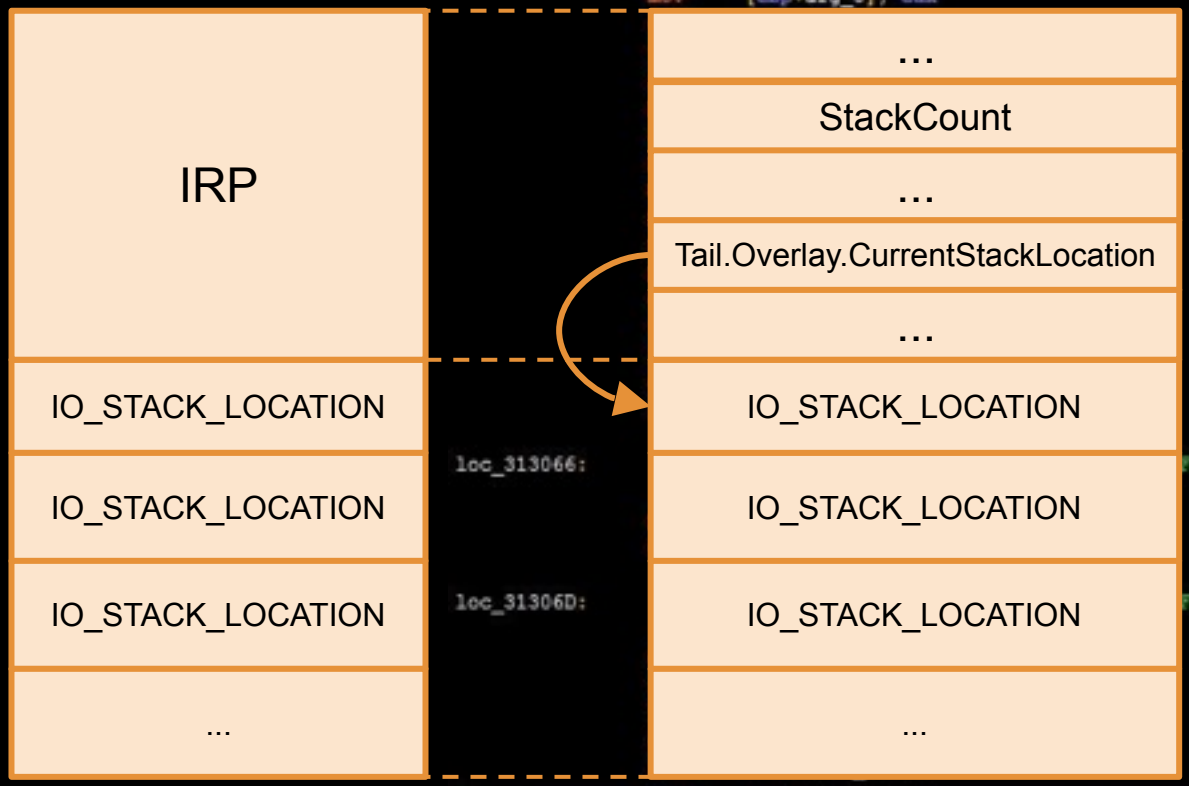


I/O Request Packet (IRP)

One IO_STACK_LOCATION structure allocated per device in the device stack

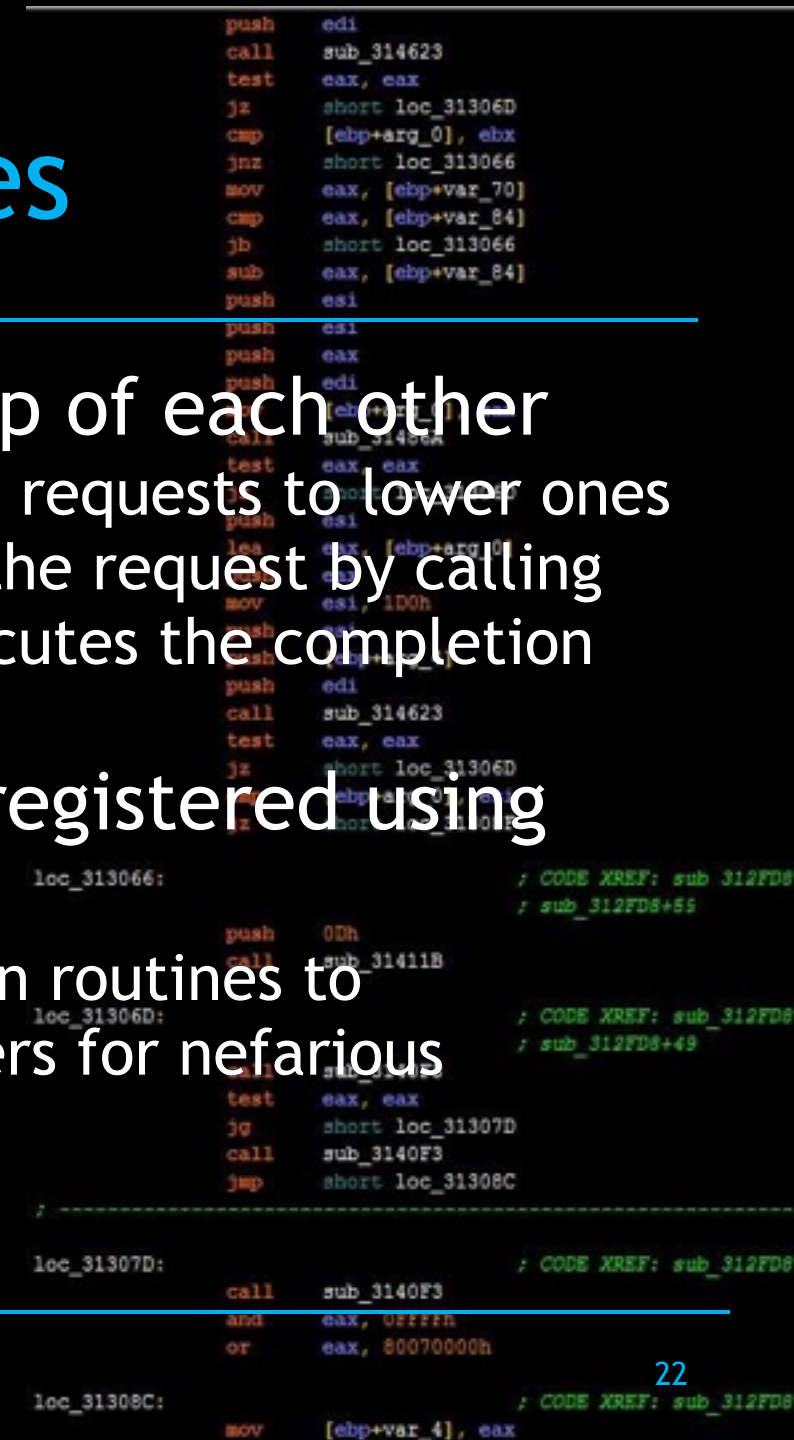
Each device is responsible for preparing the next IO_STACK_LOCATION

CurrentStackLocation pointer moves upward, towards the IRP



Completion Routines

- Devices are layered on top of each other
  - Higher level drivers can pass requests to lower ones
  - Whichever layer completes the request by calling IoCompleteRequest() → executes the completion routine
- Completion routines are registered using IoSetCompletionRoutine()
  - Rootkits often set completion routines to manipulate lower level drivers for nefarious purposes!

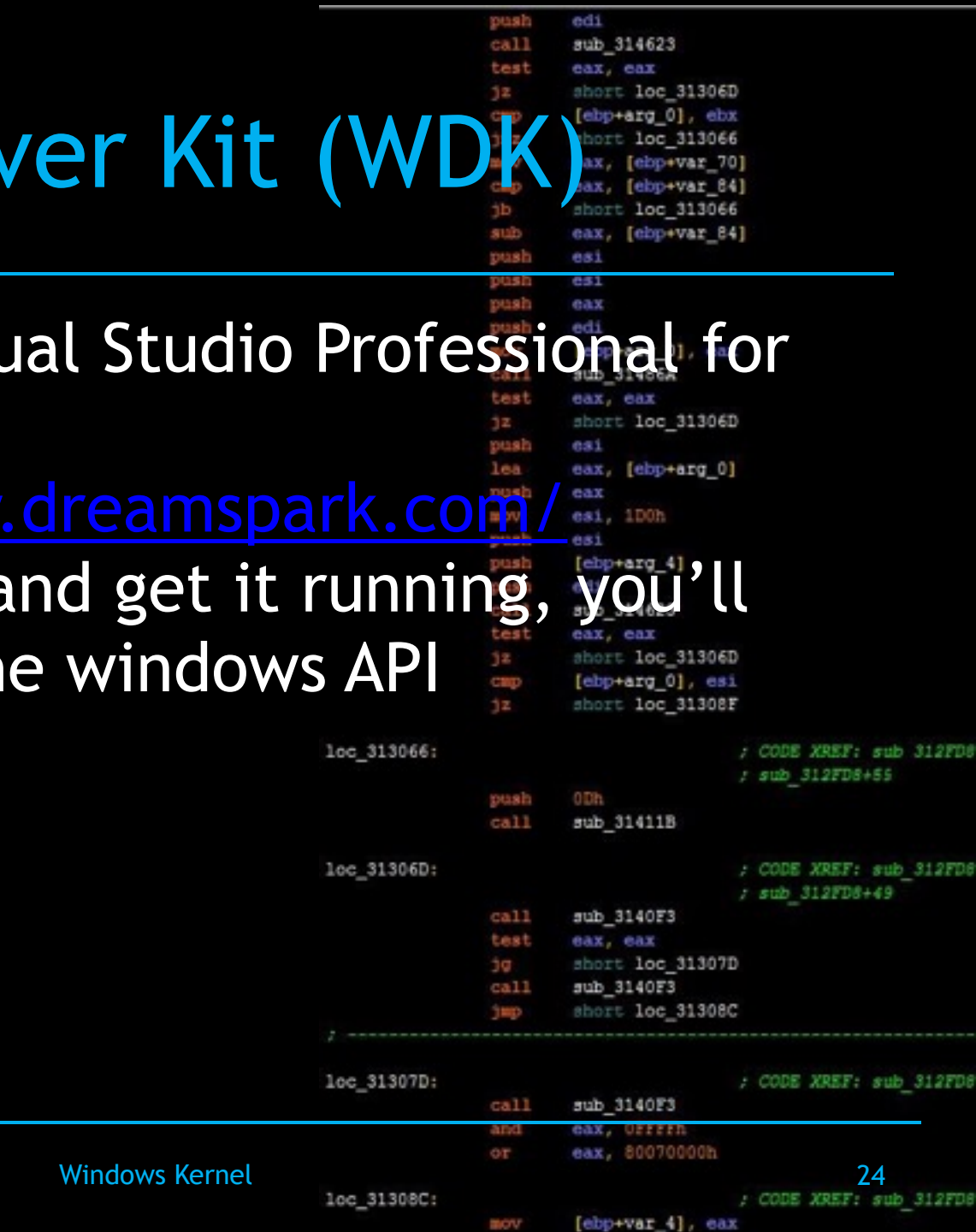


Agenda

- Kernel Basics
- Windows API
- Windows Drivers & Devices
- Kernel Debugging

Windows Driver Kit (WDK)

- You all have Visual Studio Professional for free!
  - https://www.dreamspark.com/
  - Download it and get it running, you'll need it for the windows API

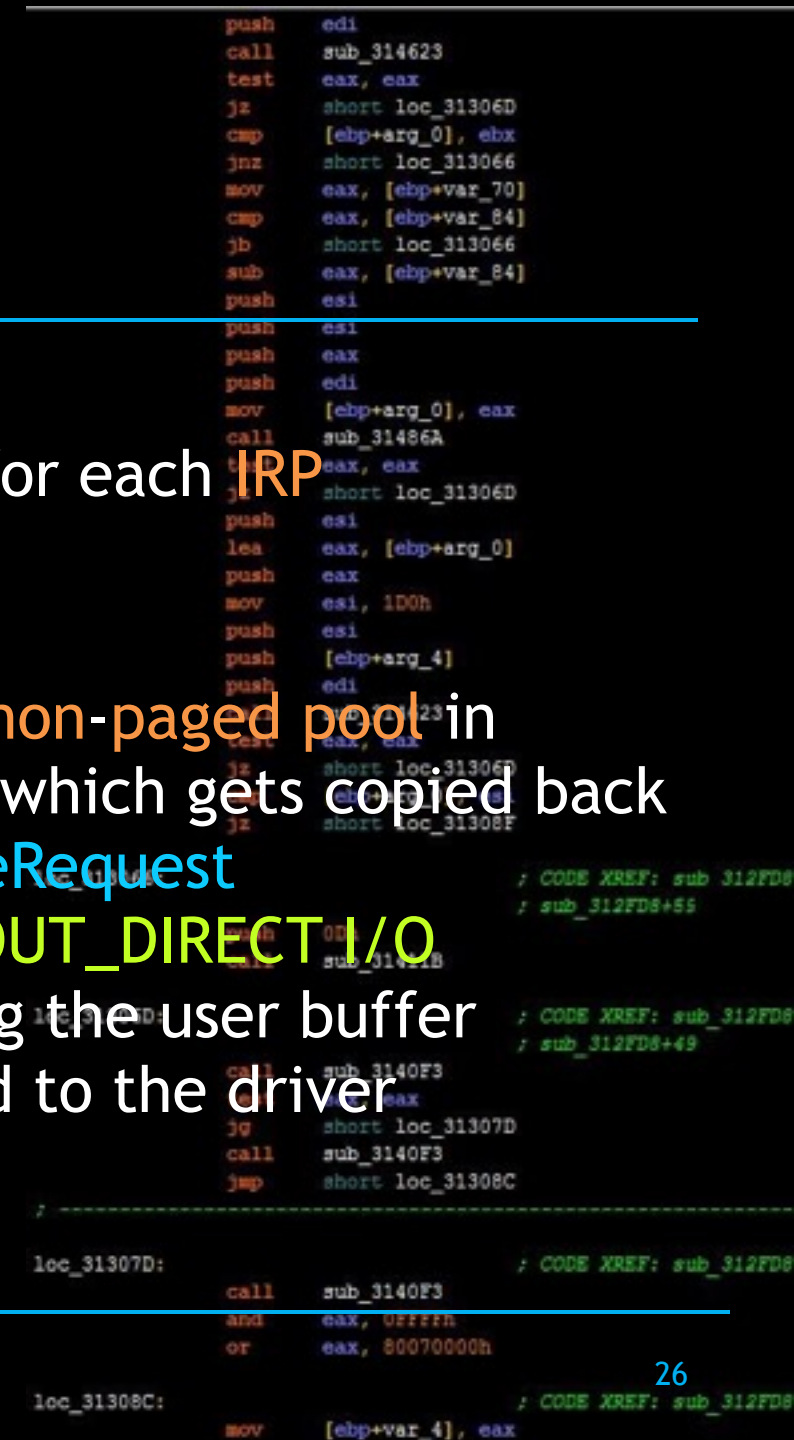


Driver Basics

- DriverEntry
  - Think of a driver as a DLL for the kernel
    - The DriverEntry function is immediately executed when a driver is loaded
    - Initializes any driver-specific settings and IRP routines
- Driver Objects
  - One per driver, tracks driver data for the kernel
- Device Objects
  - One per device, contains device-specific data

Driver Basics

- IRP Handling
  - DriverEntry makes routines for each IRP MajorFunction
  - METHOD_BUFFERED I/O
    - User buffer is copied into a non-paged pool in AssociatedIrp.SystemBuffer, which gets copied back to userland upon IoCompleteRequest
  - METHOD_IN_DIRECT/METHOD_OUT_DIRECT I/O
    - Driver gets an MDL describing the user buffer
  - Neither - The raw data is passed to the driver



IOCTL

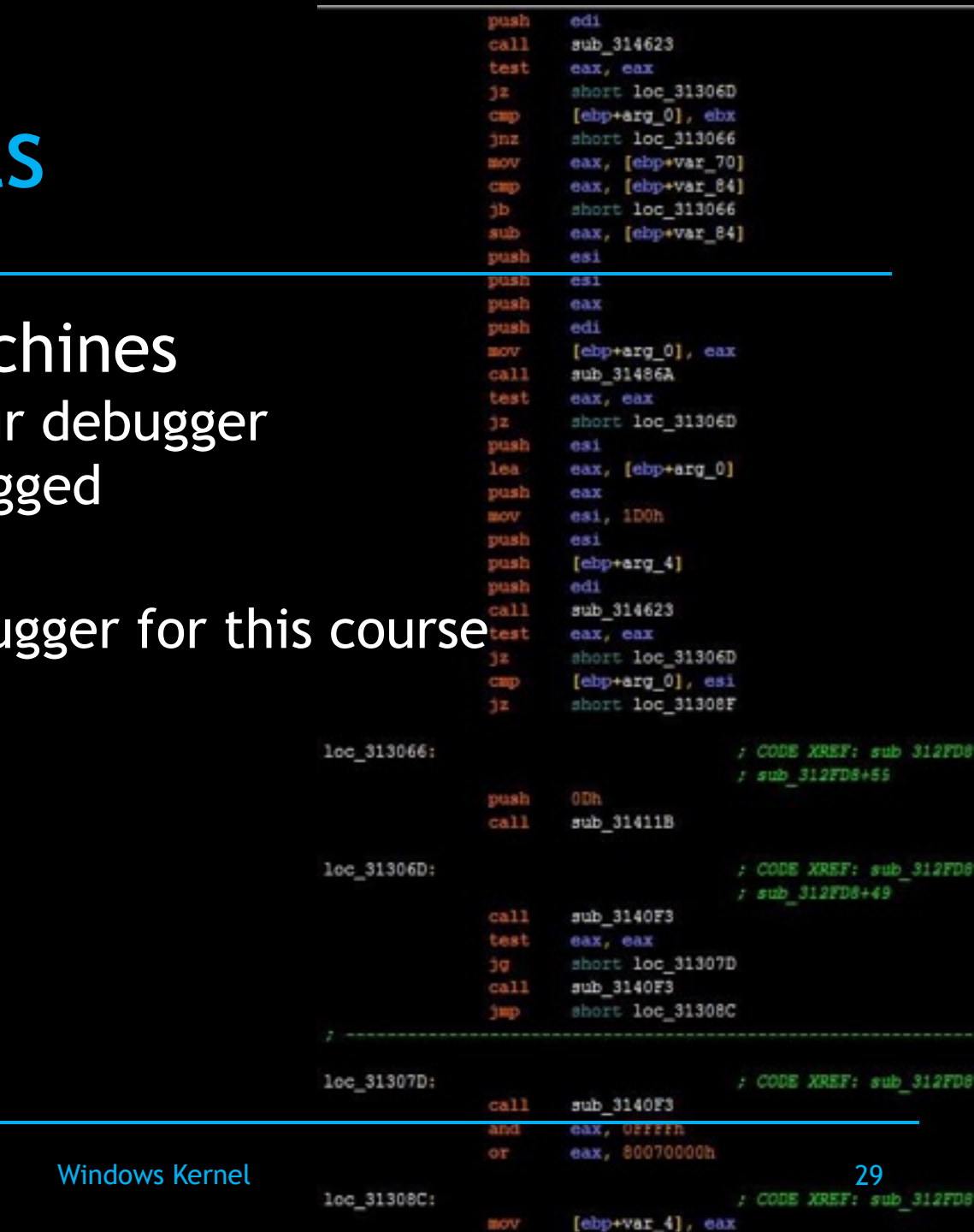
- IRP_MJ_DEVICE_CONTROL
  - Special IRP MajorFunction type
  - The driver switches over an IOCTL code
  - See skeleton-rootkit for usage

Agenda

- Kernel Basics
- Windows API
- Windows Drivers & Devices
- Kernel Debugging

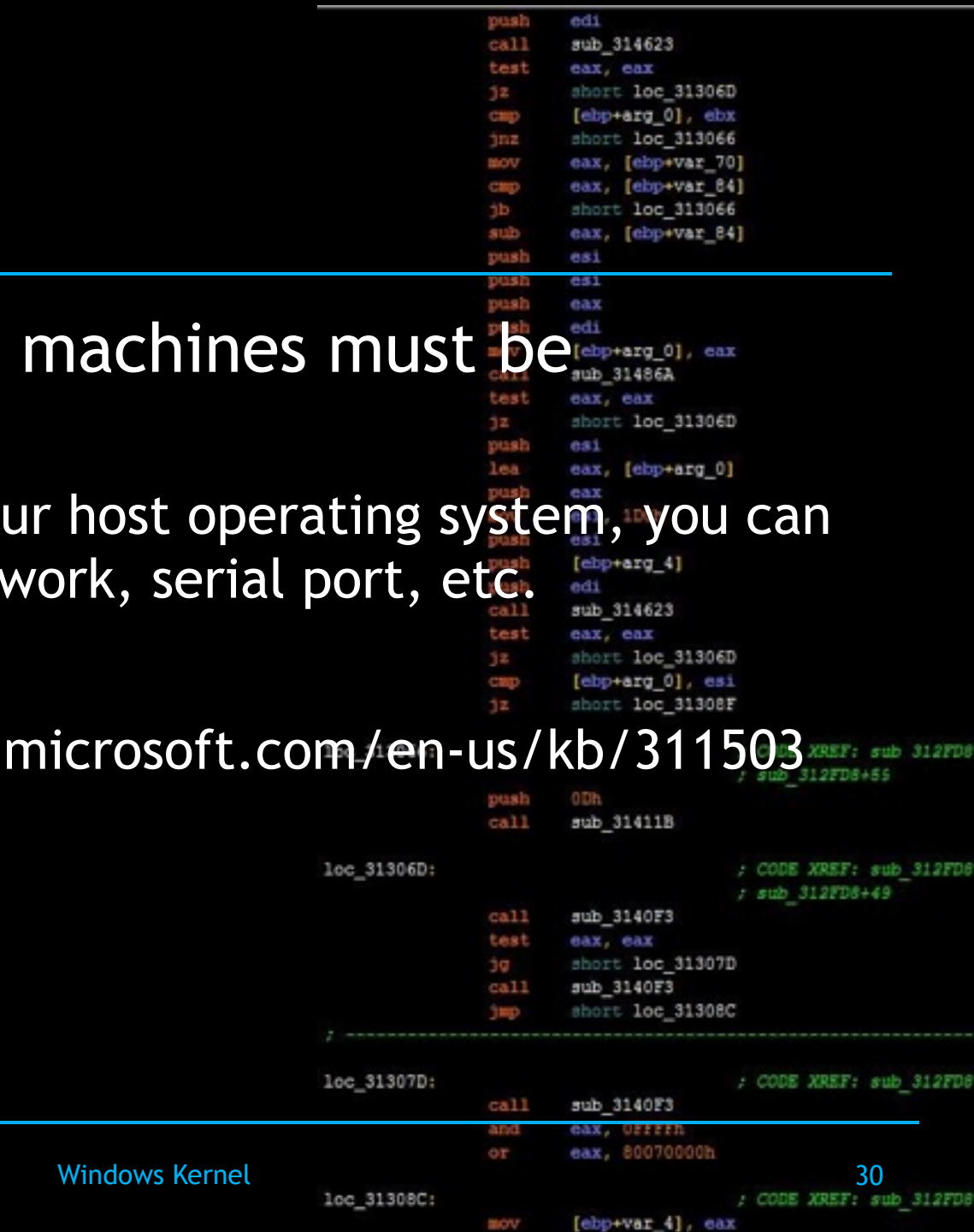
Requirements

1. Two virtual machines
   a. One running your debugger
   b. One being debugged
2. WinDBG
   a. Our chosen debugger for this course



Set Up

- Your two virtual machines must be connected
  - Depending on your host operating system, you can use a virtual network, serial port, etc.
- Symbols
  - https://support.microsoft.com/en-us/kb/311503



Demo Driver!

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+55
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

Questions?

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+55
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

References

1. Dang, Bruce, and Alexandre Gazet. *Practical Reverse Engineering: X86, X64, ARM, Windows Kernel, Reversing Tools, and Obfuscation*. Print.